D R A F T

The OSKit: The Flux Operating System Toolkit Version 0.97

The Flux Research Group

Department of Computer Science
University of Utah

Salt Lake City, UT, USA 84112

http://www.cs.utah.edu/projects/flux/oskit/

oskit@flux.cs.utah.edu

January 15, 1999

Contents

Part I

Design and Organization

Chapter 1

Introduction

Caveat: This document is really two different documents in one. Much of the expository text (rationale, extended descriptions) belongs in an overview, background, or introductory document. The more concise "man pages" belong in an API reference manual. And, of course, a tutorial is needed. Lacking a tutorial, we suspect the best thing for new users to do is to scan this introductory chapter and then look over and play with some of the small example programs, outlined below in section 1.6.1. Feedback appreciated, and bear with us.

1.1 Goals and Scope

The OSKit is a framework and set of modularized components and library code, together with extensive documentation, for the construction of operating system kernels, servers, and other OS-level functionality. Its purpose is to provide, as a set of easily reusable modules, much of the infrastructure "grunge" that usually takes up a large percentage of development time in any operating system or OS-related project, and allow developers to concentrate their efforts on the unique and interesting aspects of the new OS in question. The goal is for someone to be able to take the OSKit and immediately have a base on which they can start concentrating on "real" OS issues such as scheduling, VM, IPC, file systems, security, or whatever. Alternately they can concentrate on the real language issues raised by using advanced languages inside operating systems, such as Java, Lisp, Scheme, or ML—instead of spending six months first writing boot loader code, startup code, device drivers, kernel printf and malloc code, a kernel debugger, etc. With the recent addition of extensive multithreading and sophisticated scheduling support, the OSKit also provides a modular platform for embedded applications.

Although it can provide a complete OS environment for many domains, the primary intention of this toolkit is not to "write the OS for you"; we certainly want to leave the OS writing to the OS writer. The dividing line between the "OS" and the "OS toolkit," as we see it, is basically the line between what OS writers *want* to write and what they would otherwise *have* to write but don't really want to. Naturally this will vary between different OS groups and developers. If you really want to write your own x86 protected-mode startup code, or have found a clever way to do it "better," you're perfectly free to do so and simply not use the corresponding code in our toolkit. However, our goal is that the toolkit be modular enough that you can still easily use *other* parts of it to fill in other functional areas you don't want to have to deal with yourself (or areas that you just don't have time to do "yet").

As such, the toolkit is designed to be usable either as a whole or in arbitrary subsets, as requirements dictate. It can be used either as a set of support libraries to be linked into an operating system kernel and/or its support programs, or it can be used merely as a collection of "spare parts": example source code to be ripped apart and cannibalized for whatever purpose. (Naturally, we prefer that the toolkit be used in library fashion, since this keeps a cleaner interface between the toolkit and the OS and makes them both easier to maintain; however, we recognize that in some situations this will not be practical for technical or political reasons.)

The toolkit is also intended to be useful for things that aren't kernels but are OS-related, such as boot loaders or OS-level servers running on top of a microkernel.

1.2 Road Map

The facilities provided by the OSKit are currently organized into three main categories, corresponding to the three main sections of this manual: *interfaces*, *function libraries*, and *component libraries.* Note that the distinction between the two types of libraries has lessened since the majority of this document was written, and in some cases is rather arbitrary. There is also a small *Utilities* section (V) describing useful OSKit programs; this currently describes only the "NetBoot" utility.

1.2.1 Interfaces

The OSKit's interfaces are a set of clean, object-oriented interfaces specified in the framework of the Component Object Model (COM), described in Chapter 4. These interfaces are made available through thoroughly commented public C header files, with corresponding documentation in Part II of this manual. For example, the OSKit provides a "block I/O" interface for communication between file systems and disk device drivers, a "network I/O" interface for communication between network device drivers and protocol stacks, and a file system interface similar to the "VFS" interface in BSD. These interfaces are used and shared by the various OSKit components in order to provide consistency and allow them to be used together easily. The OS developer may additionally use or build on these interfaces in defining the fundamental structure of the client OS, but is not required to do so; alternatively, the developer may simply use them when writing the "glue" code necessary to incorporate a particular OSKit component into a new or existing OS environment.

1.2.2 Function Libraries

The OSKit's function libraries provide basic low-level services in a traditional C-language function-oriented style. For example, the OSKit provides libraries exporting kernel bootstrap support, standard C library functions, and executable program loading. These libraries are designed to be usable and controllable in a very fine-grained manner, generally on a function-by-function basis, allowing the client OS to easily use particular library functions while leaving out or individually overriding other functions. The dependencies between library functions are minimized, as are dependencies of library functions on other libraries; where these dependencies inevitably exist, they are well-defined and explicitly exposed to the OS developer. In general, the implementation details of these libraries are intentionally exposed rather than hidden. The function libraries make only minimal use of the OSKit's object-oriented COM interfaces, instead generally defining their own function-oriented interfaces in ordinary C header files. This design strategy provides maximum power and flexibility to the OS developer at the cost of increasing the dependence of the client OS on the implementation details of the libraries; we have found this to be a good design tradeoff for the low-level OSKit facilities which usually must be customized extensively in order to fit into any particular OS environment.

Following is a summary of the function libraries currently provided by the OSKit along with the chapter numbers in which each is described:

9 `liboskit_c`: A simple, minimal C library which minimizes dependencies with the environment and between modules, to provide common C library services in a restricted OS environment. For example, this library provides many standard string, memory, and other utility functions, as well as a formatted I/O facility (e.g., `printf`) designed for easy use in restricted environments such as kernels.

10 `liboskit_kern`: Kernel support code for setting up a basic OS kernel environment, including providing default handlers for traps and interrupts and such. This library includes many general utilities useful in kernel code, such as functions to access special processor registers, set up and manipulate page tables, and switch between processor modes (e.g., between real and protected mode on the x86). Also includes facilities for convenient source-level remote debugging of OS kernels under development.

11 `liboskit_smp`: More kernel support code, this library deals with setting up a multiprocessor system to the point where it can be used by the operating system. Also (to be) included are message-passing routines and synchronization primitives, which are necessary to flush remote TLBs.

4 `liboskit_com`: Utility functions for handling COM interfaces, and a suite of generic wrapper components. These wrappers map one OSKit COM interface to another, or implement simple functionality such as synchronization by creating proxy COM objects that wrap the COM objects provided by more primitive components.

6 `liboskit_dev`: This library provides default implementations of the "glue" functions required by device drivers and other "large" encapsulated components (networking, filesystems) imported from other operating systems, running in the OSKit Device Driver (née "fdev") framework. (The framework's current name reveals its heritage; today, a more accurate name would be the "OS Environment" framework.) The default implementations of these functions are designed for simple kernels using `liboskit_kern`; for more elaborate kernel (or user-mode) environments, the client OS will have to override some or all of the functions in this library.

The following four libraries are provided mainly for convenience; they are not as general or as well documented as the other libraries but are still useful. See the source directories and READMEs for details.

`liboskit_startup`: Contains functions to start up and initialize various OSKit components, making it easier to write OSKit programs.

`liboskit_unsupp`: Contains various unsupported hacks and utilities.

`liboskit_unix`: Provides the necessary support to debug and run certain OSKit components on FreeBSD in user-mode. One can tweak this to run on Linux.

`liboskit_fudp`: Provides a "Fake UDP" implementation: a restricted send-only no-fragmenting version of UDP. This can be useful with the hpfq library (in Chapter 29).

1.2.3 Component Libraries

Finally, the OSKit's component libraries provide generally higher-level functionality in a more standard, coarse-grained, object-oriented "black box" design philosophy. Although the OSKit's "components" are also packaged into ordinary link libraries by default, their structure represents more of a component-oriented design than a traditional C function-oriented design. In contrast with the OSKit's function libraries, the component libraries typically export relatively few public entrypoints in comparison to the amount of functionality provided. For example, in the Linux and BSD device driver component libraries (see Chapters 30 and 31), each entire device driver is represented by a single function entrypoint which is used to initialize and register the driver as a whole. The client OS generally interacts with these components through the OSKit's object-oriented COM interfaces, allowing many components and instances of components to coexist and interact as defined by the OS developer. This design strategy has proven to be most appropriate when incorporating large chunks of existing code from existing systems such as BSD and Linux, where it is more important to hide the details of the original environment than to provide the most flexibility.

Following is a summary of the component libraries currently provided by the OSKit along with the chapter numbers in which each is described:

- **POSIX emulation and libraries**

  14 `liboskit_freebsd_c`: Complete POSIX-like C library derived from FreeBSD, providing both single- and multithreaded configurations. This library is an alternative to the minimal C library `liboskit_c` (above), intended for programs that use substantial POSIX-like functionality or need thread-safety. The FreeBSD "system call" layer is replaced with glue code that uses the OSKit COM interfaces, while the higher-level code is mostly unchanged from that found in FreeBSD.

  15 `liboskit_freebsd_m`: Complete standard math library (taken from FreeBSD's `libm`). The functions in this library will commonly be needed by programs that use floating point.

- **Memory management**

16 `liboskit_lmm`: A flexible memory management library that can be used to manage either physical or virtual memory. This library supports many special features needed by OS-level code, such as multiple memory types, allocation priorities, and arbitrary alignment and placement constraints for allocated blocks.

17 `liboskit_amm`: The *Address Map Manager* library manages collections of resources where each element of a collection has a name (address) and some set of attributes. Examples of resources that might be managed by address maps include swap space and process virtual address space.

18 `liboskit_svm`: The *Simple Virtual Memory* library uses the AMM library to define a simple virtual-memory interface for a single address space that can provide memory protection and paging to a block device such as a disk partition. (`unsupported/redzone.c` provides a stack redzone for single-threaded kernels, without all of SVM.)

- **Threads, synchronization, and scheduling**

19 `liboskit_threads`: This library provides support for multithreaded kernels, including POSIX threads, synchronization, scheduling, and stack guards. Scheduling policies are the standard POSIX Round-Robin and FIFO. Experimental support for CPU inheritance scheduling, a hierarchical framework for arbitrary scheduling policies, is also provided, although not integrated or robust. Provided policies include rate-monotonic, stride (WFQ), and lottery scheduling.

- **Development aids**

20 `liboskit_memdebug`: This library provides debugging versions of `malloc` et al that check for a variety of bugs related to memory-allocation (overruns, use of freed blocks, etc).

21 `liboskit_gprof`: Run-time support code for an OSKit kernel to collect profiling data about itself and report it at the end of a run. Profiling data can be collected for kernel code compiled with the "-pg" option (as for use with Unix `gprof`), and profiled "_p" versions of all OSKit libraries are provided to profile OSKit code used by the application kernel.

- **Simple disk/file reading and loading**

22 `liboskit_diskpart`: A generic library that recognizes various common disk partitioning schemes and produces a complete "map" of the organization of any disk. This library provides a simple way for the OS to find relevant or "interesting" disk partitions, as well as to easily provide high-level access to arbitrary disk partitions through various naming schemes; BSD- and Linux-compatible naming mechanisms are provided as defaults.

23 `liboskit_fsread`: A simple read-only file system interpretation library supporting various common types of file systems including BSD FFS, Linux ext2fs, and MINIX file systems. This library is typically used in conjunction with the partition library to provide a convenient way for the OS to read programs and data off of hard disks or floppies. Again, this functionality is often needed at boot time even in operating systems that otherwise would not require it. This code is also extremely useful in constructing boot loaders.

24 `liboskit_exec`: A generic executable interpreter and loader that supports popular executable formats such as `a.out` and ELF, either during bootstrap or during general operation. (Even microkernel systems, which normally don't load executables, generally must have a way to load the first user-level program; the OSKit's small, simple executable interpreter is ideally suited to this purpose.)

- **Filesystem implementations**

25 `liboskit_linux_fs`: Encapsulated Linux 2.0.29 filesystem code. Includes the Linux VFS layer supporting ext2fs, the primary Linux filesystem, as well as numerous other PC filesystems supported under Linux.

26 `liboskit_netbsd_fs`: Encapsulated NetBSD 1.2 filesystem code. Includes the BSD VFS layer supporting the local FFS filesystem.

- **Networking implementations**

  27 `liboskit_freebsd_net`: Encapsulated FreeBSD 2.1.7.1 networking code. Includes socket layer and protocol support wrapped to use the OSKit's framework.

  28 `liboskit_bootp`: This library provides a simple interface for performing the BOOTP protocol (RFC 1048/1533) on Ethernet devices to retrieve a canonical set of parameters from a server, based on the client's Ethernet hardware address.

  29 `liboskit_hpfq`: This library provides hierarchical proportional-share control of outgoing network link bandwidth, described in the Bennet/Zhang SIGCOMM'96 paper.

- **Device driver implementations**

  30 `liboskit_linux_dev`: Encapsulated Linux 2.0.29 device driver set. Currently includes over 50 block (SCSI, IDE) and network drivers wrapped to use the OSKit's device driver framework. See the source file `linux/dev/README` for a list of devices and their status.

  31 `liboskit_freebsd_dev`: Encapsulated FreeBSD 2.1.7.1 device driver set. Currently includes eight TTY (virtual console and serial line, including mouse) drivers wrapped to use the OSKit's device driver framework.

- **Video and window manager implementations**

  32 `liboskit_wimpi`: Simple hierarchical windowing system based on MGR, with only simple drawing and window management operations.

  33 `liboskit_*video*`: Basic video support, with two implementations: one encapsulating all of SVGALIB 1.3.0, and one based on XFree86 3.3.1, but with only the S3 driver currently supported. We also provide support for X11 clients.

1.3 Overall Design Principles

This section describes some of the general principles and policies we followed in designing the components of the OSKit. This section is most relevant for people developing or modifying the toolkit itself; however, this information may also help users of the toolkit to understand it better and to be able to use it more effectively.

- Cleanly separate and clearly flag architecture- and platform-specific facilities. Although the OSKit currently only runs on the x86 architecture, we plan to port it to other architectures such as the StrongARM in the future. (We will also help with ports by others, e.g., to the DEC Alpha.) Architecture-specific and platform-specific features and interfaces are tagged in this document with boxed icons, e.g., $\boxed{\text{X86}}$ indicating the Intel x86 processor architecture, and $\boxed{\text{X86 PC}}$ representing x86-based PC platforms.

- Attempt to make the OSKit's *interfaces* portable even in situations where their *implementation* can't be. Although by its nature a large percentage of the implementation of the OSKit is inherently machine-dependent, the interfaces to many of its facilities are very generic and should be usable on most any architecture or platform. For example, the OSKit's device driver interfaces are almost completely portable, even though the device drivers themselves generally aren't.

- Document inter-module dependencies within each library. For function libraries, this means documenting the dependencies between individual functions; for component libraries, this means documenting the dependencies between the different components collected in each library. This policy contrasts to most other third-party libraries, which are usually documented "black box" fashion: you are given descriptions of the "public" interfaces, and that's it. Although with such libraries you could in principle use part of the library independently from the rest or override individual library components with your own implementations, there is no documentation describing *how* to do so. Even if such documentation existed, these libraries often aren't well enough modularized internally, so replacing one library component would require understanding and dealing with a complicated web of relationships with other components.

The downside of this policy is that exposing the internal composition of the libraries this way leaves less room for the implementation to change later without affecting the client-visible interfaces. However, we felt that for the purposes of the OSKit, allowing the client more flexibility in using the library is more important than hiding implementation details.

- Where there is already a standard meaning associated with a symbol, use it. For example, our toolkit assumes that `putchar()` means the same thing as it does under normal POSIX, and is used the same way, even if it works very differently in a kernel environment. Similarly, the toolkit's startup code starts the kernel by calling the standard `main()` function with the standard `argc` and `argv` parameters, even if the kernel was booted straight off the hardware.

- It is generally safe to call initialization routines more than once with no harmful side-effects, except when the documentation says otherwise. Although this is often unnecessary in particular situations, it increases the overall robustness of the components and decreases hair-pulling since clients can always follow the simple rule, "if in doubt, initialize it."

1.4 Configuring the OSKit

The OSKit follows the GNU conventions for configuration, building, and installation; see the `INSTALL` file in the top-level source directory for general instructions on using GNU `configure` scripts. In short, you need to run the `configure` script that is in the top-level source directory of the OSKit; this script will attempt to guess your system type and locate various required tools such as the C compiler. You can configure the OSKit to build itself in its own source directory, simply by moving to that directory and typing `./configure`, or you can build the OSKit into a separate object directory by changing to that directory and running the `configure` script from there. For example, using a separate object directory allows you to put the object files on a local disk if the sources come across NFS, or on a partition that isn't backed up. Additionally, you can have multiple configurations of the OSKit at once (with different options or whatever), each in its own object tree but sharing the same sources.

To cross-compile the OSKit for another architecture, you will need to specify the host machine type (the machine that the OSKit will run on) and the build machine type (the machine on which you are building the toolkit), using the `--build=`*`machine`* and `--host=`*`machine`* options. Since the OSKit is a standalone package and does not use any include files or libraries other than its own, the operating system component of the host machine type is not directly relevant to the configuration of the OSKit. However, the host machine designator as a whole is used by the `configure` script as a name prefix to find appropriate cross-compilation tools. For example, if you specify '`--host=i486-linux`', the `configure` script will search for build tools called `i486-linux-gcc`, `i486-linux-ar`, `i486-linux-ld`, etc. Among other things, which tools are selected determines the object format of the created images (e.g., ix86-linux-* tools create ELF format, while ix86-mach-* tools create `a.out` format).

The OSKit's `configure` script accepts various standard options; for a full listing of the supported options, run `configure --help`. In addition, the `configure` script also supports the following options specific to the OSKit:

`--enable-debug`: Turn on debugging support in the compiler, and include debugging and sanity checking code within the OSKit. This option increases code size and reduces performance slightly, but also increases the likelihood of errors being detected quickly.

`--enable-profiling`: Generates profiling versions of all of the OSKit libraries; in keeping with the standard convention, the profiling versions of the libraries are suffixed with `_p`.

`--disable-asserts`: Compiles out assert() calls, for those who live on the edge.

`--enable-hpfq`: Build the HPFQ library and the special liboskit_linux_dev that's required.

`--enable-unixexamples`: Generates support code and example programs to debug and run some OSKit components in user-mode on top of Unix. FreeBSD is the only version of Unix on which this works unchanged. Tweaking it for other versions of Unix should not be hard and we've done so and run at least part of it on Linux.

`--enable-doc`: The build process will attempt to format the OSKit documentation. In addition to taking a long time (the complete documentation is over 475 pages), it requires LaTeX and `dvips`. Pre-formatted .ps and .html files are provided in the source tree.

`--enable-linux-bogomips=VALUE`: This option is only relevant to the Linux device driver and filesystem libraries. Prevents the Linux "BogoMips" calibration and instead hardcodes it to the value given. If the `=VALUE` is omitted, the default value of 300 is taken. *Warning: specifying this option may cause the Linux device drivers and filesystems to behave incorrectly. Use at your own risk.*

Before you do the actual OSKit build, there are some steps you can choose to take to make the build go faster, by not compiling parts you do not need.

- Edit `<oskit/dev/linux_ethernet.h>` to contain only lines for drivers you will need. In addition to making your builds go faster, with some hardware this may even be neccessary since some of the probing by incompatible drivers can hang your machine.
- Edit `<oskit/dev/linux_scsi.h>` to contain only lines for drivers you will need.
- Edit `<oskit/fs/linux_filesystems.h>` to contain only lines for filesystems you will need.
- Edit the top-level `GNUmakefile` (created by `configure`) to set `SUBDIRS` to only the directories you will need. This step should only be done after you are somewhat experienced with the OSKit and know what directories you need. An alternative to this is to specify the subdirs directly in `GNUmakefile.in` instead of using the value `@SUBDIRS@`. This has the advantage of persisting after future `configure` runs, which create a new `GNUmakefile` from `GNUmakefile.in`.
- Set your `CFLAGS` environment variable to `-O1 -pipe`. This will speed up compiles by doing less optimization than the default `-O2`, and may also make the compiler go faster if your compiler doesn't write its temporary files into a memory-based filesystem.

1.5 Building the OSKit

Building the OSKit currently *requires* these tools and versions:

- GNU `make`
- GNU CC (`gcc`) version 2.7.x. More recent `gcc` versions and `egcs` may work but have not been well tested.
- GNU `binutils` version 2.8.x, or 2.9.1 with BFD 2.9.1.

To build the OSKit, go to the top-level source directory (or the top-level object directory, if you configured the toolkit to build in a separate object directory), and run GNU `make` (e.g., just '`make`' on Linux systems, or '`gmake`' on BSD systems). Note that the OSKit *requires* GNU `make`. Don't even think about trying to get it to work with another version of `make`. You'll be better off porting GNU `make` to your system. Really. To avoid confusion, the OSKit's makefiles are named `GNUmakefile` rather than just `Makefile`; this way, if you accidentally run the wrong `make` utility, it will simply complain that it can't find any makefile, instead of producing an obscure error.

To build or rebuild only one specific part of the OSKit, such as one of the libraries, you can simply go into the appropriate subdirectory of the object tree and run GNU `make` from there. The top-level `GNUmakefile` essentially does nothing except recursively run `make` in each subdirectory, so it is always safe to run any OSKit makefile manually. A few OSKit directories depend on others being built first—for example, the example kernels cannot be built until the OSKit libraries they use have been generated—but most of the OSKit libraries can be built completely independently of each other.

Once the toolkit is built, you can install it with '`make install`'. By default, the libraries will go into `/usr/local/lib` and the header files into `/usr/local/include`, unless you specified a `--prefix` on the `configure` command line. All of the OSKit header files are installed in an `oskit/` subdirectory (e.g.

`/usr/local/include/oskit`), so they should not conflict with any header files already present. Although the libraries are installed in the main library directory (e.g., `/usr/local/lib`) and not a subdirectory, all the library names are prefixed with `oskit_` to avoid conflicts with other unrelated libraries you may want to place there. For example, the OSKit's minimal C library is named `liboskit_c.a` rather than just `libc.a`, allowing you to install a "real" C library in the same directory if desired.

The standard `make` variables such as `CFLAGS` and `LDFLAGS` are used by the OSKit's build rules but are not actually defined by any of the OSKit makefiles; thus they are available for use on the `make` command line. For example, you can type '`make CFLAGS="-save-temps"`' to cause GCC to leave its intermediate files in the object directory, such as the preprocessed C source and the assembly language compiler output. (Internally, the OSKit makefiles use OSKIT_ prefixes on most standard makefile variables.)

1.6 Using the OSKit

To use the OSKit, simply link your kernel (or servers, or whatever) with the appropriate libraries. Detailed information on how to use each library is provided in the appropriate chapters in this document. For initial experimentation with the OSKit, you can simply hack on the example kernels or create new kernels in the same directory (see Section 1.6.1 for a tour through our examples); however, once your own system grows beyond the stage of a simple demo kernel you will probably want to set up a separate source tree and link in the OSKit components from the installation directory.

Linking libraries into a kernel may seem strange at first, since all of the existing OS kernels that we have encountered seem to have a strong "anti-library" do-everything-yourself attitude. However, the linker can link libraries into a kernel just as easily as it can link them into application programs; we believe that the primary reason existing kernels avoid libraries is because the available libraries aren't *designed* to be used in kernels; they make too many assumptions about the environment they run in. Filling that gap is the purpose of the OSKit.

All of the OSKit libraries are designed so that individual components of each library can be replaced easily; we have taken pains to document the dependencies clearly so that clients can override whatever components they need to, without causing unexpected results. In fact, in many cases, particularly in the function libraries, it is *necessary* to override certain functions or symbols in order to make effective use of the toolkit. To override a library function or any other symbol defined by a library, just define your own version of it in your kernel or other client program; the linker will ensure that your definition is used instead of the library's. We strongly suggest that you use the linker to replace components of the OSKit, instead of making changes directly in the OSKit source (except, of course, to fix bugs in the OSKit). Maintaining a clean separation between the OSKit and your kernel will make things much easier when upgrading to a new version of the OSKit.

1.6.1 Example Kernels

If you are starting a new OS kernel, or just want to experiment with the OSKit in a "standalone" fashion, an easy way to begin is with one of the example "kernels" in the `examples` directory. These examples build up from the kernel equivalent of a "Hello World" application, demonstrating how to use various facilities separately or together, such as the base environment initialization code in `kern`, the minimal console driver code, the minimal C library, the remote debugging stub, and device, filesystem and network COM interfaces. The code implementing these examples is almost as small and simple as equivalent ordinary user-level applications would be because they fully rely on the OSKit to provide the underlying infrastructure necessary to get started. The compilation and linking rules in the `GNUmakerules` files for these example programs demonstrate how to link kernels for various startup environments.

The following example "kernels," and many more, are provided. See the `examples/README` file for a list of most of them, and their source for more detail.

- `hello`: What else? Absolutely useless, but you must have one!
- $\boxed{\text{X86 PC}}$ `multiboot`: Prints out info passed by the boot loader and info about the CPU.

- X86 PC `anno_test`: Simple example of the use of trap and interrupt annotations. See section 10.19 for details on annotations.

- `timer_com`: Shows a way to use timer interrupts.

- `blkio`: Demonstrates use of basic low-level disk access.

- `linux_fs_com`: Shows how to use the low-level filesystem interfaces to the Linux filesystems.

- `bmodfs`: Shows use of the BMOD filesystem and the POSIX layer over the OSKit FS interfaces. See section 10.20 for details on the BMOD filesystem.

- `pingreply`: Shows how to use the low-level network access.

- `socket_bsd`: Demonstrates use of the BSD socket layer over the low-level network interfaces.

- X86 PC `smp`: A simple MultiBoot kernel that demonstrates how to use the SMP support.

1.6.2 Booting Kernels

The example kernels, as well as custom kernels you build using the OSKit, can be booted from either the GRUB, Linux, Mach, or BSD boot loaders, from MS-DOS directly, or from the NetBoot "meta-kernel." (NetBoot is described in Section 34.) GRUB and NetBoot can boot the kernels as-is, since they directly support the MultiBoot standard, whereas the other boot loaders need the kernel to be in a different format. This conversion can be done with the `mkbsdimage`, `mklinuximage`, and `mkdosimage` "boot adapter" scripts, which are automatically built and installed with the OSKit when configured for the appropriate host. See comments in each script for the argument syntax.

- The `mklinuximage` script is installed with the OSKit when configured for a Linux or other ELF-based host; given a MultiBoot boot image, it creates a standard Linux boot image that can be loaded from LILO or other Linux boot loaders.

- The `mkbsdimage` script is installed when the OSKit is configured for a Mach or BSD host; its script creates an NMAGIC `a.out` image from a MultiBoot image that can be loaded from any of the BSD or Mach boot loaders. Note that `mkbsdimage` requires GNU `ld` to work properly: on BSD systems, which don't normally use GNU `ld`, you will have to build and install GNU `ld` manually.

- The `mkdosimage` script is installed when the OSKit is configured for DOS-based targets such as `i386-msdos` and `i386-moss`. Like the `mkbsdimage` script, `mkdosimage` requires GNU `ld`; you may have to build and install GNU `ld` first, configured as a cross-inker for an MS-DOS target, before `mkdosimage` will build properly.

- The `mkmbimage` is installed with all OSKit configurations And, unlike the other scripts, doesn't do any conversion. It simply allows for the combining of a kernel and additional files into one MultiBoot image. The resulting image can be used with MultiBoot boot loaders such as GRUB and NetBoot.

For example, the following command creates a bootable BSD-style image named '`Image`':

% `mkbsdimage` *hello*

the `mk`*type*`image` scripts can also do more complex things, such as combining an arbitrary number of additional files or "boot modules" into the image. See 10.14 and the scripts for more info.

For details on the MultiBoot standard see Section 10.14.12.

1.6.3 Command line arguments

The various boot adapters convert their respective command-line formats, such as the `boothowto` word in the BSD boot loader, into the string format used by MultiBoot-compliant operating systems. The OSKit expects this string to be in a certain format, which looks like:

progname `[`¡boot-opts and env-vars¿ `--]` ¡args to main¿

Note that if no `--` is present then all of the args will be passed to `main`.

The default OSKit MultiBoot startup code then converts this string into a C-style `argv/argc` pair, an `environ` global array, and a set of booting-options in the `oskit_bootargv/oskit_bootargc` global variables.

The `argv/argc` pair and the `environ` array are passed to main, the latter as the third parameter commonly called `envp`. The booting-options in `oskit_bootargv/oskit_bootargc` are interpreted by the default OSKit console startup code and the following flags have special meaning:

`-h` Use the serial line for the console. See also the `-f` flag. The serial port to use is determined by the `cons_com_port` variable in `libkern`'s `base_console.c`;

`-d` Enable debugging via GDB over a serial line; The serial port to use is determined by the `gdb_com_port` variable in `libkern`'s `base_console.c`. This port may differ from the serial console port, in fact it is advantageous to do so.

`-p` Enable profiling. The OS must have been compiled accordingly. See Section 21 for more details on profiling;

`-k` Enable "killswitch" support. This allows one to kill the running kernel by sending characters to the second serial line;

`-f` When using a serial console, run it at 115200 baud instead of the default 9600. This is a Utah extension and is not in BSD.

These flags are decidedly BSD-centric, but that is because at Utah we most commonly boot OSKit kernels from the FreeBSD boot-loader.

In addition, if the NetBoot booting program is being used, then an additional parameter will be present in `oskit_bootargv`:

`-retaddr` ***address*** This specifies a location in physical memory where the OS can jump to and return control to NetBoot. The default `_exit` routine in `libkern`'s `base_console.c` uses this value when exiting.

Chapter 2

Execution Environments

2.1 Introduction

Because the components of the OSKit are intended to be usable in many different kernel and user-mode environments, it is important that their requirements be defined fully, not only in terms of interface dependencies, but also in terms of *execution environment*. A component's execution environment involves many implicit and sometimes subtle factors such as whether and in what cases the component is reentrant. A client using a component must either use an execution environment directly compatible with the environment expected by the component, or it must be able to *provide* an environment in which the component can run by adding appropriate glue code. For example, most of the OSKit's components are not inherently thread- or multiprocessor-safe; although they can be used in multithreaded or multiprocessor environments, they rely on the client OS to provide the necessary locking as part of the "glue" the client OS uses to interface with the component.

In order to make it reasonably easy for the client OS to adapt to the execution environment requirements of OSKit components, the execution models used by the OSKit components are purposely kept as simple and easy to understand as possible without sacrificing significant functionality or performance. Another factor driving the OSKit components' execution models is the goal of being able to integrate large amounts of code, such as device drivers and network protocol stacks, virtually unmodified from traditional kernels such as BSD and Linux; this requirement inevitably places some restrictions on the execution models of the OSKit components derived from these source bases. However, in general, even the execution models of these complex OSKit components are considerably simpler and more well-defined than the original execution environments of the legacy systems from which the components were adapted; this simplification is enabled by carefully-designed OSKit glue code surrounding the legacy code which emulates and hides from the OSKit user the more subtle aspects of the legacy execution environments.

Since the OSKit includes components with a wide range of size and complexity, and as a result different components naturally tend to have different levels of dependence on their execution environment, the OSKit defines a set of standard execution models arranged on a continuum from simplest and least restrictive to most complex and demanding on the client OS. This range of execution models allows the client OS to adopt the simpler OSKit components quickly and with minimal fuss, while still providing all the detailed environment information necessary for the client OS to incorporate the most demanding components. For example, the basic memory-management libraries, LMM and AMM, use an extremely simple execution models with very few restrictions, allowing them to be used immediately in almost any context. The device driver libraries, on the other hand, necessarily place much greater demands on the client since they must deal with interrupts, DMA, and other hardware facilities closely tied to the execution environment; however, these requirements are defined explicitly and generically so that with a little effort even these components can be used in many different contexts.

The remaining sections of this chapter describe each of the execution models used in the OSKit, in order from simplest to most complex. In general, each succeeding execution model builds on and extends the previous model.

2.2 Pure Model

The *pure* execution model is the most basic of the OSKit execution environments; it has the following properties:

- Pure functions and components have *no* implicit global state. For example, they do not define or use any global or static variables; they only use and manipulate data passed explicitly the client. For example, many functions in the minimal C library, such as `memcpy` and `strlen`, are pure by nature in that they only touch data areas passed in parameters by the client and have no other side-effects. As a less trivial example, the LMM and AMM memory manager components include functions that maintain state across calls, but only in explicit data structures visible to the client.

- Pure functions and components are fully reentrant and thread-safe with respect to disjoint data sets. For example, if the client OS uses the LMM to manage several separate and disjoint memory pools, then the *lmm* functions may be run concurrently on two different processors in a multiprocessor system without synchronization, as long as each processor is manipulating a different LMM memory pool. This property is a natural consequence of the fact that pure OSKit components maintain no implicit global state.

- Pure functions and components are *not* reentrant or thread-safe with respect to overlapping data sets. For example, just as it is not safe to make several `memcpy` calls concurrently with overlapping destination buffers, it is not safe to call LMM functions concurrently on the same memory pool. This is true for interrupt-style as well as thread-style concurrency: for example, the client OS must not call an LMM function on a particular memory pool from within an interrupt handler if the interrupt handler might have interrupted another LMM function call using the same memory pool. In order to use these components in an interruptible or multithreaded/multiprocessor environment, the client OS must wrap them with appropriate synchronization code, such as locking or interrupt priority management, in order to ensure that only one call can be made at a time for a given data set.

- Pure functions and components are *not* reentrant with respect to overlapping data sets even during callbacks from the component to the client OS. In other words, callbacks are assumed to be atomic as far as the component is concerned. For example, the address map manager, AMM, makes calls back to the client OS to allocate and free memory for use in maintaining the address map, as well as to perform client-defined processing when address map regions are split or joined together. During the processing of one of these callbacks, the client OS must not attempt to make a reentrant call back into the AMM operating on the same address map.

Figure 2.1 illustrates the pure execution environment. Since pure functions and components contain no implicit global state, separate "instances" or uses of these components by the client can be treated as completely independent objects: although each individual instance of the component is single-threaded and non-reentrant, the client OS can manage synchronization between independent instances of the component in any way it chooses.

2.3 Impure Model

Components that use the *impure* execution model act just like those operating in the pure model, except that they may contain global shared state and therefore must be treated as a single "instance" for synchronization and reentrancy purposes. For example, many of the functions in `liboskit_kern`, the kernel support library, set up and access global processor registers and data structures, and are therefore impure. Similarly, some of the functions in the minimal C library, such as `malloc` and its relatives, inherently require the use of global state and therefore are impure.

The impure execution model has the following properties:

- Impure functions and components may depend on implicit global state, such as global or static variables or special processor registers.

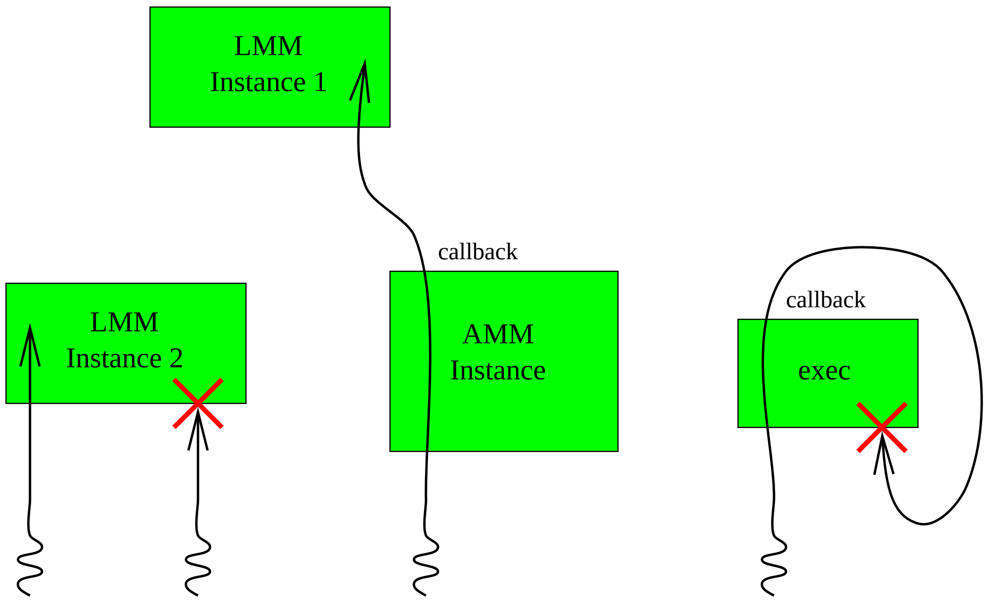

Figure 2.1: Illustration of the execution model used by pure components. Separate components, and separate instances of each component, are fully independent and have no implicit shared global state; therefore they can be invoked concurrently with no synchronization. However, each individual instance of a component (e.g., a particular LMM memory pool) is single-threaded and non-reentrant; the client OS must avoid concurrent calls to that instance, as well as recursive calls to the same instance through callbacks.

- Impure functions and components are *not* reentrant or thread-safe, except when explicitly stated otherwise. In order to use these components in an interruptible or multithreaded/multiprocessor environment, the client OS must provide appropriate synchronization code. Many impure components and functions provide explicit synchronization hooks for the convenience of the client OS. For example, the minimal C library's `malloc` functions make calls to `mem_lock` and `mem_unlock`, which are empty functions by default but can be overridden by the client to provide real synchronization; see Section 9.5 for more information.

- Impure functions and components are *not* reentrant even during callbacks from the component to the client OS, except when explicitly stated otherwise. In other words, callbacks are assumed to be atomic as far as the component is concerned.

2.4 Blocking Model

The *blocking* execution model extends the impure model to support non-preemptive multithreading; it is essentially the execution model used in traditional Unix-like kernels such as BSD and Linux. Components that use the blocking model have the same properties as those that use the impure model, except that they are re-entrant with respect to *some* callback functions; these functions are known as *blocking* functions. This means that, whenever the component makes a call to a blocking function, the client OS may re-enter the component in a different context, e.g., in the context of a different thread or processor. The set of callback functions that are assumed to be blocking is part of the component's contract with the client OS; in general, a function is blocking unless it is explicitly stated to be nonblocking.

In order to use a blocking-model component in a fully preemptive, interruptible, or multiprocessor environment, the client OS must do essentially the same thing to adapt to the component's execution model as it would for a pure or impure component: namely, it must surround the component with a lock which is taken before entry to the component and released on exit. However, because the component supports re-entrancy through callbacks that are defined to be blocking functions, the client OS's implementations of these callback functions may release the component's lock temporarily and re-acquire it before returning into the component, thereby allowing other concurrent uses of the component.

2.5 Interruptible Blocking Model

The *interruptible blocking* execution model, unlike the other models, allows a component to be re-entered at arbitrary points under certain conditions. In the interrupt model, there are two “levels” in which a component’s code may execute: interrupt level and process level. (Note that in this context we use the term “process” only because it is the traditional term used in this context; the components in fact have no notion of an actual “process.”) The interrupt model also assumes a one-bit *interrupt enable* flag, which the component can control through well-defined callback functions which must be provided by the client OS. When the component is running at either level and interrupts are enabled, the component may be re-entered at interrupt level, typically to execute an *interrupt handler* of some kind. To be specific, the properties of the interruptible blocking model are as follows:

1. Each component is a single-threaded execution domain: only one (virtual or physical) CPU may execute code in the component at a given time. For example, on a multiprocessor, only one processor may be allowed to execute in a component set at a time at process level; this can be handled by placing a global lock around the component. (Different components can be allowed to execute concurrently, as long as the client OS takes appropriate precautions to keep them fully independent of each other.) Similarly, if the host OS is preemptible, then the OS must ensure that if a component is preempted, then that component will not be re-entered in another context before the first context has finished or entered a blocking function.

2. Multiple process-level activities may be outstanding at a given time in a component, as long as only one is actually *executing* at a time (as required by rule 1). A subset of the callback functions provided by the client OS are defined as *blocking functions*; whenever one of these functions is called, the host OS may start a new activity in the component, or switch back to other previously blocked activities.

3. The host OS supplies each outstanding activity with a separate stack, which is retained across blocking function calls. Stacks are only relinquished by a component when the operation completes and the component’s code returns from the original call that was used to invoke it.

4. Code in a component always runs at one of two levels: process level or interrupt level. Whenever the host OS invokes a component through its interface, it enters the component at one of these two levels. Typically, some of the component’s exported functions or methods can be invoked only at process level, while others can be invoked only at interrupt level, and there may be a few that can be invoked at either level; which functions can be invoked at which levels is part of the component’s interface (its contract with the client OS), and thus is defined in the component’s description. Typically, most of a component’s entrypoints can only be invoked at process level; therefore, entrypoints for which no level is explicitly specified can be entered only at process level.

5. Both process-level and interrupt-level execution in a component can be interrupted at any time by interrupt handlers in the component, except when the code has disabled interrupts using `osenv_intr_disable` (see Section 6.15.1).

6. When a component is entered at process level, the component assumes that interrupts are enabled. The component may temporarily disable interrupts during processing, but must re-enable them before returning to the client OS. Similarly, when a component is entered at interrupt level (e.g., a hardware interrupt handler in a device driver), interrupts are assumed to be initially disabled as if `osenv_intr_disable` had been called implicitly before entering the handler. However, the component may re-enable interrupts, at which point the client OS is allowed to interrupt the component again with other interrupt-level activities.

7. Interrupt-level activities must be strictly stacked. In other words, when the client OS interrupts a process-level activity in a component, that interrupt-level activity must be allowed to run to completion before the client OS may resume the process-level activity. Similarly, if an interrupt-level activity is itself interrupted, then the most recent interrupt-level activity must finish before the client OS may resume previous interrupt-level activities. This constraint is generally satisfied automatically if the

client OS is running on a uniprocessor and uses only a single stack for both process-level and interrupt-level processing; however, the OSKit components do not require the client OS to use only a single stack as long as it meets these re-entrancy requirements.

8. Code in a component that may run at interrupt level may not call blocking callback functions provided by the client OS; only nonblocking callback functions may be called at interrupt level.

Although on the surface it may appear that these requirements place severe restrictions on the host OS, the required execution model can in fact be provided quite easily even in most kernels supporting other execution models. The following sections describe some example techniques for providing this execution model.

2.5.1 Use in multiprocessor kernels

Global spin lock: The easiest way to provide the required execution model for interruptible, blocking components in a nonpreemptive, process-model, multiprocessor kernel such as Mach 3.0 is to place a single global spin lock around all code running in the device driver framework. A process must acquire this lock before entering driver code, and release it after the operation completes. (This includes both process-level entry through the component's interface, and interrupt-level entry into the components' interrupt handlers.) In addition, all blocking callback functions which the host OS supplies should release the global lock before blocking and acquire the lock again after being woken up. This way, other processors, and other processes on the same processor, can run code in the same or other drivers while the first operation is blocked.

Note that this global lock must be handled carefully in order to avoid deadlock situations. A simple, "naive" non-reentrant spin lock will not work, because if an interrupt occurs on a processor that is already executing process-level driver code, and that interrupt tries to lock the global lock again, it will deadlock because the lock is already held by the process-level code. The typical solution to this problem is to implement the lock as a "reentrant" lock, so that the same processor can lock it twice (once at process level and once at interrupt level) without deadlocking.

Another strategy for handling the deadlock problem is for the host OS simply to disable interrupts before acquiring the global spin lock and enable interrupts after releasing it, so that interrupt handlers are only called while the process-level device driver code is blocked. (In this case, the `osenv_intr_enable` and `osenv_intr_disable` calls, provided by the OS to the drivers, would do nothing because interrupts are always disabled during process-level execution.) This strategy is not recommended, however, because it will increase interrupt latency and break some existing partially-compliant device drivers which busy-wait at process level for conditions set by interrupt handlers.

Spin lock per component: As a refinement to the approach described above, to achieve better parallelism, the host OS kernel may want to maintain a separate spin lock for each component. This way, for example, a network device driver can be run on one processor while a disk device driver is being run on another. This parallelism is allowed by the framework because components are fully independent and do not share data with each other directly. Of course, the client OS must be careful to maintain this independence in the way it uses the components: for example, if the client OS wants to have one component make calls to another (e.g., to connect a file system component to a disk device driver), and it wants the two components to be separately synchronized and use separate locks, the client OS must interpose some of its own code to release one lock and acquire the other during calls from one component to the other.

2.5.2 Use in preemptive kernels

The issues and solutions for implementing the required execution model in preemptive kernels are similar to those for multiprocessor kernels: basically, locks are used to protect the component's code. Again, the locking granularity can be global or per-component (or anything in between, as the OS desires). However, in this case, a blocking lock must be used rather than a simple spin lock because the lock must continue to be held if a process running the component's code is preempted. (Note the distinction between OS-level "blocking," which can occur at any time during execution of the component's code but is made invisible to the component's code through the use of locks; and component-level "blocking," which only occurs when a component calls a blocking function.)

An alternative solution to the preemption problem is simply to disable preemption while running the component's code. This solution is likely to be simpler in terms of implementation and to have less overhead, but it may greatly increase thread dispatch latency, possibly defeating the purpose of kernel preemption in the first place.

2.5.3 Use in multiple-interrupt-level kernels

Many existing kernels, particularly those derived from Unix or BSD, implement a range of "interrupt priority levels," typically assigning different levels to different classes of devices such as block, character, or network devices. In addition, some processor architectures, such as the 680x0, directly support and require the use of some kind of IPL-based scheme. Although the OSKit device drivers and other OSKit components do not directly support a notion of interrupt priority levels, it can be simulated fairly easily in IPL-based kernels by assigning a particular IPL to each component used by the kernel. In this case, the `osenv_intr_disable` routine provided by the kernel does not disable *all* interrupts, but instead only disables interrupts at the interrupt priority level that the client OS has assigned to the calling component, and at all lower priority levels. This way, although the code in each component is only aware of interrupts being "enabled" or "disabled," the host OS can in effect enforce a general IPL-based scheme.

An obvious limitation, of course, is that all of the device drivers in a particular driver set must generally have the same IPL. However, this is usually not a problem, since the drivers in a set are usually closely related anyway.

2.5.4 Use in interrupt-model kernels

Many small kernels use a pure interrupt model internally rather than a traditional process model; this basically means that there is only one kernel stack per *processor* rather than one kernel stack per process, and therefore kernel code can't block without losing all of the state on its stack. This is probably the most difficult environment in which to use the framework, since the framework fundamentally assumes one stack per outstanding component invocation. Nevertheless, there are a number of reasonable ways to work around this mismatch of execution model, some of which are described briefly below as examples:

- **Switch stacks while running driver code.** Before the kernel invokes a component operation (e.g., makes a `read` or `write` request), it allocates a special "alternate" kernel stack, possibly from a "pool" of stacks reserved for this purpose. This alternate stack is associated with the outstanding operation until the operation completes; the kernel switches to the alternate stack before executing process-level component code, and switches back to the per-processor kernel stack when the driver blocks or returns. Depending on the details of the kernel's execution model, the kernel may also have to switch back to the per-processor stack when the process-level component code is interrupted, due to an event such as a hardware interrupt or a page fault occurring while copying data into or out of a user-mode process's address space. However, note that stack switching is only required when running process-level component code; interrupt handlers in this execution model are already "interrupt model" code and need no special adaptation.

- **Run process-level device driver code on a separate kernel thread.** If the kernel supports kernel threads in some form (threads that run using a traditional process model but happen to execute in the kernel's address space), then process-level component code can be run on a kernel thread. Basically, the kernel creates or otherwise "fires off" a new kernel thread for each new component operation invoked, and the thread terminates when the operation is complete. (If thread creation and termination are expensive, then a "pool" of available threads can be cached.) The kernel must ensure that the threads active in a particular component at a given time cannot preempt each other arbitrarily except in the blocking functions defined by this framework; one way to do this is with locks (see Section 2.5.2). Conceptually, this solution is more or less isomorphic to the stack switching solution described above, since a context switch basically amounts to a stack switch; only the low-level details are really different.

- **Run the device drivers in user mode.** If a process-model environment cannot easily be provided or simulated within the kernel, then the best solution may be to run components in user mode, as ordinary threads running on top of the kernel. Of course, this solution brings with it various potential complications and efficiency problems; however, in practice they may be fairly easily surmountable, especially in kernels that already support other kinds of user-level OS components such as device drivers, file systems, etc.
- **Run the device drivers at an intermediate privilege level.** Some processor architectures, such as the x86 and PA-RISC, support multiple privilege levels besides just "supervisor mode" and "user mode." Kernels for such architectures may want to run blocking OSKit components under this framework at an intermediate privilege level, if this approach results in a net win in terms of performance or implementation complexity. Alternatively, on most architectures, the kernel may be able to run blocking OSKit components in user mode but with an address map identical to the kernel's, allowing them direct access to physical memory and other important kernel resources.

Part II

Interfaces

Chapter 3

Introduction to OSKit Interfaces

The OSKit's interfaces provide clean, well-defined intra-process or intra-kernel protocols that can be used to define the interaction between different modules of an operating system. For example, the OSKit provides a "block I/O" interface for communication between file systems and disk device drivers, a "network I/O" interface for communication between network device drivers and protocol stacks, and a file system interface similar to the "VFS" interface in BSD. However, the OSKit's interfaces were designed with a number of properties that make them much more useful as parts of a toolkit than are comparable traditional OS-level interfaces. These properties partly stem from the use of the Component Object Model (COM), described in Chapter 4 as the underlying framework in which the interfaces are defined, and partly just from careful interface design with these properties in mind. The primary important properties of the OSKit interfaces include:

- **Extensibility**. Anyone can define new interfaces or extend existing ones with no need to interact with a centralized authority. Components can simultaneously use and export interfaces defined by many different sources.
- **Simplicity**. All of the OSKit's interfaces take a minimalist design strategy: only the most obvious and fundamental features are included in the base interfaces. Get it right first; frills and optimizations can be added later through additional or extended interfaces.
- **Full extensibility while retaining interoperability**. Adding support in a component for a new or extended interface does not cause existing clients to break. Similarly, adding support in a client for a new interface does not make the client cease to work with existing components.
- **Clean separation between components**. The clean object model used by the OSKit interfaces ensures that components do not develop implicit dependencies on each other's internal state or implementation. Such dependencies can still be introduced explicitly when desirable for performance reasons, by defining additional specialized interfaces, but the object model helps prevent them from developing accidentally.
- **Orthogonality of interfaces**. Like the OSKit components that use them, the interfaces themselves are designed to be as independent of and orthogonal to each other as possible, so that exactly the set of interfaces needed in a particular situation can be used without requiring a lot of other loosely related interfaces to be implemented or used as well.
- **No required standardized infrastructure code**. The OSKit interfaces can be used irrespective of which actual OSKit components are used, if any; they do not require any fixed "support libraries" of any kind which all clients must link with in order to use the interfaces. This is one important difference between the application of COM in the OSKit versus its original Win32 environment, which requires all components to link with a standard "COM Library."
- **Efficiency**. The basic cost of invocation of an OSKit COM interface is no more than the cost of a virtual function call in C++. Through the use of additional specialized interfaces even this cost can be eliminated in performance-critical situations.

- **Automation**. The simplicity and consistent design conventions used by the OSKit's interfaces make them amenable to use with automated tools, such as Flick, the Flux IDL Compiler[1], in the future.
- **Binary compatibility**. The Component Object Model and the OSKit's interfaces are designed to support not only source-level but binary-level compatibility across future generations of components and clients.

As with all other parts of the OSKit, the client is not required to use the OSKit's interfaces as the primary inter-module interfaces within the system being designed. Similarly, the client may use only some of the interfaces and not others, or may use the OSKit's interfaces as a base from which to build more powerful, efficient interfaces specialized to the needs of the system being developed. Naturally, since the specific components provided in the OSKit must have *some* interface, they have been designed to use the standardized OSKit interfaces so that they can easily be used together when desired; however, the OS developer can choose whether to adopt these interfaces as primary inter-module interfaces in the system, or simply to use them to connect to particular OSKit components that the developer would like to use.

3.1 Header File Conventions

This section describes some specific important properties of the design and organization of the OSKit header files.

3.1.1 Basic Structure

All of the OSKit's public header files are installed in an `oskit` subdirectory of the main installation directory for header files (e.g., `/usr/local/include` by default). Assuming client code is compiled with the main include directory in its path, this means that OSKit-specific header files are generally included with lines of the form '`#include <oskit/foo.h>`'. This is also the convention used by all of the internal OSKit components. Confining all the OSKit headers into a subdirectory in this way allows the client OS to place its own header files in the same header file namespace with complete freedom, without worrying about conflicting with OSKit header files.

The OSKit follows this rule even for header files with well-known, standardized names: for example, the ANSI/POSIX header files provided by the minimal C library (e.g., `string.h`, `stdlib.h`, etc.) are all located in a header file subdirectory called `oskit/c`. On the surface this may seem to make it more cumbersome for the client OS to use these headers and hence the minimal C library, since for example it would have to '`#include <oskit/c/string.h>`' instead of just the standard '`#include <string.h>`'. However, this problem can easily be solved simply by adding the `oskit/c` subdirectory to the C compiler's include path (e.g., add `-I/usr/local/include/oskit/c` to the GCC command line); in fact this is exactly what most of the OSKit components themselves do. Furthermore, strictly confining the OSKit headers to the `oskit` subdirectory, it makes it possible for the client OS and the OSKit itself to have *several* different sets of "standard" header files coexisting in the same directory structure: for example, the OSKit components derived from Linux or BSD typically leave `oskit/c` out of the compiler's include path and instead use the native OS's header files; this makes it much easier to incorporate legacy code with minimal changes.

3.1.2 Namespace Cleanliness

A similar namespace cleanliness issue applies to the actual symbols defined by many the OSKit header files. In particular, all OSKit header files defining COM interfaces, as well as any related header files that they cross-include such as `oskit/types.h` and `oskit/error.h`, *only* define symbols having a prefix of `oskit_`, `OSKIT_`, `osenv_`, or `OSENV_`. This rule allows these headers to be included along with arbitrary other headers from different environments without introducing a significant chance of name conflicts. In fact, the OSKit components derived from legacy systems, such as the Linux driver set and the FreeBSD drivers and TCP/IP stack, depend on this property, to allow them to include the OSKit headers defining the COM interfaces

[1]http://www.cs.utah.edu/projects/flux/flick/

they are expected to export, along with the native Linux or BSD header files that the legacy code itself relies on.

Once again, this rule creates a potential problem for header files whose purpose is to declare standard, well-known symbols, such as the minimal C library header files. For example, `string.h` clearly should declare `memcpy` simply as `memcpy` and not as `oskit_memcpy` or somesuch, since in the latter case the "C library" wouldn't be conforming to the standard C library interface. However, there are many types, structures, and definitions that are needed in both the minimal C library headers and the COM interfaces: for example, both the `oskit_ttystream` COM interface and the minimal C library's `termios.h` need to have some kind of `termios` structure; however, in the former case a disambiguating `oskit_` prefix is required, whereas in the latter case such a prefix is not allowed. Although technically these two `termios` structures exist in separate contexts and could be defined independently, for practical purposes it would be very convenient for them to coincide, to avoid having to perform unnecessary conversions in code that uses both sets of headers. Therefore, the solution used throughout the OSKit header files is to define the non-prefixed versions of equivalent symbols with respect to the prefixed versions whenever possible: for example, the `errno.h` provided by the minimal C library simply does a '`#include <oskit/error.h>`' and then defines all the `errno` values with lines of the form:

```
#define EDOM    OSKIT_EDOM
#define ERANGE  OSKIT_ERANGE
```

Unfortunately this is not possible for structures since C does not have a form of `typedef` statement for defining one structure tag as an alias for another. Therefore, the few structures that need to exist in both contexts (such as the `termios` structure mentioned above) are simply defined twice. Since these structures are generally well-defined and standardized by ANSI C, POSIX, or CAE, they are not expected to change much over time, so the added maintenance burden should not be significant and is vastly outweighed by the additional flexibility provided by the clean separation of namespaces.

3.2 Common Header Files

This section describes a few basic header files that are used throughout the OSKit and are cross-included by many of the other OSKit headers.

3.2.1 `boolean.h`: boolean type definitions

SYNOPSIS

```
#include <oskit/boolean.h>
```

DESCRIPTION

Defines the fundamental values `TRUE` and `FALSE` for use with the machine-dependent `oskit_bool_t` type.

3.2.2 `compiler.h`: compiler-specific macro definitions

SYNOPSIS

```
#include <oskit/compiler.h>
```

DESCRIPTION

Defines a variety of macros used to hide compiler-dependent ways of doing things.

`OSKIT_BEGIN_DECLS, OSKIT_END_DECLS`: All function prototypes should be surrounded by these macros, so that a C++ compiler will identify them as C functions.

`OSKIT_INLINE`: Identifies a function as being inline-able.

`OSKIT_PURE`: Identifies a function as "pure." A pure function has no side-effects: it examines no values other than its arguments and changes no values other than its return value.

`OSKIT_NORETURN`: Identifies a function as never returning (e.g., `_exit`).

`OSKIT_STDCALL`: Indicates that a function uses an alternative calling convention compatibile with COM. In particular, this option indicates the called function will pop its parameters unless there were a variable number of them.

3.2.3 `config.h`: OSKit configuration-specific definitions

SYNOPSIS

```
#include <oskit/config.h>
```

DESCRIPTION

This file is generated by the `configure` program. It identifies a number of environment-dependent parameters. Currently these are all related to the compiler and assembler used to build the OSKit.

`HAVE_CR4`: Defined if the assembler supports the %cr4 register.

`HAVE_DEBUG_REGS`: Defined if the assembler supports the debug registers.

`HAVE_P2ALIGN`: Defined if the assembler supports the `.p2align` pseudo-op.

`HAVE_CODE16`: Defined if your assembler supports the `.code16` pseudo-op.

`HAVE_WORKING_BSS`: Defined if your assembler allows `.space` within `.bss` segments.

`HAVE_PACKED_STRUCTS`: Defined if your compiler groks `__attribute__((packed))` on structs.

`HAVE_PURE`: Defined if your compiler groks `__attribute__((pure))` on functions.

`HAVE_NORETURN`: Defined if your compiler groks `__attribute__((noreturn))` on functions.

`HAVE_STDCALL`: Defined if your compiler groks `__attribute__((stdcall))` on functions.

3.2.4 `machine/types.h`: basic machine-dependent types

Synopsis

```
#include <oskit/machine/boolean.h>
```

Description

This header file defines a number of types whose exact definitions are dependent on the processor architecture and compiler in use.

The following set of types are guaranteed to be *exactly* the indicated width regardless of processor architecture; they are used to get around the fact that different C compilers assign different meanings to the built-in C types such as `int` and `long`:

`oskit_s8_t`: Signed 8-bit integer

`oskit_s16_t`: Signed 16-bit integer

`oskit_s32_t`: Signed 32-bit integer

`oskit_s64_t`: Signed 64-bit integer

`oskit_u8_t`: Unsigned 8-bit integer

`oskit_u16_t`: Unsigned 16-bit integer

`oskit_u32_t`: Unsigned 32-bit integer

`oskit_u64_t`: Unsigned 64-bit integer

`oskit_f32_t`: 32-bit floating point type

`oskit_f64_t`: 64-bit floating point type

The following types depend in various ways on the target processor architecture:

`oskit_bool_t`: This type represents the most efficient integer type for storage of simple boolean values; on typical architectures it is the smallest integer type that the processor can handle with no extra overhead.

`oskit_addr_t`: This is an unsigned integer type the same size as a pointer, which can therefore be used to hold virtual or physical addresses and offsets.

`oskit_size_t`: This is an unsigned integer type equivalent to `oskit_addr_t`, except that it is generally used to represent the size of a data structure or memory buffer, or a difference between two `oskit_addr_t`s.

`oskit_ssize_t`: This is a signed integer type the same size as `oskit_size`.

`oskit_reg_t`: This is an unsigned integer type of the same size as a general-purpose processor register; it is generally but not necessarily always equivalent to `oskit_addr_t`.

`oskit_sreg_t`: This is a signed integer type the same size as `oskit_reg_t`.

3.2.5 `types.h`: basic machine-independent types

Description

This header file defines a few basic types which are used throughout many of the OSKit's COM interfaces. These types correspond to standard POSIX types traditionally defined in `sys/types.h`; however, this does not mean that the client OS must use these types as its standard definitions for the corresponding POSIX symbols; it only means that the client must use these types when interacting with OSKit components through the OSKit's COM interfaces. All of the type names are prefixed by `oskit_`, for exactly this reason.

`oskit_dev_t`: Type representing a device number; used in the `oskit_stat` structure in the OSKit's file system interfaces. Note that the OSKit's file system components themselves don't know or care about the actual assignment or meaning of device numbers; it is up to the client OS to determine how these device numbers are used, if at all.

`oskit_ino_t`: Type representing a file serial number ("inode" number), again used in the OSKit's file system interfaces.

`oskit_nlink_t`: Type representing the link count of a file.

`oskit_pid_t`: Type representing a process ID. This type is used in a few COM interfaces, such as the `oskit_ttystream` interface which includes POSIX-like job control methods. Currently the OSKit currently does not include any process management facilities, but this type and the related methods that use it are included in case such a facility is added in the future.

`oskit_uid_t`: Type representing a traditional POSIX user ID. The current OSKit components do not know or care about POSIX security, but for example the NetBSD file system component knows how to store and retrieve user IDs in BSD file system partitions.

`oskit_gid_t`: Type representing a traditional POSIX group ID. The same considerations apply as for `oskit_uid_t`.

`oskit_mode_t`: Type representing a file type and access permissions bit mask; again used in the file system interface.

`oskit_off_t`: Type representing a 64-bit file offset; used in the file system interface.

`oskit_wchar_t`: Unicode "wide" character.

Chapter 4

The Component Object Model

The Component Object Model (COM) is an architecture and infrastructure for building fast, robust, and extensible component-based software. This chapter describes the basic subset of COM that is used by the OSKit; the complete COM specification is available from Microsoft's web site[1].

At its lowest level, COM is merely a language-independent binary-level standard defining how software components within a single address space can rendezvous and interact with each other efficiently, while retaining a sufficient degree of separation between these components so that they can be developed and evolved independently. To achieve this goal, COM specifies a standard format for *dynamic dispatch tables* associated with objects. These dispatch tables are similar in function to the virtual function tables ("vtables") used in C++, but they are specified at the binary level rather than the language level, and they include additional functionality: in particular, a standardized run-time type determination ("narrowing") facility, and reference counting methods. This minimal basis allows a software component to dynamically determine the types of interfaces supported by another unknown component and negotiate a common "language" or set of interfaces through which further interaction can take place. ("Parlez vous Fran cais? Sprechen Sie Deutsch?") COM builds a whole range of services on top of this basic facility, such as cross-address-space RPC (MIDL), object linking and embedding (OLE), scripting (OLE Automation), etc. However, it is primarily this lowest-level facility that is used by and relevant to the OSKit.

4.1 Objects and Interfaces

The COM dynamic dispatch facility revolves around the fundamental concepts of *objects* and *interfaces*. An object in COM is a fairly abstract concept, not necessarily associated with a particular data structure in memory like C++ or Java objects. A COM object can be implemented in any language and can maintain its internal state in any way it chooses; as far as COM is concerned, the only thing relevant about the object is its interfaces. A client accessing a COM object generally does not have direct access to the actual data contained in the object; instead COM objects are only accessible through the set of interfaces the object exports. A reference to a COM object is really just a pointer to one of the object's interfaces. An object may support any number of interfaces; each interface represents one particular "view" of the object, or one "protocol" through which the object can be accessed. Each interface has its own dynamic dispatch table, consisting of a few standard methods (function pointers) whose calling conventions and semantics are defined by COM, followed by an arbitrary number of custom methods whose calling conventions and semantics are specific to that particular interface.

Although the COM specification defines a few basic interfaces, anyone can independently define new COM interfaces, and in fact the OSKit defines quite a number of such interfaces. COM interfaces are identified by 128-bit globally unique identifiers (GUIDs), which are algorithmically generated through various standardized mechanisms; this avoids all the accidental collisions that can easily occur when human-readable names are used as identifiers. COM GUIDs are the equivalent to and compatible with the Universally Unique Identifiers (UUIDs) used in the Distributed Computing Environment (DCE) originally developed at Apollo and the

[1]http://www.microsoft.com/com/

Open Software Foundation (OSF). Although COM interfaces generally also have human-readable names such as `IUnknown` and `IStream`, these names are only for the programmer's benefit at development time; they get compiled out in the final program and only the GUIDs are used at run-time.

4.1.1 Interface Inheritance and the `IUnknown` Interface

COM interfaces can directly extend other COM interfaces in single-inheritance relationships, simply by adding additional methods to the dispatch table defined by the base interface and/or further restricting the semantic requirements defined by the base interface. (Derived interfaces cannot relax or weaken the requirements of the base interface, since that would violate the whole principle of subtyping.) Multiple inheritance of COM interfaces cannot be implemented simply by extending dispatch tables this way, since in this case there would be multiple mutually conflicting dispatch tables to extend; however, the effect of multiple inheritance can be achieved by making the object support multiple independent interfaces using the querying mechanism described below. Note that the only form of inheritance of relevance to COM is *subtyping*, or inheritance of interfaces (types), as opposed to *subclassing*, or implementation inheritance: COM doesn't care how an object is implemented, only what interfaces it exports.

Ultimately, every COM interface is derived from a single standard universal base interface, known as `IUnknown` in the COM standard and `oskit_iunknown` in the OSKit headers. This minimal COM interface contains only three standard methods, `query`, `addref`, and `release`, which provide the basic administrative facilities that all COM objects are expected to implement. These basic facilities, which essentially provide run-time type determination and reference counting, are described in the following sections.

4.1.2 Querying for Interfaces

The first function pointer slot in the dynamic dispatch table of any COM interface always points to the standard `query` method (called `QueryInterface` in the COM specification); this means that a reference to *any* COM interface can always be used to find any of the other interfaces the object supports. To determine if a COM object supports a particular interface, the client simply calls the standard query method on the object, passing the GUID of the desired interface as a parameter. If the object supports the requested interface, it returns a pointer to that interface; this returned pointer is generally, though not always, a different pointer from the one the client already had, since it points to a different interface, even though the interface refers to the same underlying logical object. The client can then interact with the object through the methods defined by this interface. On the other hand, if the object doesn't support the requested interface, the query method simply returns an error.

Since interfaces are identified only by simple GUIDs and do not directly contain any detailed information about the methods defined by the interface, the client must already know before it requests the interface what methods the interface supports and what their semantics are. In other words, the basic COM query mechanism only allows the client and to determine the common set of interfaces both it and the object already understand; it does not directly enable the client to learn about other arbitrary unknown interfaces the object might support. Such a dynamic invocation interface can be built on top of the basic COM infrastructure, and in fact that is exactly what is done in ActiveX/OLE Automation scripting; however, the OSKit currently does not support or use this extended invocation facility.

Semantics of the Query Operation

The COM standard specifies that all COM objects must support the standard query operation, and furthermore, the query operation must have certain well-defined semantics. In particular, an object's interfaces must be:

- **Static**: If a query on one specific interface to obtain another interface succeeds the first time, then it must also succeed on subsequent calls; similarly, if it fails, it must continue to fail on subsequent calls.
- **Symmetric**: A query on an interface for the *same* interface must succeed.
- **Reflexive**: If a query on interface A for interface B succeeds, then a query on interface B for interface A must also succeed.

- **Transitive**: If a query on interface A for interface B succeeds, and a query on interface B for interface C succeeds, then a query on interface C for interface A must succeed.

However, note that subsequent queries for a given interface identifier on a given object are not required always to return the same pointer. This allows objects to create interfaces dynamically upon request and free them later when they are no longer in use, independently of the lifetime of the object itself.

As a special exception to this rule, queries on an object for the `IUnknown` interface must always return the same pointer; this allows clients to perform a reliable object identity test between two arbitrary COM interface pointers by querying each for their `IUnknown` interfaces and comparing the pointers. However, as an *approximate* object identity test, in which occasional false negative answers can be tolerated (i.e., two objects appear different when they are in fact the same), it is sufficient simply to compare two pointers having the same interface type (i.e., the same interface identifier): although objects sometimes export multiple "copies" of an interface, as in certain multiple inheritance or interface caching scenarios, this is rare enough that simple pointer comparison can work well as a heuristic.

4.1.3 Reference Counting

In addition to the basic interface negotiation mechanism provided by standard query method, every COM interface must also export two additional standard methods, `addref` and `release`, which are used to control the object's life cycle through reference counting. Whenever a client receives a pointer to a COM interface, e.g., as the result of a call to a method on a different interface, the client is considered to have one *reference* to the interface. When the client is done using this reference, it must call the `release` method on that interface so that the object will know when it is no longer in use and can delete itself. If the client needs to copy the reference, e.g., to give it to some third party while still retaining a reference itself, the client must call the `addref` method on the interface. Eventually, both the client and the third party will call `release` on their respective pointers to this interface (possibly at different times and in arbitrary order); the object can then be destroyed only when *both* outstanding references are released.

In COM, only interfaces are reference counted, not the objects themselves. After a client has obtained a reference to a particular interface, it must call the `release` method on *exactly* that interface, and not a different interface referring to the same object. This allows object implementations to maintain individual reference counts on each of their interfaces if they choose. Object implementations are also free to maintain only a single reference count for the entire object, in which case the `addref` and `release` methods on all the object's interfaces will happen to do the same thing; however, clients must not depend on this behavior.

Cycles

As with all reference counted systems, there is a danger of cycles developing among COM objects and preventing proper garbage collection. For example, if object A holds a reference to object B, and object B holds a reference to object A, then assuming nothing is specifically done to change this situation, the reference counts of both objects will remain nonzero and the objects will effectively "keep each other alive" indefinitely even if nothing else in the system still refers to either of those objects. COM does not provide any automatic facility to solve this problem: instead, it must be solved manually by careful design.

One particular technique that can often be used to avoid cycles when two objects both need to maintain pointers to each other is to make one of the objects maintain a reference not to the other object itself, but rather to an *inner object* which is logically (and possibly physically) part of the main object but is independently reference counted. For example, often a client needs to register a *callback object* of some kind, such as the network-packet-receive callback object passed to the `open` method of the `oskit_netdev` interface. To avoid cycles, the client object should not pass a reference to *itself* as the callback, but instead should create a separate, independently reference counted callback object which is logically contained in the main client object, and pass a reference to the callback object rather than the main object. This way, the client can keep a reference to the server object, and the server can keep a reference to the callback object, but no cycle will develop and garbage collection works properly.

4.2 Reference and Memory Management Conventions

Since the set of methods and semantics attached to a COM interface represented by a particular GUID only needs to be informally defined by the designer of the interface and understood by the programmers writing code that uses that interface, the exact semantics of the interface can be arbitrary - COM makes no explicit restrictions on the interface's methods and semantics as long as they are meaningful and well-defined. However, for convenience and consistency, and to make it more practical for COM interfaces to be defined in a form usable by automated tools such as IDL compilers, COM provides a set of method invocation conventions which interface designers are recommended to use when possible. These conventions mainly deal with the allocation and deallocation of memory across method calls, and similarly, the allocation and deallocation of COM object references.

As with typical interface definition languages (IDLs), COM defines three basic logical types of parameters, each with its own standard semantic rules for memory and object reference management. Although COM interfaces do not need to be defined in any IDL, this categorization makes COM's conventions consistent with common IDLs and makes COM interfaces easier to define in IDLs when necessary:

- `in` parameters are parameters through which the caller passes information to the callee. Memory is allocated and freed by the caller: if the callee wants to retain a copy of the data after returning from the call, it must allocate its own memory area and copy the data into it. The callee must not modify or deallocate the memory since it is merely "on loan" from the caller. Similarly, COM object references are allocated and freed by the caller: the callee merely "borrows" the object reference during the call, and if it wants to retain its own reference to the object after returning from the call, it must call the `addref` method on the interface pointer before returning.

- `out` parameters are parameters through which the callee passes information back to the caller. In this case, memory buffers and object references are allocated by the callee and freed by the caller: in other words, the method allocates the buffers or references itself and then hands the responsibility for their management to the caller on return.

- `in-out` parameters are parameters through which the caller passes information to the callee at call time, and the callee later passes information back to the caller at return time. In this case, the caller initially allocates the parameters and initializes them with their initial "ingoing" values; however, during the call, the callee may free and reallocate some or all of these parameters and reinitialize them with the final "outgoing" values to be passed back to the caller. After the call is complete, the caller is then responsible for deallocating these final memory buffers and object references.

By convention, the return value of a COM interface method is normally used to return a generic success/failure code to the caller (see Section 4.3, below). However, sometimes methods are instead defined to return something else as their return value; in this case, the return value can be thought of as an `out` parameter for purposes of memory and object reference management.

4.3 Error Handling

The COM specification defines a standard 32-bit namespace for error codes, which the OSKit adopts as the error code namespace for all of its exported interfaces, both COM and conventional. COM error return values are divided into three fields: a one-bit *severity* flag, indicating success (zero) or failure (one), a 15-bit *facility*, providing a broad categorization of error, and finally, a 16-bit *code*, which indicates the specific error and is only meaningful with respect to a particular facility.

Unfortunately, the management of error codes is one of the relatively few parts of COM in which Microsoft-centrism appears in force. Ideally, error codes should be GUIDs, just like interface identifiers; however, this would be too cumbersome and inefficient in practice. Therefore, COM divides the error code namespace into two categories: globally-unique, centrally allocated error codes, and interface-specific error codes. Most interfaces are expected to use the interface-specific range; these error codes are only meaningful when returned from methods on a particular COM interface, and their meanings are defined as part of that

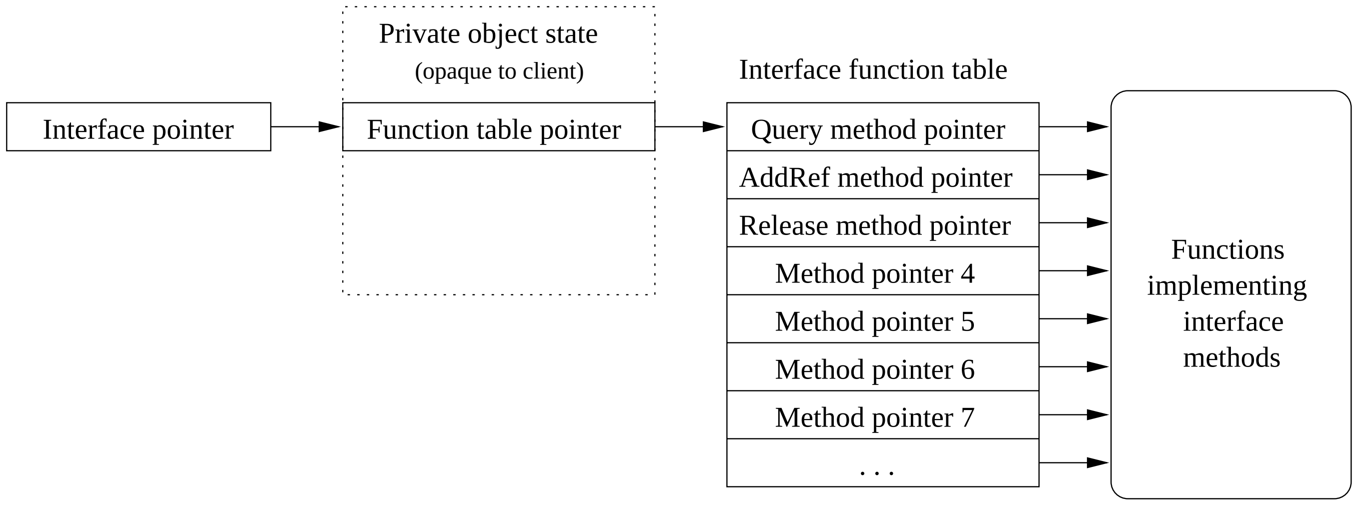

Figure 4.1: Diagram of the structure of a COM interface. The client holds a pointer to a pointer to a table of function pointers; the pointers in the function table point to the object's implementations of the methods exported to the client through the interface.

COM interface. However, as may be expected, most of Microsoft's COM interfaces use centrally administrated error codes since they are much easier to deal with in large software systems and, conveniently, Microsoft happens to be the "central administrative authority." Furthermore, again as may be expected, Microsoft does not readily allocate facility codes to third parties.

The OSKit defines a number of error codes that need to be valid across a large number of interfaces, both COM and non-COM; it would be difficult or impossible to make these error codes fit into the "interface-specific" paradigm. Further, since only one small 64K range has been assigned for interface-specific error codes, out of the two billion possible values, we felt that using values in the interface-specific range to represent errors that are treated as globally unique by OSKit components would be just asking for trouble in the long term, since in such a small namespace collisions are inevitable. Therefore, for global error codes used by the OSKit, we have informally allocated (i.e., stolen) facility code 0xf10 for our purposes. For the specific assignment of error codes in this range, see the description of `oskit/error.h`, in Section 4.6.2. However, the OSKit still uses the interface-specific error code range, when appropriate, for error codes only meaningful to a particular interface.

4.4 Binary Issues

Since COM is a binary-level standard, it defines the exact in-memory layout of COM interfaces and their function tables as seen by clients of those interfaces. Any object can implement any COM interface in whatever way it chooses, as long as it conforms to these basic rules.

4.4.1 Interface Structure

Figure 4.1 shows a diagram of the structure of a COM interface. From the client's viewpoint, a reference to a COM interface is a pointer to a pointer (the function table pointer) to a table of pointers to functions (the dynamic dispatch table). The function table pointer is generally just a part of a larger data structure representing the internal state of the object, illustrated in the figure by the dotted box; however, only the function table pointer itself is visible to the client. To call one of the interface's methods, the client follows the interface pointer to the function table pointer, then dereferences the function table pointer to find the function table, looks up the appropriate function pointer in the table, and finally calls the function. In general, the first parameter to this function will be a copy of the original interface pointer the client started with, allowing the object implementation to locate its private object state quickly and easily.

4.4.2 Calling Conventions

The specific calling conventions used in a method call is also standardized by the COM specification but depends on the processor architecture. On the Intel x86 architecture, where different types of calling conventions abound, the standard calling conventions for COM interfaces are the `stdcall` conventions defined by Microsoft and used throughout the Win32 API. Note that these calling conventions generally do *not* match the default calling conventions of any particular C or C++ compiler, so implementors of COM interfaces must be careful to use the appropriate declarations. In the OSKit, the `OSKIT_COMCALL` macro can be used to declare functions and function pointers to use standard COM calling conventions, regardless of the compiler or processor architecture in use; see 4.6.1 for more details.

4.5 Source Issues

The OSKit header files defining COM interfaces do not use the traditional Win32 type names used in the corresponding Microsoft header files; instead, they follow the naming and style conventions used in the rest of the OSKit. For example, the type representing a 32-bit unsigned integer is called `oskit_u32_t` instead of `DWORD` as in Win32, and the type representing the standard COM stream interface is named `oskit_stream_t` instead of `IStream`. This is done for two reasons:

- To retain a consistent overall style with the rest of the OSKit, and present a consistent naming conventions to clients.
- To avoid name conflicts with actual Win32 header files if the OSKit is actually used in a Win32 or similar environment.

However, all of the COM interfaces in the OSKit use the standard COM function calling and interface layout conventions, so that binary-level compatibility with Win32 environments is possible (though it hasn't been tried yet). Since COM is primarily a binary-level rather than source-level standard, this appeared to be the best approach to retaining compatibility with COM while maximizing the flexibility and ease-of-use of the OSKit.

4.6 COM Header Files

This section describes the general COM-related public header files provided by the OSKit.

4.6.1 `com.h`: basic COM types and constants

DESCRIPTION

This header file defines various types and other symbols for defining and using COM interfaces. Most of these symbols correspond directly to similar symbols in the Win32 API; however, all of the names are prefixed with `oskit_` to avoid conflicts with actual Win32 headers or other symbols used in the client OS environment, and they are named according to the standard OSKit conventions for consistency with the rest of the OSKit.

The `oskit_guid` structure and corresponding type `oskit_guid_t` define the format of DCE/COM globally unique identifiers:

```
struct oskit_guid {
    oskit_u32_t  data1;     /* Data - often time stamp      */
    oskit_u16_t  data2;     /* Data                         */
    oskit_u16_t  data3;     /* Data                         */
    oskit_u8_t   data4[8];  /* Data - often MAC address     */
};
```

Additionally, the related preprocessor macro `OSKIT_GUID` can be used to declare initializers for GUID structures.

The type `oskit_iid_t` is defined as an alias for `oskit_guid_t`, and is specifically used for COM interface identifiers (IIDs).

The following preprocessor symbols are defined for constructing and testing COM error codes:

`OSKIT_SUCCEEDED`: Evaluates to true if the error code parameter indicates success (the high bit is zero).

`OSKIT_FAILED`: Evaluates to true if the error code parameter indicates failure (the high bit is one).

`OSKIT_ERROR_SEVERITY`: Extracts the the severity (success/failure) flag from the supplied error parameter.

`OSKIT_ERROR_FACILITY`: Extracts the the facility code (bits 16-30) from the supplied error parameter.

`OSKIT_ERROR_CODE`: Extracts the the code portion (low 16 bits) of the supplied error parameter.

`OSKIT_S_OK`: Defined as zero, the standard return code for COM methods indicating "all's well, nothing to report."

`OSKIT_S_TRUE`: Defined as zero, the same as `OSKIT_S_OK`; this is used when the method returns a true/false flag of some kind on success.

`OSKIT_S_FALSE`: Defined as one (which, as a COM error value, still indicates success); used when the method returns a true/false flag of some kind on success. Note that this representation is exactly reversed from normal C conventions for boolean flags; it's a unfortunate inherited Microsoftism.

Finally, the following macros, whose exact definitions are compiler-specific, are used in declarations of functions and function pointers for COM interfaces, to ensure that the standard COM calling conventions are used:

`OSKIT_COMCALL`: Declares a function to use standard COM calling conventions, known as `stdcall` conventions in the COM specification. This tag must be placed in the function prototype between the return value and the symbol being declared, e.g., `oskit_error_t OSKIT_COMCALL` `query(...)`. (Note that the GNU C compiler also allows the tag to be placed at the end of the prototype, but this placement is not compatible with other compilers and therefore not recommended.)

`OSKIT_COMDECL`: This is simply a shorthand for `oskit_error_t OSKIT_COMCALL`; it is used in declarations of normal COM methods which return an error code as the result.

`OSKIT_COMDECL_U`: This is simply a shorthand for `oskit_u32_t OSKIT_COMCALL`; it is generally used in declarations of the `addref` and `release` methods common to all COM interfaces, which return integer reference counts as their result.

`OSKIT_COMDECL_V`: This is simply a shorthand for `void OSKIT_COMCALL`; it is used in declarations of COM methods having no return value.

4.6.2 `error.h`: error codes used in the OSKit COM interfaces

Description

This header file defines the type `oskit_error_t`, representing a COM error status; it is equivalent to the `HRESULT` type in Win32. It also defines a number of specific error codes that are widely applicable and used throughout the OSKit.

The following symbols correspond directly to standard COM errors, and use the standard values; they differ only in the `OSKIT_` prefix added to the names to avoid conflicts with other header files the client may use.

`OSKIT_E_UNEXPECTED`: Unexpected error

`OSKIT_E_NOTIMPL`: Not implemented

`OSKIT_E_NOINTERFACE`: Interface not supported

`OSKIT_E_POINTER`: Bad pointer

`OSKIT_E_ABORT`: Operation aborted

`OSKIT_E_FAIL`: General failure

`OSKIT_E_ACCESSDENIED`: Access denied

`OSKIT_E_OUTOFMEMORY`: Out of memory

`OSKIT_E_INVALIDARG`: Invalid argument

The following symbols correspond to the `errno` values defined by the 1990 ISO/ANSI C standard:

`OSKIT_EDOM`: Argument out of domain

`OSKIT_ERANGE`: Result too large

The following symbols correspond to the `errno` values defined by the 1990 POSIX.1 standard; although many of them are never actually generated by existing OSKit components, the full set is included for completeness:

`OSKIT_E2BIG`: Argument list too long

`OSKIT_EACCES`: Permission denied

`OSKIT_EAGAIN`: Resource temporarily unavailable

`OSKIT_EBADF`: Bad file descriptor

`OSKIT_EBUSY`: Device busy

`OSKIT_ECHILD`: No child processes

`OSKIT_EDEADLK`: Resource deadlock avoided

`OSKIT_EEXIST`: File exists

`OSKIT_EFAULT`: Bad address. This is the same as `OSKIT_E_POINTER`.

`OSKIT_EFBIG`: File too large

`OSKIT_EINTR`: Interrupted system call

`OSKIT_EINVAL`: Invalid argument. This is the same as `OSKIT_E_INVALIDARG`.

`OSKIT_EIO`: Input/output error

`OSKIT_EISDIR`: Is a directory

`OSKIT_EMFILE`: Too many open files

`OSKIT_EMLINK`: Too many links

`OSKIT_ENAMETOOLONG`: File name too long

`OSKIT_ENFILE`: Max files open in system

`OSKIT_ENODEV`: Operation not supported by device

`OSKIT_ENOENT`: No such file or directory

`OSKIT_ENOEXEC`: Exec format error

`OSKIT_ENOLCK`: No locks available

`OSKIT_ENOMEM`: Cannot allocate memory. This is the same as `OSKIT_E_OUTOFMEMORY`.

`OSKIT_ENOSPC`: No space left on device

`OSKIT_ENOSYS`: Function not implemented. This is the same as `OSKIT_E_NOTIMPL`.

`OSKIT_ENOTDIR`: Not a directory

`OSKIT_ENOTEMPTY`: Directory not empty

`OSKIT_ENOTTY`: Inappropriate ioctl

`OSKIT_ENXIO`: Device not configured

`OSKIT_EPERM`: Operation not permitted. This is the same as `OSKIT_E_ACCESSDENIED`.

`OSKIT_EPIPE`: Broken pipe

`OSKIT_EROFS`: Read-only file system

`OSKIT_ESPIPE`: Illegal seek

`OSKIT_ESRCH`: No such process

`OSKIT_EXDEV`: Cross-device link

The following symbols correspond to the `errno` values added by the 1993 POSIX.1 standard (real-time extensions); although most of them are never actually generated by existing OSKit components, they are included for completeness:

`OSKIT_EBADMSG`: Bad message

`OSKIT_ECANCELED`: Operation canceled

`OSKIT_EINPROGRESS`: Operation in progress

`OSKIT_EMSGSIZE`: Bad message buffer length

`OSKIT_ENOTSUP`: Not supported

The following symbol corresponds to the `errno` value added by the 1996 POSIX.1 standard:

`OSKIT_ETIMEDOUT`: Operation timed out

The following symbols correspond to the `errno` values defined by the 1994 X/Open Unix CAE standard, and not defined by one of the above standards. Most of them are related to networking, and are therefore used by the OSKit networking components; a few are not used at all by the OSKit (such as the “reserved” and STREAMS-related codes), but are included for completeness.

`OSKIT_EADDRINUSE`: Address in use

`OSKIT_EADDRNOTAVAIL`: Address not available

`OSKIT_EAFNOSUPPORT`: Address family unsupported

`OSKIT_EALREADY`: Already connected

`OSKIT_ECONNABORTED`: Connection aborted

`OSKIT_ECONNREFUSED`: Connection refused

`OSKIT_ECONNRESET`: Connection reset

`OSKIT_EDESTADDRREQ`: Destination address required

`OSKIT_EDQUOT`: Reserved

`OSKIT_EHOSTUNREACH`: Host is unreachable

`OSKIT_EIDRM`: Identifier removed

`OSKIT_EILSEQ`: Illegal byte sequence

`OSKIT_EISCONN`: Connection in progress

`OSKIT_ELOOP`: Too many symbolic links

`OSKIT_EMULTIHOP`: Reserved

`OSKIT_ENETDOWN`: Network is down

`OSKIT_ENETUNREACH`: Network unreachable

`OSKIT_ENOBUFS`: No buffer space available

`OSKIT_ENODATA`: No message is available

`OSKIT_ENOLINK`: Reserved

`OSKIT_ENOMSG`: No message of desired type

`OSKIT_ENOPROTOOPT`: Protocol not available

`OSKIT_ENOSR`: No STREAM resources

`OSKIT_ENOSTR`: Not a STREAM

`OSKIT_ENOTCONN`: Socket not connected

`OSKIT_ENOTSOCK`: Not a socket

`OSKIT_EOPNOTSUPP`: Operation not supported on socket

`OSKIT_EOVERFLOW`: Value too large

`OSKIT_EPROTO`: Protocol error

`OSKIT_EPROTONOSUPPORT`: Protocol not supported

`OSKIT_EPROTOTYPE`: Socket type not supported

`OSKIT_ESTALE`: Reserved

`OSKIT_ETIME`: Stream ioctl timeout

`OSKIT_ETXTBSY`: Text file busy

`OSKIT_EWOULDBLOCK`: Operation would block

4.7 `oskit_iunknown`: base interface for all COM objects

The `oskit_iunknown` interface, known as `IUnknown` in the Win32 API, serves as the basis for all other COM interfaces; it provides the following three methods which all COM interfaces are required to support:

`query`: Query for a different interface to the same object.

`addref`: Increment the reference count on the interface.

`release`: Decrement the reference count on the interface.

4.7.1 `query`: Query for a different interface to the same object

Synopsis

`#include <oskit/com.h>`

`OSKIT_COMDECL` **query**(`oskit_iunknown_t *`*obj*, `const oskit_iid_t *`*iid*, [out] `void **`*ihandle*);

Description

Given a reference to any of an object's COM interfaces, this method allows the client to obtain a reference to any of the other interfaces the object exports by querying for a specific interface identifier (IID).

Parameters

obj: The object being queried.

iid: The interface identifier of the requested interface.

ihandle: On success, the requested interface pointer is returned in this parameter. The client must `release` the returned reference when it is no longer needed.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error. Usually the only error code this method returns is `OSKIT_E_NOINTERFACE`, indicating that the object does not support the requested interface.

4.7.2 `addref`: Increment an interface's reference count

Synopsis

`#include <oskit/com.h>`

`OSKIT_COMDECL_U` **addref**(`oskit_iunknown_t *`*obj*);

Description

This method adds one to the interface's reference count (or to the object's reference count, if the object implements only one counter for all its interfaces). A corresponding call must later be made to the `release` method.

Parameters

obj: The interface on which to increment the reference count.

RETURNS

Returns the new reference count. This return code should only be used for debugging purposes, as its value is generally unstable at run time and its behavior depends on the object's implementation.

4.7.3 `release`: Release a reference to an interface

SYNOPSIS

`#include <oskit/com.h>`

`OSKIT_COMDECL_U` **release**(`oskit_iunknown_t` `*`*obj*);

DESCRIPTION

This method decrements the interface's reference count (or the object's reference count, if the object implements only one counter for all its interfaces). The object destroys itself if its reference count drops to zero. Note that the client must be careful never to release a reference too many times, or to release a reference to a different interface from the one on which `addref` was called, or chaos will surely ensue.

PARAMETERS

obj: The interface on which to release a reference.

RETURNS

Returns the new reference count. This return code should only be used for debugging purposes, as its value is generally unstable at run time and its behavior depends on the object's implementation.

4.8 `oskit_stream`: standard interface for byte stream objects

The `oskit_stream` COM interface supports reading and writing to stream objects, and corresponds to the Microsoft COM `IStream` interface[2].

The `oskit_stream` COM interface inherits from `oskit_iunknown`, and has the following additional methods:

`read`: Read from this object, starting at the current seek pointer.

`write`: Write to this object, starting at the current seek pointer.

`seek`: Change the seek pointer of this object.

`setsize`: Change the size of this object.

`copyto`: Copy from this object to another stream object.

`commit`: Commit all changes to this object.

`revert`: Revert to last committed version of this object.

`lockregion`: Lock a region of this object.

`unlockregion`: Unlock a region of this object.

`stat`: Get attributes of this object.

`clone`: Create a new stream object for the same underlying object.

4.8.1 `read`: Read from this stream, starting at the seek pointer

SYNOPSIS

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_read**(`oskit_stream_t` **f*, `void` **buf*, `oskit_u32_t` *len*, [out] `oskit_u32_t` **out_actual*);

DESCRIPTION

This method reads no more than `len` bytes into `buf` from this stream, starting at the current seek pointer of this stream. `out_actual` is set to the actual number of bytes read.

PARAMETERS

f: The object from which to read.

buf: The buffer into which the data is to be copied.

len: The maximum number of bytes to read.

out_actual: The actual number of bytes read.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

[2]http://www.microsoft.com/msdn/sdk/platforms/doc/activex/src/if_r2z_62.htm

4.8.2 `write`: Write to this stream, starting at the seek pointer

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_write**(`oskit_stream_t` **f*, `const void` **buf*, `oskit_u32_t` *len*, [out] `oskit_u32_t` **out_actual*);

Description

This method writes no more than `len` bytes from `buf` into this stream, starting at the current seek pointer of this stream. `out_actual` is set to the actual number of bytes written.

Parameters

f: The object to which to write.

buf: The buffer from which the data is to be copied.

len: The maximum number of bytes to write.

out_actual: The actual number of bytes written.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.3 `seek`: Change the seek pointer of this stream

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_seek**(`oskit_stream_t` **f*, `oskit_s64_t` *ofs*, `oskit_seek_t` *whence*, [out] `oskit_u64_t` **out_newpos*);

Description

This method changes the seek pointer of this stream. If `whence` is `OSKIT_SEEK_SET`, then `ofs` is used as the new seek pointer value. If `whence` is `OSKIT_SEEK_CUR`, then the new seek pointer value is set to the sum of `ofs` and the former seek pointer value. If `whence` is `OSKIT_SEEK_END`, then the new seek pointer value is set to the sum of `ofs` and the size of the stream object. The new seek pointer value is returned via `out_newpos`.

Parameters

f: The object whose seek pointer is to be changed.

ofs: The relative offset used in computing the new seek pointer.

whence: The location that ofs to which ofs is relative.

out_newpos: The new seek pointer value.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.4 `setsize`: Set the size of this object

Synopsis

> `#include <oskit/com/stream.h>`
>
> `OSKIT_COMDECL` **oskit_stream_setsize**(`oskit_stream_t` **f*, `oskit_u64_t` *new_size*);

Description

> This method sets the size of this stream to `new_size` bytes. If `new_size` is larger than the former size of this stream, then the contents of the stream between its former end and its new end are undefined.
>
> The seek pointer is not affected by this method.

Parameters

> *f*: The object whose size is to be changed.
>
> *new_size*: The new size in bytes for this object.

Returns

> Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.5 `copyto`: Copy data from this object to another stream object

Synopsis

> `#include <oskit/com/stream.h>`
>
> `OSKIT_COMDECL` **oskit_stream_copyto**(`oskit_stream_t` **f*, `oskit_stream_t` **dst*, `oskit_u64_t` *size*, [out] `oskit_u64_t` **out_read*, [out] `oskit_u64_t` **out_written*);

Description

> This method copies `size` bytes from the current seek pointer in this stream to the current seek pointer in `dst`.
>
> Both seek pointers are updated by this method. This method is functionally equivalent to performing an `oskit_stream_read` on the source stream followed by an `oskit_stream_write` on the destination stream.

Parameters

> *f*: The source stream from which to copy.
>
> *dst*: The destination stream to which to copy.
>
> *size*: The number of bytes to copy.
>
> *out_read*: The actual number of bytes read from the source.
>
> *out_written*: The actual number of bytes written to the destination.

Returns

> Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.6 `commit`: Commit all changes to this object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_commit**(`oskit_stream_t` **f*, `oskit_u32_t` *commit_flags*);

Description

This method flushes all changes made to this stream object to the next level storage object.

Parameters

f: The object to commit.

commit_flags: Conditions for performing the commit operation.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.7 `revert`: Revert to last committed version of this object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_revert**(`oskit_stream_t` **f*);

Description

This method changes the state of this stream object to its last committed state if the stream is a transacted object. Otherwise, this method does nothing.

Parameters

f: The object to revert.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.8 `lockregion`: Lock a region of this object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_lockregion**(`oskit_stream_t` **f*, `oskit_u64_t` *offset*, `oskit_u64_t` *size*, `oskit_u32_t` *lock_type*);

Description

This method locks a range of this stream object, where the range starts at the specified byte `offset` and extends for the specified `size` bytes.

Parameters

f: The object to lock.

offset: The starting byte offset of the range to be locked.

size: The length in bytes of the range.

lock_type: The type of lock to apply.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.9 `unlockregion`: Unlock a region of this object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_unlockregion**(`oskit_stream_t` **f*, `oskit_u64_t` *offset*, `oskit_u64_t` *size*, `oskit_u32_t` *lock_type*);

Description

This method unlocks a range of this stream object, where the range starts at the specified byte `offset` and extends for the specified `size` bytes.

The parameters must match the parameters used in a prior `oskit_stream_lockregion` call.

Parameters

f: The object to unlock.

offset: The starting byte offset of the range to be unlocked.

size: The length in bytes of the range.

lock_type: The type of lock to release.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.10 `stat`: Get attributes of this object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_stat**(`oskit_stream_t` **f*, [out] `oskit_stream_stat_t` **out_stat*, `oskit_u32_t` *stat_flags*);

Description

This method returns the attributes of this stream object. `out_stat` is a pointer to an `oskit_stream_stat_t` structure, defined as follows:

```
struct oskit_stream_stat {
    oskit_char_t    *name;   /* string name (optional)    */
    oskit_u32_t     type;    /* type of object            */
    oskit_u64_t     size;    /* size in bytes             */
};
```

Parameters

f: The object whose attributes are desired.

out_stat: The attributes of the stream object.

stat_flags: Which attributes to obtain.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.8.11 `clone`: Create a new stream object for the same underlying object

Synopsis

`#include <oskit/com/stream.h>`

`OSKIT_COMDECL` **oskit_stream_clone**(`oskit_stream_t` **f*, [out] `oskit_stream_t` ***out_stream*);

Description

This method creates a new stream object for the same underlying object, with a distinct seek pointer. The seek pointer of the new object is initially set to the current seek pointer of this object.

Subsequent modifications of data within one stream object are visible to readers of the other object; likewise, locking on either object affects the other object.

Parameters

f: The object to be cloned.

out_stream: The new stream object

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.9 Services Registry

The services registry allows components to lookup and rendezvous with an arbitrary "service" using the interface ID (IID) of the desired interface. More than one interface supporting a particular IID can be registered. For example, the C library queries the registry for the lock manager (see Section 4.12) so that it can allocate locks to properly protect its internal data structures when running in a multithreaded environment. The `services` registry supports the following interface functions.

`oskit_register`: Register an interface in the services registry.

`oskit_unregister`: Unregister a previously registered interface.

`oskit_lookup`: Obtain a list of all the registered interfaces with a specified IID.

`oskit_lookup_first`: Lookup the first interface registered for a specified IID.

4.9.1 `oskit_register`: Register an interface in the services registry

Synopsis

`#include <oskit/com/services.h>`

`oskit_error_t` **oskit_register**(`const` *struct oskit_guid *iid*, `void` **interface*);

Description

Register a COM interface in the services registry. An additional reference on the interface is taken. More than one interface may be registered for a particular IID. Attempts to register an interface that is already registered will succeed, although the registry will remain unchanged and no additional references will be taken.

Parameters

iid: The `oskit_guid` of the COM interface being registered.

interface: The COM interface being registered.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.9.2 `oskit_unregister`: Unregister a previously registered interface

Synopsis

`#include <oskit/com/services.h>`

`oskit_error_t` **oskit_unregister**(`const` *struct oskit_guid *iid*, `void` **interface*);

Description

Unregister a COM interface that has been previously registered in the services registry. The reference on the interface that was taken in `oskit_register` is released.

Parameters

iid: The `oskit_guid` of the COM interface being registered.

interface: The COM interface being registered.

Returns

Returns 0 on success, or OSKIT_E_INVALIDARG if the specified IID and COM interface is not in the registry.

4.9.3 `oskit_lookup`: Obtain a list of all COM interfaces registered for an IID

Synopsis

`#include <oskit/com/services.h>`

`oskit_error_t` **oskit_lookup**(`const` *struct oskit_guid *iid*, [out] `void` `***`*out_interface_array*);

Description

Look up the set of interfaces that have been registered with a particular IID, returning an array of COM interfaces. The client is responsible for releasing the references on the interfaces, and deallocating the array. By default, the first interface registered is the first interface placed in the array.

Parameters

iid: The `oskit_guid` of the COM interface being looked up..

out_interface_array: The array of COM interfaces registered for the given IID.

Returns

Returns the number of COM interfaces found, or 0 if there were no matches.

4.9.4 `oskit_lookup_first`: Obtain the first COM interface registered for an IID

Synopsis

`#include <oskit/com/services.h>`

`oskit_error_t` **oskit_lookup_first**(`const` *struct oskit_guid *iid*, [out] `void` `**`*out_interface*);

Description

Look up the first COM interface that has been registered with a particular IID. The client is responsible for releasing the reference on the interface.

Parameters

iid: The `oskit_guid` of the COM interface being looked up..

out_interface: The first COM interface registered for the given IID.

Returns

Always returns 0, setting `out_interface` to NULL if there was no match.

4.10 `oskit_lock`: Thread-safe lock interface

The `oskit_lock` COM interface allows components to protect data structures from concurrent access by multiple threads. The interface is intended to be generic so that components do not need to know the specifics of any particular thread system. The user of a lock should be prepared for the possibilty that the thread will be put to sleep if the lock cannot be granted. There are two variants supported; a regular lock and a critical lock. A critical lock differs only in that interrupts are blocked while the lock is held. The `oskit_lock` COM interface inherits from `oskit_iunknown`, and has the following additional methods:

`lock`: Lock a lock.

`unlock`: Unlock a lock.

4.10.1 `lock`: Lock a lock

Synopsis

`#include <oskit/com/lock.h>`

`OSKIT_COMDECL` **oskit_lock_lock**(`oskit_lock_t` **lock*);

Description

This method attempts to lock `lock`. If the lock cannot be immediately granted, the current thread is put to sleep until the lock can be granted.

Parameters

lock: The `oskit_lock` COM interface for the lock.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.10.2 `lock`: Unlock a lock

Synopsis

`#include <oskit/com/lock.h>`

`OSKIT_COMDECL` **oskit_lock_unlock**(`oskit_lock_t` **lock*);

Description

This method unlocks `lock`. If there are any threads waiting for the lock, one will be woken up and granted the lock.

Parameters

lock: The `oskit_lock` COM interface for the lock.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.11 `oskit_condvar`: Condition variable interface

The `oskit_condvar` COM interface allows components to wait for conditions. The interface is intended to be generic so that components do not need to know the specifics of any particular thread system. A condition is typically combined with an `oskit_lock` object to faciliate building monitor type objects. Attempting to wait on a condition without supplying a locked `oskit_lock` object results in undefined behaviour. The `oskit_lock` COM interface inherits from `oskit_iunknown`, and has the following additional methods:

`wait`: Wait on a condition variable.

`signal`: Signal a condition variable.

`broadcast`: Broadcast a condition variable.

4.11.1 `wait`: Wait on a condition variable

Synopsis

`#include <oskit/com/condvar.h>`

`OSKIT_COMDECL` **oskit_condvar_wait**(`oskit_condvar_t` **condvar*, `oskit_lock_t` **lock*);

Description

This method causes the current thread is to wait until the condition variable is signaled or broadcast. The `oskit_lock` object must be locked when called. The lock is released prior to waiting, and reacquired before returning.

Parameters

condvar: The `oskit_condvar` COM interface for the condition variable.

lock: The `oskit_lock` COM interface for the lock.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.11.2 `signal`: Signal a condition variable

Synopsis

`#include <oskit/com/condvar.h>`

`OSKIT_COMDECL` **oskit_condvar_signal**(`oskit_condvar_t` **condvar*);

Description

Wake up exactly one thread waiting on the condition variable object `condvar`.

Parameters

condvar: The `oskit_condvar` COM interface for the condition variable.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.11.3 `broadcast`: Broadcast a condition variable

Synopsis

```
#include <oskit/com/condvar.h>
OSKIT_COMDECL oskit_condvar_broadcast(oskit_condvar_t *condvar);
```

Description

Wake up all threads waiting on the condition variable object `condvar`.

Parameters

condvar: The `oskit_condvar` COM interface for the condition variable.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.12 `oskit_lock_mgr`: Lock manager: Interface for creating locks and condition variables

The lock manager is registered in the services registry (See Section 4.9) when the threading system (if it is included) is initialized. Components that need to protect data structures can query the services registry for the lock manager. Since the lock manager will only be registered by the threading system, the client can assume that the absence of a lock manager implies a single threaded system (locks are unnecessary). The `oskit_lock_mgr` COM interface inherits from `oskit_iunknown`, and has the following additional methods:

`allocate_lock`: Allocate a lock object.

`allocate_critical_lock`: Allocate a critical lock object.

4.12.1 `allocate_lock`: Allocate a thread-safe lock

Synopsis

`#include <oskit/com/lock_mgr.h>`

`OSKIT_COMDECL` **oskit_lock_mgr_allocate_lock**(`oskit_lock_mgr_t` **lmgr*, [out] `oskit_lock_t` **out_lock*);

Description

This method returns an `oskit_lock_t` COM interface in `out_lock`.

Parameters

lmgr: The lock manager COM interface.

out_lock: The `oskit_lock` COM interface for the new lock.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.12.2 `allocate_critical_lock`: Allocate a critical thread-safe lock

Synopsis

`#include <oskit/com/lock_mgr.h>`

`OSKIT_COMDECL` **oskit_lock_mgr_allocate_critical_lock**(`oskit_lock_mgr_t` **lmgr*, [out] `oskit_lock_t` **out_lock*);

Description

This method returns an `oskit_lock_t` COM interface in `out_lock`. The lock is flagged as critical so that interrupts are blocked while the lock is held.

Parameters

lmgr: The lock manager COM interface.

out_lock: The `oskit_lock` COM interface for the new lock.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

4.12.3 `allocate_condvar`: Allocate a condition variable

Synopsis

`#include <oskit/com/lock_mgr.h>`

`OSKIT_COMDECL` **oskit_lock_mgr_allocate_condvar**(`oskit_lock_mgr_t` **lmgr*, [out] `oskit_condvar_t` **out_condvar*);

Description

This method returns an `oskit_condvar_t` COM interface in `out_condvar`. Condition variables may be used in conjuction with locks to form monitors.

Parameters

lmgr: The lock manager COM interface.

out_condvar: The `oskit_condvar` COM interface for the new condition variable.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 5

Input/Output Interfaces

This chapter defines the I/O-related COM interfaces which are defined by header files in the `oskit/io` directory. Most of these interfaces are fairly generic and can be used in a wide variety of situations. Some of these interfaces, such as the `bufio` interface, are extensions to other more primitive interfaces, and allow objects to export the same functionality in different forms, permitting clients to select the service that most directly meets their needs thereby reducing interface crossing overhead and increasing overall performance.

5.1 `oskit_absio`: Absolute I/O Interface

The `oskit_absio` interface supports reading from and writing to objects at specified absolute offsets, with no concept of a seek pointer. The `oskit_absio` interface is identical to the `oskit_blkio` COM interface, except that the block size is always one, since absolute IO is byte-oriented. In fact, an object that supports byte-granularity reads and writes can easily export both `oskit_blkio` and `oskit_absio` using exactly the same function pointer table, simply by implementing an `oskit_blkio` interface that always returns one from `getblocksize`, and then returning a pointer to that interface on queries for either `oskit_blkio` or `oskit_absio`.

The `oskit_absio` COM interface inherits from `IUnknown`, and has the following additional methods:

`read`: Read from this object, starting at the specified offset.

`write`: Write to this object, starting at the specified offset.

`getsize`: Get the current size of this object.

`setsize`: Set the current size of this object.

5.1.1 `read`: Read from this object, starting at specified offset

SYNOPSIS

`#include <oskit/io/absio.h>`

`OSKIT_COMDECL` **oskit_absio_read**(`oskit_absio_t` **f*, `void` **buf*, `oskit_off_t` *offset*, `oskit_size_t` *amount*, [out] `oskit_size_t` **out_actual*);

DESCRIPTION

This method reads no more than `amount` bytes into `buf` from this object, starting at `offset`. `out_actual` is set to the actual number of bytes read.

PARAMETERS

f: The object from which to read.

buf: The buffer into which the data is to be copied.

offset: The offset in this object at which to start reading.

amount: The maximum number of bytes to read.

out_actual: The actual number of bytes read.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.1.2 `write`: Write to this object, starting at specified offset

SYNOPSIS

`#include <oskit/io/absio.h>`

`OSKIT_COMDECL` **oskit_absio_write**(`oskit_absio_t` **f*, `const void` **buf*, `oskit_off_t` *offset*, `oskit_size_t` *amount*, [out] `oskit_size_t` **out_actual*);

Description

This method writes no more than `amount` bytes from `buf` into this object, starting at `offset`. `out_actual` is set to the actual number of bytes written.

Parameters

f: The object to which to write.

buf: The buffer from which the data is to be copied.

offset: The offset in this object at which to start writing.

amount: The maximum number of bytes to write.

out_actual: The actual number of bytes written.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.1.3 `getsize`: Get the size of this object

Synopsis

`#include <oskit/io/absio.h>`

`OSKIT_COMDECL` **oskit_absio_getsize**(`oskit_absio_t` **f*, [out] `oskit_off_t` **out_size*);

Description

This method returns the current size of this object in bytes.

Parameters

f: The object whose size is desired.

out_size: The current size in bytes of this object.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.1.4 `setsize`: Set the size of this object

Synopsis

`#include <oskit/io/absio.h>`

`OSKIT_COMDECL` **oskit_absio_setsize**(`oskit_absio_t` **f*, `oskit_off_t` *new_size*);

Description

This method sets the size of this object to `new_size` bytes. If `new_size` is larger than the former size of this object, then the contents of the object between its former end and its new end are undefined.

Note that some absolute I/O objects may be fixed-size, such as objects representing preallocated memory buffers; in such cases, this method will always return `OSKIT_E_NOTIMPL`.

Parameters

f: The object whose size is to be changed.

new_size: The new size in bytes for this object.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.2 `oskit_asyncio`: Asynchronous I/O Interface

XXX This section needs work.

The `oskit_asyncio` interface provides interfaces in support of basic asynchronous I/O, based on registered callback objects. This can be used, for example, to implement Unix `SIGIO` or `select` or `POSIX.1b aio`.

The `oskit_asyncio` COM interface inherits from `IUnknown`, and has the following additional methods:

`poll`: Poll for currently pending asynchronous I/O conditions.

`add_listener`: Add a callback object for async I/O events.

`remove_listener`: Remove a previously registered callback object.

`readable`: Returns the number of bytes that can be read.

5.3 `oskit_blkio`: Block I/O Interface

The `oskit_blkio` interface supports reading and writing of raw data in units of fixed-sized blocks which are some power of two. This interface is identical to the `oskit_absio` interface except for the addition of a `getblocksize` method; in fact, an object that supports byte-granularity reads and writes can easily export both `oskit_blkio` and `oskit_absio` using exactly the same function pointer table, simply by implementing an `oskit_blkio` interface that always returns one from `getblocksize`, and then returning a pointer to that interface on queries for either `oskit_blkio` or `oskit_absio`.

The `oskit_blkio` interface inherits from `IUnknown`, and has the following additional methods:

`getblocksize`: Return the minimum block size of this block I/O object.

`read`: Read from this object, starting at the specified offset.

`write`: Write to this object, starting at the specified offset.

`getsize`: Get the current size of this object.

`setsize`: Set the current size of this object.

5.3.1 `getblocksize`: Return the minimum block size of this block I/O object

Synopsis

`#include <oskit/io/blkio.h>`

`OSKIT_COMDECL_U` **oskit_blkio_getblocksize**(`oskit_blkio_t` **f*);

Description

This method simply returns the block size of the object, which must be a power of two. Calls by the client to read from or write to the object must only use offsets and sizes that are evenly divisible by this block size.

Parameters

f: The block I/O object.

Returns

Returns the block size of the object.

5.3.2 `read`: Read from this object, starting at specified offset

Synopsis

`#include <oskit/io/blkio.h>`

`OSKIT_COMDECL` **oskit_blkio_read**(`oskit_blkio_t` **f*, `void` **buf*, `oskit_off_t` *offset*, `oskit_size_t` *amount*, [out] `oskit_size_t` **out_actual*);

Description

This method reads no more than `amount` bytes into `buf` from this object, starting at `offset`. `out_actual` is set to the actual number of bytes read.

Parameters

f: The object from which to read.

buf: The buffer into which the data is to be copied.

offset: The offset in this object at which to start reading. Must be a multiple of the object's block size.

amount: The maximum number of bytes to read. Must be a multiple of the object's block size.

out_actual: The actual number of bytes read. Must be a multiple of the object's block size.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.3.3 `write`: Write to this object, starting at specified offset

Synopsis

`#include <oskit/io/blkio.h>`

`OSKIT_COMDECL` **oskit_blkio_write**(`oskit_blkio_t` **f*, `const void` **buf*, `oskit_off_t` *offset*, `oskit_size_t` *amount*, [out] `oskit_size_t` **out_actual*);

Description

This method writes no more than `amount` bytes from `buf` into this object, starting at `offset`. `out_actual` is set to the actual number of bytes written.

Parameters

f: The object to which to write.

buf: The buffer from which the data is to be copied.

offset: The offset in this object at which to start writing. Must be a multiple of the object's block size.

amount: The maximum number of bytes to write. Must be a multiple of the object's block size.

out_actual: The actual number of bytes written. Must be a multiple of the object's block size.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.3.4 `getsize`: Get the size of this object

Synopsis

`#include <oskit/io/blkio.h>`

`OSKIT_COMDECL` **oskit_blkio_getsize**(`oskit_blkio_t` **f*, [out] `oskit_off_t` **out_size*);

Description

This method returns the current size of this object in bytes.

Parameters

f: The object whose size is desired.

out_size: The current size in bytes of this object. Must be a multiple of the object's block size.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.3.5 `setsize`: Set the size of this object

Synopsis

`#include <oskit/io/blkio.h>`

`OSKIT_COMDECL` **oskit_blkio_setsize**(`oskit_blkio_t` **f*, `oskit_off_t` *new_size*);

Description

This method sets the size of this object to `new_size` bytes. If `new_size` is larger than the former size of this object, then the contents of the object between its former end and its new end are undefined.

Note that some block I/O objects may be fixed-size, such as objects representing physical disks or partitions; in such cases, this method will always return `OSKIT_E_NOTIMPL`.

Parameters

f: The object whose size is to be changed.

new_size: The new size in bytes for this object. Must be a multiple of the object's block size.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.4 `oskit_bufio`: Buffer-based I/O interface

The `oskit_bufio` interface extends the `oskit_absio` interface, providing additional alternative methods of accessing the object's data. In particular, for objects whose data is stored in an in-memory buffer of some kind, this interface allows clients to obtain direct access to the buffer itself so that they can read and write data using loads and stores, rather than having to copy data into and out of the buffer using the `read` and `write` methods. In addition, this interface provides similar methods to allow clients to "wire" the buffer's contents to physical memory, enabling DMA-based hardware devices to access the buffer directly.

However, note that only the read/write methods, inherited from `oskit_absio`, are mandatory; the others may consistently fail with `OSKIT_E_NOTIMPL` if they cannot be implemented efficiently in a particular situation. In that case, the caller must use the basic `read` and `write` methods instead to copy the data. In other words, `oskit_bufio` object implementations are not *required* to implement direct buffer access, either software- or DMA-based; the purpose of this interface is merely to allow them to provide this optional functionality easily and consistently. In general, the `map` and `wire` methods should only be implemented if they can be done more efficiently than simply copying the data. Further, even if a buffer I/O implementation does implement `map` and/or `wire` it may allow only one mapping or wiring to be in effect at once, failing if the client attempts to map or wire the buffer a second time before the first mapping is undone. Similarly, on some buffer I/O implementations, these operations may only work on certain areas of the buffer or only when the request has certain size or alignment properties: for example, a buffer object that stores data in discontiguous segments, such as BSD's `mbuf` system, may only allow a buffer to be mapped if the requested region happens to fall entirely within one segment. Thus, the client of a `bufio` object should call the `map` or `wire` methods whenever it can take advantage of direct buffer access, but must always be prepared to fall back to the basic copying methods.

A particular buffer object may be semantically read-only or write-only; it is assumed that parties passing `bufio` objects around will agree upon this as part of their protocols. For a read-only buffer, the `write` method may or may not fail, and a mapping established using the `map` method may or may not actually be a read-only memory mapping; it is the client's responsibility not to attempt to write to the buffer. Similarly, for a write-only buffer, the `read` method may or may not fail; it is the client's responsibility not to attempt to read from the buffer.

The `oskit_bufio` interface extends the `oskit_absio` interface with the following additional methods:

`map`: Map some or all of this buffer into locally accessible memory.

`unmap`: Release a previously mapped region of this buffer.

`wire`: Wire a region of this buffer into contiguous physical memory.

`unwire`: Unwire a previously wired region of this buffer.

`copy`: Create a copy of the specified portion of this buffer.

5.4.1 `map`: Map some or all of this buffer into locally accessible memory

SYNOPSIS

`#include <oskit/io/bufio.h>`

`OSKIT_COMDECL` **map**(`oskit_bufio_t` **io*, [out] `void` ***addr*, `oskit_off_t` *offset*, `oskit_size_t` *amount*);

DESCRIPTION

This method attempts to map some or all of this buffer into memory directly accessible to the client, so that the client can access it using loads and stores. The operation may or may not succeed, depending on the parameters and the implementation of the object; if it fails, the client must be prepared to fall back to the basic `read` and `write` methods. If the mapping operation succeeds, the pointer returned is not guaranteed to have any particular alignment.

If a call to the `map` method requests only a subset of the buffer to be mapped, the object may actually map more than the requested amount; however, since no information is passed back indicating how much of the buffer was actually mapped, the client must only attempt to access the region it requested.

Note that this method does not necessarily twiddle with virtual memory, as its name may seem to imply; in fact in most cases in which it is implemented at all, it just returns a pointer to a buffer if the data is already in locally-accessible memory.

Parameters

io: The object whose contents are to be mapped.

addr: On success, the method returns in this parameter the address at which the client can directly access the requested buffer region.

offset: The offset into the buffer of the region to be mapped.

size: The size of the region to be mapped.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.4.2 `unmap`: Release a previously mapped region of this buffer

Synopsis

`#include <oskit/io/bufio.h>`

`OSKIT_COMDECL` **unmap**(`oskit_bufio_t *`*io*, `void *`*addr*, `oskit_off_t` *offset*, `oskit_size_t` *amount*);

Description

After a successful call to the `map` method, the client should call this method after it is finished accessing the buffer directly, so that the buffer object can clean up and free any resources that might be associated with the mapping.

The *addr* parameter passed to this method must be exactly the value returned by the `map` request, and the *offset* and *amount* parameters must be exactly the same as the values previously passed in the corresponding `map` call. In other words, clients must only attempt to unmap whole regions; they must not attempt to unmap only part of a region, or to unmap two previously mapped regions in one call, even if the two regions appear to be contiguous in memory.

Parameters

io: The object whose contents are to be mapped.

addr: The address of the mapped region, as returned from the corresponding `map` call.

offset: The offset into the buffer of the mapped region, as passed to the corresponding `map` call.

size: The size of the mapped region, as passed to the corresponding `map` call.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.4.3 `wire`: Wire a region of this buffer into contiguous physical memory

Synopsis

`#include <oskit/io/bufio.h>`

`OSKIT_COMDECL` **wire**(`oskit_bufio_t` **io*, [out] `oskit_addr_t` **phys_addr*, `oskit_off_t` *offset*, `oskit_size_t` *amount*);

Description

This method attempts to wire down some or all of this buffer into memory directly accessible by DMA hardware. The operation may or may not succeed, depending on the parameters and the implementation of the object; if it fails, the client must be prepared to fall back to the basic `read` and `write` methods.

If the wiring operation succeeds, the physical address of the buffer is guaranteed not to change or otherwise become invalid until the region is unwired or the `bufio` object is released. The wired buffer is not guaranteed to have any particular alignment or location properties: for example, on a PC, if the device that is going to be accessing the buffer requires memory below 16MB, then it must be prepared to use appropriate bounce buffers if the wired buffer turns out to be above 16MB.

If a call to the `wire` method requests only a subset of the buffer to be mapped, the object may actually wire more than the requested amount; however, since no information is passed back indicating how much of the buffer was actually wired, the client must only attempt to use the region it requested.

Parameters

io: The object whose contents are to be wired.

addr: On success, the method returns in this parameter the physical address at which DMA hardware can directly access the requested buffer region.

offset: The offset into the buffer of the region to be wired.

size: The size of the region to be wired.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.4.4 `unwire`: Unwire a previously wired region of this buffer

Synopsis

`#include <oskit/io/bufio.h>`

`OSKIT_COMDECL` **unwire**(`oskit_bufio_t` **io*, `void` **addr*, `oskit_off_t` *offset*, `oskit_size_t` *amount*);

Description

After a successful call to the `wire` method, the client should call this method after the hardware is finished accessing the buffer directly, so that the buffer object can clean up and free any resources that might be associated with the wiring.

The *addr* parameter passed to this method must be exactly the value returned by the `wire` request, and the *offset* and *amount* parameters must be exactly the same as the values previously passed in the corresponding `wire` call. In other words, clients must only attempt to unwire whole regions; they must not attempt to unwire only part of a region, or to unwire two previously wired regions in one call, even if the two regions appear to be contiguous in physical memory.

Parameters

io: The object whose contents are to be wired.

addr: The address of the wired region, as returned from the corresponding `map` call.

offset: The offset into the buffer of the wired region, as passed to the corresponding `wire` call.

size: The size of the wired region, as passed to the corresponding `wire` call.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.4.5 `copy`: Create a copy of the specified portion of this buffer

Synopsis

`#include <oskit/io/bufio.h>`

`OSKIT_COMDECL` **copy**(`oskit_bufio_t` **io*, `oskit_off_t` *offset*, `oskit_size_t` *amount*, [out] `oskit_bufio_t` ***out_io*);

Description

This method attempts to create a logical copy of a portion of this buffer object (possibly the whole buffer), returning a new `oskit_bufio` object representing the copy. As with the `map` and `wire` methods, this method should only be implemented by an object if it can be done more efficiently than a simple "brute-force" copy using `read`. For example, in virtual memory environments, the object may be able to use copy-on-write optimizations. Similarly, if the buffer's contents are stored in special memory not efficiently accessible to the processor, such as memory on a video or coprocessor board, this method could use on-board hardware to perform a much faster copy.

Parameters

io: The object whose contents are to be copied.

offset: The offset into the buffer of the region to be copied.

size: The size of the region to be copied.

out_io: On success, this parameter holds the `bufio` object representing the newly created copy of the buffer's contents.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.5 `oskit_netio`: Network packet I/O interface

This interface builds on the above interfaces to provide a clean and simple but powerful interface for passing network packets between device drivers and protocol stacks, and possibly between layers of a protocol stack as well.

The `oskit_netio` interface uses a symmetric sender-driven model for asynchronous communication. Each party involved (e.g., the network device driver and the protocol stack) must implement a `netio` object and pass a reference to its own `netio` object to the other party. For example, in the `oskit_netdev` interface, which represents a network device of some kind, this exchange of `netio` objects occurs when the protocol stack or other client opens the device. The `oskit_netio` interface defines only a single additional method beyond the basic methods inherited from `oskit_iunknown`; this method, appropriately named `push`, is used to "push" a network packet to the "other" party. For example, when a network device driver receives a packet from the hardware, the driver calls the `push` method on the `netio` object provided by the protocol stack; conversely, when the protocol stack needs to send a packet, it calls the `netio` object returned by the device driver at the time the device was opened. Thus, a `netio` object essentially represents a "packet consumer."

The following section describes the specifics of the `push` method.

5.5.1 `push`: Push a packet through to the packet consumer

Synopsis

```
#include <oskit/io/netio.h>
OSKIT_COMDECL push(oskit_netio_t *io, oskit_bufio *buf, oskit_size_t size);
```

Description

This method feeds a network packet to the packet consumer represented by the `netio` object; what the consumer does with the packet depends entirely on who the consumer is and how it is configured. The packet is contained in a `bufio` object which must be at least the size of the packet, but may be larger; the *size* parameter on the `push` call indicates the actual size of the packet.

If the consumer needs to hold on to the provided `bufio` object after returning from the call, it must call `addref` on the `bufio` object to obtain its own reference; then it must release this reference at some later time when it is done with the buffer. Otherwise, if the consumer doesn't obtain its own reference, the caller may recycle the buffer as soon as the call returns.

The passed buffer object is logically read-only; the consumer must not attempt to write to it. The size parameter to this call is the actual size of the packet; the size of the buffer, as returned by the `getsize` method, may be larger than the size of the packet.

Parameters

io: The `oskit_netio` interface representing the packet consumer.

buf: The `oskit_bufio` interface to the buffer object containing the packet.

size: The actual size of the packet; must be less than or equal to the size of the buffer object.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.6 `oskit_posixio`: posix I/O interface

The `oskit_posixio` interface defines the minimal POSIX I/O interface that any POSIX I/O object (file, device, pipe, socket, etc) can be expected to support. Only per-object methods are provided by this interface. Additional I/O operations are supported through separate interfaces, such as the `oskit_stream` interface and `oskit_absio` COM interface.

The `oskit_posixio` COM interface inherits from `oskit_iunknown`, and has the following additional methods:

`stat`: Get this object's attributes.

`setstat`: Set this object's attributes.

`pathconf`: Get this object's value for a configuration option variable.

5.6.1 `stat`: Get attributes of this object

SYNOPSIS

`#include <oskit/io/posixio.h>`

`OSKIT_COMDECL` **oskit_posixio_stat**(`oskit_posixio_t` **f*, [out] `oskit_stat_t` **out_stats*);

DESCRIPTION

This method returns the attributes of this object. Depending on the type of object, only some of the attributes may be meaningful. `out_stats` is a pointer to a `oskit_stat_t` structure defined as follows:

```
struct oskit_stat {
    oskit_dev_t         dev;      /* device on which inode resides    */
    oskit_ino_t         ino;      /* inode's number                   */
    oskit_mode_t        mode;     /* file mode                        */
    oskit_nlink_t       nlink;    /* number of hard links to file     */
    oskit_uid_t         uid;      /* user id of owner                 */
    oskit_gid_t         gid;      /* group id of owner                */
    oskit_dev_t         rdev;     /* device number, for device files  */
    oskit_timespec_t    atime;    /* time of last access              */
    oskit_timespec_t    mtime;    /* time of last data modification   */
    oskit_timespec_t    ctime;    /* time of last attribute change    */
    oskit_off_t         size;     /* size in bytes                    */
    oskit_u64_t         blocks;   /* blocks allocated for file        */
    oskit_u32_t         blksize;  /* optimal block size in bytes      */
};
```

PARAMETERS

f: The object whose attributes are desired.

out_stats: The attributes of the specified object.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.6.2 `setstat`: Set the attributes of this object

SYNOPSIS

`#include <oskit/io/posixio.h>`

`OSKIT_COMDECL` **oskit_posixio_setstat**(`oskit_posixio_t` **f*, `oskit_u32_t` *mask*, `const oskit_stat_t` **stat*);

DESCRIPTION

This method sets the attributes specified in `mask` to the values specified in `stat`. `mask` may be any combination of the following:

`OSKIT_STAT_MODE`: Set the file mode, except for the file type bits, as in the Unix `chmod` system call.

`OSKIT_STAT_UID`: Set the file user id, as in the Unix `chown` system call.

`OSKIT_STAT_GID`: Set the file group id, as in the Unix `chown` system call.

`OSKIT_STAT_SIZE`: Set the file size, as in the Unix `truncate` system call.

`OSKIT_STAT_ATIME`: Set the file's last access timestamp to a particular value, as in the Unix `utimes` system call with a non-`NULL` parameter.

`OSKIT_STAT_MTIME`: Set the file's last data modification timestamp to a particular value, as in the Unix `utimes` system call with a non-`NULL` parameter.

`OSKIT_STAT_UTIMES_NULL`: Set the file's last access timestamp and data modification timestamp to the current time, as in the Unix `utimes` system call with a `NULL` parameter.

Typically, this method is not supported for symbolic links.

PARAMETERS

f: The object whose attributes are to be changed.

mask: The attributes to be changed.

stat: The new attribute values.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.6.3 `pathconf`: Get value of a configuration option variable

SYNOPSIS

`#include <oskit/io/posixio.h>`

`OSKIT_COMDECL` **oskit_posixio_pathconf**(`oskit_posixio_t` **f*, `oskit_s32_t` *option*, [out] `oskit_s32_t` **out_val*);

DESCRIPTION

This method returns the value of the specified configuration option variable for this object. The value of `option` may be one of the following:

`OSKIT_PC_LINK_MAX`: Get the maximum file link count.

`OSKIT_PC_MAX_CANON`: Get the maximum size of the terminal input line.

`OSKIT_PC_MAX_INPUT`: Get the maximum input queue size.

`OSKIT_PC_NAME_MAX`: Get the maximum number of bytes in a filename.

`OSKIT_PC_PATH_MAX`: Get the maximum number of bytes in a pathname.

`OSKIT_PC_PIPE_BUF`: Get the maximum atomic write size to a pipe.

`OSKIT_PC_CHOWN_RESTRICTED`: Determine whether use of chown is restricted.

`OSKIT_PC_NO_TRUNC`: Determine whether too-long pathnames produce errors.

`OSKIT_PC_VDISABLE`: Get value to disable special terminal characters.

`OSKIT_PC_ASYNC_IO`: Determine whether asynchronous IO is supported.

`OSKIT_PC_PRIO_IO`: Determine whether prioritized IO is supported.

`OSKIT_PC_SYNC_IO`: Determine whether synchronized IO is supported.

Parameters

f: The object from which to obtain a configuration option value

option: The configuration option variable

out_val: The value of the configuration option value.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7 `oskit_ttystream`: Interface to Unix TTY-like streams

This interface extends the standard COM `IStream` interface with POSIX/Unix TTY functionality, such as methods to control serial port settings, enable, disable, and control line editing, flush the input and output queues, etc.

This interface is currently exported by character-oriented device drivers incorporated into the OSKit from legacy systems such as BSD and Linux, in which full Unix TTY functionality can be provided easily. In the future, these drivers are expected to export more minimal, lower-level interfaces instead of or in addition to this interface; however, in the short term, this interface allows clients to obtain full Unix terminal functionality quickly and easily.

The `oskit_ttystream` interface inherits from `oskit_stream`, and has the following additional methods:

`getattr`: Get the stream's current TTY attributes.

`setattr`: Set the stream's TTY attributes.

`sendbreak`: Send a break signal over the line.

`drain`: Wait until all buffered output has been transmitted.

`flush`: Discared buffered input and/or output data.

`flow`: Suspend or resume data transmission or reception.

In addition, this header file defines a structure called `oskit_termios`, corresponding to the standard POSIX `termios` structure, and a set of related definitions used to specify terminal-related settings. See the POSIX and Unix standards for details on the exact contents and meaning of this structure.

5.7.1 `getattr`: Get the stream's current TTY attributes

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **getattr**(`oskit_ttystream_t *`*tty*, [out] `struct oskit_termios *`*attr*);

Description

This method retrieves the current line settings of this stream and returns them in the specified `oskit_termios` structure. This method corresponds to the POSIX `tcgetattr` function; see the POSIX standard for details.

Parameters

tty: The TTY stream object to query.

attr: The structure to be filled with the current line settings.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7.2 `setattr`: Set the stream's TTY attributes

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **setattr**(`oskit_ttystream_t *`*tty*, `const struct oskit_termios *`*attr*);

Description

This method sets the line settings of this stream based on the specified `oskit_termios` structure. This method corresponds to the POSIX `tcsetattr` function; see the POSIX standard for details.

Parameters

tty: The TTY stream object to modify.

attr: The structure containing the new line settings.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7.3 `sendbreak`: Send a break signal

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **sendbreak**(`oskit_ttystream_t` **tty*, `oskit_u32_t` *duration*);

Description

On streams controlling asynchronous serial communication, this method sends a break signal (a continuous stream of zero-valued bits) for a specific duration. This method corresponds to the POSIX `tcsendbreak` function; see the POSIX standard for details.

Parameters

tty: The TTY stream on which to send the break.

duration: The duration of the break signal to send. If this parameter is zero, then the duration will be between 0.25 and 0.5 seconds.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7.4 `drain`: Wait until all buffered output has been transmitted

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **drain**(`oskit_ttystream_t` **tty*);

Description

This method waits until any buffered output data that has been written to the stream is successfully transmitted. This method corresponds to the POSIX `tcdrain` function; see the POSIX standard for details.

Parameters

tty: The TTY stream object to drain.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7.5 `flush`: Discared buffered input and/or output data

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **flush**(`oskit_ttystream_t` **tty*, `int` *queue_selector*);

Description

This method discards any buffered output data that has not yet been transmitted, and/or any buffered input data that has not yet been read, depending on the *queue_selector* parameter. This method corresponds to the POSIX `tcflush` function; see the POSIX standard for details.

Parameters

tty: The TTY stream object to flush.

queue_selector: Must be one of the following:

- `OSKIT_TCIFLUSH`: Flush the input buffer.
- `OSKIT_TCOFLUSH`: Flush the output buffer.
- `OSKIT_TCIOFLUSH`: Flush the input and output buffers.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

5.7.6 `flow`: Suspend or resume data transmission or reception

Synopsis

`#include <oskit/io/ttystream.h>`

`OSKIT_COMDECL` **flow**(`oskit_ttystream_t` **tty*, `int` *action*);

Description

This method controls the transmission or reception of data on this TTY stream. This method corresponds to the POSIX `tcflow` function; see the POSIX standard for details.

Parameters

tty: The TTY stream object to control.

action: Must be one of the following:

- `OSKIT_TCOOFF`: Suspend output.
- `OSKIT_TCOON`: Restart output.
- `OSKIT_TCIOFF`: Transmit a STOP character.
- `OSKIT_TCION`: Transmit a START character.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 6

OSKit Device Driver (OS Environment) Framework

6.1 Introduction

Note: the framework's obsolete name as "device driver framework" is historical baggage from its first client component, imported device drivers. Today, a more accurate name would be the "OS Environment" framework. It provides the API and glue used by all "large" encapsulated components (devices, networking, filesystems) imported from other operating systems. We'll change the name and documentation in the future.

A note on organization and content: this chapter really contains three quite separate parts: a general narrative about execution models, some very sketchy documentation of the "up-side" device interfaces, and the bulk covers the "osenv" interfaces. A later chapter (12) talks sketchily about the default implementation of the interfaces found here.

The OSKit device driver framework is a device driver interface specification designed to allow *existing* device drivers to be borrowed from well-established operating systems in source form, and used unchanged to provide extensive device support in new operating systems or other programs that need device drivers (e.g., hardware configuration management utilities). With appropriate glue, this framework can also be used in an existing operating system to augment the drivers already supported by the OS. (We believe it's possible to extend the framework to accomodate drivers in binary form.) This chapter describes the device driver framework itself; other chapters later in this document describe specific libraries provided as part of the OSKit that provide driver and kernel code implementing or supporting this interface.

The primary goals of this device driver framework are, in order from most to least important:

1. **Breadth of hardware coverage.** There is a tremendous range of common hardware available these days, each typically supporting its own device programming interface and requiring a special device driver. Device drivers for a given device are generally only available for a few operating systems, depending on how well-established the particular device and OS is. Thus, in order to achieve maximum hardware coverage, the framework must be capable of incorporating device drivers originally written for a variety of different operating systems.

2. **Adaptability to different environments.** This device driver framework is intended to be useful not only in traditional Unix-like kernels, but also in operating systems with widely different structures, e.g., kernels written in a "stackless" interrupt model, or kernels that run all device drivers as user mode programs, or kernels that do not support virtual memory.

3. **Ease-of-use.** It should be reasonably easy for an OS developer to add support for this framework to a new or existing OS. The set of support functions the OS developer must supply should be kept as small and simple as possible, and there should be few "hidden surprises" lurking in the drivers. In situations where existing device drivers supported by the OSKit have special requirements that the OS must satisfy in order to use them, these requirements are clearly documented in the relevant chapters.

4. **Performance.** In spite of the above constraints, device drivers should be able to run under this framework with as little unnecessary overhead as possible. Performance issues are discussed further in Section 6.5.

Since the most important goal of this framework is to achieve wide hardware coverage by making use of existing drivers, and not to define a new model or interface for writing drivers, it is somewhat more demanding and restricting in terms of OS support than would be ideal if we were writing entirely new device drivers from scratch. Other device driver interface standards, such as DDI/DKI and UDI, are not designed to allow easy adaptation of existing drivers; instead, they are intended to define and restrict the interfaces and environment used by *new* drivers specially written for those interfaces, so that these new drivers will be as widely useful as possible. For example, UDI requires all conforming drivers to be implemented in a nonblocking interrupt model; this theoretically allows UDI drivers to run easily in either process-model or interrupt-model kernels, but at the same time it eliminates all possibility of adapting existing traditional process-model drivers to be UDI conformant without extensive changes to the drivers themselves. Hopefully, at some point in the future, one of these more generic device driver standards will become commonplace enough so that conforming device drivers are available for "everything"; however, until then, the OSKit device driver framework takes a compromise approach, being designed to allow easy adaptation of a wide range of existing drivers while keeping the primary interface as simple and flexible as possible.

6.1.1 Full versus partial compliance

Because the range of existing drivers to be adopted under this framework is so diverse in terms of the assumptions and restrictions made by the drivers, it would be impractical to define the requirements of the framework as a whole to be the "union" of all the requirements of all possible drivers. For example, if we had taken that approach, then the framework would only be usable in kernels in which all physical memory is directly mapped into the kernel's virtual address space at identical addresses, because *some* drivers will not work unless that is the case. This restriction would make the framework completely unusable in many common OS environments, even though there are plenty of drivers available that *don't* make the $virtual = physical$ assumption and should work fine in OS environments that don't meet that requirement.

For this reason, we have defined the framework itself to be somewhat more generic than is suitable for "all" existing drivers, and to account for the remaining "problematic" drivers, we make a distinction between *full* and *partial compliance*. A fully compliant driver is a driver that makes no additional assumptions or requirements beyond those defined as part of the basic driver framework; these drivers should run in any environment that supports the framework. A partially compliant driver is a driver that is compliant with the framework, *except* that it makes one or more additional restrictions or requirements, such as the $virtual = physical$ requirement mentioned above. For each partially-compliant driver provided with the OSKit, the exact set of additional restrictions made by the driver are clearly documented and provided in both human- and machine-readable form so that a given OS environment can make use of the framework as a whole while avoiding drivers that will not work in the environment it provides.

6.2 Organization

In a typical OS environment in which all device drivers run in the kernel, Figure 6.1 illustrates the basic organization of the device driver framework.

The heavy black horizontal lines represent the actual interfaces comprising the framework, which are described in this chapter. There are two primary interfaces: the *device driver interface* (or just "driver interface"), which the OS kernel uses to invoke the device drivers; and the *driver-kernel interface* (or just "kernel interface"), which the device drivers use to invoke kernel support functions. The kernel implements the kernel interface and uses the driver interface; the drivers implement the driver interface and use the kernel interface.

Chapter 12 describes a library supplied as part of the OSKit that provides facilities to help the OS implement the kernel interface and use the driver interface effectively. Default implementations suitable in typical kernel environments are provided for many operations; the OS can use these default implementations or not, as the situation demands.

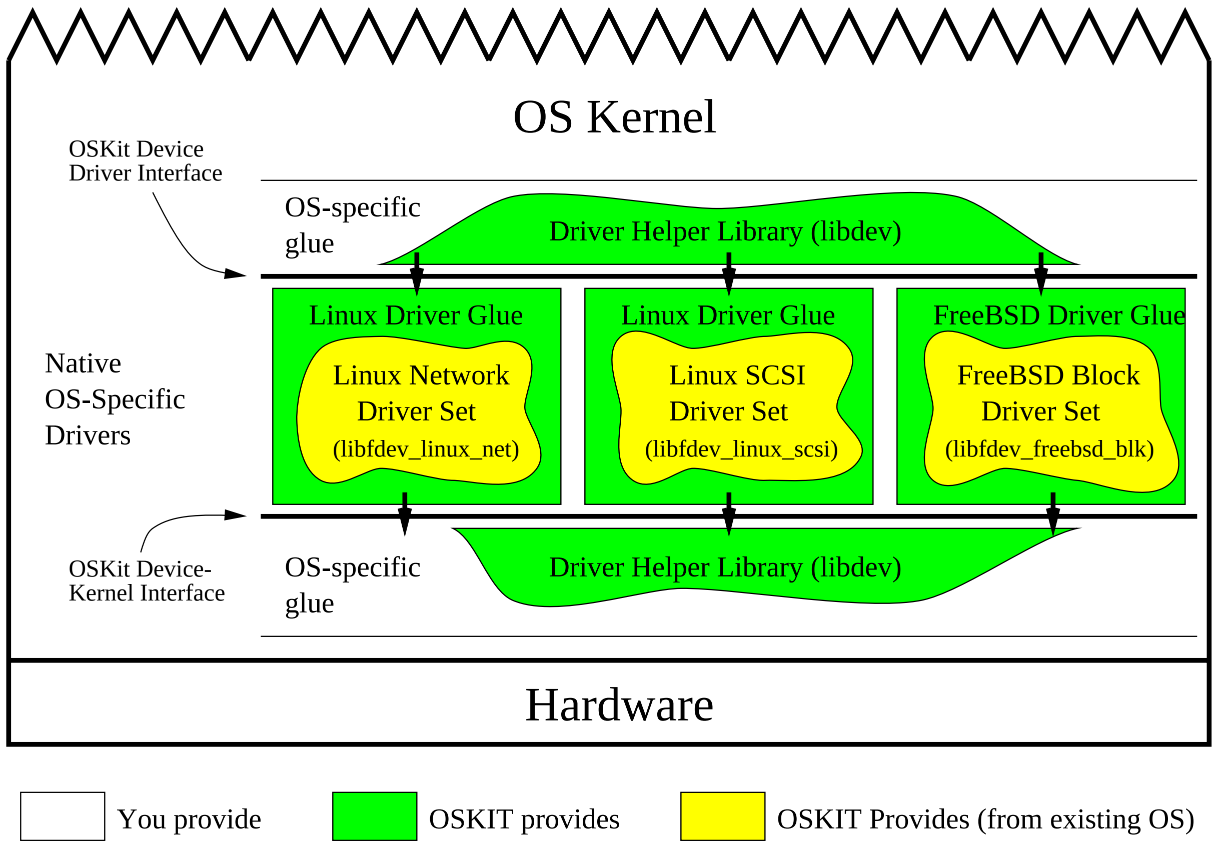

Figure 6.1: Organization of OSKit Device Driver Framework in a typical kernel

Several chapters in Part IV describe device driver sets supplied with the OSKit for use in environments supporting the OSKit device driver framework. Since the Flux project is not in the driver writing business, and does not wish to be, these driver sets are derived from existing kernels, either unchanged or with as little code modified as possible so that the versions of the drivers in the OSKit can easily be kept up-to-date with the original source bases from which they are derived.

6.3 Driver Sets

Up to this point we have used the term "device driver set" fairly loosely; however, in the context of the OSKit device driver framework, this term has a very important, specific meaning. A driver set is a set of related device drivers that work together and are fairly tightly integrated together. Different driver sets running in a given environment are independent of each other and oblivious to each other's presence. Drivers within a set may share code and data structures internally in arbitrary ways; however, code in different driver sets may *not* directly share data structures. (Different driver sets may share code, but only if that code is "pure" or operates on a disjoint set of data structures: for example, driver sets may share simple functions such as `memcpy`.)

Of course, the surrounding OS can maintain shared data structures in whatever way it chooses; this is the only way drivers in different sets can interact with each other. For example, if a kernel is using a FreeBSD device driver to drive one network card and a Linux driver to drive another, then the kernel can take IP packets coming in on one card and route them out through the other card, but the network device drivers themselves are completely oblivious to each other's presence.

Some driver sets may contain only a single driver; this is ideal for modularity purposes, since in this case each such driver is independent of all others. Also, given some effort on the part of the OS, some multi-driver sets can be "split up" into multiple single-driver sets and used independently; Section 6.4.1 describes one way this can be done.

In essence, each driver set represents an "encapsulated environment" with a well-defined interface and a clearly-bounded set of state. The concept of a driver set has important implications throughout the device driver framework, especially in terms of execution environment and synchronization; the following sections describe these aspects of the framework in more detail.

Note that currently all "osenv" code in the same address space is essentially a single driver set. We are planning on changing this to allow drivers to be independant from each other. Currently, the only way to achieve this is to run them in separate address spaces.

6.4 Execution Model

Device drivers running in the OSKit device driver framework use the interruptible, blocking execution model, defined in Section 2.5, and all of the constraints and considerations described in that section generally apply to OSKit device drivers. However, there are a few execution model issues specific to device drivers, which are dealt with here.

6.4.1 Use in out-of-kernel, user-mode device drivers

In some situations, for reasons of elegance, modularity, configuration flexibility, robustness, or even (in some cases) performance, it is desirable to run device drivers in user mode, as "semi-ordinary" application programs. This is done as a matter of course by some microkernels. There is nothing in the OSKit device driver framework that prevents its device drivers from executing in user mode, and in fact the framework was deliberately designed with support for user-mode device drivers in mind.

Figure 6.2 illustrates an example system in which device drivers are located in user-mode processes. In this case, all of the code within a given driver set is part of the user-level device driver process, and the "surrounding" OS-specific code, which makes calls to the drivers through the driver interface, and provides the functions in the "kernel interface," is not actually kernel code at all but, rather, "glue" code that handles communication with the kernel and other processes. For example, many of the functions in the driver-kernel

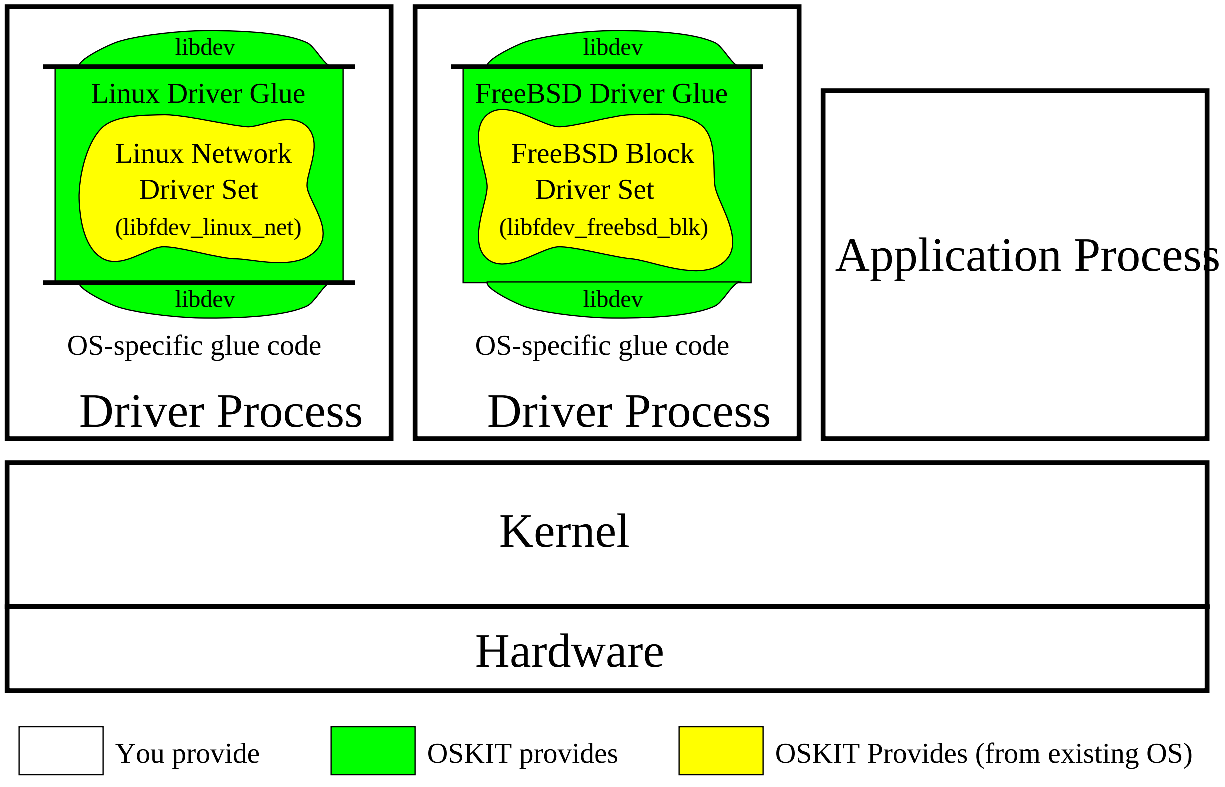

Figure 6.2: Using the framework to create user-mode device drivers

interface, such as the calls to allocate interrupt request lines, will be implemented by this glue code as system calls to the "actual" kernel, or as remote procedure calls to servers in other processes.

Device driver code running in user space will typically run in the context of ordinary threads; the execution environment required by the driver framework can be built on top of these threads in different ways. For example, the OS-specific glue code may run on only a single thread and use a simple coroutine mechanism to provide a separate stack for each outstanding process-level device driver operation; alternately, multiple threads may be used, in which case the glue code will have to use locking to provide the nonpreemptive environment required by the framework.

Dispatching interrupt handlers in these user-mode drivers can be handled in various ways, depending on the environment and kernel functionality provided. For example, interrupt handlers may be run as "signal handers" of some kind "on top of" the thread(s) that normally execute process-level driver code; alternatively, a separate thread may be used to run interrupt handlers. In the latter case, the OS-specific glue code must use appropriate locking to ensure that process-level driver code does not continue to execute while interrupt handlers are running.

Shared interrupt request lines

One particularly difficult problem for user-level drivers in general, and especially for user-level drivers built using this framework, is supporting shared interrupt lines. Many platforms, including PCI-based PCs, allow multiple unrelated devices to send interrupts to the processor using a single request line; the processor must then sort out which device(s) actually caused the interrupt by checking each of the possible devices in turn. With user-level drivers, the code necessary to perform this checking is typically part of the user-mode device driver, since it must access device-specific registers. Thus, in a "naive" implementation, when the kernel receives a device interrupt, it must notify *all* of the drivers hooked to that interrupt, possibly causing many unnecessary context switches for every interrupt.

The typical solution to this problem is to allow device drivers to "download" small pieces of "disambiguation" code into the kernel itself; the kernel then chains together all of the code fragments for a particular interrupt line, and when an interrupt occurs, the resulting code sequence determines exactly which device(s) caused the interrupt, and hence, which drivers need to be notified. This solution works fine for "native" drivers designed specifically for the kernel in question; however, there is no obvious, straightforward way to support such a feature in the driver framework.

For this reason, until a better solution can be found, the following policy applies to using shared interrupts in this framework: for a given shared interrupt line, either the kernel must unconditionally notify all registered drivers running under this framework, and take the resulting performance hit; or else the drivers running under this framework will not support shared interrupts at all. (Native drivers written specifically for the kernel in question can still use the appropriate facilities to support shared interrupt lines efficiently.)

6.5 Performance

Since this framework emphasizes breadth, adaptability, and ease-of-use over raw performance, the performance of device drivers running under this framework is likely to suffer somewhat; how much depends on how well-matched the particular driver is to the driver framework and to the host OS. Various factors can influence driver performance: for example, if the OS's network code does not match the network drivers in terms of whether scatter/gather message buffers are supported or required, performance is likely to suffer somewhat due to extra copying between the driver and the OS's network code. The OS developer will have to take these issues into account when selecting *which* sets of device drivers to use (e.g., FreeBSD versus Linux network drivers). If the device driver sets are chosen carefully and the OS's driver support code is designed well, in many cases it should be possible to use these drivers with minimal performance loss.

Another consideration is how extensively the OS should rely on this device driver framework. There is nothing preventing the OS from maintaining its own (probably smaller) collection of "native" drivers designed and tuned for the particular OS; this way, the OS can achieve maximum performance for particularly common or performance-critical hardware devices, and use the larger set of device drivers easily available through this framework to provide support for other types of hardware that otherwise wouldn't be supported at all. This approach of combining native and emulated drivers is likely to be especially important for kernels that are not well matched to the existing drivers this framework was designed around: e.g., "stackless" interrupt model kernels which must run emulated device drivers on special threads or in user space.

For a very rough idea of the performance of drivers and kernels using this framework, see the results in our SOSP'97 paper "The Flux OSKit: A Substrate for OS and Language Research." Performance results for a related but less formal and less encapsulated framework can be found in the USENIX'96 paper "Linux Device Driver Emulation in Mach."

6.6 Device Driver Initialization

When the host OS is ready to start using device drivers in this framework, it typically calls a *probe function* for each driver set it uses; this function initializes the drivers and checks for hardware devices supported by any of the drivers in the set. If any such devices are found, they are *registered* with the host OS by calling a registration routine specific to the type of bus on which the device resides (e.g., ISA, PCI, SCSI). The host OS can then record this information internally so that it knows which devices are available for later use. The OS can implement device registration any way it chooses; however, the driver support library (`libdev`) provided by the OSKit provides a default implementation of a registration mechanism which builds a single "hardware tree" representing all known devices; see Section 12.2 for more information.

When a device driver discovers a device, it creates a *device node* structure representing the device. The device node structure can be of arbitrary size, and most of its contents are private to the device driver. However, the first part of the device node is always a structure of type `oskit_device_t`, defined in `oskit/dev/dev.h`, which contains generic information about the device and driver needed by the OS to make use of the device. In addition, depending on the device's type, there may be additional information available to the host OS, as described in the following section.

6.7 Device Classification

Device nodes have types that follow a C++-like single-inheritance subtyping relationship, where `oskit_device_t` is the ultimate ancestor or "supertype" of all device types.

In general, the host OS *must* know what *class* of device it is talking to in order to make use of it properly. On the other hand, it is *not* strictly necessary for the host OS to recognize the specific device *type*, although it may be able to make better use of the device if it does.

The block device class has the following attributes:

- All input and output is synchronously driven by the host OS, through calls to the read and write methods of the associated `blkio` object; the driver never calls the asynchronous I/O functions defined in Section 5.2. I/O operations always complete "promptly": barring device driver or hardware bugs, reads and writes are never delayed indefinitely due to external conditions. (This contrasts with network devices, for example, where input is received when another machine sends a message, not when the host OS asks for input.)

- There may be a minimum read/write granularity, or *block size*, which may be obtained through the `getblocksize` method. The block size is always a power of two (e.g., typically 512 for most disks), and is always less than the processor's minimum page size (`PAGE_SIZE`, Section 10.2.1). The *offset* and *count* parameters of all read/write calls made by the host OS to this device driver must be an even multiple of this block size. For block devices with no minimum read/write granularity, the driver specifies a block size of 1 (i.e., one-byte granularity).

- Block devices may have removable media, such as floppy drives, CD-ROM drives, or removable hard drives. The device driver provides an indication to the OS of whether or not the device supports removable media.

The character device class has the following characteristics:

- Output is synchronous, directed by the host OS, but input is asynchronous, directed by the external device.

- Incoming and outgoing data consists of a stream of bytes; there is no larger minimum read/write granularity. Multiple bytes of data can be sent and received in one operation, but this is just an optimization; there is no semantic difference from handling each byte individually.

The network device class has the following characteristics:

- Output is synchronous, directed by the host OS, but input is asynchronous, directed by the external device.

- Data is handled in units of packets; one send or receive operation is performed for each packet.

- Packets sent and received typically have specific size and format restrictions, depending on the specific network type (e.g., ethernet, myrinet).

Note that it would certainly be possible to decompose these device classes into a deeper type hierarchy. For example, in abstract terms it might make sense to arrange character and network devices under a single supertype representing "asynchronous" devices. However, since the structure representing this "abstract supertype" would contain essentially nothing in terms of actual code or data, this additional level was not deemed useful for the driver framework. Of course, the OS is free to use any type hierarchy (or non-hierarchy) it desires for its own data structures representing devices, drivers, etc.

6.8 Buffer Management

XXX overview

6.9 Asynchronous I/O

While asynchronous I/O is not directly suported by the OSKit device interface, it is possible to create an asychronous interface in the OS itself, which calls the blocking fdev functions.

6.10 Other Considerations

XXX some rare, poorly-designed hardware does not work right if long delays occur while programming the devices. (This is supposedly the case for some IDE drives, for example.) For this reason, reliability and hardware compatibility may be increased by implementing `osenv_intr_disable` as a function that *really does* disable all interrupts on the processor in question.

XXX Symbol name conflicts among libraries... For each existing driver set, provide a list of "reserved" symbols used by the set.

XXX This should be moved somewhere else:

> All functions may block, except those specifically designated as nonblocking.
>
> All functions may be called at any time, including during driver initialization. In other words, all of the functionality exposed by this interface must be present and fully operational by the time the device drivers are initialized.

6.11 Common Device Driver Interface

This section describes the OSKit device driver interfaces that are common to all types of drivers and hardware.

6.11.1 `dev.h`: common device driver framework definitions

Synopsis

```
#include <oskit/dev/dev.h>
```

XXX
oskit_dev_init
oskit_X_init_X
oskit_dump_drivers
oskit_dev_probe
oskit_dump_devices
rtc_get and rtc_set interfaces (Real time clock).

6.12 Driver Memory Allocation

The OS must provide routines for drivers to call to allocate memory for the private use of the drivers, as well as for I/O buffers and other purposes. The OSKit device driver framework defines a single set of memory allocation functions which all drivers running under the framework call to allocate and free memory.

Device drivers often need to allocate memory in different ways, or memory of different types, for different purposes. For this reason, the device driver framework defines a set of flags provided to each memory allocation function describing how the allocation is to be done, or what type of memory is required.

As with other aspects of the OSKit device driver framework, the `libdev` library provides default implementations of the memory allocation functions, but these implementations may be replaced by the OS as desired. The default implementations make a number of assumptions which are often invalid in "real" OS kernels; therefore, these functions will often be overridden by the client OS. Specifically, the default implementation assumes:

- The LMM pool `malloc_lmm` is used to manage kernel memory.
- Memory allocation and deallocation never block.
- All memory allocation functions can be called at interrupt time.
- All allocated blocks are physically as well as virtually contiguous.

Additionally, the default routines which deal with physical memory addresses make these assumptions:

- Virtual address is the same as the physical address.
- Paging is not enabled.

6.12.1 `osenv_memflags_t`: memory allocation flags

Synopsis

XXX typedef unsigned osenv_memflags_t;

Description

All of the memory allocation functions used by device drivers in the OSKit device framework take a parameter of type `osenv_memflags_t`, which is a bit field describing various option flags that affect how memory allocation is done. Device drivers often need to allocate memory that satisfies certain constraints, such as being physically contiguous, or page aligned, or accessible to DMA controllers. These flags abstract out these various requirements, so that all memory allocation requests made by device drivers are sent to a single set of routines; this design allows the OS maximum flexibility in mapping device memory allocation requests onto its internal kernel memory allocation mechanisms.

Routing all memory allocations through a single interface this way may have some impact on performance, due to the cost of decoding the *flags* argument on every allocation or deallocation call. However, this cost is expected to be small compared to the typical cost of actually performing the requested operation.

The specific flags currently defined are as follows:

`OSENV_AUTO_SIZE`: The memory allocator must keep track of the size of allocated blocks allocated using this flag; in this case, the value *size* parameter passed in the corresponding `osenv_mem_free` call is meaningless. For blocks allocated without this flag set, the caller (device driver) promises to keep track of the size of the allocated block, and pass it back to `osenv_mem_free` on deallocation.

It is possible for the OS to implement these memory allocation routines so that they ignore the `OSENV_AUTO_SIZE` flag and simply *always* keep track of block sizes themselves. However, note that in some situations, doing so may produce extremely inefficient memory usage. For example, if the OS memory allocation mechanism prefixes each block with a word containing the block's length, then any request by a device driver to allocate a page-aligned page (or some other naturally-aligned, power-of-two-sized block) will consume that page *plus* the last word of the previous page. If many successive allocations are done in this way, only every *other* page will be usable, and half of the available memory will be wasted. Therefore, it is generally a good idea for the memory allocation functions to pay attention to the `OSENV_AUTO_SIZE` flag, at least for allocations with alignment restrictions.

`OSENV_NONBLOCKING`: If set, this flag indicates that the memory allocator must not block during the allocation or deallocation operation. More specifically, the flag indicates that the device driver code must not be run in the context of other, concurrent processes while the allocation is taking place. Any calls to the allocation functions from interrupt handlers *must* specify the `OSENV_NONBLOCKING` flag.

`OSENV_PHYS_WIRED`: Indicates that the must must be non-pageable. Accesses to the returned memory must not fault.

`OSENV_PHYS_CONTIG`: Indicates the underlying physical memory must be contiguous.

`OSENV_VIRT_EQ_PHYS`: Indicates the virtual address must *exactly* equal the physical address so the driver may use them interchangeably. The `OSENV_PHYS_CONTIG` flag must also be set whenever this flag is set.

`OSENV_ISADMA_MEM`: This flag applies only to machines with ISA busses or other busses that are software compatible with ISA, such as EISA, MCA, or PCI. It indicates that the memory allocated must be appropriate for DMA access using the system's built-in DMA controller. In particular, it means that the buffer must be physically contiguous, must be entirely contained in the low 16MB of physical memory, and must not cross a 64KB boundary. (By implication, this means that allocations using this flag are limited to at most 64KB in size.) The `OSENV_PHYS_CONTIG` flag must also be set if this flag is set.

`OSENV_X861MB_MEM`: This flag only applies to x86 machines, in which some device drivers may need to call 16-bit real-mode BIOS routines. Such drivers may need to allocate physical memory in the low 1MB region accessible to real-mode code; this flag allows drivers to request such memory. This is not used by existing driver sets.

6.12.2 `osenv_mem_alloc`: allocate memory for use by device drivers

SYNOPSIS

`void *`**osenv_mem_alloc**`(oskit_size_t` *size*, `osenv_memflags_t` *flags*, `unsigned` *align*);

DIRECTION

Component → OS, Blocking

DESCRIPTION

This function is called by the drivers to allocate memory. Allocate the requested amount of memory with the restrictions specified by the *flags* argument as described above.

XXX: While this is defined as blocking, the current glue code cannot yet handle this blocking, as it is not prepared for another request to enter the component. This will be fixed.

PARAMETERS

size: Amount of memory to allocate.

flags: Restrictions on memory.

align: Boundary on which memory should be aligned, which must be a power of two, or 0 which means the same as 1 (no restrictions).

Returns

Returns the address of the allocated block in the driver's virtual address space, or NULL if not enough memory was available.

6.12.3 `osenv_mem_free`: free memory allocated with osenv_mem_alloc

Synopsis

`void` **osenv_mem_free**(`void *`*block*, `osenv_memflags_t` *flags*, `oskit_size_t` *size*);

Direction

Component → OS, Blocking

Description

Frees a memory block previously allocated by `osenv_mem_alloc`.

XXX: While this is defined as blocking, the current glue code cannot yet handle this blocking, as it is not prepared for another request to enter the component. This will be fixed.

Parameters

block: A pointer to the memory block, as returned from `osenv_mem_alloc`.

flags: Flags indicating deallocation semantics required. Only `OSENV_AUTO_SIZE` and `OSENV_NONBLOCKING` are meaningful in this context. `OSENV_AUTO_SIZE` must be set *if and only if* it was set during the allocation, and `OSENV_NONBLOCKING` indicates that the deallocation operation must not block.

size: If *flags* doesn't include `OSENV_AUTO_SIZE`, then this parameter *must* be the size requested when this block was allocated. Otherwise, the value of the *size* parameter is meaningless.

6.12.4 `osenv_mem_get_phys`: find the physical address of an allocated block

Synopsis

`oskit_addr_t` **osenv_mem_get_phys**(`oskit_addr_t` *va*);

Direction

Component → OS, Nonblocking

Description

Returns the physical address associated with a given virtual address. Virtual address should refer to a memory block as returned by `osenv_mem_alloc`. XXX does it have to be the exact same pointer, or just a pointer in the block? In systems which do not support address translation, or for blocks allocated with `OSENV_VIRT_EQ_PHYS`, this function returns *va*.

The returned address is only valid for the first page of the indicated block unless it was allocated with `OSENV_PHYS_CONTIG`. In a system supporting paging, the result of the operation is only guaranteed to be accurate if `OSENV_PHYS_WIRED` was specified when the block was allocated. XXX other constraints?

PARAMETERS

va: The virtual address of a memory block, as returned from `osenv_mem_alloc`.

RETURNS

Returns the PA for the associated (wired) VA. XXX zero (or something else) if VA is not valid?

6.12.5 `osenv_mem_get_virt`: find the virtual address of an allocated block

SYNOPSIS

`oskit_addr_t` **osenv_mem_get_virt**(`oskit_addr_t` *pa*);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

Returns the virtual address of an allocated physical memory block. Can only be called with the physical address of blocks that have been allocated with `osenv_mem_alloc`. XXX or else what?

XXX error codes?

XXX If the Linux glue uses this, and gets and error, should the physical memory be mapped (by the glue) (if it is not in the address space) and re-try?

PARAMETERS

pa: The physical memory location.

RETURNS

Returns the VA for the mapped PA.

6.12.6 `osenv_mem_phys_max`: find the largest physical memory address

SYNOPSIS

`oskit_addr_t` **osenv_mem_phys_max**(`void`);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

Returns the top of physical memory, which is noramlly equivelent to the amount of physical RAM in the machine. Note that memory-mapped devices may reside higher in physical memory, but this is the largest address normal RAM could have.

RETURNS

Returns the amount of physical memory.

6.12.7 `osenv_mem_map_phys`: map physical memory into kernel virtual memory

Synopsis

`int` **osenv_mem_map_phys**(`oskit_addr_t` *pa*, `oskit_size_t` *length*, `void **`*kaddr*, `int` *flags*);

Direction

Component → OS, Blocking

Description

Allocate kernel virtual memory and map the caller supplied physical addresses into it. The address and length must be aligned on a page boundary.

This function is intended to provide device drivers access to memory-mapped devices.

An osenv_mem_unmap_phys interface will likely be added in the future.

XXX: While this is defined as blocking, the current glue code cannot yet handle this blocking, as it is not prepared for another request to enter the component. This will be fixed.

Flags:

`PHYS_MEM_NOCACHE`: Inhibit cache of data in the specified memory.

`PHYS_MEM_WRITETHROUGH`: Data cached from the specified memory must be synchronously written back on writes.

Parameters

pa: Starting physical address.

length: Amount of memory to map.

kaddr: Kernel virtual address allocated and returned by the kernel that maps the specified memory.

flags: Memory mapping attributes, as described above.

Returns

Returns 0 on success, non-zero on error.

6.13 DMA

This section is specific to ISA devices utilizing the Direct Memory Access controller.

If the OS wishes to support devices that utilize DMA, then basic routines must be provided to allow access to the DMA controller.

The Linux drivers directly access the DMA controller themselves, with macros and with embedded assembly. All devices that utilize the DMA controller must be in the same driver set, as there is not way to arbitrate between different driver sets. Because this shortcoming is in the encapsulated drivers, and would take significant effort to correct, we have not provided an interface to access the DMA controller, although we may in the future.

6.13.1 `osenv_isadma_alloc`: Reserve a DMA channel

SYNOPSIS

`int` **osenv_isadma_alloc**(`int` *channel*);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

This requests a DMA channel.

If sucessfull, the driver must be able to directly manipulate the ISA DMA controller.

PARAMETERS

channel: The DMA channel to reserve.

RETURNS

Returns 0 on success, non-zero if already allocated.

6.13.2 `osenv_isadma_free`: Release a DMA channel

SYNOPSIS

`void` **osenv_isadma_free**(`int` *channel*);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

This releases a DMA channel. The DMA channel must have already been reserved by the driver.

PARAMETERS

channel: The DMA channel to release.

6.14 I/O Ports

Many devices have a concept of "I/O space". In general, multiple devices cannot share the same range of I/O ports. Unfortunately, there are a few exceptions, most notably the keyboard and PS/2 mouse, and the Floppy and IDE controllers.

Many of the device drivers assume they may access port 0x80, for use in timing loops. This is not used in most computers, although POST cards are used to display the last value written to that port.

6.14.1 `osenv_io_avail`: Check availability of a range of ports

Synopsis

`oskit_bool_t` **osenv_io_avail**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

Direction

Component → OS, Nonblocking

Description

Returns true (nonzero) if the range is free; false (zero) if any ports in the range are already allocated.

Parameters

port: The start of the I/O range.

size: The number of ports to check.

Returns

Returns 0 (false) if any part of the range is unavailable, non-zero otherwise.

6.14.2 `osenv_io_alloc`: Allocate a range of ports

Synopsis

`oskit_error_t` **osenv_io_alloc**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

Direction

Component → OS, Nonblocking

Description

Returns 0 if the range is free, or an error code if any ports in the range are already allocated.

XXX: shared ports?

XXX: Default implementation panics if range is allocated.

Note: this is based on the assumption that I/O space is not mapped through the MMU. On a system where this is not the case (memory mapped I/O), osenv_mem_map_phys should be used instead.

Parameters

port: The start of the I/O range.

size: The number of ports to check.

6.14.3 `osenv_io_free`: Release a range of ports

SYNOPSIS

`void` **osenv_io_free**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

Releases a range previously allocated. All ports in the range must have been allocated by the device.

PARAMETERS

port: The start of the I/O range.

size: The number of ports to check.

6.15 Hardware Interrupts

Shared interrupts are supported, as long as OSENV_IRQ_SHAREABLE is requested by all devices wishing to use the interrupt.

In a given driver environment in this framework, there are only two "interrupt levels": enabled and disabled. In the default case in which all device drivers of all types are linked together into one large driver environment in an OS kernel, this means that whenever one driver masks interrupts, it masks *all* device interrupts in the system.[1]

However, an OS can implement multiple interrupt priority levels, as in BSD or Windows NT, if it so desires, by creating separate "environments" for different device drivers. For example, if each driver is built into a separate, dynamically-loadable module, then the `osenv_intr_` calls in different driver modules could be resolved by the dynamic loader to `spl`-like routines that switch between different interrupt priority levels. For example, the `osenv_intr_disable` call in network drivers may resolve to `splnet`, whereas the same call in a disk driver may be mapped to `splbio` instead.

6.15.1 `osenv_intr_disable`: prevent interrupts in the driver environment

Synopsis

`void` **osenv_intr_disable**(`void`);

Direction

Component → OS, Nonblocking

Description

Disable further entry into the calling driver set through an interrupt handler. This can be implemented either by directly disabling interrupts at the interrupt controller or CPU, or using some software scheme.

XXX Merely needs to prevent intrs from being dispatched to the driver set. Drivers may see spurious interrupts if they briefly cause interrupts while disabled.

XXX Timing-critical sections need interrupts actually disabled.

6.15.2 `osenv_intr_enable`: allow interrupts in the driver environment

Synopsis

`void` **osenv_intr_enable**(`void`);

Direction

Component → OS, Nonblocking

Description

Enable interrupt delivery to the calling driver set. This can be implemented either by directly enabling interrupts at the interrupt controller or CPU, or using some software scheme.

[1]**Rationale:** The Linux device drivers work this way, and we can't provide more than what we have to work with. This also makes the OS interface simpler, and may allow the basic operations to be faster due to this simplicity.

6.15.3 `osenv_intr_enabled`: determine the current interrupt enable state

SYNOPSIS

`int` **osenv_intr_enabled**(`void`);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

Returns the driver's view of the current interrupt status.

RETURNS

Returns true if interrupts are currently enabled, false otherwise. XXX 1 and 0 instead of true and false?

6.15.4 `osenv_irq_alloc`: allocate an interrupt request line

SYNOPSIS

`int` **osenv_irq_alloc**(`int` *irqnum*, `void` *(*handler)*(void *), void *data, int flags);

DIRECTION

Component → OS, Blocking

DESCRIPTION

Allocate an interrupt request line and attach the specified handler to it. On interrupt, the kernel must pass the *data* argument to the handler.

XXX: interrupts should be "disabled" when the handler is invoked.

XXX: This has not been verified to function correctly if an incomming request is processed while this is blocked.

Flags:

`OSENV_IRQ_SHAREABLE`: If this flag is specified, the interrupt request line can be shared between multiple devices. On interrupt, the OS will call each handler attached to the interrupt line. Without this flag set, the OS is free to return an error if another handler is attached to the interrupt request line. (If shared, ALL handlers must have this set).

PARAMETERS

irqnum: The interrupt request line to allocate.

handler: Interrupt handler.

data: Data passed by the kernel to the interrupt handler.

flags: Flags indicating special behavior. Only OSENV_IRQ_SHAREABLE is currently used.

RETURNS

Returns 0 on success, non-zero on error.

6.15.5 `osenv_irq_free`: Unregister the handler for the interrupt

Synopsis

`void` **osenv_irq_free**(`int` *irqnum*, `void` *(*handler)*(void *), void *data);

Direction

Component → OS, Nonblocking

Description

Removes the indicated interrupt handler. The handler is only removed if it was registered with `osenv_irq_alloc` for the indicated interrupt request line and with the indicated *data* pointer.

Parameters

irq: The physical interrupt line.

handler: The function handler's address. This is necessary if multiple handlers are registered for the same interrupt.

data: The data value registered with `osenv_irq_alloc`.

6.15.6 `osenv_irq_disable`: Disable a single interrupt line

Synopsis

`void` **osenv_irq_disable**(`int` *irq*);

Direction

Component → OS, Nonblocking

Description

Prevents a specific interrupt line from delivering an interrupt. Can be done in software or by disabling at the interrupt controller.

If the interrupt does occur while disabled, it should be delivered as soon as `osenv_irq_enable` is called (see that section for details).

Parameters

irq: The physical interrupt line.

6.15.7 `osenv_irq_enable`: Enable a single interrupt line

Synopsis

`void` **osenv_irq_enable**(`int` *irq*);

Direction

Component → OS, Nonblocking

Description

This allows the specified interrupt to be received, provided interrupts are enabled. (e.g., os-env_intr_enabled also returns true)

Parameters

irq: The physical interrupt line.

6.15.8 `osenv_irq_pending`: Determine if an interrupt is pending for a single line

Synopsis

`int` **osenv_irq_pending**(`int` *irq*);

Direction

Component → OS, Nonblocking

Description

Determine if an interrupt is pending for the specified interrupt line.

Parameters

irq: The physical interrupt line.

Returns

Returns 1 if an interrupt is pending for the indicated line, 0 otherwise.

6.16 Sleep/Wakeup

The current driver model only allow one thread or request into the driver set at a time. However, if the driver set is waiting for an external event and can handle another request while it is waiting, then the driver sleeps.

The default implementation of sleep busy-waits on the event, as it is not possible for it to do more without knowledge of the operating sysmte environment it is in.

6.16.1 `osenv_sleep_init`: prepare to put the current process to sleep

Synopsis

`#include <oskit/dev/dev.h>`

`void` **osenv_sleep_init**(`osenv_sleeprec_t` **sleeprec*);

Direction

Component → OS, Nonblocking

Description

This function initializes a "sleep record" structure in preparation for the current process's going to sleep waiting for some event to occur. The sleep record is used to avoid races between actually going to sleep and the event of interest, and to provide a "handle" on the current activity by which `osenv_wakeup` can indicate which process to awaken.

Parameters

sleeprec: A pointer to the process-private sleep record.

6.16.2 `osenv_sleep`: put the current process to sleep

Synopsis

`#include <oskit/dev/dev.h>`

`int` **osenv_sleep**(`osenv_sleeprec_t` **sleeprec*);

Direction

Component → OS, Blocking

Description

The driver calls this function at process level to put the current activity (process) to sleep until some event occurs, typically triggered by a hardware interrupt or timer handler. The driver must supply a pointer to a process-private "sleep record" variable (*sleeprec*), which is typically just allocated on the stack by the driver. The *sleeprec* must already have been initialized using `osenv_sleep_init`. If the event of interest occurs after the `osenv_sleep_init` but before the `osenv_sleep`, then `osenv_sleep` will return immediately without blocking.

Parameters

sleeprec: A pointer to the process-private sleep record, already allocated by the driver and initialized using `osenv_sleep_init`.

Returns

Returns the wakeup status value provided to `osenv_wakeup`.

6.16.3 `osenv_wakeup`: wake up a sleeping process

Synopsis

`#include <oskit/dev/dev.h>`

`void` **osenv_wakeup**(`osenv_sleeprec_t *`*sleeprec*, `int` *wakeup_status*);

Direction

Component → OS, Nonblocking

Description

The driver calls this function to wake up a process-level activity that has gone to sleep (or is preparing to go to sleep) waiting on some event. The value of `wakeup_status` is subsequently returned to the caller of `osenv_sleep`, making it possible to indicate various wakeup conditions (such as abnormal termination). It is harmless to wake up a process that has already been woken.

Parameters

sleeprec: A pointer to the sleep record of the process to wake up. Must actually point to a valid sleep record variable that has been properly initialized using `osenv_sleep_init`.

wakeup_status: The status to be returned from `osenv_sleep`. OSKIT_WAKEUP indicates normal wakeup, while other status values indicate other conditions.

6.17 Driver-Kernel Interface: Timing

The device support code relies on the OS to provide timers to control events. Unfortunately, timers are in a state of flux, and there are currently too many ways to do almost the same thing. We will be cleaning this up.

Meanwhile... the interface provided by the host OS is currently at the osenv_timer layer. However, we plan on moving the abstraction layer down to a simple "PIT" interface. (The existing osenv_timer_pit code is similar to the planned interface).

When we move to an osenv_pit interface, the driver glue code will use an intermediate timer 'device driver' which will provide the higher-level functionality currently in the osenv_timer interface. The motivation for this is to make the OS-provided interface as simple as possible and to build extra functionality on top.

dev/clock.c is an example device driver built on the osenv_timer interface. It could be implemented on top of an osenv_pit interface as easily as on the osenv_timer interface.

The current implementation of the default osenv_timer code is based on the osenv_timer_pit interface. osenv_timer_pit is *not* currently defined as part of the osenv API, but merely exists for implementation convenience. However, over-riding the osenv_timer_pit implementation is probably the easiest way to provide a different implementation of the osenv_timer interface.

The default osenv_timer implementation also provides an osenv_timer_shutdown hook for use by the host operating system. osenv_timer_shutdown disables the osenv_timer.

6.17.1 `osenv_timer_init`: Initialize the timer support code

Synopsis

`void` **osenv_timer_init**(`void`);

Direction

Component → OS, Nonblocking

Description

XXX: Belongs in libdev.a section

Intiializes the timer code.

6.17.2 `osenv_timer_register`: Request a timer handler be called at the specified frequency

Synopsis

`void` **osenv_timer_register**(`void` *(*func)*(void), int freq);

Direction

Component → OS, Nonblocking

Description

Requests that the function `func` gets called `freq` times per second.

XXX: Default implementation currently only works for `freq` equal to 100.

Parameters

func: Address of function to be called.

freq: Times per second to call the handler.

6.17.3 `osenv_timer_unregister`: Request a timer handler not be called

SYNOPSIS

`void` **osenv_timer_unregister**(`void` *(*func)*(void), int freq);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

The function pointer and frequency must be identically equal to parameters on a previous osenv_timer_register call.

PARAMETERS

func: Address of function to be called.

freq: Times per second the handler was called.

6.17.4 `osenv_timer_spin`: Wait for a specified amount of time without blocking.

SYNOPSIS

`void` **osenv_timer_spin**(`long` *nanosec*);

DIRECTION

Component → OS, Nonblocking

DESCRIPTION

This allows a driver component to block for a specified amount of time (usually for hardware to catch up) without blocking. Unlike with osenv_sleep, this cannot give up the process-level lock.

PARAMETERS

nanosec: Time to spin, in nanoseconds.

6.18 Misc

All output goes throught the osenv_vlog interface.

The following log priorities are defined. From highest priority to lowest, they are: `OSENV_LOG_EMERG`, `OSENV_LOG_ALERT`, `OSENV_LOG_CRIT`, `OSENV_LOG_ERR`, `OSENV_LOG_WARNING`, `OSENV_LOG_NOTICE`, `OSENV_LOG_INFO`, and `OSENV_LOG_DEBUG` which correspond the the log priorities used by both BSD and Linux.

6.18.1 `osenv_vlog`: OS environment's output routine

Synopsis

`void` **osenv_vlog**(`int` *priority*, `const char *`*fmt*, `va_list` *args*);

Direction

Component → OS, Nonblocking

Description

This is the output interface to the device driver framework. All output must go through this interface, so the OS may decide what to do with it.

Normal `printf`-type calls should get converted to the OSENV_LOG_INFO priority.

Parameters

priority:

fmt: `printf`-style message format

args: Any parameters required by the output format

6.18.2 `osenv_log`: OS environment's output routine

Synopsis

`void` **osenv_log**(`int` *priority*, `const char *`*fmt*, `...`);

Direction

Component → OS, Nonblocking

Description

Front-end to osenv_vlog

Parameters

priority: Priority of the message.

fmt: `printf`-style message format

...: Any parameters required by the output format

6.18.3 `osenv_vpanic`: Abort driver set operation

Synopsis

`void` **osenv_vpanic**(`const char *`*fmt*, `va_list` *args*);

Direction

Component → OS, Nonblocking

Description

This function should only be called if the device driver framework can no longer continue and cannot exit gracefully.

The driver's 'native' `panic` calls will get resolved to this function call.

This should be provided by the OS to provide a graceful way of dealing with a situation that prevents the drivers from continuing.

Parameters

fmt: `printf`-style message format

args: Any parameters required by the output format

6.18.4 `osenv_panic`: Abort driver set operation

Synopsis

`void` **osenv_panic**(`const char *`*fmt*, `...`);

Direction

Component → OS, Nonblocking

Description

Front-end to osenv_vpanic

Parameters

fmt: `printf`-style message format

...: Any parameters required by the output format

6.19 Device Registration

Nothing here yet, sorry. See Section 12.2 for a tiny bit more information on our current default implementation of device registration. More information can be gained from the extensively commented header files in the directory `<oskit/dev>`, starting with file `device.h`.

6.20 Block Storage Device Interfaces

This section is incomplete. Block device interfaces now provide an `open` *method which returns a per-open* `blkio` *object through which block reads and writes are done. See Section 5.3. In the absence of other documentation, the example programs will be helpful.*

XXX describe oskit_blkdev, blksize, etc.

6.21 Serial Device Interfaces

XXX: This section is in severe need of an update.

Character device support is provided in the OSKit using device drivers from FreeBSD.

6.22 Driver-Kernel Interface: X86 PC ISA device registration

6.22.1 `osenv_isabus_addchild`: add a device node to an ISA bus

XXX: new device tree management

The *address* parameter is used to uniquely identify the device on the ISA bus. For example, if there are two identical NE2000 cards plugged into the machine, the *address* will be be the only way the host OS can distinguish them, because all of the other parameters of the device will be identical. If *address* is in the range 0–0xffff (0–65535), it is interpreted as a port number in I/O space; otherwise, it is interpreted as a physical memory address. For devices that use any I/O ports for communication with software, the base of the "primary" range of I/O ports used by the device should be used as the *address*; a physical memory address should be used only for devices that *only* communicate through memory-mapped I/O.

6.22.2 `osenv_isabus_remchild`: remove a device node from an ISA bus

Chapter 7

OSKit File System Framework

7.1 Introduction

The OSKit file system framework has parallel goals to the OSKit device driver framework; the framework provides a file system interface specification designed to allow existing filesystem implementations to be borrowed from well-established operating systems in source form and used mostly unchanged to provide file system support in new operating systems. The framework is also designed to allow file systems to be implemented in diverse ways and then composed together.

The OSKit file system framework encompasses a collection of COM interfaces used by the client operating system to invoke the file system libraries, and a collection of interfaces used by the file system libraries to request services from the client operating system. The individual file system libraries supply additional interfaces to the client operating system for initialization, and may supply additional interfaces for supporting extended features unique to particular file system implementations.

The OSKit File, Directory and Open File COM interfaces inherit from several general COM interfaces, such as Stream, Absolute IO and POSIX IO. The inheritance relationships among these COM interfaces are shown in Figure 7.1. Refer to Section 4 for more details on COM interfaces.

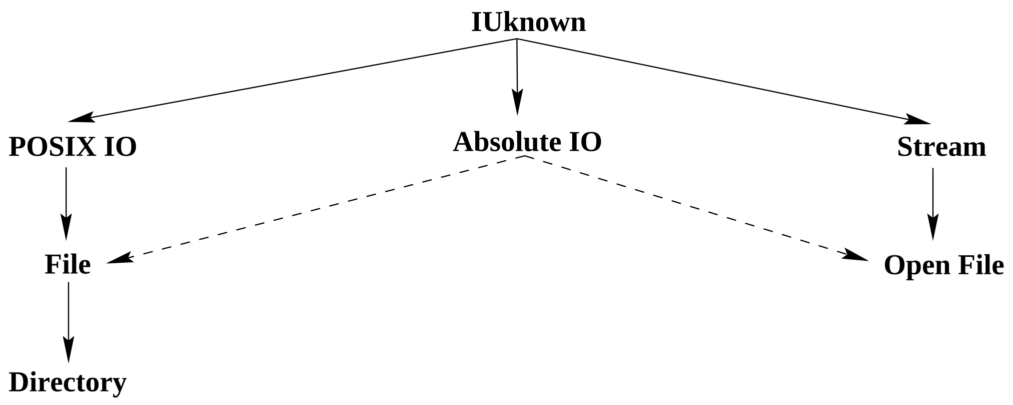

Figure 7.1: Interface hierarchy. Solid lines indicate that the child interface directly inherits the methods of the parent interface. Dashed lines indicate that the child interface may optionally support the parent interface; this may be determined by querying the object.

7.2 `oskit_principal`: Principal Interface

The `oskit_principal` COM interface defines an interface for obtaining identity information about a principal (aka subject or client). The filesystem libraries obtain an `oskit_principal` object for the current client by invoking `oskit_get_call_context` on `oskit_principal_iid`.

The `oskit_principal` COM interface inherits from `IUnknown`, and has one additional method:

`getid`: Obtain identity attributes of principal.

7.2.1 `getid`: Get the identity attributes of this principal

Synopsis

`#include <oskit/principal.h>`

`oskit_error_t` **oskit_principal_getid**(`oskit_principal_t` **p*, [out] `oskit_identity_t` **out_id*);

Direction

filesystem library → client OS

Description

This method returns the identity attributes of this principal. `out_id` is a pointer to an `oskit_identity_t` structure defined as follows:

```
struct oskit_identity {
    oskit_uid_t   uid;       /* effective user id      */
    oskit_gid_t   gid;       /* effective group id     */
    oskit_u32_t   ngroups;   /* number of groups       */
    oskit_u32_t   *groups;   /* supplemental groups    */
};
```

Parameters

p: The principal whose identity is desired.

out_id: The identity attributes of this principal.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.3 oskit_filesystem: File System Interface

The oskit_filesystem COM interface defines an interface for operating on a filesystem, which is a logical collection of files and a tree-structured namespace with a single root. The filesystem itself exists independent of any given namespace, for there is no notion of mounts in this interface. That functionality must be implemented at a higher level.

The oskit_filesystem COM interface inherits from IUnknown, and has the following additional methods:

statfs: Get attributes of this filesystem

sync: Write this filesystem's data to permanent storage

getroot: Get this filesystem's root directory

remount: Update this filesystem's mount flags

unmount: Forcibly unmount this filesystem

lookupi: Lookup a file by inode number

7.3.1 statfs: Get attributes of this filesystem

Synopsis

```
#include <oskit/fs/filesystem.h>
oskit_error_t oskit_filesystem_statfs(oskit_filesystem_t *f, [out] oskit_statfs_t *out_stats);
```

Direction

client OS → filesystem library

Description

This method returns the attributes of this filesystem. out_stats is a pointer to an oskit_statfs_t structure defined as follows:

```
struct oskit_statfs {
    oskit_u32_t  bsize;     /* file system block size                  */
    oskit_u32_t  frsize;    /* fundamental file system block size      */
    oskit_u32_t  blocks;    /* total blocks in fs in units of frsize   */
    oskit_u32_t  bfree;     /* free blocks in fs                       */
    oskit_u32_t  bavail;    /* free blocks avail to non-superuser      */
    oskit_u32_t  files;     /* total file nodes in file system         */
    oskit_u32_t  ffree;     /* free file nodes in fs                   */
    oskit_u32_t  favail;    /* free file nodes avail to non-superuser  */
    oskit_u32_t  fsid;      /* file system id                          */
    oskit_u32_t  flag;      /* mount flags                             */
    oskit_u32_t  namemax;   /* max bytes in a file name                */
};
```

Parameters

f: The filesystem whose attributes are desired.

out_stats: The attributes of the specified filesystem.

Returns

Returns 0 on success, or an error code specified in <oskit/error.h>, on error.

7.3.2 `sync`: Synchronize in-core filesystem data with permanent storage

Synopsis

`#include <oskit/fs/filesystem.h>`

`oskit_error_t` **oskit_filesystem_sync**(`oskit_filesystem_t` **f*, `oskit_bool_t` *wait*);

Direction

client OS → filesystem library

Description

This method writes all of this filesystem's data back to permanent storage. If `wait` is `TRUE`, then the call does not return until all pending data has been completely written.

Parameters

f: The filesystem to sync.

wait: `TRUE` if the call should wait for completion.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.3.3 `getroot`: Return a reference to the root directory of this filesystem

Synopsis

`#include <oskit/fs/filesystem.h>`

`oskit_error_t` **oskit_filesystem_getroot**(`oskit_filesystem_t` **f*, [out] `oskit_dir_t` ***out_dir*);

Direction

client OS → filesystem library

Description

This method returns a reference to the root directory of this filesystem. `out_dir` is a pointer to the `oskit_dir` COM interface for the root directory.

Parameters

f: The filesystem whose root directory is desired.

out_dir: The `oskit_dir` COM interface for the root directory.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.3.4 `remount`: Update the mount flags of this filesystem

Synopsis

`#include <oskit/fs/filesystem.h>`

`oskit_error_t` **oskit_filesystem_remount**(`oskit_filesystem_t` **f*, `oskit_u32_t` *flags*);

Direction

client OS → filesystem library

Description

This method changes the mount flags associated with this filesystem. For example, this method might be used to change a filesystem from read-only to read-write, or vice versa.

Parameters

f: The filesystem whose flags are to be changed.

flags: The new mount flags value.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.3.5 `unmount`: Forcibly unmount this filesystem

Synopsis

`#include <oskit/fs/filesystem.h>`

`oskit_error_t` **oskit_filesystem_unmount**(`oskit_filesystem_t` **f*);

Direction

client OS → filesystem library

Description

This method forcibly unmounts this filesystem. Ordinarily, a filesystem is unmounted when the last reference to it is released; in contrast, this method forces an unmount regardless of external references to the filesystem, and is consequently unsafe. Subsequent attempts to use references to the filesystem or to use references to files within the filesystem may yield undefined results.

Parameters

f: The filesystem to be forcibly unmounted.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.3.6 `lookupi`: Lookup a file by inode number

Synopsis

`#include <oskit/fs/filesystem.h>`

`oskit_error_t` **oskit_filesystem_lookupi**(`oskit_filesystem_t` **f*, `oskit_ino_t` *ino*, [out] `oskit_file_t` ***out_file*);

Direction

client OS → filesystem library

Description

This method looks up a file given its inode number. If the inode number is invalid, the behavior is undefined.

Parameters

f: The filesystem to find the inode in.

ino: The inode number of the file to find.

out_file: Upon success, will point to a oskit_file_t for the file.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.4 oskit_file: File Interface

The `oskit_file` COM interface defines an interface for operating on a file. The interface does not imply any per-open state; per-open methods are defined by the `oskit_openfile` COM interface.

The `oskit_file` COM interface inherits from the `oskit_posixio` COM interface, and has the following additional methods:

`sync`: Write this file's data and metadata to permanent storage.

`datasync`: Write this file's data to permanent storage.

`access`: Check accessibility of this file.

`readlink`: Read the contents of this symbolic link.

`open`: Create an open instance of this file.

`getfs`: Get the filesystem in which this file resides.

Additionally, an `oskit_file` object may export a `oskit_absio` COM interface; this may be determined by querying the object.

7.4.1 sync: Write this file's data and metadata to permanent storage

Synopsis

`#include <oskit/fs/file.h>`

`oskit_error_t` **oskit_file_sync**(`oskit_file_t` **f*, `oskit_bool_t` *wait*);

Direction

client OS → filesystem library

Description

This method synchronizes the in-core copy of this file's data and metadata with the on-disk copy. If `wait` is `TRUE`, then the call does not return until all pending data has been completely written.

Parameters

f: The file to sync.

wait: `TRUE` if the call should wait for completion.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.4.2 datasync: Write this file's data to permanent storage

Synopsis

`#include <oskit/fs/file.h>`

`oskit_error_t` **oskit_file_datasync**(`oskit_file_t` **f*, `oskit_bool_t` *wait*);

Direction

client OS → filesystem library

Description

This method synchronizes the in-core copy of this file's data with the on-disk copy. The file metadata need not be sychronized by this method. If `wait` is `TRUE`, then the call does not return until all pending data has been completely written.

Parameters

f: The file to sync.

wait: `TRUE` if the call should wait for completion.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.4.3 `access`: Check accessibility of this file

Synopsis

`#include <oskit/fs/file.h>`

`oskit_error_t` **oskit_file_access**(`oskit_file_t` **f*, `oskit_amode_t` *mask*);

Direction

client OS → filesystem library

Description

This method checks whether the form of access specified by `mask` would be granted. `mask` may be any combination of `OSKIT_R_OK` (read access), `OSKIT_W_OK` (write access), or `OSKIT_X_OK` (execute access). If the access would not be granted, then this method will return the error that would be returned if the actual access were attempted.

Parameters

f: The file whose accessibility is to be checked.

mask: The access mask.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.4.4 `readlink`: Read the contents of this symbolic link

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_file_readlink**(`oskit_file_t` **f*, `char` **buf*, `oskit_u32_t` *len*, [out] `oskit_u32_t` **out_actual*);

Direction

client OS → filesystem library

DESCRIPTION

If this file is a symbolic link, then this method reads the contents of the symbolic link into `buf`. No more than `len` bytes will be read. `out_actual` will be set to the actual number of bytes read.

PARAMETERS

f: The symbolic link file.

buf: The buffer into which the contents are to be copied.

len: The maximum number of bytes to read.

out_actual: The actual bytes read from the symlink.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

If the file is not a symbolic link, then `OSKIT_E_NOTIMPL` is returned.

7.4.5 `open`: Create an open instance of this file

SYNOPSIS

```
#include <oskit/fs/file.h>
#include <oskit/fs/openfile.h>
```

`oskit_error_t` **oskit_file_open**(`oskit_file_t` **f*, `oskit_oflags_t` *flags*, [out] `oskit_openfile_t` `**`*out_openfile*);

DIRECTION

client OS $\rightarrow$ filesystem library

DESCRIPTION

This method returns an `oskit_openfile` COM interface for an open instance of this file. `flags` specifies the file open flags, as defined in `<oskit/fs/file.h>`. If `OSKIT_O_TRUNC` is specified, then the file will be truncated to zero length.

This method may only be used on regular files and directories. Directories may not be opened with `OSKIT_O_WRONLY`, `OSKIT_O_RDWR` or `OSKIT_O_TRUNC`.

This method may return success but set `*out_openfile` to `NULL`, indicating that the requested operation is allowed but the filesystem does not support per-open state; the client operating system must provide this functionality.

PARAMETERS

f: The file to open.

flags: The open flags.

out_openfile: The `oskit_openfile` COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.4.6 `getfs`: Get the filesystem in which this file resides

Synopsis

`#include <oskit/fs/file.h>`

`oskit_error_t` **oskit_file_getfs**(`oskit_file_t` **f*, [out] `oskit_filesystem_t` ***out_fs*);

Direction

client OS → filesystem library

Description

Returns the `oskit_filesystem` COM interface for the filesystem in which this file resides.

Parameters

f: The file whose filesystem is desired.

out_fs: The filesystem in which the file resides.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5 `oskit_dir`: Directory Interface

The `oskit_dir` COM interface defines an interface for operating on a directory. The interface does not imply any per-open state; per-open methods are defined by the `oskit_openfile` COM interface.

The `oskit_dir` COM interface inherits from the `oskit_file` COM interface, and has the following additional methods:

`lookup`: Lookup a file in this directory.

`create`: Create a regular file in this directory.

`link`: Link a file into this directory.

`unlink`: Unlink a file from this directory.

`rename`: Rename a file from this directory.

`mkdir`: Create a directory in this directory.

`rmdir`: Remove a directory from this directory.

`getdirentries`: Read entries from this directory.

`mknod`: Create a special file in this directory.

`symlink`: Create a symlink in this directory.

`reparent`: Create a virtual directory from this directory.

Additionally, an `oskit_dir` object may export a `oskit_absio` COM interface; this may be determined by querying the object.

All name parameters to directory methods must be a single component, ie an entry in one of the specified directories. With the exception of `rename`, name parameters always refer to entries in the target directory itself.

7.5.1 `lookup`: Look up a file in this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_lookup**(`oskit_dir_t *`*d*, `const char *`*name*, [out] `oskit_file_t **`*out_file*);

Direction

client OS → filesystem library

Description

This method returns the `oskit_file` COM interface for the file named by `name` in this directory. The `name` may only be a single component; multi-component lookups are not supported. If the file is a symbolic link, then `out_file` will reference the symbolic link itself.

Parameters

d: The directory to search.

name: The name of the file.

out_file: The `oskit_file` COM interface for the file.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.2 `create`: Create a regular file in this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_create**(`oskit_dir_t *`*d*, `const char *`*name*, `oskit_bool_t` *excl*, `oskit_mode_t` *mode*, [out] `oskit_file_t **`*out_file*);

Direction

client OS → filesystem library

Description

This method is the same as `oskit_dir_lookup`, except that if the file does not exist, then a regular file will be created with the specified `name` and `mode`.

If a file with `name` already exists, and `excl` is `TRUE`, then `OSKIT_EEXIST` will be returned.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

d: The directory to search.

name: The name of the file.

excl: `TRUE` if an error should be returned if the file exists

mode: The file mode to use if creating a new file.

out_file: The `oskit_file` COM interface for the file.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.3 `link`: Link a file into this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_link**(`oskit_dir_t *`*d*, `const char *`*name*, `oskit_file_t *`*file*);

Direction

client OS → filesystem library

Description

This method adds an entry for `file` into this directory, using `name` for the new directory entry. Typically, this is only supported if `file` resides in the same filesystem as `d`.

`file` may not be a symbolic link.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

d: The directory to search.

name: The name for the new link.

file: The file to be linked.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.4 `unlink`: Unlink a file from this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_unlink**(`oskit_dir_t *`*d*, `const char *`*name*);

Direction

client OS → filesystem library

Description

This method removes the directory entry for `name` from `d`.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

d: The directory to search.

name: The name of the file to be unlinked.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.5 `rename`: Rename a file from this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_rename**(`oskit_dir_t *`*old_dir*, `const char *`*old_name*, `oskit_dir_t *`*new_dir*, `const char *`*new_name*);

Direction

client OS → filesystem library

Description

This method atomically links the file named by `old_name` in `old_dir` into `new_dir`, using `new_name` for the new directory entry, and unlinks `old_name` from `old_dir`.

If a file named `new_name` already exists in `new_dir`, then it is first removed. In this case, the source and target files must either both be directories or both be non-directories, and if the target file is a directory, it must be empty.

Typically, this is only supported if `new_dir` resides in the same filesystem as `old_dir`.

The `old_name` and `new_name` may each only be a single component; multi-component lookups are not supported.

Parameters

old_dir: This directory.

old_name: The name of the file to be renamed.

new_dir: The target directory.

new_name: The name for the new directory entry.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.6 `mkdir`: Create a subdirectory in this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_mkdir**(`oskit_dir_t` **d*, `const char` **name*, `oskit_mode_t` *mode*);

Direction

client OS → filesystem library

Description

This method creates a new subdirectory in this directory, with the specified `name` and `mode`.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

dir: The directory in which to create the subdirectory.

name: The name of the new subdirectory.

mode: The mode for the new subdirectory.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.7 `rmdir`: Remove a subdirectory from this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_rmdir**(`oskit_dir_t` **d*, `const char` **name*);

Direction

client OS → filesystem library

Description

This method removes the subdirectory named `name` from this directory. Typically, this is only supported if the subdirectory is empty.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

dir: The directory in which the subdirectory resides.

name: The name of the subdirectory.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.8 getdirentries: Read one or more entries from this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_getdirentries**(`oskit_dir_t` **d*, `oskit_u32_t` **inout_ofs*, `oskit_u32_t` *nentries*, [out] `oskit_dirents_t` **out_dirents*);

Direction

client OS → filesystem library

Description

This method reads one or more entries from this directory. On entry, `inout_ofs` contains the offset of the first entry to be read. Before returning, this method updates the value at `inout_ofs` to contain the offset of the next entry after the last entry returned in `out_dirents`. The returned value of `inout_ofs` is opaque; it should only be used in subsequent calls to this method.

This method will return at least `nentries` entries if there are at least that many entries remaining in the directory; however, this method may return more entries.

The client operating system must free the contents of `out_dirents` when they are no longer needed.

`out_dirents` is a pointer to an `oskit_dirents_t` structure defined as follows:

```
struct oskit_dirent {
    char          *name;   /* entry name    */
    oskit_ino_t   ino;     /* entry inode   */
};
struct oskit_dirents {
    struct        oskit_dirent *dirents;   /* array of entries    */
    oskit_u32_t   count;                   /* number of entries   */
};
```

Parameters

d: The directory to read.

inout_ofs: On entry, the offset of the first entry to read. On exit, the offset of the next entry to read.

nentries: The minimum desired number of entries.

out_dirents: The directory entries.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.9 `mknod`: Create a special file node in this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_mknod**(`oskit_dir_t *`*d*, `const char *`*name*, `oskit_mode_t` *mode*, `oskit_dev_t` *dev*);

Direction

client OS → filesystem library

Description

This method creates a device special file in this directory, with the specified `name` and file `mode`, and with the specified device number `dev`. The device number is opaque to the filesystem library.

The `name` may only be a single component; multi-component lookups are not supported.

Parameters

d: The directory in which to create the node

name: The name of the new node.

mode: The mode for the new node.

dev: The device number for the new node.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.10 `symlink`: Create a symbolic link in this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_symlink**(`oskit_dir_t *`*d*, `const char *`*link_name*, `char *`*dest_name*);

Direction

client OS → filesystem library

Description

This method creates a symbolic link in this directory, named `link_name`, with contents `dest_name`.

The `link_name` may only be a single component; multi-component lookups are not supported.

Parameters

d: The directory in which to create the symlink.

link_name: The name of the new symlink.

dest_name: The contents of the new symlink.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.5.11 `reparent`: Create a virtual directory from this directory

Synopsis

`#include <oskit/fs/dir.h>`

`oskit_error_t` **oskit_dir_reparent**(`oskit_dir_t *`*d*, `oskit_dir_t *`*parent*, [out] `oskit_dir_t **`*out_dir*);

Direction

client OS → filesystem library

Description

This method creates a virtual directory `out_dir` which refers to the same underlying directory as `d`, but whose logical parent directory is `parent`. If `parent` is `NULL`, then the logical parent directory of `out_dir` will be itself.

Lookups of the parent directory entry ('..') in the virtual directory will return a reference to the logical parent directory.

This method may be used to provide equivalent functionality to the Unix `chroot` operation.

Parameters

d: The directory

parent: The logical parent directory

out_dir: The `oskit_dir` COM interface for the virtual directory

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.6 `oskit_openfile`: Open File Interface

The `oskit_openfile` COM interface defines an interface for operating on an open instance of a file.

The `oskit_openfile` COM interface inherits from the `oskit_stream` COM interface, and has the following additional method:

`getfile`: Get the underlying file object to which this open file refers.

Additionally, an `oskit_openfile` object may export a `oskit_absio` COM interface; this may be determined by querying the object.

7.6.1 `getfile`: Get the underlying file object to which this open file refers

Synopsis

`#include <oskit/fs/openfile.h>`

`oskit_error_t` **oskit_openfile_getfile**(`oskit_openfile_t` **f*, [out] `oskit_file_t` ***out_file*);

Direction

client OS → filesystem library

Description

This method returns the `oskit_file` COM interface for the underlying file object to which this open file refers.

Parameters

f: The open file whose underlying file is desired.

out_file: The `oskit_file` COM interface for the underlying file.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.7 Dependencies on the Client Operating System

This section describes the interfaces which must be provided by the client operating system to the filesystem library.

These interfaces consist of:

`oskit_get_call_context`: Obtain information about client context.

`fs_delay`: Wait for a period of time to elapse.

`fs_vprintf`: Generate formatted output to stdout.

`fs_vsprintf`: Generate formatted output to a string.

`fs_panic`: Perform any cleanup and exit.

`fs_gettime`: Obtain the current time.

`fs_tsleep`: Wait for a wakeup on a channel or a timeout.

`fs_wakeup`: Awaken any threads waiting on a channel.

`fs_malloc`: Allocate memory.

`fs_realloc`: Resize a chunk of allocated memory.

`fs_free`: Free memory.

Default implementations of the `fs_*` functions are provided in `liboskit_fs`.

7.7.1 `oskit_get_call_context`: Get the caller's context

Synopsis

`#include <oskit/com.h>`

`oskit_error_t` **oskit_get_call_context**(`oskit_guid_t *`*iid*, [out] `void **`*out_if*);

Direction

filesystem library → client OS

Description

This function returns the requested COM interface for the current caller.

Typically, this is used to obtain the `oskit_principal` object for the current client of the filesystem library.

Parameters

iid: The desired COM interface identifier.

out_if: The COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.7.2 `fs_delay`: Wait for a period of time to elapse

Synopsis

`#include <oskit/fs/fs.h>`

`void` **fs_delay**(`oskit_u32_t` *n*);

Direction

filesystem library → client OS

Description

Wait for `n` microseconds to elapse.

Parameters

n: The number of microseconds to delay.

7.7.3 `fs_vprintf`: Generate formatted output to stdout

Synopsis

`#include <oskit/fs/fs.h>`

`void` **fs_vprintf**(`char *`*fmt*, `__oskit_va_list` *ap*);

Direction

filesystem library → client OS

Description

This function generates formatted output to `stdout`. The client operating system defines what is meant by `stdout`; the output may be displayed on the console, directed to a log, returned to the current client, etc.

Parameters

fmt: The format for the output.

ap: The list of arguments for the output.

7.7.4 `fs_vsprintf`: Generate formatted output to a string

Synopsis

`#include <oskit/fs/fs.h>`

`void` **fs_vsprintf**(`char *`*s*, `char *`*fmt*, `__oskit_va_list` *ap*);

Direction

filesystem library → client OS

Description

This function generates formatted output to a string.

PARAMETERS

s: The string into which to copy the output.

fmt: The format for the output.

ap: The list of arguments for the output.

7.7.5 `fs_panic`: Cleanup and exit

SYNOPSIS

`#include <oskit/fs/fs.h>`

`void` **fs_panic**(`void`);

DIRECTION

filesystem library → client OS

DESCRIPTION

This function cleans up and exits. If this function returns rather than exiting, the filesystem library may be very unhappy.

7.7.6 `fs_gettime`: Get the current time

SYNOPSIS

`#include <oskit/fs/fs.h>`

`oskit_error_t` **fs_gettime**([out] `oskit_timespec_t` **out_tsp*);

DIRECTION

filesystem library → client OS

DESCRIPTION

This function returns the current time. `out_tsp` is a pointer to an `oskit_timespec_t` structure, defined as follows:

```
struct oskit_timespec {
    oskit_time_t  tv_sec;   /* seconds          */
    oskit_s32_t   tv_nsec;  /* and nanoseconds  */
};
```

PARAMETERS

out_tsp: The current time.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.7.7 `fs_tsleep`: Wait for a wakeup on a channel or for a timeout

SYNOPSIS

`#include <oskit/fs/fs.h>`

`oskit_error_t` **fs_tsleep**(`void *`*chan*, `oskit_u32_t` *pri*, `char *`*wmesg*, `oskit_u32_t` *timo*);

DIRECTION

filesystem library → client OS

DESCRIPTION

This function waits for a wakeup to occur on sleep channel `chan` or until `timo` microseconds have elapsed. Upon waking up, the thread is assigned priority `pri`. If `timo` is zero, then this function waits until a wakeup occurs.

A sleep channel is simply an identifier for an event; typically, `chan` will be the address of some variable used by the filesystem library.

PARAMETERS

chan: The sleep channel.

pri: The priority upon wakeup.

wmesg: A description of the sleep state.

timo: The timeout for the sleep.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

7.7.8 `fs_wakeup`: Wakeup any threads waiting on this channel

SYNOPSIS

`#include <oskit/fs/fs.h>`

`void` **fs_wakeup**(`void *`*chan*);

DIRECTION

filesystem library → client OS

DESCRIPTION

This function awakens any threads waiting on this sleep channel.

A sleep channel is simply an identifier for an event; typically, `chan` will be the address of some variable used by the filesystem library.

PARAMETERS

chan: The sleep channel.

7.7.9 `fs_malloc`: Allocate memory from the heap

Synopsis

`#include <oskit/fs/fs.h>`

`void *`**fs_malloc**(`oskit_u32_t` *size*);

Direction

filesystem library → client OS

Description

This function returns a pointer to a chunk of contiguous memory of at least `size` bytes. The client operating system need not provide any alignment guarantees for the chunk of memory.

Parameters

size: The desired number of bytes.

Returns

Returns a pointer to the chunk of memory, or NULL on error.

7.7.10 `fs_realloc`: Resize a chunk of allocated memory

Synopsis

`#include <oskit/fs/fs.h>`

`void *`**fs_realloc**(`void *`*curaddr*, `oskit_u32_t` *newsize*);

Direction

filesystem library → client OS

Description

This function resizes the chunk of allocated memory referenced by `curaddr` to be at least `newsize` bytes, and returns a pointer to the new chunk. If successful, the old chunk is freed.

The client operating system need not provide any alignment guarantees for the new chunk of memory.

Parameters

curaddr: The address of the old chunk.

newsize: The new size in bytes.

Returns

Returns a pointer to the new chunk of memory, or NULL on error.

7.7.11 `fs_free`: Free a chunk of allocated memory

SYNOPSIS

`#include <oskit/fs/fs.h>`

`void` **fs_free**(`void *`*addr*);

DIRECTION

filesystem library → client OS

DESCRIPTION

This function frees a chunk of allocated memory referenced by `addr`. `addr` must be an address returned previously by a call to `fs_malloc` or `fs_realloc`.

PARAMETERS

addr: The address of the chunk.

Chapter 8

OSKit Networking Framework

8.1 Introduction

The OSKit networking framework encompasses a collection of COM interfaces used by the client operating system to invoke the networking libraries. The individual networking libraries supply additional interfaces to the client operating system for initialization, and may supply additional interfaces for supporting extended features unique to particular networking protocol implementations.

At this point, we have only one interface, the oskit_socket interface, defined. Additional interfaces for configuration, routing, etc., are future work.

8.2 `oskit_socket`: Socket Interface

The `oskit_socket` COM interface defines an interface which capture the semantics of a socket as defined in the corresponding POSIX/CAE standards. The `oskit_socket` COM interface inherits from `oskit_posixio`. It can be queried for an `oskit_stream` interface. This query will always be successful, but the resulting `oskit_stream` instance might not support all methods. Generally, at least `read` and `write` will be supported. The `oskit_socket` COM interface provides in addition to the `oskit_posixio` COM interface the following methods:

`accept`: accept a connection on a socket

`bind`: bind a name to a socket

`connect`: initiate a connection on a socket

`shutdown`: shut down part of a full-duplex connection

`listen`: listen for connections on a socket

`getsockname`: get socket name

`getpeername`: get name of connected peer

`getsockopt`: get options on sockets

`setsockopt`: set options on sockets

`sendto`: send a message from a socket

`recvfrom`: receive a message from a socket

`sendmsg`: send a message from a socket

`recvmsg`: receive a message from a socket

Note that these methods are not minimal, but correspond very closely to the traditional BSD interfaces.

> **Note:** the following paragraphs have a certain likelihood to change. The main reason for this is the obviously undesirable connection between the way socket factories and the socket interface interact. On a more positive note, everything right now is so close to the BSD interfaces that the reader familiar with those shouldn't have any problems understanding these.

8.2.1 `oskit_socket_factory_t`: socket factories

SYNOPSIS

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_factory_create**(*oskit_socket_factory_t *factory*, `oskit_u32_t` *domain*, `oskit_u32_t` *type*, `oskit_u32_t` *protocol*, [out] `oskit_socket_t **`*newsocket*);

DESCRIPTION

Socket instances are created by *socket factories*.

A socket factory is an instance of the `oskit_socket_factory` COM interface. Implementations of this interface will be provided by the networking stack(s) included in the OSKit. This interface implements a single method corresponding to the `socket(2)` call in addition to the `oskit_iunknown` interface.

Each instance of socket has a type and a protocol associated with it. This type and protocol is given to the socket by its factory, and cannot be changed during the lifetime of that socket instance.

Parameters

factory: The socket factory used to create this socket.

domain: The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. Some common formats are

`OSKIT_PF_LOCAL`	Host-internal protocols
`OSKIT_PF_INET`	DARPA Internet protocols
`OSKIT_PF_ISO`	ISO protocols
`OSKIT_PF_CCITT`	ITU-T protocols, like X.25
`OSKIT_PF_NS`	Xerox Network Systems protocols

`OSKIT_PF_INET` is the only format for which the OSKit currently contains an implementation.

type: The socket will have the indicated *type*, which specifies the semantics of communication. Currently defined types are

`OSKIT_SOCK_STREAM`	stream socket
`OSKIT_SOCK_DGRAM`	datagram socket
`OSKIT_SOCK_RAW`	raw-protocol interface
`OSKIT_SOCK_RDM`	reliably-delivered message
`OSKIT_SOCK_SEQPACKET`	sequenced packet stream

An `OSKIT_SOCK_STREAM` type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. An `OSKIT_SOCK_DGRAM` socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). An `OSKIT_SOCK_SEQPACKET` socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length. `OSKIT_SOCK_RAW` sockets provide access to internal network protocols and interfaces.

protocol: The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified. The protocol number to use is particular to the communication domain in which communication is to take place.

Protocols for the `OSKIT_PF_INET` protocol family are defined in `oskit/c/netinet/in.h`.

newsocket: The new `oskit_socket_t` instance that was created.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.2 `accept`: accept a connection on a socket

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_accept**(`oskit_socket_t *`*s*, [out] `struct` *oskit_sockaddr *name*, [in/out] `oskit_size_t *`*anamelen*, [out] `struct` *oskit_socket **newopenso*);

Description

The `accept` method extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of s and returns it. The socket must have been bound to an address with `bind` and it must be listening for connections after a `listen`.

If no pending connections are present on the queue, `accept` blocks the caller until a connection is present.

PARAMETERS

s: The socket from which connections are to accepted.

name: Filled with the address of the connecting entity as known to the communication layer.

anamelen: Initially, the amount of space pointed to by name, on return it will contain the amount actually used.

newopenso: Newly created socket.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.3 `bind`: bind a name to a socket

SYNOPSIS

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_bind**(`oskit_socket_t` **s*, `const` *struct oskit_sockaddr *name*, `oskit_size_t` *namelen*);

DESCRIPTION

`bind` assigns a name to an unnamed socket. When a socket is created, it exists in a name space (address family) but has no name assigned. `bind` requests that *name* be assigned to the socket.

PARAMETERS

s: The socket to which a name is to be bound.

name: The name to which the socket is to be bound.

namelen: The length of *name* in bytes.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.4 `connect`: initiate a connection on a socket

SYNOPSIS

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_connect**(`oskit_socket_t` **s*, `const` *struct oskit_sockaddr *name*, `oskit_size_t` *namelen*);

DESCRIPTION

If *s* is of type `OSKIT_SOCK_DGRAM`, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type `OSKIT_SOCK_STREAM`, this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully `connect` only once; datagram sockets may use `connect` multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Parameters

s: The socket from which the connection is to be initiated.

name: The address of the entity to which the connection is to be established.

namelen: The length of *name* in bytes.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.5 `shutdown`: shut down part of a full-duplex connection

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_shutdown**(`oskit_socket_t` **s*, `oskit_u32_t` *how*);

Description

The `shutdown` call causes all or part of a full-duplex connection on the socket *s* to be shut down.

Parameters

s: The socket which is to be shut down.

how: Specifies what is to be disallowed:

how = 0	receives
how = 1	sends
how = 2	sends and receives

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.6 `listen`: listen for connections on a socket

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_listen**(`oskit_socket_t` **s*, `oskit_u32_t` *backlog*);

Description

A willingness to accept incoming connections and a queue limit for incoming connections are specified with `listen`, and then the connections are accepted with `accept`. The `listen` call applies only to sockets of type `OSKIT_SOCK_STREAM` or `OSKIT_SOCK_SEQPACKET`.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of connection refused, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

Parameters

s: The socket where connections will be accepted.

backlog: Maximum number of pending connections.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.7 `getsockname`: get socket name

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_getsockname**(`oskit_socket_t` **s*, [out] `struct` *oskit_sockaddr *asa*, [in/out] `oskit_size_t` **anamelen*);

Description

`getsockname` returns the current name for the specified socket.

Parameters

s: The socket whose name is to be determined.

name: Contains the name of the socket upon return.

anamelen: Initially, the amount of space pointed to by name, on return it will contain the amount actually used, i.e., the actual size of the name.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.8 `getpeername`: get name of connected peer

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_getpeername**(`oskit_socket_t` **s*, [out] `struct` *oskit_sockaddr *asa*, [in/out] `oskit_size_t` **anamelen*);

Description

`getpeername` returns the name of the peer connected to socket *s*.

Parameters

s: The socket connected to the peer whose name is to be returned.

name: Contains the peer's name upon return.

anamelen: Initially, the amount of space pointed to by name, on return it will contain the amount actually used. The name is truncated if the buffer provided is too small.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.9 getsockopt, setsockopt: get and set options on sockets

SYNOPSIS

#include <oskit/net/socket.h>

oskit_error_t **oskit_socket_getsockopt**(oskit_socket_t **s*, oskit_u32_t *level*, oskit_u32_t *name*, [out] void **val*, [in/out] oskit_size_t **valsize*);

oskit_error_t **oskit_socket_setsockopt**(oskit_socket_t **s*, oskit_u32_t *level*, oskit_u32_t *name*, const *void *val*, oskit_size_t *valsize*);

DESCRIPTION

getsockopt and setsockopt manipulate the options associated with a socket. Options may exist at multiple protocol levels.

PARAMETERS

s: The socket whose options are to be queried or set.

level: When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as OSKIT_SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to IPPROTO_TCP.

name: *name* and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. Definitions for socket level options are described below. Options at other protocol levels vary in format and name.

Most socket-level options utilize an int parameter for *val*. For setsockopt, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. OSKIT_SO_LINGER uses a struct oskit_linger parameter, which specifies the desired state of the option and the linger interval (see below). OSKIT_SO_SNDTIMEO and OSKIT_SO_RCVTIMEO use a struct timeval parameter, defined in <oskit/c/sys/time.h>

The following options are recognized at the socket level. Except as noted, each may be examined with getsockopt and set with setsockopt.

OSKIT_SO_DEBUG	enables recording of debugging information
OSKIT_SO_REUSEADDR	enables local address reuse
OSKIT_SO_REUSEPORT	enables duplicate address and port bindings
OSKIT_SO_KEEPALIVE	enables keep connections alive
OSKIT_SO_DONTROUTE	enables routing bypass for outgoing messages
OSKIT_SO_LINGER	linger on close if data present
OSKIT_SO_BROADCAST	enables permission to transmit broadcast messages
OSKIT_SO_OOBINLINE	enables reception of out-of-band data in band
OSKIT_SO_SNDBUF	set buffer size for output
OSKIT_SO_RCVBUF	set buffer size for input
OSKIT_SO_SNDLOWAT	set minimum count for output
OSKIT_SO_RCVLOWAT	set minimum count for input
OSKIT_SO_SNDTIMEO	set timeout value for output
OSKIT_SO_RCVTIMEO	set timeout value for input
OSKIT_SO_TYPE	get the type of the socket (get only)
OSKIT_SO_ERROR	get and clear error on the socket (get only)

OSKIT_SO_DEBUG enables debugging in the underlying protocol modules. OSKIT_SO_REUSEADDR indicates that the rules used in validating addresses supplied in bind should allow reuse of local addresses. OSKIT_SO_REUSEPORT allows completely duplicate bindings by multiple clients if they all set OSKIT_SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. `OSKIT_SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and clients using the socket are notified when attempting to send data. `OSKIT_SO_DONTROUTE` indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

`OSKIT_SO_LINGER` controls the action taken when unsent messages are queued on a socket and the socket is released. If the socket promises reliable delivery of data and `OSKIT_SO_LINGER` is set, the system will block on the last `release` attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the `setsockopt` call when `OSKIT_SO_LINGER` is requested. If `OSKIT_SO_LINGER` is disabled, the last `release` will succeed immediately.

The option `OSKIT_SO_BROADCAST` requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the `OSKIT_SO_OOBINLINE` option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with `recv` or `read` calls without the `OSKIT_MSG_OOB` flag. Some protocols always behave as if this option were set.

`OSKIT_SO_SNDBUF` and `OSKIT_SO_RCVBUF` are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. An absolute limit may be places on these values.

`OSKIT_SO_SNDLOWAT` is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed.

The default value for `OSKIT_SO_SNDLOWAT` is set to a convenient size for network efficiency, often 1024.

`OSKIT_SO_RCVLOWAT` is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for `OSKIT_SO_RCVLOWAT` is 1. If `OSKIT_SO_RCVLOWAT` is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

`OSKIT_SO_SNDTIMEO` is an option to set a timeout value for output operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error `OSKIT_EWOULDBLOCK` if no data were sent.

This timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output.

`OSKIT_SO_RCVTIMEO` is an option to set a timeout value for input operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for input operations to complete.

This timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `OSKIT_EWOULDBLOCK` if no data were received.

Finally, `OSKIT_SO_TYPE` and `OSKIT_SO_ERROR` are options used only with `getsockopt`. `OSKIT_SO_TYPE` returns the type of the socket, such as `OSKIT_SOCK_STREAM`. `OSKIT_SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

val, valsize: The parameters *val* and *valsize* are used to access option values for `setsockopt`. For `getsockopt` they identify a buffer in which the value for the requested option(s) are to be returned. For `getsockopt`, *valsize* initially contains the size of the buffer pointed to by *val*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *val* may be `NULL`.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.10 `recvfrom`, `recvmsg`: receive a message from a socket

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_recvfrom**(`oskit_socket_t *`*s*, [out] `void *`*buf*, `oskit_size_t` *len*, `oskit_u32_t` *flags*, [out] `struct` *oskit_sockaddr *from*, [in/out] `oskit_size_t *`*fromlen*, [out] `oskit_size_t *`*retval*);

`oskit_error_t` **oskit_socket_recvmsg**(`oskit_socket_t *`*s*, [in/out] `struct` *oskit_msghdr *msg*, `oskit_u32_t` *flags*, [out] `oskit_size_t *`*retval*);

Description

`recvfrom` and `recvmsg` are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

Note: The `recv` library function can be implemented using `recvfrom` with a nil *from* parameter.

If no messages are available at the socket, the receive call waits for a message to arrive. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount equested; this behavior is affected by the socket-level options `OSKIT_SO_RCVLOWAT` and `OSKIT_SO_RCVTIMEO` described in `getsockopt`.

Parameters

s: The socket from the message is to be received.

buf: Buffer in which the message is to be copied.

len: Length of the buffer provided.

flags: The *flags* argument is formed by or'ing one or more of the values:

`OSKIT_MSG_OOB`	process out-of-band data
`OSKIT_MSG_PEEK`	peek at incoming message
`OSKIT_MSG_WAITALL`	wait for full request or error

The `OSKIT_MSG_OOB` flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The `OSKIT_MSG_PEEK` flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The `OSKIT_MSG_WAITALL` flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if an error or disconnect occurs, or the next data to be received is of a different type than that returned.

from: If *from* is non-nil, and the socket is not connection-oriented, the source address of the message is filled in.

fromlen: Initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there.

msg: The `recvmsg` method uses a `struct oskit_msghdr` structure to minimize the number of directly supplied parameters.

```
struct oskit_msghdr {
    oskit_addr_t   msg_name;              /*  optional address             */
    oskit_u32_t    msg_namelen;           /*  size of address              */
    struct         oskit_iovec *msg_iov;  /*  scatter/gather array         */
    oskit_u32_t    msg_iovlen;            /*  # elements in msg_iov        */
    oskit_addr_t   msg_control;           /*  ancillary data, see below    */
    oskit_u32_t    msg_controllen;        /*  ancillary data buffer len    */
    oskit_u32_t    msg_flags;             /*  flags on received message    */
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. *msg_iov* and *msg_iovlen* describe scatter gather locations.

msg_control, which has length *msg_controllen*, points to a buffer for other protocol control related messages or other miscellaneous ancillary data.

The *msg_flags* field is set on return according to the message received. `OSKIT_MSG_EOR` indicates end-of-record; the data returned completed a record (generally used with sockets of type `OSKIT_SOCK_SEQPACKET`). `OSKIT_MSG_TRUNC` indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. `OSKIT_CMSG_TRUNC` indicates that some control data were discarded due to lack of space in the buffer for ancillary data. `OSKIT_MSG_OOB` is returned to indicate that expedited or out-of-band data were received.

retval: Contains the number of characters received, i.e., the total length of the message upon return. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

8.2.11 `sendto`, `sendmsg`: send a message from a socket

Synopsis

`#include <oskit/net/socket.h>`

`oskit_error_t` **oskit_socket_sendto**(`oskit_socket_t` **s*, `const` *void *buf*, `oskit_size_t` *len*, `oskit_u32_t` *flags*, `const` *struct oskit_sockaddr *to*, `oskit_size_t` *tolen*, [out] `oskit_size_t` **retval*);

`oskit_error_t` **oskit_socket_sendmsg**(`oskit_socket_t` **s*, `const` *struct oskit_msghdr *msg*, `oskit_u32_t` *flags*, [out] `oskit_size_t` **retval*);

Description

`sendto`, `sendmsg` are used to transmit a message to another socket. The C library `send` may be implemented by passing a NULL *to* parameter. It may be used only when the socket is in a connected state, while `sendto` and `sendmsg` may generally be used at any time.

Send will block if no messages space is available at the socket to hold the message to be transmitted.

Parameters

s: The socket from which the message is to be sent.

buf:

len: *len* gives the length of the message. If the message is too long to pass atomically through the underlying protocol, the error `OSKIT_EMSGSIZE` is returned, and the message is not transmitted.

flags: The *flags* parameter may include one or more of the following:

`OSKIT_MSG_OOB`	process out-of-band data
`OSKIT_MSG_PEEK`	peek at incoming message
`OSKIT_MSG_DONTROUTE`	bypass routing, use direct interface
`OSKIT_MSG_EOR`	data completes record
`OSKIT_MSG_EOF`	data completes transaction

The flag `OSKIT_MSG_OOB` is used to send "out-of-band" data on sockets that support this notion (e.g. `OSKIT_SOCK_STREAM`); the underlying protocol must also support "out-of-band" data. `OSKIT_MSG_EOR` is used to indicate a record mark for protocols which support the concept. `OSKIT_MSG_EOF` requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for `OSKIT_SOCK_STREAM` sockets in the `OSKIT_PF_INET` protocol family.

to, tolen: The address of the target is given by *to* with *tolen* specifying its size.

msg: See `recvmsg` for a description of the `oskit_msghdr` structure.

retval: Upon return **retval* contains the number of characters sent.

Returns

Returns 0 on success. No indication of failure to deliver is implicit in a send. Locally detected errors are indicated by an error code specified in `<oskit/error.h>`.

Part III

Function Libraries

Chapter 9

Minimal C Library: `liboskit_c.a`

9.1 Introduction

The OSKit's minimal C library is a subset of a standard ANSI/POSIX C library designed specifically for use in kernels or other restricted environments in which a "full-blown" C library cannot be used. The minimal C library provides many simple standard functions such as string, memory, and formatted output functions: functions that are often useful in kernels as well as application programs, but because ordinary application-oriented C libraries are unusable in kernels, must usually be reimplemented or manually "pasted" into the kernel sources with appropriate modifications to make them usable in the kernel environment. The versions of these functions provided by the OSKit minimal C library, like the other components of the OSKit, are designed to be as generic and context-independent as possible, so that they can be used in arbitrary environments without the developer having to resort to the traditional manual cut-and-paste methods. This cleaner strategy brings with it the well-known advantages of careful code reuse: the kernel itself becomes smaller and simpler due to fewer extraneous "utility" functions hanging around in the sources; it is easier to maintain both the kernel, for the above reason, and the standard utility functions it uses, because there is only one copy of each to maintain; finally, the kernel can easily adopt new, improved implementations of common performance-critical functions as they become available, simply by linking against a new version of the minimal C library (e.g., new versions of `memcpy` or `bzero` optimized for particular architectures or newer family members of a given architecture).

In general, the minimal C library provides *only* functions specified in the ANSI C or POSIX.1 standards, and only a subset thereof. Furthermore, the provided implementations of these functions are designed to be as independent as possible from each other and from the environment in which they run, allowing arbitrary subsets of these functions to be used when needed without pulling in any more functionality than necessary and without requiring the OS developer to provide significant support infrastructure. For example, all of the "simple" functions which merely perform some computation on or manipulation of supplied data, such as the string instructions, are guaranteed to be completely independent of each other.

The functions that are inherently environment-dependent in some way, such as `printf`, which assumes the existence of some kind of "standard output" or "console," are implemented in terms of other clearly specified, environment-dependent functions. Thus, in order to use the minimal C library's implementation of `printf`, the OS developer must provide appropriate `console_putchar` and `console_putbytes` routines to be used to write characters to whatever acts as the "standard output" in the current environment. All such dependencies between C library functions are explicitly stated in this document, so that it is always clear what additional functions the developer must supply in order to make use of a set of functions provided by the minimal C library.

Since almost all of the functions and definitions provided by the OSKit minimal C library implement well-known, well-defined ANSI and POSIX C library interfaces which are amply documented elsewhere, we do not attempt to describe the purpose and behavior of each function in this chapter. Instead, only the peculiarities relevant to the minimal C library, such as implementation interdependencies and side effects, are described here.

Note that many files and functions in the minimal C library are derived or taken directly from other source code bases, particularly Mach and BSD. Specific attributions are made in the source files themselves.

9.2 posix Interface

Some of the functions provided in the minimal C library depend on lower level I/O routines in the POSIX library (see Section 13) to provide mappings to the appropriate OSKit COM interfaces. For example, `fopen` in the C library will chain to `open` in the POSIX library, which in turn will chain to the appropriate `oskit_dir` and `oskit_file` COM operations. Certain initialization routines in the POSIX library may need to be called; refer to Section 13.3 for details.

9.3 Unsupported Features

The following features in many C libraries are deliberately unsupported by the minimal C library, for reasons described below, and will remain unsupported unless a compelling counterargument arises:

- **File I/O:** There is no support for file operations in the minimal C library. While "stream" operations like `fopen` *are* defined in the minimal C library, these functions depend on the extended POSIX support contained in the POSIX library (see Section 13) to provide the necessary low level file operations such as `open`, `read`, and `write`. Programs that do not use any of the stream operations need not link with POSIX library.

- **Locales:** Typical programs that use the minimal C library, particularly kernels, are generally not the kinds of programs that need extensive internationalization support from the C library functions they use. In practice, the string-related minimal C library functions are typically used for printing diagnostic messages and allowing the user to select boot time parameters such as the root partition; for these purposes, simplicity and compactness are generally more important than multilingual flexibility. If a particular (rare) kernel *does* want full internationalization support in the C library functions it uses, and is prepared to pay the price in size and complexity, then it can instead use the full internationalized implementations from standard application-oriented C libraries, rather than the simple ones provided by the minimal C library.

- **Multibyte characters:** These are not supported for basically the same reasons as for locales.

- **I/O buffering:** Although the OSKit minimal C library provides high-level I/O functions such as `fprintf`, `fputc`, `fread`, etc., these functions do no buffering, and instead simply translate directly into calls to low-level I/O routines (e.g., `read` and `write`). We chose this strategy because typical programs that use the minimal C library only want to use high-level I/O functions for the convenience they provide (particularly formatted I/O), not for the performance benefits of buffering. Full I/O buffering generally comes with a great deal of C library code size and complexity, and add many additional dependencies to the environment (e.g., memory allocation for buffers, detection of line disciplines). Furthermore, the mere act of buffering I/O implies a major assumption about the environment and the use of these functions: in particular, it assumes that the underlying low-level I/O operations have high per-invocation overhead and that the high-level I/O operations are called at fine enough granularity to make this overhead a problem in practice. This assumption is often invalid for clients of the minimal C library, which generally use I/O functions only sporadically if at all, rather than intensively as many user-level applications do; and in any case, one of the primary goals of the minimal C library is to avoid such assumptions in the first place. For these reasons, we felt that I/O buffering is neither necessary nor appropriate for the minimal C library to perform.

- **Floating-point math:** In general, most kernels and other programs likely to use the minimal C library do not perform much, if any, floating point arithmetic; in many cases they never even access the FPU other than to save and restore its state on context switches. For this reason, all of the floating-point math functions that are a standard part of most C libraries are omitted from the minimal C library.

  There is limited support for printing floating point numbers, however. If this feature is desired, `doprnt.c` can be compiled with `-DDOPRNT_FLOAT` to enable the use of the `%f` format specifier.

9.4 Header Files

When the OSKit is installed using `make install`, a set of standard ANSI/POSIX-defined header files, containing definitions and function prototypes for the minimal C library, are installed in the selected `include` directory under the subdirectory `oskit/c/`. For example, the version of the ANSI C header file `string.h` provided with the minimal C library is installed as *prefix*`/include/oskit/c/string.h`. These header files are installed in a subdirectory rather than in the top level `include` directory so that if the OSKit is installed in a standard place shared by other packages and/or system files, such as `/usr` or `/usr/local`, the minimal C library's header files will not conflict with header files provided by normal application-oriented C libraries, nor will applications "accidentally" use the minimal C library's header files when they really want the normal C library's header files.

There are two main ways a kernel or other program can explicitly use the OSKit minimal C library's header files. The first is by including the `oskit/c/` prefix directly in all relevant `#include` statements; e.g., '`#include <oskit/c/string.h>`' instead of '`#include <string.h>`'. However, since this method effectively makes the client code somewhat specific to the OSKit minimal C library by hard-coding OSKit-specific pathnames into the `#include` statements, this method should generally only be used if for some reason the code in question is extremely dependent on the OSKit minimal C library in particular, and it would never make sense for it to include corresponding header files from a different C library.

For typical code using the minimal C library, which simply needs "a `printf`" or "a `strcpy`," the preferred method of including the library's header files is to code the `#include` lines without the `oskit/c/` prefix, just as in application code using an ordinary C library, and then add an appropriate `-I` (include directory) directive to the compiler command line so that the `oskit/c/` directory will be scanned automatically for these header files before the top-level `include` directory and other include directories in the system are searched. Typically this `-I` directive can be added to the `CFLAGS` variable in the `Makefile` used to build the program in question. In fact, the OSKit itself uses this method to allow code in other toolkit components and in the minimal C library itself to make use of definitions and functions provided by the minimal C library. (Of course, these dependencies are clearly documented, so that if you want to use other OSKit components but not the minimal C library, or only part of the minimal C library, it is possible to do so cleanly.)

Except when otherwise noted, all of the definitions and functions described in this section are very simple, have few dependencies, and behave as in ordinary C libraries. Functions that are not self-contained and interact with the surrounding environment in non-trivial ways (e.g., the memory allocation functions) are described in more detail in later sections.

9.4.1 `a.out.h`: semi-standard `a.out` file format definitions

Description

This header file simply cross-includes the header file `oskit/exec/a.out.h`, which is part of the executable interpreter library (see Section 24.1.2) and provides a minimal set of definitions describing `a.out`-format executable and object files. Although this header file is not standard ANSI or POSIX (thank goodness!), it is a fairly strong Unix tradition, and is especially relevant to operating system code, and therefore is provided as part of the OSKit.

9.4.2 `alloca.h`: explicit stack-based memory allocation

Description

This header file defines the `alloca` pseudo-function, which allows C code to dynamically allocate memory on the calling function's stack frame, which will be freed automatically when the function returns. This header is not ANSI or POSIX but is a fairly well-established tradition. The implementation of this function currently depends on being compiled with gcc.

9.4.3 `assert.h`: program diagnostics facility

Description

This header file provides a standard `assert` macro as described in the C standard. All uses of the `assert` macro are compiled out (they generate no code) if the preprocessor symbol `NDEBUG` is defined before this header file is included.

9.4.4 `ctype.h`: character handling functions

Description

This header file provides implementations of the following standard character handling functions:

`isascii`: Tests if a character is in the range 0–127. This is not supplied in ISO C but exists on many systems.

`isalnum`: Tests if a character is alphanumeric.

`isalpha`: Tests if a character is alphabetic.

`iscntrl`: Tests if a character is a control character.

`isdigit`: Tests if a character is a decimal digit.

`isgraph`: Tests if a character is a printable non-space character.

`islower`: Tests if a character is a lowercase letter.

`isprint`: Tests if a character is a printable character, including space.

`ispunct`: Tests if a character is a punctuation mark.

`isspace`: Tests if a character is a whitespace character of any kind.

`isupper`: Tests if a character is a uppercase letter.

`isxdigit`: Tests if a character is a hexadecimal digit.

`toascii`: Converts an integer into a 7-bit ASCII character.

`tolower`: Converts a character to lowercase.

`toupper`: Converts a character to uppercase.

The implementations of these functions provided by the minimal C library are directly-coded inline functions, and do not reference any global data structures such as character type arrays. They do not support locales (see Section 9.3), and only recognize the basic 7-bit ASCII character set (all characters above 126 are considered to be control characters).

9.4.5 `errno.h`: error numbers

Description

This file declares the global `errno` variable, and defines symbolic constants for all the `errno` values defined in the ISO/ANSI C, POSIX.1, and UNIX standards. They are provided mainly for the convenience of clients that can benefit from standardized error codes and do not already have their own error handling scheme and error code namespace. The symbols defined in this header file have the same values as the corresponding symbols defined in `oskit/error.h` (see 4.6.2), which are the error codes used through the OSKit's COM interfaces; this way, error codes from arbitrary OSKit components can be used directly as `errno` values at least by programs that use the minimal C library.

The main disadvantage of using COM error codes as `errno` values is that, since they don't start from around 0 like typical Unix errno values, it's impossible to provide a traditional Unix-style `sys_errlist` table for them. However, they are fully compatible with the POSIX-blessed `strerror` and `perror` routines, and in any case the minimal C library is not intended to support "legacy" applications directly - for that purpose, a "real" C library would be more appropriate, and such a C library would probably use more traditional `errno` values, doing appropriate translation when interacting with COM interfaces.

9.4.6 `fcntl.h`: POSIX low-level file control

Description

This header file defines prototypes for the low-level POSIX functions `creat` and `open`, and provides symbolic constants for the POSIX open mode flags (`O_*`). Neither `creat` nor `open` are defined in the minimal C library, but instead are defined in the POSIX library (see Section 13).

The open mode constants defined by this header are identical to and interchangeable with the corresponding constants defined in `oskit/fs/file.h` for the `oskit_file` COM interface (see 7.4. These definitions are provided so that clients may standardize on a single set of defintions, which are the same as those used by the COM components. For example, the FreeBSD C library includes this header file, thus providing compatibility between the the two libraries and the disk-based file systems.

9.4.7 `float.h`: constants describing floating-point types

Description

This header file provides the standard set of symbols required by the ISO C standard describing various characteristics of the `float`, `double`, and `long double` types. There is nothing special about the OSKit's definition of these symbols; see the ANSI/ISO C or Single UNIX standard for detailed information about this header file.

9.4.8 `limits.h`: architecture-specific limits

Description

This header file defines the following standard symbols describing architecture-specific limits of basic numeric types:

`CHAR_BIT`: Number of bytes in a `char`.

`CHAR_MAX`: Maximum value of a `char`.

`CHAR_MIN`: Minimum value of a `char`.

`SCHAR_MAX`: Maximum value of a `signed char`.

`SCHAR_MIN`: Minimum value of a `signed char`.

`UCHAR_MAX`: Maximum value of a `unsigned char`.

`SHRT_MAX`: Maximum value of a `short`.

`SHRT_MIN`: Minimum value of a `short`.

`USHRT_MAX`: Maximum value of a `unsigned short`.

`INT_MAX`: Maximum value of a `int`.

`INT_MIN`: Minimum value of a `int`.

`UINT_MAX`: Maximum value of a `unsigned int`.

`LONG_MAX`: Maximum value of a `long`.

`LONG_MIN`: Minimum value of a `long`.

`ULONG_MAX`: Maximum value of a `unsigned long`.

`SSIZE_MAX`: Maximum value of a `size_t`.

The minimal C library's `limits.h` does *not* define any of the POSIX symbols describing operating system-specific limits, such as maximum number of open files, since the minimal C library has know way of knowing how it will be used and thus what these values should be.

9.4.9 `malloc.h`: memory allocator definitions

Description

This header file defines common types and functions used by the minimal C library's default memory allocation functions. This header file is *not* a standard POSIX or X/Open CAE header file; instead its purpose is to expose the implementation of the `malloc` facility so that the client can fully control it and use it in arbitrary contexts.

The `malloc` package implements the following standard allocation routines (also defined in `stdlib.h`).

`malloc`: Allocate a chunk of memory in the caller's heap.

`mustmalloc`: Like `malloc`, but calls `panic` if the allocation fails.

`memalign`: Allocate a chunk of aligned memory.

`calloc`: Allocate a zero-filled chunk of memory.

`mustcalloc`: Like `calloc`, but calls `panic` if the allocation fails.

`realloc`: Changes the allocated size of a chunk of memory while preserving the contents of that memory.

`free`: Releases a chunk of memory.

The base C library also provides additional routines that allocate chunks of memory that are naturally aligned. The user must keep track of the size of each allocated chunk and free the memory with `sfree` rather than the ordinary `free`.

`smalloc`: Allocate a chunk of user-managed memory in the caller's heap.

`smemalign`: Allocate an aligned chunk of user-managed memory.

`scalloc`: Currently not implemented.

`srealloc`: Currently not implemented.

`sfree`: Free a user-managed chunk of memory previously allocated by an `s*` allocation.

The following are specific to the LMM implementation. They take an additional flag to allow requests for specific types of memory.

`mallocf`: Allocate a chunk of user-managed memory in the caller's heap.

`memalignf`: Allocate an aligned chunk of user-managed memory.

`smallocf`: Allocate a chunk of user-managed memory in the caller's heap.

`smemalignf`: Allocate an aligned chunk of user-managed memory.

The following functions are frequently overridden by the client OS:

`morecore`: Called by `malloc` and `realloc` varients when an attempt to allocate memory from the LMM fails. The default version does nothing.

`mem_lock`: Called to ensure exclusive access to the underlying LMM. The default version does nothing.

`mem_unlock`: Called when exclusive access is no longer needed. The default version does nothing.

See Section 9.5 for details on these functions.

9.4.10 `math.h`: floating-point math functions and constants

Description

This header file provides function prototypes for the math functions conventionally found in `libm`, the standard C math library. Although these functions are not part of the minimal C library, an implementation of the math functions is available in the FreeBSD math library; see Chapter 15 for details. This header file also defines various floating-point constants, such as the value of π, as described in the Unix CAE specification. Since these functions and their implementations are fully standard, they are not described in further detail here; refer to the ISO C and Unix standards for more information.

9.4.11 `netdb.h`: definitions for network database operations

Description

This header file defines structures and prototypes for Internet domain name service (DNS) operations, such as finding the IP address for a host name and vice versa.

9.4.12 `setjmp.h`: nonlocal jumps

Description

This header provides definitions for the minimal `setjmp/longjmp` facility provided in the minimal C library. This facility differs from standard ones in two ways:

- Floating-point state is not saved and restored, since in many kernel environments it is important that the kernel itself not make use of floating point registers.
- Signal state is not saved and restored, since the minimal C library has no concept of signals.

In summary, this header file defines the following symbols:

`jmp_buf`: An array type describing a buffer for `setjmp` to save state in.

`setjmp`: Function to record the current stack and register state.

`longjmp`: Function to return to a previously saved state.

9.4.13 `signal.h`: signal handling

Description

The minimal C library has no support for signals, and thus does not implement any of the functions prototyped in this header file. The header file is here for client OSes that wish to support POSIX signal semantics.

9.4.14 `stdarg.h`: variable arguments

Description

This header provides definitions for accessing variable argument lists. It simply chains to x86-specific definitions.

`va_list`: Type used to declare local state variable used in traversing the variable argument list.

`va_start`: Initializes the `va_list` state variable. Must be called before `va_arg` or `va_end`.

`va_arg`: This macro returns the value of the next argument in the variable argument list, and advances the `va_list` state variable.

`va_end`: This macro is called after all the arguments have been read.

9.4.15 `stddef.h`: common definitions

DESCRIPTION

This header file defines the symbol `NULL` and the type `size_t` if they haven't been defined already. It also defines `wchar_t` and the `offsetof` macro.

9.4.16 `stdio.h`: standard input/output

DESCRIPTION

This header provides definitions for the standard input and output facilities provided by the minimal C library. Many of these routines simply chain to the low-level I/O routines in the POSIX library, and do no buffering.

`putchar`: Output a character to `stdout`.

`puts`: Output a string to a stream.

`printf`: Formatted output to `stdout`.

`vprintf`: Formatted output to `stdout` with a `stdarg.h` `va_list` argument.

`sprintf`: Formatted output to a string buffer.

`snprintf`: Formatted output of up to len characters into a string buffer.

`vsprintf`: Formatted output to a string buffer with a `stdarg.h` `va_list` argument.

`vsnprintf`: Formatted output of up to len characters into a string buffer with a `stdarg.h` `va_list` argument.

`getchar`: Input a character from `stdin`.

`gets`: Input a string from `stdin`.

`fgets`: Input a string from a stream.

`fopen`: Open a stream.

`fclose`: Close a stream.

`fread`: Read bytes from a stream.

`fwrite`: Write bytes to a stream.

`fputc`: Output a character to a stream.

`fputs`: Output a string to a stream.

`fgetc`: Input a character from a stream.

`fprintf`: Formatted output to a stream.

`vfprintf`: Formatted output to a stream with a `stdarg.h` `va_list` argument.

`fscanf`: Formatted input from a stream.

`fseek`: Reposition a stream.

`feof`: Check for end-of-file in an input stream.

`ftell`: Return the current position in a stream.

`rewind`: Reset a stream to the beginning.

`hexdump`: Print a buffer in hexdump style.

`putc`: Macro-expanded to fputc.

9.4.17 `stdlib.h`: standard library functions

DESCRIPTION

This header file defines the symbol `NULL` and the type `size_t` if they haven't been defined already, and provides prototypes for the following functions in the minimal C library:

`atol`: Convert an ASCII decimal number into a `long`.

`strtol`: Convert an ASCII number into a `long`.

`strtoul`: Convert an ASCII number into an `unsigned long`.

`strtod`: Convert ASCII string to `double`.

`malloc`: Allocate a chunk of memory in the caller's heap.

`mustmalloc`: Like `malloc`, but calls `panic` if the allocation fails.

`calloc`: Allocate a zero-filled chunk of memory.

`mustcalloc`: Like `calloc`, but calls `panic` if the allocation fails.

`realloc`: Changes the allocated size of a chunk of memory while preserving the contents of that memory.

`free`: Releases a chunk of memory.

`exit`: Cause normal program termination; see Section 9.8.1.

`abort`: Cause abnormal program termination; see Section 9.8.2.

`panic`: Cause abnormal termination and print a message. Not a standard C function; see Section 9.8.3.

`atexit`: Register a function to be called on exit.

`getenv`: Search for a string in the environment.

Prototypes for the following functions are also provided, but they are not implemented in the minimal C library. See the FreeBSD C library in Section 14.

`abs`: Compute the absolute value of an integer.

`atoi`: Convert an ASCII decimal number into an `int`.

`atof`: Convert ASCII string to `double`.

`qsort`: Sort an array of objects.

`rand`: Compute a pseudo-random integer. Not thread safe; uses static data.

`srand`: Seed the pseudo-random number generator. Not thread safe; uses static data.

9.4.18 `string.h`: string handling functions

DESCRIPTION

This header file defines the symbol `NULL` if it hasn't been defined already, and provides prototypes for the following functions in the minimal C library:

`memcpy`: Copy data from one location in memory to another. Our implementation behaves correctly when source and destination overlap.

`memmove`: Like `memcpy` but is guaranteed to behave correctly when source and destination overlap.

`memset`: Set the contents of a block of memory to a uniform value.

`strlen`: Find the length of a null-terminated string.

`strcpy`: Copy a string to another location in memory.

`strncpy`: Copy a string, up to a specified maximum length.

`strdup`: Return a copy of a string in newly-allocated memory. Depends on `malloc`, Section 9.5.2.

`strcat`: Concatenate a second string onto the end of a first.

`strncat`: Concatenate two strings, up to a specified maximum length.

`strcmp`: Compare two strings.

`strncmp`: Compare two strings, up to a specified maximum length.

`strchr`: Find the first occurrence of a character in a string.

`strrchr`: Find the last occurrence of a character in a string.

`strstr`: Find the first occurrence of a substring in a larger string.

`strtok`: Scan for tokens in a string. Not thread safe; uses static data.

`strpbrk`: Locate the first occurrence in a string of one of several characters.

`strspn`: Find the length of an initial span of characters in a given set.

`strcspn`: Measure a span of characters *not* in a given set.

`strerror`: Returns a pointer to a message string for an error number.

The following deprecated functions are provided for compatibility with existing code:

`bcopy`: Copy data from one location in memory to another.

`bzero`: Clear the contents of a memory block to zero.

`index`: Find the first occurrence of a character in a string.

`rindex`: Find the last occurrence of a character in a string.

9.4.19 `strings.h`: string handling functions (deprecated)

Description

For compatibility with existing software, a header file called `strings.h` is provided which acts as a synonym for `string.h` (Section 9.4.18).

9.4.20 `sys/gmon.h`: GNU profiling support definitions

Description

GNU profiling support definitions.

9.4.21 `sys/ioctl.h`: I/O control definitions

Description

Format definitions for 'ioctl' commands. From BSD4.4.

9.4.22 `sys/mman.h`: memory management and mapping definitions

Description

This file includes constant definitions and function prototypes for memory management operations.

`mmap`: Map a file into a region of memory.

`mprotect`: Change the protections associated with an `mmap`ed region.

`munmap`: Unmap a file from memory.

None of these routines are implemented in the minimal C library.

The defined constant values are the same as traditional BSD, though the values of `PROT_READ` and `PROT_EXEC` are reversed.

9.4.23 `sys/reboot.h`: reboot definitions (deprecated)

Description

Definitions the arguments to the reboot system call.

9.4.24 `sys/signal.h`: signal handling (deprecated)

Description

This header simply includes the base C library `signal.h`.

9.4.25 `sys/stat.h`: file operations

Description

This header includes constant definitions and function prototypes for file operations.

`chmod`: Change the access mode of a file.

`fchmod`: Change the access mode of a file descriptor.

`stat`: Get statistics on a named file.

`lstat`: Get statistics on a named file without following symbolic links.

`fstat`: Get statistics on an open file by file descriptor.

`mkdir`: Create a directory.

`mkfifo`: Create a fifo.

`mknod`: Create a special file.

`umask`: Get/set creation mode mask.

None of these routines are implemented in the minimal C library. Refer to the POSIX library in Section 13.

9.4.26 `sys/termios.h`: terminal handling functions and definitions (deprecated)

Description

This header simply includes the base C library `termio.h`.

9.4.27 `sys/time.h`: timing functions

Description

This header includes constant definitions and function prototypes for timing and related functions, none of which are implemented in the minimal C library. Refer to the POSIX library (Section 13) and the FreeBSD C library (Section 14) for implementation of these functions.

9.4.28 `sys/wait.h`: a POSIX wait specification

Description

Note that the minimal C library has no support for processes, and thus doesn't implement any of the functions prototyped in this header file. The header file is here in case client OSes wish to support POSIX `wait` semantics.

9.4.29 `sys/types.h`: general POSIX types

Description

General POSIX types.

9.4.30 `termios.h`: terminal handling functions and definitions

Description

The minimal C library does not fully support termios. Some of the termio stuff is implemented elsewhere to support OSKit devices.

9.4.31 `unistd.h`: POSIX standard symbolic constants

Description

This file contains the required symbolic constants for a POSIX system. These include the symbolic `access` and `seek` constants:

`R_OK`: Test for read permission.

`W_OK`: Test for write permission.

`X_OK`: Test for execute permission.

`F_OK`: Test for file existence.

`SEEK_SET`: Set file offset to value.

`SEEK_CUR`: Set file offset to current plus value.

`SEEK_END`: Set file offset to EOF plus value.

This file defines no POSIX compile-time or execution-time constants. Additionally defined are the constants:

`STDIN_FILENO`: File descriptor for `stdin`.

`STDOUT_FILENO`: File descriptor for `stdout`.

`STDERR_FILENO`: File descriptor for `stderr`.

prototypes for standard POSIX functions:

`_exit`: Terminate a process.

`access`: Check file accessibility.

`close`: Close a file.

`lseek`: Reposition read/write file offset.

`read`: Read from a file.

`unlink`: Remove directory entries.

`write`: Write to a file.

Of the above routines, only `_exit` is considered part of the minimal C library. The remaining functions are part of the extended POSIX environment. Refer to Section 13 for details.

9.4.32 `utime.h`: file times

DESCRIPTION

This file defines the `utimbuf` structure, as well as the prototype for the POSIX function `utime`, which sets the access and modification times of a named file. This function is not implemented in the minimal C library. Refer to Section 13 for details.

9.4.33 `sys/utsname.h`: system identification

DESCRIPTION

This file defines the `utsname` structure, as well as the prototype for the POSIX function `uname`, which returns a series of null terminated strings of information identifying the current system. This function is not implemented in the minimal C library. Refer to Section 13 for details.

9.5 Memory Allocation

All of the default memory allocation functions in the minimal C library are built on top of the OSKit LMM, described in Chapter 16.

There are three families of memory allocation routines available in the minimal C library. First is the standard `malloc`, `realloc`, `calloc`, and `free`. These work as in any standard C library.

The second family, `smalloc`, `smemalign`, and `sfree`, assume that the caller will keep track of the size of allocated memory blocks. Chunks allocated with `smalloc`-style functions must be freed with `sfree` rather than the normal `free`. These functions are not part of the POSIX standard, but are *much* more memory efficient when allocating many power-of-two-size chunks naturally aligned to their size (e.g., when allocating naturally-aligned pages or superpages). The normal `memalign` function attaches a prefix to each allocated block to keep track of the block's size, and the presence of this prefix makes it impossible to allocate naturally-aligned, natural-sized blocks successively in memory; only every *other* block can be used, greatly increasing fragmentation and effectively halving usable memory. (Note that this fragmentation property is not peculiar to the OSKit's implementation of `memalign`; most versions of `memalign` produce have this effect.)

The third family, `mallocf`, `memalignf`, `smallocf`, and `smemalignf`, allow LMM flags to be passed to the more common allocation routines. These are useful for allocating memory of a specific type (see 16.2). Memory allocated with these routines should be freed with `free` or `sfree` as appropriate.

All of the memory management functions, if they are unable to allocate a block out of the LMM pool, call the `morecore` function and then retry the allocation if `morecore` returns non-zero. The default behavior for this function is simply to return 0, signifying that no more memory is available. In environments in which a dynamically growable heap is available, you can override the `morecore` function to grow the heap as appropriate.

All of the memory allocation functions make calls to `mem_lock` and `mem_unlock` to protect access to the LMM pool under all of these services. The default implementation of these synchronization functions in the minimal C library is to do nothing. However, when the C library is initialized (see Section 9.7.1 or Section 14.7.1), a query for the lock manager will be made (See Section 4.12) to determine if there is a default implementation of locks available, and will use that implementation to guarantee thread/SMP safety. The absence of a lock manager implementation implies a single threaded environment, and thus locks are unnecessary. Additionally, they can be overridden with functions that acquire and release a lock of some kind appropriate to the environment in order to make the allocation functions thread- or SMP-safe. Also, note that if you link in `liboskit_kern` before `liboskit_c`, the kernel support library provides its own default implementation of `mem_lock` and `mem_unlock`, which call `base_critical_enter` and `base_critical_leave` respectively; this provides simple and robust, though probably far from optimal, memory allocation protection for kernel code running on the bare hardware.

9.5.1 `malloc_lmm`: LMM pool used by the default memory allocation functions

SYNOPSIS

```
#include <oskit/c/malloc.h>
extern lmm_t malloc_lmm;
```

DESCRIPTION

The LMM pool used by all default memory allocation functions either directly or indirectly.

In the base environemnt, this LMM is initialized at boot time to contain all the physical memory available in the system (see Section 10.11). "Available memory" means all that is not used by base environment data structures or by the OS kernel image itself.

9.5.2 `malloc`: allocate uninitialized memory

SYNOPSIS

```
#include <oskit/c/malloc.h>
```

`void *`**malloc**`(size_t` *size*);

DESCRIPTION

Standard issue `malloc` function. Calls `mallocf` with flags value zero to allocate the memory.

PARAMETERS

size: Size in bytes of desired allocation.

RETURNS

Returns a pointer to the allocated memory or zero if none.

9.5.3 `mustmalloc`: allocate uninitialized memory and panic on failure

SYNOPSIS

`#include <oskit/c/malloc.h>`

`void *`**mustmalloc**`(size_t` *size*);

DESCRIPTION

Calls `malloc` to allocate memory, `assert`ing that the return is non-zero; i.e., `mustmalloc` will `panic` if no memory is available.

Note that if `NDEBUG` is defined, `assert` will do nothing and this routine is identical to `malloc`.

PARAMETERS

size: Size in bytes of desired allocation.

RETURNS

Returns a pointer to the allocated memory if it returns at all.

9.5.4 `memalign`: allocate aligned uninitialized memory

SYNOPSIS

`#include <oskit/c/malloc.h>`

`void *`**memalign**`(size_t` *alignment*, `size_t` *size*);

DESCRIPTION

Allocate uninitialized memory with the specified byte alignment; e.g., an alignment value of 32 will return a block aligned on a 32-byte boundary. Calls `memalignf` with flags value zero to allocate the memory.

Note that the alignment is *not* the same as used by the underlying LMM routines. The alignment parameter in LMM calls is the number of low-order bits that should be zero in the returned pointer.

PARAMETERS

alignment: Desired byte-alignment of the returned block.

size: Size in bytes of desired allocation.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.5 `calloc`: allocate cleared memory

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**calloc**(`size_t` *nelt*, `size_t` *eltsize*);

Description

Standard issue `calloc` function. Calls `malloc` to allocate the memory and `memset` to clear it.

Parameters

nelt: Number of elements being allocated.

eltsize: Size of each element.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.6 `mustcalloc`: allocate cleared memory and panic on failure

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**mustcalloc**(`size_t` *nelt*, `size_t` *eltsize*);

Description

Calls `calloc` to allocate memory, `assert`ing that the return is non-zero; i.e., `mustcalloc` will `panic` if no memory is available.

Note that if `NDEBUG` is defined, `assert` will do nothing and this routine is identical to `calloc`.

Parameters

nelt: Number of elements being allocated.

eltsize: Size of each element.

Returns

Returns a pointer to the allocated memory if it returns at all.

9.5.7 `realloc`: change the size of an existing memory block

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**realloc**(`void *`*buf*, `size_t` *new_size*);

Description

Standard issue `realloc` function. Calls `malloc` if *buf* is zero, otherwise calls `lmm_alloc` to allocate an entirely new block of memory, uses `memcpy` to copy the old block, and `lmm_free`s that block when done.

May call `morecore` if the initial attempt to allocate memory fails.

Parameters

buf: Pointer to memory to be enlarged.

new_size: Desired size of resulting block.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.8 `free`: release an allocated memory block

Synopsis

`#include <oskit/c/malloc.h>`

`void` **free**(`void *`*buf*);

Description

Standard issue `free` function. Calls `lmm_free` to release the memory.

Note that `free` must only be called with memory allocated by one of: `malloc`, `realloc`, `calloc`, `mustmalloc`, `mustcalloc`, `mallocf`, `memalign`, or `memalignf`.

Parameters

buf: Pointer to memory to be freed.

9.5.9 `smalloc`: allocated uninitialized memory with explicit size

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**smalloc**(`size_t` *size*);

Description

Identical to `malloc` except that the user must keep track of the size of the allocated chunk and pass that size to `sfree` when releasing the chunk.

Calls `smallocf` with flags value zero to allocate the memory.

Parameters

size: Size in bytes of desired allocation.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.10 `smemalign`: allocate aligned memory with explicit size

SYNOPSIS

```
#include <oskit/c/malloc.h>
void *smemalign(size_t alignment, size_t size);
```

DESCRIPTION

Identical to `memalign` except that the user must keep track of the size of the allocated chunk and pass that size to `sfree` when releasing the chunk.

Allocates uninitialized memory with the specified byte alignment; e.g., an alignment value of 32 will return a block aligned on a 32-byte boundary. Calls `smemalignf` with flags value zero to allocate the memory.

Note that the alignment is *not* the same as used by the underlying LMM routines. The alignment parameter in LMM calls is the number of low-order bits that should be zero in the returned pointer.

PARAMETERS

alignment: Desired byte-alignment of the returned block.

size: Size in bytes of desired allocation.

RETURNS

Returns a pointer to the allocated memory or zero if none.

9.5.11 `sfree`: release a memory block with explicit size

SYNOPSIS

```
#include <oskit/c/malloc.h>
void sfree(void *buf, size_t size);
```

DESCRIPTION

Frees a block of memory with the indicated size. Calls `lmm_free` to release the memory.

Note that `sfree` must only be called with memory allocated by one of: `smalloc`, `smallocf`, `smemalign`, or `smemalignf` and that the size given must match that used on allocation.

PARAMETERS

buf: Pointer to memory to be freed.

size: Size of memory block being freed.

9.5.12 `mallocf`: allocate uninitialized memory with explicit LMM flags

SYNOPSIS

```
#include <oskit/c/malloc.h>
void *mallocf(size_t size, unsigned int flags);
```

Description

Allocates uninitialized memory from `malloc_lmm`. The interface is similar to `malloc` but with an additional *flags* parameter which is passed to `lmm_alloc`.

For kernels running in the base environment on an x86, meaningful values for *flags* are as described in Section 10.11.1.

Parameters

size: Size in bytes of desired allocation.

flags: Flags to pass to `lmm_alloc`.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.13 `memalignf`: allocate aligned uninitialized memory with explict LMM flags

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**memalignf**(`size_t` *alignment*, `size_t` *size*, `unsigned int` *flags*);

Description

Allocate uninitialized memory with the specified byte alignment; e.g., an alignment value of 32 will return a block aligned on a 32-byte boundary. The interface is similar to `malloc` but with an additional *flags* parameter which is passed to `lmm_alloc`.

For kernels running in the base environment on an x86, meaningful values for *flags* are as described in Section 10.11.1.

Note that the alignment is *not* the same as used by the underlying LMM routines. The alignment parameter in LMM calls is the number of low-order bits that should be zero in the returned pointer.

Parameters

alignment: Desired byte-alignment of the returned block.

size: Size in bytes of desired allocation.

flags: Flags to pass to `lmm_alloc`.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.14 `smallocf`: allocated uninitialized memory with explicit size and LMM flags

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**smallocf**(`size_t` *size*, `unsigned int` *flags*);

Description

Allocates uninitialized memory from `malloc_lmm`. The interface is similar to `smalloc` but with an additional *flags* parameter which is passed to `lmm_alloc`. As with `smalloc`, the user must keep track of the size of the allocated chunk and pass that size to `sfree` when releasing the chunk.

For kernels running in the base environment on an x86, meaningful values for *flags* are as described in Section 10.11.1.

Parameters

size: Size in bytes of desired allocation.

flags: Flags to pass to `lmm_alloc`.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.15 `smemalignf`: allocate aligned memory with explicit size and LMM flags

Synopsis

`#include <oskit/c/malloc.h>`

`void *`**smemalignf**(`size_t` *alignment*, `size_t` *size*, `unsigned int` *flags*);

Description

Allocate uninitialized memory with the specified byte alignment; e.g., an alignment value of 32 will return a block aligned on a 32-byte boundary. The interface is similar to `smemalign` but with an additional *flags* parameter which is passed to `lmm_alloc`. As with `smemalign`, the user must keep track of the size of the allocated chunk and pass that size to `sfree` when releasing the chunk.

For kernels running in the base environment on an x86, meaningful values for *flags* are as described in Section 10.11.1.

Note that the alignment is *not* the same as used by the underlying LMM routines. The alignment parameter in LMM calls is the number of low-order bits that should be zero in the returned pointer.

Parameters

alignment: Desired byte-alignment of the returned block.

size: Size in bytes of desired allocation.

flags: Flags to pass to `lmm_alloc`.

Returns

Returns a pointer to the allocated memory or zero if none.

9.5.16 `morecore`: add memory to `malloc` memory pool

Synopsis

`#include <oskit/c/malloc.h>`

`int` **morecore**(`size_t` *size*);

Description

This routine is called directly or indirectly by any of the memory allocation routines in this section when a call to the underlying LMM allocation routine fails. This allows a kernel to add more memory to `malloc_lmm` as needed.

The default version of `morecore` in the minimal C library just returns zero indicating no more memory was available. Client OSes should override this routine as necessary.

Parameters

size: Size in bytes of memory that should be addeed to `malloc_lmm`.

Returns

Returns non-zero if the indicated amount of memory was added, zero otherwise.

9.5.17 `mem_lock`: Lock access to `malloc` memory pool

Synopsis

```
#include <oskit/c/malloc.h>
void mem_lock(void);
```

Description

This routine is called from any default memory allocation routine before it attempts to access `malloc_lmm`.

Coupled with `mem_unlock`, this provides a way to make memory allocation thread and MP safe. In a multithreaded client OS, these functions will use the default lock implementation as provided by the lock manager (see Section 4.12), to protect accesses to the `malloc_lmm`. Or, these functions may be overridden with a suitable synchronization primitive.

Note that the kernel support library provides defaults for `mem_lock` and `mem_unlock` that call `base_critical_enter` and `base_critical_leave` respectively. However, you'll only get these versions if you use the kernel support library and link it in *before* the minimal C library.

9.5.18 `mem_unlock`: Unlock access to `malloc` memory pool

Synopsis

```
#include <oskit/c/malloc.h>
void mem_unlock(void);
```

Description

This routine is called from any default memory allocation routine after all accesses to `malloc_lmm` are complete.

Coupled with `mem_lock`, this provides a way to make memory allocation thread and MP safe. In a multithreaded client OS, these functions will use the default lock implementation as provided by the lock manager (see Section 4.12), to protect accesses to the `malloc_lmm`. Or, these functions may be overridden with a suitable synchronization primitive.

Note that the kernel support library provides defaults for `mem_lock` and `mem_unlock` that call `base_critical_enter` and `base_critical_leave` respectively. However, you'll only get these versions if you use the kernel support library and link it in *before* the minimal C library.

9.6 Standard I/O Functions

The versions of `sprintf`, `vsprintf`, `sscanf`, and `vsscanf` provided in the OSKit's minimal C library are completely self-contained; they do not pull in the code for `printf`, `fprintf`, or other "file-oriented" standard I/O functions. Thus, they can be used in any environment, regardless of whether some kind of console or file I/O is available.

The routines `printf`, `puts`, `putchar`, `getchar`, etc., are all defined in terms of `console_putchar`, `console_getchar`, `console_puts`, and `console_putbytes`. This means that you can get working formatted "console" output merely by providing an appropriate implementation of the aforementioned console functions. In the base environment, these routines are defined in the kernel library (see Section 10.13).

The standard I/O functions that actually take a `FILE*` argument, such as `fprintf` and `fwrite`, and as such are fundamentally dependent on the notion of files, are implemented in terms of the low-level I/O functions in the POSIX library (see Section 13). However, unlike in "real" C libraries, the high-level file I/O functions provided by the minimal C library only implement the minimum of functionality to provide the basic API: in particular, they do no buffering, so for example an `fwrite` translates directly to a `write`. This design reduces code size and minimizes interdependencies between functions, while still providing familiar, useful services such as formatted file I/O.

9.7 Initialization

9.7.1 `oskit_init_libc`: Initialize the OSKit C library

SYNOPSIS

`void` **oskit_init_libc**(`void`);

DESCRIPTION

`oskit_init_libc` allows for internal initializatons to be done. This routine should be called when the operating system is initialized, typically at the beginning of the `main` program.

9.8 Termination Functions

9.8.1 `exit`: terminate normally

Description

`exit` calls up to 32 functions installed via `atexit` in reverse order of installation before it calls `_exit`.

`_exit`, which terminates the calling process in Unix, calls `oskit_libc_exit` with the exit status code (see Section 13.4.2).

9.8.2 `abort`: terminate abnormally

Description

`abort` calls `_exit(1)`.

9.8.3 `panic`: terminate abnormally with an error message

9.9 Miscellaneous Functions

9.9.1 `ntohl`: convert 32-bit long word from network byte order

9.9.2 `ntohs`: convert 16-bit short word from network byte order

9.9.3 `hexdump`: print a buffer as a hexdump

Synopsis

`#include <oskit/c/stdio.h>`

`void` **hexdumpb**(`void *`*base*, `void *`*buf*, `int` *nbytes*);
`void` **hexdumpw**(`void *`*base*, `void *`*buf*, `int` *nwords*);

Description

These functions print out a buffer as a hexdump. For example (the box is included):

```
.---------------------------------------------------------------------------.
| 00000000      837c240c 00741dc7 05007010 00000000      .|$..t....p..... |
| 00000010      008b4424 0ca30470 10008b04 24a30870      ..D$...p....$..p |
| 00000020      1000eb2c c7050070 10000100 0000833c      ...,...p.......< |
| 00000030      2400740a c7050070 10000200 00008b44      $.t....p.......D |
`---------------------------------------------------------------------------'
```

The first form treats the buffer as an array of *bytes* whereas the second treats the buffer as an array of *words*. This distinction is only important on little-endian machines and only affects the appearance of the four middle columns of hex numbers—the last column of output is identical for both.

Parameters

base: What the first column of output should start at. Passing zero will make the first column show the offset within the buffer rather than an absolute address, which is what happens when `base` equals `buf`.

buf: The address of what to dump.

nbytes: How many bytes to dump.

nwords: How many words to dump.

Chapter 10

Kernel Support Library: `liboskit_kern.a`

10.1 Introduction

The kernel support library, `libkern.a`, supplies a variety of functions and other definitions that are primarily of use in OS kernels. (In contrast, the other parts of the OSKit are more generic components useful in a variety of environments *including*, but not limited to, OS kernels.) The kernel support library contains all the code necessary to create a minimal working "kernel" that boots and sets up the machine for a generic "OS-friendly" environment. For example, on the x86, the kernel support library provides code to get into protected mode, set up default descriptor tables, etc. The library also includes a remote debugging stub, providing convenient source-level debugging of the kernel over a serial line using GDB's serial-line remote debugging protocol. As always, all components of this library are optional and replaceable, so although some pieces may be unusable in some environments, others should still work fine.

10.1.1 Machine-dependence of code and interfaces

This library contains a much higher percentage of machine-dependent code than the other libraries in the toolkit, primarily because this library deals with heavily machine-dependent facilities such as page tables, interrupt vector tables, trap handling, etc. The library attempts to hide *some* machine-dependent details from the OS by providing generic, machine-independent interfaces to machine-dependent library code. For example, regardless of the architecture and boot loading mechanism in use, the kernel startup code included in the library always sets up a generic C-compatible execution environment and starts the kernel by calling the well-known `main` routine, just as in ordinary C programs. However, the library makes no attempt to provide a complete architecture-independence layer, since such a layer would have to make too many assumptions about the OS that is using it. For example, although the library provides page table management routines, these routines have fairly low-level, architecture-specific interfaces.

10.1.2 Generic versus Base Environment code

The functionality provided by the kernel support library is divided into two main classes: the generic support code, and the *base environment*. The generic support contains simple routines and definitions that are almost completely independent of the particular OS environment in which they are used: for example, the generic support includes symbolic definitions for bits in processor registers and page tables, C wrapper functions to access special-purpose processor registers, etc. The generic support code should be usable in any OS that needs it.

The base environment code, on the other hand, is somewhat less generic in that it is designed to create, and function in, a well-defined default or "base" kernel execution environment. Out of necessity, this code makes more assumptions about how it is used, and therefore it is more likely that parts of it will not be usable to a particular client OS. For example, on the x86 architecture, the base environment code sets up a default global descriptor table containing a "standard" set of basic, flat-model segment descriptors, as well as a few extra slots reserved for use by the client OS. This "base GDT" is likely to be sufficient for many kernels, but may not be usable to kernels that make more exotic uses of the processor's GDT. In order to allow piecemeal replacement of the base environment as necessary, the assumptions made by the code and the intermodule dependencies are clearly documented in the sections covering the base environment code.

10.1.3 Road Map

Following is a brief summary of the main facilities provided by the library, indexed by the section numbers of the sections describing each facility:

10.2 **Machine-independent Facilities:** Types and constants describing machine-dependent information such as word size and page size. For example, types are provided which, if used properly, allow machine-independent code to compile easily on both 32-bit and 64-bit architectures. Also, functions are provided for various generic operations such as primitive multiprocessor synchronization and efficient bit field manipulation.

10.3 X86 **Generic Low-level Definitions:** Header files describing x86 processor data structures and registers, as well as functions to access and manipulate them. Includes:

- Bit definitions of the contents of the flags, control, debug, and floating point registers.
- Inline functions and macros to read and write the flags, control, debug, segment registers, and descriptor registers (IDTR, GDTR, LDTR, TR).
- Macros to read the Pentium timestamp counter (useful for fine-grained timing and benchmarking) and the stack pointer.
- Structure definitions for architectural data structures such as far pointers, segment and gate descriptors, task state structures, floating point save areas, and page tables, as well as generic functions to set up these structures.
- Symbolic definitions of the processor trap vectors.
- Macros to access I/O ports using the x86's `in` and `out` instructions.
- Assembly language support macros to smooth over the differences in target object formats, such as ELF versus `a.out`.

10.4 X86 PC **Generic Low-level Definitions:** Generic definitions for standard parts of the PC architecture, such as IRQ assignments, the programmable interrupt controller (PIC), and the keyboard controller.

10.5 X86 **Processor Identification and Management:** Functions to identify the CPU and available features, to enter and leave protected mode, and to enable and disable paging.

10.6–10.10 X86 **Base Environment Setup:** Functions that can be used individually or as a unit to set up a basic, minimal kernel execution environment on x86 processors: e.g., a minimal GDT, IDT, TSS, and kernel page tables.

10.11–10.13 X86 PC **Base Environment Setup:** Functions to set up a PC's programmable interrupt controller (PIC) and standard IRQ vectors, to manage a PC's low (1MB), middle (16MB) and upper memory, and to provide simple non-interrupt-driven console support.

10.14 X86 PC **MultiBoot Startup:** Complete startup code to allow the kernel to be booted from any MultiBoot-compliant boot loader easily. Includes code to parse options and environment variables passed to the kernel by the boot loader, and to find and use *boot modules* loaded with the kernel.

10.15 X86 PC **Raw BIOS Startup:** Complete startup code for boot loaders and other programs that need to be loaded directly by the BIOS at boot time. This startup code takes care of all aspects of switching from real to protected mode and setting up a 32-bit environment, and provides mechanisms to call back to 16-bit BIOS code by running the BIOS in either real mode or v86 mode (your choice).

10.16 X86 PC **DOS Startup:** This startup code is similar to the BIOS startup code, but it expects to be loaded in a 16-bit DOS environment: useful for DOS-based boot loaders, DOS extenders, or prototype kernels that run under DOS. Again, this code fully handles mode switching and provides DOS/BIOS callback mechanisms.

10.17 **Kernel Debugging Facilities:** A generic, machine-independent remote GDB stub is provided which supports the standard serial-line GDB protocol. In addition, machine-dependent default trap handling and fault-safe memory access code is provided to allow the debugging stub to be used "out of the box" on x86 PCs.

10.19 **Kernel Annotation Facility:** Macros and functions to associate additional information with ranges of kernel text or data. Annotations allow, for example, a kernel to mark a range of kernel text so that a special function is invoked whenever an exception or interrupt occurs within that range. This facility is useful for implementing rollback routines.

10.2 Machine-independent Facilities

This section includes machine-independent types, constants, macros, and functions that every supported architecture provides. These are used extensively within the OSKit itself as well as by the applications built on the OSKit.

10.2.1 `page.h`: Page size definitions

SYNOPSIS

```
#include <oskit/machine/page.h>
```

DESCRIPTION

This file provides of following symbols, which define the architectural page size of the architecture for which the OSKit is configured:

`PAGE_SIZE`: The number of bytes on each page. It can always be assumed to be a power of two.

`PAGE_SHIFT`: The number of low address bits *not* translated by the MMU hardware. `PAGE_SIZE` is always $2^{\texttt{PAGE\_SHIFT}}$.

`PAGE_MASK`: A bit mask with the low-order `PAGE_SHIFT` address bits set. Always equal to `PAGE_SIZE` − 1. **WARNING**: Some systems (like linux) define this to be `~(PAGE_SIZE - 1)`, be careful that the definitions match what the code expects!

In addition, the following macros are provided for convenience in performing page-related manipulations of addresses:

`atop(`*addr*`)`: Converts a byte address into a page frame number, by dividing by `PAGE_SIZE`.

`ptoa(`*page*`)`: Converts a page frame number into an integer (`oskit_addr_t`) byte address, by multiplying by `PAGE_SIZE`.

`round_page(`*addr*`)`: Returns *addr* rounded up to the next higher page boundary. If *addr* is already on a page boundary, it is returned unchanged.

`trunc_page(`*addr*`)`: Returns *addr* rounded down to the next lower page boundary. If *addr* is already on a page boundary, it is returned unchanged.

`page_aligned(`*addr*`)`: Evaluates to true (nonzero) if *addr* is page aligned, or false (zero) if it isn't.

Note that many modern architectures support multiple page sizes. On such architectures, the page size defined in this file is the *minimum* architectural page size, i.e., the finest granularity over which the MMU has control. Since there seems to be no sufficiently generic and useful way that this header file could provide symbols indicating which "other" page sizes the architecture supports, making good use of larger pages probably must be done in machine-dependent code.

Some operating systems on some architectures do not actually support the minimum architectural page size in software; instead, they aggregate multiple architectural pages together into larger "logical pages" managed by the OS software. On such operating systems, it would be inappropriate for general OS or application code to use the `PAGE_SIZE` value provided by `oskit/page.h`, since this value would be smaller (more fine-grained) than the OS software actually supports, and therefore inappropriate. However, this is purely a high-level OS issue; like other parts of the toolkit, no one is required to use this header file if it is inappropriate in a particular situation.

This file was originally derived from Mach's `vm_param.h`.

10.2.2 `spin_lock.h`: Spin locks

SYNOPSIS

```
#include <oskit/machine/spin_lock.h>
```

DESCRIPTION

This file provides the architecture-dependent definition of the `spin_lock_t` "spin lock" data type and associated manipulation macros. This facility provides a basic locking mechanism which can be used with preemptive threading or on a multi-processor.

`spin_lock_t`: Typedef for the spin lock data type.

`spin_lock_init(`*s*`)`: Initialize a spin lock.

`spin_lock_locked(`*s*`)`: Check if a spin lock is locked.

`spin_unlock(`*s*`)`: Unlock a spin lock.

`spin_try_lock(`*s*`)`: Attempt to lock a spin lock. Returns 0 if successful, nonzero if unsuccessful.

`spin_lock(`*s*`)`: Busy wait until the lock is free. On return the lock has been acquired.

This header file is taken from CMU's Mach kernel.

10.2.3 `queue.h`: Generic queues

SYNOPSIS

```
#include <oskit/queue.h>

struct queue_entry {
   struct   queue_entry *next;   /* next element       */
   struct   queue_entry *prev;   /* previous element   */
};

typedef struct queue_entry *queue_t;
typedef struct queue_entry queue_head_t;
typedef struct queue_entry queue_chain_t;
typedef struct queue_entry *queue_entry_t;
```

DESCRIPTION

Macros and structures for implementation of a "queue" data type. The implementation uses a doubly-linked list and supports operations to insert and delete anywhere in the list.

`queue_init(`*q*`)`: Initialize the given queue.

`queue_first(`*q*`)`: Returns the first entry in the queue.

`queue_next(`*q*`)`: Returns the entry after an item in the queue.

`queue_last(`*q*`)`: Returns the last entry in the queue.

`queue_prev(`*q*`)`: Returns the entry before an item in the queue.

`queue_end(`*q, qe*`)`: Tests whether a new entry is really the end of the queue.

`queue_empty(`*q*`)`: Tests whether a queue is empty.

`queue_enter(`*q, elt, type, field*`)`: Insert a new element at the tail of the queue.

`queue_enter_first(`*head, elt, type, field*`)`: Insert a new element at the head of the queue.

`queue_enter_before(`*head, nelt, elt, type, field*`)`: Insert a new element before the indicated element.

`queue_enter_after(`*`head, pelt, elt, type, field`*`)`: Insert a new element after the indicated element.

`queue_remove(`*`head, elt, type, field`*`)`: Remove an arbitrary item from the queue.

`queue_remove_first(`*`head, entry, type, field`*`)`: Remove and return the entry at the head of the queue.

`queue_remove_last(`*`head, entry, type, field`*`)`: Remove and return the entry at the tail of the queue.

`queue_assign(`*`to, from, type, field`*`)`: Move an element in a queue to a new piece of memory.

`queue_iterate(`*`head, elt, type, field`*`)`: Iterate over each item in the queue. Generates a 'for' loop, setting elt to each item in turn (by reference).

This header file is taken from CMU's Mach kernel.

10.2.4 `debug.h`: debugging support facilities

Synopsis

```
#include <oskit/debug.h>
```

Description

This file contains simple macros and functions to assist in debugging. Many of these facilities are intended to be used to "annotate" programs permanently or semi-permanently in ways that reflect the code's proper or desired behavior. These facilities typically change their behavior depending on whether the preprocessor symbol `DEBUG` is defined: if it is defined, then extra code is introduced to check invariants and such; when `DEBUG` is not defined, all of this debugging code is "compiled out" so that it does not result in any size increase or efficiency loss in the resulting compiled code.

The following macros and functions are intended to be used as permanent- or semi-permanent annotations to be sprinkled throughout ordinary code to increase its robustness and clarify its invariants and assumptions to human readers:

`assert(`*`cond`*`)`: This is a standard `assert` macro, like (and compatible with) the one provided in `oskit/c/assert.h`. If `DEBUG` is defined, this macro produces code that evaluates *cond* and calls `panic` (see Section 9.8.3) if the result is false (zero). When an assertion fails and causes a panic, the resulting message includes the source file name and line number of the assertion that failed, as well as the text of the *cond* expression used in the assertion. If `DEBUG` is not defined, this macro evaluates to nothing (an empty statement), generating no code.

Assertions are typically used to codify assumptions made by a code sequence, e.g., about the parameters to a function or the conditions on entry to or exit from a loop. By placing explicit `assert` statements in well-chosen locations to verify that the code's invariants indeed hold, a thicker "safety net" is woven into the code, which tends to make bugs manifest themselves earlier and in much more obvious ways, rather than allowing incorrect results to "trickle" through the program's execution for a long time, sometimes resulting in completely baffling behavior. Assertions can also act as a form of documentation, clearly describing to human readers the exact requirements and assumptions in a piece of code.

`otsan()`: If `DEBUG` is defined, this macro unconditionally causes a `panic` with the message "off the straight and narrow!," along with the source file name and line number, if it is ever executed. It is intended to be placed at code locations that should never be reached if the code is functioning properly; e.g., as the `default` case of a `switch` statement for which the result of the conditional expression should *always* match one of the explicit case values. If `DEBUG` is not defined, this macro evaluates to nothing.

`do_debug(`*`stmt`*`)`: If `DEBUG` is defined, this macro evaluates to *stmt*; otherwise it evaluates to nothing. This macro is useful in situations where an `#ifdef DEBUG` ... `#endif` block would otherwise be used over just a few lines of code or a single statement: it produces the same effect, but is smaller and less visually intrusive.

The following macros and functions are primarily intended to be used as temporary scaffolding during debugging, and removed from production code:

`void dump_stack_trace(void)`: This function dumps a human-readable backtrace of the current function call stack to the console, using `printf`. The exact content and format of the printed data is architecture-specific; however, the output is typically a list of instruction pointer or program counter values, each pointing into a function on the call stack, presumably to the return point after the function call to the next level. You can find out what function these addresses reside in by running the Unix `nm` utility on the appropriate executable file image, sorting the resulting symbol list if necessary, and looking up the address in the sorted list. Alternatively, for more precise details, you can look up the exact instruction addresses in a disassembly of the executable file, e.g., by using GNU `objdump` with the '`-d`' option.

`here()`: This macro generates code that simply prints the source file name and line number at which the macro was used. This macro can be extremely useful when trying to nail down the precise time or code location at which a particular bug manifests itself, or to determine the sequence of events leading up to it. By sprinkling around calls to the `here` macro in appropriate places, the program will dump regular status reports of its location every time it hits one of these macros, effectively producing a log of "interesting" events ("interesting" being defined according to the placement of the `here` macro invocations). Using the `here` macro this way is equivalent to the common practice of sprinkling `printf`'s around and watching the output, except it is easier because the `here` invocation in each place does not have to be "tailored" to make it distinguishable from the other locations: each use of the `here` macro is self-identifying.

If `DEBUG` is not defined, the `here` macro is not defined at all; this makes it obvious when you've accidentally left invocations of this macro in a piece of code after it has been debugged.

`debugmsg(`*`printfargs`*`)`: This macro is similar to `here`, except it allows a formatted message to be printed along with the source file name and line number. *printfargs* is a complete set of arguments to be passed to the `printf` function, including parentheses: for example, '`debugmsg(("foo is %d", foo));`'. A newline is automatically appended to the end of the message. This macro is generally useful as a wrapper for `printf` for printing temporary run-time status messages during execution of a program being debugged.

As with `here`, if `DEBUG` is not defined, the `debugmsg` macro is not defined at all, in order to make it obvious if any invocations are accidentally left in production code.

Note that only `panic` and `dump_stack_trace` are real functions; the others are simply macros.

10.2.5 `base_critical`: simple critical section support

SYNOPSIS

`#include <oskit/base_critical.h>`

`void` **base_critical_enter**(`void`);

`void` **base_critical_leave**(`void`);

Description

Functions to implements a simple "global critical region." These functions are used throughout the OSKit to ensure proper serialization for various "touchy" but non-performance critical activities such as panicing, rebooting, debugging, etc. This critical region can safely be entered recursively; the only requirement is that `enter`s match exactly with `leave`s.

The implementation of this module is machine-dependent, and generally disables interrupts and, on multiprocessors, grabs a recursive spin lock.

10.3 X86 Generic Low-level Definitions

This section covers useful macros, definitions, and routines that are specific to the Intel x86 processor architecture but that are independent of the interrupt control, bus structure, and other ancillary functions traditionally associated with a "PC." Those facilities are covered in section 10.4.

10.3.1 `asm.h`: assembly language support macros

Synopsis

```
#include <oskit/x86/asm.h>
```

Description

This file contains convenience macros useful when writing x86 assembly language code in AT&T/GAS syntax. This header file is directly derived from Mach, and similar headers are used in various BSD kernels.

Symbol name extension: The following macros allow assembly language code to be written that coexists with C code compiled for either ELF or `a.out` format. In `a.out` format, by convention an underscore (_) is prefixed to each public symbol referenced or defined by the C compiler; however, the underscore prefix is not used in ELF format.

`EXT(`*name*`)`: Evaluates to _*name* in `a.out` format, or just *name* in ELF. This macro is typically used when referring to public symbols defined in C code.

`LEXT(`*name*`)`: Evaluates to _*name*: in `a.out` format, or *name*: in ELF. This macro is generally used when defining labels to be exported to C code.

`SEXT(`*name*`)`: Evaluates to the string literal "_*name*" in `a.out` format, or "*name*" in ELF. This macro can be used in GCC inline assembly code, where the code is contained in a string constant; for example: `asm("...; call "SEXT(foo)"; ...");`

Alignment: The following macros relate to alignment of code and data:

`TEXT_ALIGN`: Evaluates to the preferred alignment of instruction entrypoints (e.g., functions or branch targets), as a power of two. Currently evaluates to 4 (16-byte alignment) if the symbol `i486` is defined, or 2 (4-byte alignment) otherwise.

`ALIGN`: A synonym for `TEXT_ALIGN`.

`DATA_ALIGN`: Evaluates to the preferred minimum alignment of data structures. Currently it is always defined as 2, although in some cases a larger value may be preferable, such as the processor's cache line size.

`P2ALIGN(`*alignment*`)`: Assembly language code can use this macro to work around the fact that the `.align` directive works differently in different x86 environments: sometimes `.align` takes a byte count, whereas other times it takes a power of two (bit count). The `P2ALIGN` macro *always* takes a power of two: for example, `P2ALIGN(2)` means 4-byte alignment. By default, the `P2ALIGN` macro uses the `.p2align` directive supported by GAS; if a different assembler is being used, then `P2ALIGN` should be redefined as either `.align` *alignment* or `.align 1<<(`*alignment*`)`, depending on the assembler's interpretation of `.align`.

XXX S_ARG, B_ARG, frame stuff, ...

XXX need to make the macros more easily overridable, using ifdefs.

XXX need to clean out old trash still in the header file

XXX IODELAY macro

10.3.2 `eflags.h`: Processor flags register definitions

Synopsis

```
#include <oskit/x86/eflags.h>
```

Description

XXX

This header file can be used in assembly language code as well as C. The flags defined here correspond the the ones in the processor databooks.

`EFL_CF`: carry

`EFL_PF`: parity of low 8 bits

`EFL_AF`: carry out of bit 3

`EFL_ZF`: zero

`EFL_SF`: sign

`EFL_TF`: trace trap

`EFL_IF`: interrupt enable

`EFL_DF`: direction

`EFL_OF`: overflow

`EFL_IOPL`: IO privilege level mask. All 0's is the same as EFL_IOPL_KERNEL, while all 1's (or just EFL_IOPL) is the same as EFL_IOPL_USER.

`EFL_NT`: nested task

`EFL_RF`: resume without tracing

`EFL_VM`: virtual 8086 mode

`EFL_AC`: alignment check

`EFL_VIF`: virtual interrupt flag

`EFL_VIP`: virtual interrupt pending

`EFL_ID`: CPUID instruction support

10.3.3 `proc_reg.h`: Processor register definitions and accessor functions

Synopsis

```
#include <oskit/x86/proc_reg.h>
```

Description

XXX

This header file contains the definitions for the processor's control registers (CR0, CR4). It also contains macros for getting and setting the processor registers and flags. There is also a macro for reading the processor's cycle counter (on Pentium and above processors).

This header file is taken from CMU's Mach kernel.

10.3.4 `debug_reg.h`: Debug register definitions and accessor functions

Synopsis

```
#include <oskit/x86/debug_reg.h>
```

Description

This provides the definitions for the processor's built-in debug registers. There are also inline functions that allow the hardware-assisted breakpoints to be set.

DR0 through DR3 are the breakpoint address registers; DR6 is the status register, and DR7 is the control register.

`get_dr0()`: Returns the value in breakpoint address register 0.

`get_dr1()`: Returns the value in breakpoint address register 1.

`get_dr2()`: Returns the value in breakpoint address register 2.

`get_dr3()`: Returns the value in breakpoint address register 3.

`get_dr6()`: Returns the value in the debug status register.

`get_dr7()`: Returns the value in the debug control register.

`set_dr0(`*`val`*`)`: Sets the value in breakpoint address register 0 to *val*.

`set_dr1(`*`val`*`)`: Sets the value in breakpoint address register 1 to *val*.

`set_dr2(`*`val`*`)`: Sets the value in breakpoint address register 2 to *val*.

`set_dr3(`*`val`*`)`: Sets the value in breakpoint address register 3 to *val*.

`set_dr6(`*`val`*`)`: Sets the value in the debug status register to *val*.

`set_dr7(`*`val`*`)`: Sets the value in the debug control register to *val*.

`set_b0(`*`unsigned addr, unsigned len, unsigned rw`*`)`: Enables breakpoint register 0. Sets `dr0` to LINEAR address *addr* and updates `dr7` to enable it. *rw* must be DR7_RW_INST, DR7_RW_WRITE, DR7_RW_IO, or DR7_RW_DATA indicating the condition to break on. *len* must be DR7_LEN_1, DR7_LEN_2, or DR7_LEN_4, indicating how many bytes are covered by the register.

`set_b1(`*`unsigned addr, unsigned len, unsigned rw`*`)`: Enables breakpoint register 1.

`set_b2(`*`unsigned addr, unsigned len, unsigned rw`*`)`: Enables breakpoint register 2.

`set_b3(`*`unsigned addr, unsigned len, unsigned rw`*`)`: Enables breakpoint register 3.

10.3.5 `fp_reg.h`: Floating point register definitions and accessor functions

Synopsis

```
#include <oskit/x86/fp_reg.h>
```

Description

XXX

This file contains the structure definition for saving the floating-point state and then restoring it. It also contains definitions for the x87 control and status registers.

This header file is taken from CMU's Mach kernel.

10.3.6 `far_ptr.h`: Far (segment:offset) pointers

Synopsis

```
#include <oskit/x86/far_ptr.h>
```

Description

This contains struct definitions for creating "far pointers." Far pointers on the x86 are those that take an explicit segment in addition to the offset value.

`struct far_pointer_16`: 16-bit pointer structure which contains a 16-bit segment and a 16-bit offset. The address is computed as segment ¡¡ 4 + offset.

`struct far_pointer_32`: 48-bit pointer which contains a 32-bit offset and a 16-bit segment descriptor. Segmentation is used to determine what linear address is generated by these pointers.

10.3.7 `pio.h`: Programmed I/O functions

Synopsis

```
#include <oskit/x86/pio.h>
```

Description

These are macros for accessing IO-space directly on the x86. These instructions will generate traps if executed in user-mode without permissions (either IOPL in the eflags register or access via the io-bitmap in the tss).

`iodelay()`: A macro used to delay the processor for a short period of time, generally to wait until programmed io can complete. The actual amount of time is indeterminate, since the delay is accomplished by doing an `inb` from a nonexistent port, which depends on the processor and chipset. [1] The nominal delay value is 1uS for most machines.

`inl(`*`port`*`)`: Returns 32-bit value from *port*

`inw(`*`port`*`)`: Returns 16-bit value from *port*

`inb(`*`port`*`)`: Returns 8-bit value from *port*

`inl_p(`*`port`*`)`: *inl* followed immediately by *iodelay*

`inw_p(`*`port`*`)`: *inw* followed immediately by *iodelay*

`inb_p(`*`port`*`)`: *inb* followed immediately by *iodelay*

`outl(`*`port`*`, `*`val`*`)`: Send 32-bit *val* out *port.*

`outw(`*`port`*`, `*`val`*`)`: Send 16-bit *val* out *port.*

`outb(`*`port`*`, `*`val`*`)`: Send 8-bit *val* out *port.*

`outl_p(`*`port`*`)`: *outl* followed immediately by *iodelay*

`outw_p(`*`port`*`)`: *outw* followed immediately by *iodelay*

`outb_p(`*`port`*`)`: *outb* followed immediately by *iodelay*

The above macros have versions that begin with *i16_*, which are defined to be the same. It may be desirable to use the i16_ versions in 16-bit code in place of the normal macros for clarity.

This header file is taken from CMU's Mach kernel.

10.3.8 `seg.h`: Segment descriptor data structure definitions and constants

Synopsis

```
#include <oskit/x86/seg.h>
```

[1]The port used is 0x80, which was the page register for DMA channel 0 on the PC and PC/XT (A16-A19). The PC/AT and newer computers use port 0x87 for A16-A23 instead.

DESCRIPTION

XXX

`struct x86_desc`: Normal segment descriptors.

`struct x86_gate`: Trap, interrupt, and call gates.

`struct pseudo_descriptor`: Used to load the IDT and GDT (and LDT).

`sel_idx(sel)`: Converts the selector into an index in the descriptor table.

`ISPL(s)`: Returns the selector's privilege level.

`USERMODE(s, f)`:

`KERNELMODE(s, f)`:

`fill_descriptor(struct x86_desc *desc, unsigned base, unsigned limit, unsigned char access, unsigne`
Fill a segment descriptor.

`fill_descriptor_base(struct x86_desc *desc, unsigned base)`: Set the base address in a segment descriptor.

`fill_descriptor_limit(struct x86_desc *desc, unsigned limit)`: Set the limit in a segment descriptor.

`fill_gate(struct x86_gate *gate, unsigned offset, unsigned short selector, unsigned char access, un`
Fill an x86 gate descriptor.

This header file is based on a file in CMU's Mach kernel.

10.3.9 `gate_init.h`: Gate descriptor initialization support

SYNOPSIS

`#include <oskit/x86/gate_init.h>`

DESCRIPTION

This file contains the C structures and assembly-language macro definitions used to build x86 gate descriptor tables suitable for use by `gate_init` (see Section 10.5.10).

`struct gate_init_entry`: C structure describing a gate descriptor.

`GATE_INITTAB_BEGIN(name)`: Starts assembly-language definition of a gate descriptor table.

`GATE_ENTRY(n, entry, type)`: Initializes an element of a gate descriptor table.

`GATE_INITTAB_END`: Defines the end of a gate descriptor table.

The assembly-language macros are designed to be used while writing trap entrypoint routines. See `oskit/libkern/x86/base_trap_inittab.S` for example code that uses this facility.

10.3.10 `trap.h`: Processor trap vectors

SYNOPSIS

`#include <oskit/x86/trap.h>`

Description

XXX

This contains the definitions for the trap numbers returned by the processor when something goes ‘wrong’. These can be used to determine the cause of the trap.

`T_DIVIDE_ERROR`:

`T_DEBUG`:

`T_NMI`: non-maskable interrupt

`T_INT3`:

`T_OVERFLOW`: overflow test

`T_OUT_OF_BOUNDS`: bounds check

`T_INVALID_OPCODE`:

`T_NO_FPU`:

`T_DOUBLE_FAULT`:

`T_FPU_FAULT`:

`T_INVALID_TSS`:

`T_SEGMENT_NOT_PRESENT`:

`T_STACK_FAULT`:

`T_GENERAL_PROTECTION`:

`T_PAGE_FAULT`: T_PF_PROT: protection violation; T_PF_WRITE: write access; T_PF_USER: from user state

`T_FLOATING_POINT_ERROR`:

`T_ALIGNMENT_CHECK`:

`T_MACHINE_CHECK`:

This header file is taken from CMU’s Mach kernel.

10.3.11 `paging.h`: Page translation data structures and constants

Description

XXX

This header file is derived from Mach’s `intel/pmap.h`.

10.3.12 `tss.h`: Processor task save state structure definition

Synopsis

```
#include <oskit/x86/tss.h>

struct x86_tss {
int back_link; /* previous task's segment, if nested */
int esp0; /* initial stack pointer ... */
int ss0; /* and segment for ring 0 */
int esp1; /* initial stack pointer ... */
int ss1; /* and segment for ring 1 */
int esp2; /* initial stack pointer ... */
int ss2; /* and segment for ring 2 */
int cr3; /* CR3 - page table physical address */
int eip; /* eip */
int eflags; /* eflags */
int eax; /* eax */
int ecx; /* ecx */
int edx; /* edx */
int ebx; /* ebx */
int esp; /* current stack pointer (ring 3) */
int ebp; /* ebp */
int esi; /* esi */
int edi; /* edi */
int es; /* es */
int cs; /* cs */
int ss; /* current stack segment (ring 3) */
int ds; /* ds */
int fs; /* fs */
int gs; /* gs */
int ldt; /* local descriptor table segment */
unsigned short trace_trap; /* trap on switch to task */
unsigned short io_bit_map_offset; /* offset to IO perm bitmap */
};
```

Description

XXX

XXX only the 32-bit version

This contains the definition of a 32-bit tss. The tss used by the 80286 is incompatible with this.

This header file is taken from CMU's Mach kernel.

10.4 X86 PC Generic Low-level Definitions

XXX

This section covers useful macros, definitions, and routines for "PC"-specific features.

10.4.1 `irq_list.h`: Standard hardware interrupt assignments

Synopsis

```
#include <oskit/x86/pc/irq_list.h>
```

Description

XXX

Many of the interrupt vectors have pre-defined uses. The rest of them can be assigned to ISA, PCI, or other devices. This file contains the 'defined' interrupt usages.

10.4.2 `pic.h`: Programmable Interrupt Controller definitions

Synopsis

```
#include <oskit/x86/pc/pic.h>
```

Description

XXX

This contains definitions for the 8259(A) Programmable Interrupt Controller (PIC). In addition to numerous constants, it also contains the prototypes for several functions and macros.

`pic_init(unsigned char master_base, unsigned char slave_base)`: MASTER_PIC_BASE and SLAVES_PIC_BASE are also defined, and may be passed in as parameters.

`pic_disable_irq(unsigned char irq)`:

`pic_enable_irq(unsigned char irq)`:

`pic_test_irq(unsigned char irq)`:

`pic_enable_all`:

`pic_disable_all`:

`pic_ack(irq)`:

10.4.3 `keyboard.h`: PC keyboard definitions

Synopsis

```
#include <oskit/x86/pc/keyboard.h>
```

Description

XXX

This header file contains the register definitions for the PC keyboard. The port addresses are defined, along with the status and control bits. This would be used by a keyboard device driver, or someone manipulating they keyboard directly. (ie, to turn on and off the keyboard LEDs).

This header file is taken from CMU's Mach kernel.

10.4.4 `rtc.h`: NVRAM Register locations

Synopsis

```
#include <oskit/x86/pc/rtc.h>
```

Description

This file is taken from FreeBSD (XXX cite?) and contains definitions for the standard NVRAM, or Real Time Clock, register locations.

`rtcin(unsigned char addr)`: Returns the 8-bit value from location *addr*.

`rtcout(unsigned char addr, unsigned char val)`: Writes *val* to location *addr*.

10.5 X86 Processor Identification and Management

10.5.1 cpu_info: CPU identification data structure

SYNOPSIS

```
#include <oskit/x86/cpuid.h>
struct cpu_info {
   unsigned  stepping : 4;          /* Stepping ID          */
   unsigned  model : 4;             /* Model                */
   unsigned  family : 4;            /* Family               */
   unsigned  type : 2;              /* Processor type       */
   unsigned  feature_flags;         /* Features supported   */
   char      vendor_id[12];         /* Vendor ID string     */
   unsigned  char cache_config[16]; /* Cache information    */
};
```

DESCRIPTION

This structure is used to hold identification information about x86 processors, such as information returned by the `CPUID` instruction. The `cpuid` toolkit function, described below, fills in an instance of this structure with information about the current processor.

Note that it is expected that the `cpu_info` structure will continue to grow in the future as new x86-architecture processors are released, so client code should not depend on this structure in ways that will break if the structure's size changes.

The `family` field describes the processor family:

`CPU_FAMILY_386`: A 386-class processor.

`CPU_FAMILY_486`: A 486-class processor.

`CPU_FAMILY_PENTIUM`: A Pentium-class ("586") processor.

`CPU_FAMILY_PENTIUM_PRO`: A Pentium Pro-class ("686") processor.

The `type` field is one of the following:

`CPU_TYPE_ORIGINAL`: Original OEM processor.

`CPU_TYPE_OVERDRIVE`: OverDrive upgrade processor.

`CPU_TYPE_DUAL`: Dual processor.

The `feature_flags` field is a bit field containing the following bits:

`CPUF_ON_CHIP_FPU`: Set if the CPU has a built-in floating point unit.

`CPUF_VM86_EXT`: Set if the virtual 8086 mode extensions are supported, i.e., the `VIF` and `VIP` flags register bits, and the `VME` and `PVI` bits in `CR4`.

`CPUF_IO_BKPTS`: Set if I/O breakpoints are supported, i.e., the `DR7_RW_IO` mode defined in `x86/debug_reg.h`.

`CPUF_4MB_PAGES`: Set if 4MB superpages are supported, i.e., the `INTEL_PDE_SUPERPAGE` page directory entry bit defined in `x86/paging.h`.

`CPUF_TS_COUNTER`: Set if the on-chip timestamp counter and the `RDTSC` instruction are available.

`CPUF_PENTIUM_MSR`: Set if the Pentium model specific registers are available.

`CPUF_PAGE_ADDR_EXT`: Set if the Pentium Pro's page addressing extensions (36-bit physical addresses and 2MB pages) are available.

`CPUF_MACHINE_CHECK_EXCP`: Set if the processor supports the Machine Check exception (vector 18, or `T_MACHINE_CHECK` in `x86/trap.h`).

`CPUF_CMPXCHG8B`: Set if the processor supports the `CMPXCHG8B` instruction (also known as "double-compare-and-swap").

`CPUF_LOCAL_APIC`: Set if the processor has a built-in local APIC (Advanced Programmable Interrupt Controller), for symmetric multiprocessor support.

`CPUF_MEM_RANGE_REGS`: Set if the processor supports the memory type range registers.

`CPUF_PAGE_GLOBAL_EXT`: Set if the processor supports the global global paging extensions, i.e., the `INTEL_PDE_GLOBAL` page table entry bit defined in `x86/paging.h`.

`CPUF_MACHINE_CHECK_ARCH`: Set if the processor supports Intel's machine check architecture and the `MCG_CAP` model-specific register.

`CPUF_CMOVCC`: Set if the processor supports the `CMOV`*cc* instructions.

The `cpuid.h` header file also contains symbolic definitions for other constants such as the cache configuration descriptor values; see the header file and the Intel documentation for details on these.

10.5.2 `cpuid`: identify the current CPU

Synopsis

`#include <oskit/x86/cpuid.h>`

`void` **cpuid**([out] `struct cpu_info *`*out_info*);

Description

This function identifies the CPU on which it is running using Intel's recommended CPU identification procedure, and fills in the supplied structure with the information found.

Note that since the `cpuid` function is 32-bit code, it wouldn't run on anything less than an 80386 in the first place; therefore it doesn't bother to check for earlier processors.

Parameters

out_info: The CPU information structure to fill in.

10.5.3 `cpu_info_format`: output a cpu_info structure in ASCII form

Synopsis

`#include <oskit/x86/cpuid.h>`

`void` **cpu_info_format**(`struct cpu_info *`*info*, `void` *(*formatter)(void *data, const char *fmt, ...)*, `void *`*data*);

Description

This function takes the information in a `cpu_info` structure and formats it as human-readable text. The *formatter* should be a pointer to a `printf`-like function to be called to format the output data. The formatter function may be called multiple times to output all the relevant information.

Parameters

info: The filled-in CPU information structure to output.

formatter: The `printf`-style formatted output function to call. It will be called with the opaque *data* pointer provided, a standard C format string (*fmt*), and optionally a set of data items to format.

data: An opaque pointer which is simply passed on to the *formatter* function.

10.5.4 `cpu_info_min`: return the minimum feature set of two CPU information structures

Synopsis

`#include <oskit/x86/cpuid.h>`

`void` **cpu_info_min**(`struct cpu_info *`*id1*, `struct cpu_info *`*id2*);

Description

Determine the minimum (least-common-denominator) feature set of the two provided structures and return that. The new feature set is returned in *id1*.

Typically used on SMP systems to determine the basic feature set common across all processors in the system regardless of type.

Parameters

id1, id2: The CPU information structures to compare.

10.5.5 `cpu_info_dump`: pretty-print a CPU information structure to the console

Synopsis

`#include <oskit/x86/cpuid.h>`

`void` **cpu_info_dump**(`struct cpu_info *`*info*);

Description

This function is merely a convenient front-end to `cpu_info_format`; it simply formats the CPU information and outputs it to the console using `printf`.

10.5.6 `i16_enter_pmode`: enter protected mode

Synopsis

`#include <oskit/x86/pmode.h>`

`void` **i16_enter_pmode**(`int` *prot_cs*);

Description

This 16-bit function switches the processor into protected mode by turning on the Protection Enable (PE) bit in CR0. The instruction that sets the PE bit is followed immediately by a jump instruction to flush the prefetch buffer, as recommended by Intel documentation.

The function also initializes the CS register with the appropriate new protected-mode code segment, whose selector is specified in the *prot_cs* parameter. The *prot_cs* must evaluate to a constant, as it is used as an immediate operand in an inline assembly language code fragment.

This routine does not perform any of the other steps in Intel's recommended mode switching procedure, such as setting up the GDT or reinitializing the data segment registers; these steps must be performed separately. The overall mode switching sequence is necessarily much more dependent on various OS-specific factors such as the layout of the GDT; therefore the OSKit does not attempt to provide a "generic" function to perform the entire switch. Instead, the full switching sequence is provided as part of the base environment setup code; see Section 10.10 for more details.

10.5.7 `i16_leave_pmode`: leave protected mode

Synopsis

`#include <oskit/x86/pmode.h>`

`void` **i16_leave_pmode**(`int` *real_cs*);

Description

This 16-bit function switches the processor out of protected mode and back into real mode by turning off the Protection Enable (PE) bit in CR0. The instruction that clears the PE bit is followed immediately by a jump instruction to flush the prefetch buffer, as recommended by Intel documentation. At the same time, this function also initializes the CS register with the appropriate real-mode code segment, specified by the *real_cs* parameter.

This routine does not perform any of the other steps in Intel's recommended mode switching procedure, such as reinitializing the data segment registers; these steps must be performed separately. See Section 10.10 for information on the full mode switch implementation provided by the base environment.

10.5.8 `paging_enable`: enable page translation

Synopsis

`#include <oskit/x86/paging.h>`

`void` **paging_enable**(`oskit_addr_t` *pdir*);

Description

Loads the processor page directory using `pdir` and turns on paging.

The caller must already have created and initialized an appropriate initial page directory as described in Intel documentation. The OSKit provides convenient facilities that can be used to create x86 page directories and page tables; for more information, see Section 10.9.

This function assumes that `pdir` equivalently maps the physical memory that contains the currently executing code, the currently loaded GDT and IDT.

10.5.9 `paging_disable`: disable page translation

Synopsis

```
#include <oskit/x86/paging.h>
void paging_disable(void);
```

Description

Turns paging off and flushes the TLB.

This function assumes that the currently loaded page directory equivalently maps the physical memory that contains the currently executing code, the currently loaded GDT and IDT.

10.5.10 `gate_init`: install gate descriptors

Synopsis

```
#include <oskit/x86/gate_init.h>
void gate_init(struct x86_gate *dest, const struct gate_init_entry *src, unsigned entry_cs);
```

Description

Install entries in a processor descriptor table from the specified array of gate descriptors (see Section 10.3.9). Typically used to initialize the processor IDT with trap and interrupt vectors (see Section 10.7.4).

Parameters

dest: Pointer to the x86 descriptor table to fill in.

src: Pointer to the `gate_init_entry` array to copy from.

entry_cs: Code segment selector to associate with all entries.

Dependencies

`fill_gate`: 10.3.8

10.6 X86 Base Environment

The base environment code for the x86 architecture is designed to assist the OS developer in dealing with much of the "x86 grunge" that OS developers typically would rather not worry about. The OSKit provides easy-to-use primitives to set up and maintain various common flavors of x86 kernel environments without unnecessarily constraining the OS implementation. The base environment support on the x86 architecture is divided into the three main categories: segmentation, paging, and trap handling. The base environment support code in each category is largely orthogonal and easily separable, although it is also designed to work well together.

10.6.1 Memory Model

The x86 architecture supports a very complex virtual memory model involving both segmentation and paging; one of the goals of the OSKit's base environment support for the x86 is to smooth over some of this complexity, hiding the details that the OS doesn't want to deal with while still allowing the OS full freedom to use the processor's virtual memory mechanisms as it sees fit. This section describes the memory models supported and assumed by the base environment.

First, here is a summary of several important terms that are used heavily used in the following text; for full details on virtual, linear, and physical addresses on the x86 architecture, see the appropriate processor manuals.

- *Physical addresses* are the actual addresses seen on external I/O and memory busses, after segmentation and paging transformations have been applied.
- *Linear addresses* are absolute 32-bit addresses within the x86's paged address space, after segmentation has been applied but before page translation. The virtual addresses of simple "paging-only" architectures such as MIPS correspond to linear addresses on the x86.
- *Virtual addresses* are the logical addresses used by program code to access memory. To read an instruction or access a data item, the processor first converts the virtual address into a linear address using the segmentation mechanism, then translates the linear address to a physical address using paging.
- *Kernel virtual addresses* are the virtual addresses normally used by kernel code to access its own functions and data structures: in other words, addresses accessed through the kernel's segment descriptors.

The OSKit provides a standard mechanism, defined in `base_vm.h` (see Section 10.6.2), which is used throughout the base environment to maintain considerable independence from the memory model in effect. These facilities allow the base environment support code to avoid various assumptions about the relationships between kernel virtual addresses, linear addresses, and physical addresses. Client OS code can use these facilities as well if desired.

Of course, it is impractical for the base environment code to avoid assumptions about the memory model completely. In particular, the code assumes that, for "relevant" code and data (e.g., the functions implementing the base environment and the data structures they manipulate), kernel virtual addresses can be converted to and from linear or physical addresses by adding or subtracting an offset stored in a global variable. However, the code does *not* assume that these offsets are always the same (the client OS is allowed to change them dynamically), or that *all* available physical memory is mapped into the kernel's virtual address space, or that all linear memory is accessible through the kernel's data segment descriptors. Detailed information about the memory model assumptions made by particular parts of the base environment support are documented in the appropriate API sections.

If the OSKit's startup code is being used to start the OS, then the specific memory model in effect initially depends on the startup environment, described in later the appropriate sections. For example, for kernels booted from a MultiBoot boot loader, in the initial memory environment virtual addresses, linear addresses, and physical addresses are all exactly equal (the offsets are zero). On the other hand, for kernels loaded from DOS, linear addresses and physical addresses will still be equal but kernel virtual addresses will be at some offset depending on where in physical memory the kernel was loaded. Regardless of the initial memory setup, the client OS is free to change the memory model later as necessary.

XXX example memory maps
XXX explain how to change memory models at run-time

10.6.2 `base_vm.h`: definitions for the base virtual memory environment

Synopsis

```
#include <oskit/machine/base_vm.h>
```

Description

This header file provides generic virtual memory-related definitions commonly used throughout the base environment, which apply to both segmentation and paging. In particular, this file defines a set of macros and global variables which allow the rest of the base environment code in the toolkit (and the client OS, if it chooses) to maintain independence from the memory model in effect. These facilities allow code to avoid various assumptions about the relationships between kernel virtual addresses, linear addresses, and physical addresses.

The following variable and associated macros are provided to convert between linear and kernel virtual addresses.

`linear_base_va`: This global variable defines the address in kernel virtual memory that corresponds to address 0 in linear memory. It is used by the following conversion macros; therefore, changing this variable changes the behavior of the associated macros.

`lintokv(`*la*`)`: This macro converts linear address *la* into a kernel virtual address and returns the result as an `oskit_addr_t`.

`kvtolin(`*va*`)`: For example, the segmentation initialization code uses `kvtolin()` to calculate the linear addresses of segmentation structures to be used in segment descriptor or pseudo-descriptor structures provided to the processor.

Similarly, the following variable and associated macros convert between *physical* and kernel virtual addresses. (Conversions between linear and physical addresses can be done by combining the two sets of macros.)

`phys_mem_va`: This global variable defines the address in kernel virtual memory that corresponds to address 0 in physical memory. It is used by the following conversion macros; therefore, changing this variable changes the behavior of the associated macros.

`phystokv(`*pa*`)`: This macro converts physical address *pa* into a kernel virtual address and returns the result as an `oskit_addr_t`. The macro makes the assumption that the specified physical address *can* be converted to a kernel virtual address this way: in OS kernels that do not direct-map all physical memory into the kernel's virtual address space, the caller must ensure that the supplied *pa* refers to a physical address that *is* mapped. For example, the primitive page table management code provided by the OSKit's base environment uses this macro to access page table entries given the physical address of the page table; therefore, these functions can only be used if page tables are allocated from physical pages that are direct-mapped into the kernel's address space.

`kvtophys(`*va*`)`: This macro converts kernel virtual address *va* into a physical address and returns the result as an `oskit_addr_t`. The macro assumes that the virtual address *can* be converted directly to a physical address this way; the caller must ensure that this is the case. For example, some operating systems only direct-map the kernel's code and statically allocated data; in such kernels, *va* should only refer to statically-allocated variables or data structures. This is generally sufficient for the OSKit's base environment code, which mostly operates on statically-allocated data structures; however, the OS must of course take its chosen memory model into consideration if it uses these macros as well.

XXX real_cs

Note that there is nothing in this header file that defines or relates to "user-mode" address spaces. This is because the base environment code in the OSKit is not concerned with user mode in any way; in fact, it doesn't even care whether or not the OS kernel implements user address spaces at all. For example, boot loaders or unprotected real-time kernels built using the OSKit probably do not need any notion of user mode at all.

10.6.3 `base_cpu_setup`: initialize and activate the base CPU environment

Synopsis

```
#include <oskit/machine/base_cpu.h>
void base_cpu_setup(void);
```

Description

This function provides a single entrypoint to initialize and activate all of the processor structures necessary for ordinary execution. This includes identifying the CPU, and initializing and activating the base GDT, IDT, and TSS, and reloading all segment registers as recommended by Intel. The call returns with the CS segment set to `KERNEL_CS` (the default kernel code segment; see 10.7.1 for details), DS, ES, and SS set to `KERNEL_DS` (the default kernel data segment), and FS and GS set to 0. After the `base_cpu_setup` call completes, a full working kernel environment is in place: segment registers can be loaded, interrupts and traps can be fielded by the OS, privilege level changes can occur, etc.

This function does *not* initialize or activate the processor's paging mechanism, since unlike the other mechanisms, paging is optional on the x86 and not needed in some environments (e.g., boot loaders or embedded kernels).

The `base_cpu_setup` function is actually just a simple wrapper that calls `base_cpu_init` followed by `base_cpu_load`.

Note that it is permissible to call this function (and/or the more primitive functions it is built on) more than once. This is particularly useful when reconfiguring the kernel memory map. For example, a typical MultiBoot (or other 32-bit) kernel generally starts out with paging disabled, so it must run in the low range of linear/physical memory. However, after enabling page translation, the OS may later want to relocate itself to run at a higher address in linear memory so that application programs can use the low part (e.g., v86-mode programs). An easy way to do this with the OSKit is to call `base_cpu_setup` once at the very beginning, to initialize the basic unpaged kernel environment, and then later, after paging is enabled and appropriate mappings have been established in high linear address space, modify the `linear_base_va` variable (Section 10.6.2) to reflect the kernel's new linear address base, and finally call `base_cpu_setup` again to reinitialize and reload the processor tables according to the new memory map.

Dependencies

`base_cpu_init`:	10.6.4
`base_cpu_load`:	10.6.5

10.6.4 `base_cpu_init`: initialize the base environment data structures

Synopsis

```
#include <oskit/machine/base_cpu.h>
void base_cpu_init(void);
```

Description

This function initializes all of the critical data structures used by the base environment, including `base_cpuid`, `base_idt`, `base_gdt`, and `base_tss`, but does not actually activate them or otherwise modify the processor's execution state. The `base_cpu_load` function must be called later to initialize the processor with these structures. Separate initialization and activation functions are provided to allow the OS to customize the processor data structures if necessary before activating them.

Dependencies

`cpuid`: 10.5.2

`base_trap_init`: 10.8.2

`base_gdt_init`: 10.7.2

`base_tss_init`: 10.7.7

10.6.5 `base_cpu_load`: activate the base processor execution environment

Synopsis

`#include <oskit/machine/base_cpu.h>`

`void` **base_cpu_load**(`void`);

Description

This function loads the critical base environment data structures (in particular, the GDT, IDT, and TSS) into the processor, and reinitializes all segment registers from the new GDT as recommended in Intel processor documentation. The structures must already have been set up by a call to `base_cpu_init` and/or custom initialization code in the client OS.

This function returns with the CS segment set to `KERNEL_CS` (the default kernel code segment; see Section 10.7.1 for details), DS, ES, and SS set to `KERNEL_DS` (the default kernel data segment), and FS and GS set to 0. After the `base_cpu_load` call completes, a full working kernel environment is in place: segment registers can be loaded, interrupts and traps can be fielded by the OS, privilege level changes can occur, etc.

Dependencies

`base_gdt_load`: 10.7.3

`base_idt_load`: 10.7.5

`base_tss_load`: 10.7.8

10.6.6 `base_cpuid`: global variable describing the processor

Synopsis

`#include <oskit/machine/base_cpu.h>`

`extern struct cpu_info` **base_cpuid**;

Description

This is a global variable that is filled in by `base_cpu_init` with information about the processor on which `base_cpu_init` was called. (Alternatively, it can also be initialized manually by the OS simply by calling `cpuid(&base_cpuid)`). This structure is used by other parts of the kernel support library to determine whether or not certain processor features are available, such as 4MB superpages. See 10.5.1 for details on the contents of this structure.

Note that in a multiprocessor system, this variable will reflect the boot processor. This is generally not a problem, since most SMPs use identical processors, or at least processors in the same generation, so that they appear equivalent to OS software. (For example, it is very unlikely that you'd find an SMP that mixes 486 and Pentium processors), However, if this ever turns out to be a problem, the OS can always override the `cpuid` or `base_cpu_init` function, or just modify the contents of the `base_cpuid` variable after calling `base_cpu_init` so that it reflects the least common denominator of all the processors.

10.6.7 `base_stack.h`: default kernel stack

Synopsis

`#include <oskit/machine/base_stack.h>`

Description

Definitions related to the default kernel stack.

`BASE_STACK_SIZE`: Preprocessor constant defining the size in bytes of the default kernel stack.

`base_stack_start`: C external variable declaration for the low-address end of the stack. On the x86, this is the end toward which the stack grows.

`base_stack_end`: C external variable declaration for the high-address end of the stack (`base_stack_start` + `BASE_STACK_SIZE`). On the x86, this is the end where the stack begins.

This header file can be used in assembly language code as well as C.

10.7 X86 Base Environment: Segmentation Support

Although most modern operating systems use a simple "flat" address space model, the x86 enforces a segmentation model which cannot be disabled directly; instead, it must be set up to emulate a flat address space model if that is what the OS desires. The base environment code provides functionality to set up a simple flat-model processor environment suitable for many types of kernels, both "micro" and "macro." For example, it provides a default global descriptor table (GDT) containing various flat-model segments for the kernel's use, as well as a default task state segment (TSS).

Furthermore, even though this base environment is often sufficient, the client OS is not limited to using it exactly as provided by default: the client kernel is given the flexibility to tweak various parameters, such as virtual and linear memory layout, as well as the freedom to operate completely outside of the base environment when necessary. For example, although the base environment provides a default TSS, the OS is free to create its own TSS structures and use them when running applications that need special facilities such as v86 mode or I/O ports. Alternatively, the OS could use the default processor data structures only during startup, and switch to its own complete, customized set after initialization.

The base environment code in the OSKit generally assumes that it is running in a simple flat model, in which only one code segment and one data segment are used for all kernel code and data, respectively, and that the code and data segments are synonymous (they each map to the same range of linear addresses). The OS is free to make more exotic uses of segmentation if it so desires, as long as the OSKit code is run in a consistent environment.

XXX diagram of function call tree?

The base segmentation environment provided by the OSKit is described in more detail in the following API sections.

10.7.1 base_gdt: default global descriptor table for the base environment

Synopsis

```
#include <oskit/x86/base_gdt.h>
extern struct x86_desc base_gdt[GDTSZ];
```

Description

This variable is used in the base environment as the default global descriptor table. The default `base_gdt` definition contains `GDTSZ` selector slots, including the Intel-reserved, permanently unused slot 0.

The following symbols are defined in `base_gdt.h` to be segment selectors for the descriptors in the base GDT. These selectors can be converted to indices into the GDT descriptor array `base_gdt` by dividing by 8 (the processor reserves the low three bits of all selectors for other information).

`BASE_TSS`: A selector for the base task state segment (`base_tss`). The `BASE_TSS` segment descriptor is initialized by `base_gdt_init`, but the `base_tss` structure itself is initialized by `base_tss_init` and loaded into the processor by `base_tss_load`; see Section 10.7.6 for more details.

`KERNEL_CS`: This is the default kernel code segment selector. It is initialized by `base_gdt_init` to be a flat-model, 4GB, readable, ring 0 code segment; `base_gdt_load` loads this segment into the CS register while reinitializing the processor's segment registers.

`KERNEL_DS`: This is the default kernel data segment selector. It is initialized by `base_gdt_init` to be a flat-model, 4GB, writable, ring 0 data segment; `base_gdt_load` loads this segment into the DS, ES, and SS registers while reinitializing the processor's segment registers.

`KERNEL_16_CS`: This selector is identical to `KERNEL_CS` except that it is a 16-bit code segment (the processor defaults to 16-bit operand and addressing modes rather than 32-bit while running code in this segment), and it has a 64KB limit rather than 4GB. This selector is used when switching between real and protected mode, to provide an intermediate 16-bit protected mode execution context. It is unused in kernels that never execute in real mode (e.g., typical MultiBoot kernels).

`KERNEL_16_DS`: This selector is a data segment synonym for `KERNEL_16_CS`; it is generally only used when switching from protected mode back to real mode. It is used to ensure that the segment registers contain sensible real-mode values before performing the switch, as recommended in Intel literature.

`LINEAR_CS`: This selector is set up to be a ring 0 code segment that directly maps the entire linear address space: in other words, it has an offset of zero and a 4GB limit. In some environments, where kernel virtual addresses are the same as linear addresses, this selector is a synonym for `KERNEL_CS`.

`LINEAR_DS`: This is a data segment otherwise identical to `LINEAR_CS`.

`USER_CS`: This selector is left unused and uninitialized by the OSKit; nominally, it is intended to be used as a code segment for unprivileged user-level code.

`USER_DS`: This selector is left unused and uninitialized by the OSKit; nominally, it is intended to be used as a data segment for unprivileged user-level code.

If the client OS wants to make use of the base GDT but needs more selector slots for its own purposes, it can define its own instance of the `base_gdt` variable so that it has room for more than `GDTSZ` elements; `base_gdt_init` will initialize only the first "standard" segment descriptors, leaving the rest for the client OS's use.

On multiprocessor systems, the client OS may want each processor to have its own GDT. In this case, the OS can create a separate clone of the base GDT for each additional processor besides the boot processor, and leave the boot processor using the base GDT. Alternatively, the OS could use the base GDT only during initialization, and switch *all* processors to custom GDTs later; this approach provides the most flexibility to the OS, since the custom GDTs can be arranged in whatever way is most convenient.

10.7.2 `base_gdt_init`: initialize the base GDT to default values

SYNOPSIS

`#include <oskit/x86/base_gdt.h>`

`void` **base_gdt_init**(`void`); `void` **i16_base_gdt_init**(`void`);

DESCRIPTION

This function initializes the standard descriptors in the base GDT as described in Section 10.7.1.

For all of the standard descriptors except `LINEAR_CS` and `LINEAR_DS`, the `kvtolin` macro is used to compute the linear address to plug into the offset field of the descriptor: for `BASE_TSS`, this is `kvtolin(&base_tss)`; for the kernel code and data segments, it is `kvtolin(0)` (i.e., the linear address corresponding to the beginning of kernel virtual address space). `LINEAR_CS` and `LINEAR_DS` are always given an offset of 0.

A 16-bit version of this function, `i16_base_gdt_init`, is also provided so that the GDT can be initialized properly before the processor has been switched to protected mode. (Switching to protected mode on the x86 according to Intel's recommended procedure requires a functional GDT to be already initialized and activated.)

DEPENDENCIES

`fill_descriptor`: 10.3.8

`kvtolin`: 10.6.2

`base_gdt`: 10.7.1

`base_tss`: 10.7.6

10.7.3 `base_gdt_load`: load the base GDT into the CPU

Synopsis

`void` **base_gdt_load**(`void`); `void` **i16_base_gdt_load**(`void`);

Description

This function loads the base GDT into the processor's GDTR, and then reinitializes all segment registers from the descriptors in the newly loaded GDT. It returns with the CS segment set to `KERNEL_CS` (the default kernel code segment; see Section 10.7.1 for details), DS, ES, and SS set to `KERNEL_DS` (the default kernel data segment), and FS and GS set to 0.

Dependencies

`kvtolin`: 10.6.2

`base_gdt`: 10.7.1

10.7.4 `base_idt`: default interrupt descriptor table

Synopsis

`#include <oskit/x86/base_idt.h>`

`extern struct x86_desc` **base_idt**`[IDTSZ];`

Description

This global variable is used in the base environment as the default interrupt descriptor table. The default definition of `base_idt` in the library contains the architecturally-defined maximum of 256 interrupt vectors (`IDTSZ`).[2]

The `base_idt.h` header file does not define any symbols representing interrupt vector numbers. The lowest 32 vectors are the processor trap vectors defined by Intel; since these are not specific to the base environment, they are defined in the generic header file `x86/trap.h` (see Section 10.3.10). Standard hardware interrupt vectors are PC-specific, and therefore are defined separately in `x86/pc/irq_list.h` (see Section 10.4.1). For the same reason, there is no `base_idt_init` function, only separate functions to initialize the trap vectors in the base IDT (`base_trap_init`, Section 10.8.2), and hardware interrupt vectors in the IDT (`base_irq_init`, Section 10.12.3).

10.7.5 `base_idt_load`: load the base IDT into the current processor

Synopsis

`#include <oskit/x86/base_idt.h>`

`void` **base_idt_load**(`void`);

[2]**Rationale:** Although simple x86 PC kernels often only use the 32 processor trap vectors plus 16 interrupt vectors, *which* set of vectors are used for hardware interrupts tends to differ greatly between kernels. Some kernels also want to use well-known vectors for efficient system call emulation, such as 0x21 for DOS or 0x80 for Linux. Some bootstrap mechanisms, such as VCPI on DOS, must determine at run-time the set of vectors used for hardware interrupts, and therefore potentially need all 256 vectors to be available. Finally, making use of the enhanced interrupt facilities on Intel SMP Standard-compliant multiprocessors generally requires use of higher vector numbers, since vector numbers are tied to interrupt priorities. For all these reasons, we felt the default IDT should be of the maximum size, even though much of it is usually wasted.

Description

This function loads the `base_idt` into the processor, so that subsequent traps and hardware interrupts will vector through it. It uses the `kvtolin` macro to compute the proper linear address of the IDT to be loaded into the processor.

Dependencies

`kvtolin`: 10.6.2

`base_idt`: 10.7.4

10.7.6 `base_tss`: default task state segment

Synopsis

```
#include <oskit/x86/base_tss.h>
extern struct x86_tss base_tss;
```

Description

The `base_tss` variable provides a default task state segment that the OS can use for privilege level switching if it does not otherwise use the x86's task switching mechanisms. The x86 architecture requires every protected-mode OS to have at least one TSS even if no task switching is done; however, many x86 kernels do not use the processor's task switching features because it is faster to context switch manually. Even if special TSS segments are used sometimes (e.g., to take advantage of the I/O bitmap feature when running MS-DOS programs), the OS can still use a common TSS for all tasks that do not need to use these special features; this is the strategy taken by the Mach kernel, for example. The `base_tss` provided by the toolkit serves in this role as a generic "default" TSS.

The `base_tss` is a minimal TSS, in that it contains no I/O bitmap or interrupt redirection map. XXX The toolkit also supports an alternate default TSS with a full I/O permission bitmap, but it isn't fully integrated or documented yet.

10.7.7 `base_tss_init`: initialize the base task state segment

Synopsis

```
#include <oskit/x86/base_tss.h>
void base_tss_init(void);
```

Description

The `base_tss_init` function initializes the `base_tss` to a valid minimal state. It sets the I/O permission bitmap offset to point past the end of the TSS, so that it will be interpreted by the processor as empty (no permissions for any I/O ports). It also initializes the ring 0 stack segment selector (`ss0`) to `KERNEL_DS`, and the ring 0 stack pointer (`esp0`) to the current stack pointer value at the time of the function call, to provide a minimal working context for trap handling. Once the OS kernel sets up a "real" kernel stack, it should reinitialize `base_tss.esp0` to point to that.

Dependencies

`base_tss`: 10.7.6

10.7.8 `base_tss_load`: load the base TSS into the current processor

Synopsis

```
#include <oskit/x86/base_tss.h>
void base_tss_load(void);
```

Description

This function activates the `base_tss` in the processor using the `LTR` instruction, after clear the busy bit in the `BASE_TSS` segment descriptor to ensure that a spurious trap isn't generated.

Dependencies

`base_gdt`: 10.7.1

10.8 X86 Base Environment: Trap Handling

The first 32 vectors in the IDT are used to handle processor exceptions ("traps"). In the base OSKit environment, these vectors are initialized from the `base_trap_inittab` array (10.8.3) using the `base_trap_init` function (10.8.2). By default, each exception vector in the processor IDT is set to point to a common assembly language stub that saves a standard trap frame (10.8.1), and calls a designated high-level handler specified in the `base_trap_handlers` table (10.8.4). Initially, all the entries in this table point to `base_trap_default_handler` (10.8.5). Custom trap handlers can be installed by changing the appropriate entry in the table. The default action for all traps can be changed by overriding `base_trap_default_handler`.

This affords client OSes with a variety of choices for modifying the behavior of trap handling. By using the base trap environment unchanged (i.e., the client OS is not expecting or handling traps), all traps will produce a trap dump and panic. This behavior is sufficient for most simple OSKit applications. By setting entries `base_trap_handlers`, the client can provide its own C language trap handlers while still using the default `trap_state` structure. The OSKit remote GDB debugging package (10.17.5) does this. Finally, the client OS can override `base_trap_inittab` to allow for different high-level handlers for every exception type and/or to permit the use of a different trap state format.

10.8.1 `trap_state`: saved state format used by the default trap handler

Synopsis

```
#include <oskit/x86/base_trap.h>

struct trap_state
{
/* Saved segment registers */
unsigned int    gs;
unsigned int    fs;
unsigned int    es;
unsigned int    ds;

/* PUSHA register state frame */
unsigned int    edi;
unsigned int    esi;
unsigned int    ebp;
unsigned int    cr2; /* we save cr2 over esp for page faults */
unsigned int    ebx;
unsigned int    edx;
unsigned int    ecx;
unsigned int    eax;

/* Processor trap number, 0-31.  */
unsigned int    trapno;

/* Error code pushed by the processor, 0 if none.  */
unsigned int    err;

/* Processor state frame */
unsigned int    eip;
unsigned int    cs;
unsigned int    eflags;
unsigned int    esp;
unsigned int    ss;

/* Virtual 8086 segment registers */
unsigned int    v86_es;
unsigned int    v86_ds;
unsigned int    v86_fs;
unsigned int    v86_gs;
};
```

DESCRIPTION

This structure defines the saved state frame pushed on the stack by the default trap entrypoints provided by the base environment (see Section 10.8.3). It is also used by the `trap_dump` routine, which is used in the default environment to dump the saved register state and panic if an unexpected trap occurs; and by `gdb_trap`, the default trap handler for remote GDB debugging.

This client OS is not obligated to use this structure as the saved state frame for traps it handles; if this structure is not used, then the OS must also override (or not use) the dependent routines mentioned above.

The structure elements from `err` down corresponds to the basic trap frames pushed on the stack by the x86 processor. (For traps in which the processor does not push an error code, the default trap entrypoint code sets `err` to zero.) The structure elements from `esp` down are only pushed by traps from lower privilege (rings 1–3), and the structure elements from `v86_es` down are only pushed by traps from v86 mode.

The rest of the state frame is pushed manually by the default trap entrypoint code. The saved integer register state is organized in a format compatible with the processor's `PUSHA` instruction. However, in the slot that would otherwise hold the pushed `ESP` (which is useless since it is the trap handler's stack pointer rather than the trapping code's stack pointer), the default trap handler saves the `CR2` register (page fault linear address) during page faults.

This trap state structure is borrowed from Mach.

10.8.2 `base_trap_init`: initialize the processor trap vectors in the base IDT

SYNOPSIS

`#include <oskit/x86/base_trap.h>`

`void` **base_trap_init**(`void`);

DESCRIPTION

This function initializes the processor trap vectors in the base IDT to the default trap entrypoints defined in `base_trap_inittab`.

DEPENDENCIES

`gate_init`: 10.5.10

`base_idt`: 10.7.4

`base_trap_inittab`: 10.8.3

10.8.3 `base_trap_inittab`: initialization table for the default trap entrypoints

SYNOPSIS

`#include <oskit/x86/base_trap.h>`

`extern struct gate_init_entry` **base_trap_inittab**`[];`

Description

This gate initialization table (see Section 10.3.9) encapsulates the base environment's default trap entrypoint code. This module provides IDT entrypoints for all of the processor-defined trap vectors; each entrypoint pushes a standard state frame on the stack (see Section 10.8.1), and then calls the C function pointed to by the corresponding entry in `base_trap_handlers` array (see Section 10.8.4). Through these entrypoints, the OSKit provides the client OS with a convenient, uniform method of handling all processor traps in ordinary high-level C code.

If a trap occurs and the trap entrypoint code finds that the corresponding entry in `base_trap_handlers` is null, or if it points to a handler routine but the handler returns a nonzero value indicating failure, the entrypoint code calls `trap_dump_panic` (see Section 10.8.7) to dump the register state to the console and panic the kernel. This behavior is typically appropriate in kernels that do not expect traps to occur during proper operation (e.g., boot loaders or embedded operating systems), where a trap probably indicates a serious software bug.

On the other hand, if a trap handler *is* present and returns success (zero), the entrypoint code restores the saved state and resumes execution of the trapping code. The trap handler may change the contents of the `trap_state` structure passed by the entrypoint code; in this case, final contents of the structure on return from the trap handler will be the state restored.

All of the IDT entries initialized by the `base_trap_inittab` are trap gates rather than interrupt gates; therefore, if hardware interrupts are enabled when a trap occurs, then interrupts will still be enabled during the trap handler unless the trap handler explicitly disables them. If the OS wants interrupts to be disabled during trap handling, it can change the processor trap vectors in the IDT (vectors 0–31) into interrupt gates, or it can simply use its own trap entrypoint code instead.

Dependencies

`struct trap_state`: 10.8.1

`base_trap_handlers`: 10.8.4

`trap_dump_panic`: 10.8.7

10.8.4 `base_trap_handlers`: Array of handler routines for hardware traps

Synopsis

```
#include <oskit/x86/base_trap.h>
void (*base_trap_handlers[BASE_TRAP_COUNT]) (struct trap_state *ts);
```

Description

Contains a function pointer for every hardware trap vector. By default, all entries in this table point to `base_trap_default_handler`, which will simply dump the register state to the console and panic. The client OS can set entries in this table to point to its own trap handler function(s), or to alternative trap handlers supplied by the OSKit, such as the remote GDB debugging trap handler, `gdb_trap` (see Section 10.17.5).

Parameters

state: A pointer to the trap state structure to dump.

Returns

The trap handler returns zero (success) to resume execution, or nonzero (failure) to cause the entrypoint code to dump the register state and panic the kernel.

10.8.5 `base_trap_default_handler`: default trap handler for unexpected traps

Synopsis

`#include <oskit/x86/pc/base_trap.h>`

`void` **base_trap_default_handler**(`struct trap_state *`*state*);

Description

This routine is the default handler for all traps in the base environment. It simply displays displays the trap state information and then calls panic.

It is expected that the client OS will override entries in the `base_trap_handlers` array for traps it cares about. Alternatively, the client OS may override the `base_trap_default_handler` entrypoint entirely.

Parameters

state: A pointer to the processor state at the time of the interrupt.

Dependencies

`struct trap_state`: 10.8.1

10.8.6 `trap_dump`: dump a saved trap state structure

Synopsis

`#include <oskit/x86/base_trap.h>`

`void` **trap_dump**(`const struct trap_state *`*state*);

Description

This function dumps the contents of the specified trap state frame to the console using the `printf` function, in a simple human-readable form. The function is smart enough to determine whether the trap occurred from supervisor mode, user mode, or v86 mode, and interpret the saved state accordingly. For example, for traps from rings 1–3 or from v86 mode, the the original stack pointer is part of the saved state frame; however, for traps from ring 0, the original stack pointer is simply the end of the stack frame pushed by the processor, since no stack switch occurs in this case.

In addition, for traps from ring 0, this routine also provides a hex dump of the top of the kernel stack as it appeared when the trap occurred; this stack dump can aid in tracking down the cause of a kernel bug. `trap_dump` does not attempt to dump the stack for traps from user or v86 mode, because there seems to be no sufficiently generic way for it to access the appropriate user stack; in addition, in this case the trap might have been *caused* by a user-stack-related exception, in which case attempting to dump the user stack could lead to a recursive trap.

Parameters

state: A pointer to the trap state structure to dump.

Dependencies

`struct trap_state`: 10.8.1

`printf`: 9.6

10.8.7 `trap_dump_panic`: dump a saved trap state structure

SYNOPSIS

`#include <oskit/x86/base_trap.h>`

`void` **trap_dump_panic**(`const struct trap_state *`*state*);

DESCRIPTION

This function simply calls `trap_dump` (Section 10.8.6) to dump the specified trap state frame, and then calls `panic` (Section 9.8.3). It is invoked by the default trap entrypoint code (Section 10.8.3) if a trap occurs when there is no interrupt handler, or if there is an interrupt handler but it returns a failure indication.

PARAMETERS

state: A pointer to the trap state structure to dump.

DEPENDENCIES

`trap_dump`: 10.8.6

`panic`: 9.8.3

10.9 X86 Base Environment: Page Translation

XXX diagram of function call tree?

XXX Although a "base" x86 paging environment is defined, it is not automatically initialized by `base_cpu_init`, and paging is not activated by `base_cpu_load`. This is because unlike segmentation, paging is an optional feature on the x86 architecture, and many simple "kernels" such as boot loaders would prefer to ignore it completely. Therefore, client kernels that *do* want the base paging environment must call the functions to initialize and activate it manually, after the basic CPU segmentation environment is set up.

XXX describe assumptions made about use of page tables, e.g. 4MB pages whenever possible, always modify/unmap _exactly_ the region that was mapped.

XXX assumes that mappings are only changed or unmapped with the same size and offset as the original mapping.

XXX does not attempt to support page table sharing in any way, since this code has no clue about the relationship between address spaces; it only knows about page directories and page tables.

10.9.1 `base_paging_init`: create minimal kernel page tables and enable paging

SYNOPSIS

`#include <oskit/x86/base_paging.h>`

`void` **base_paging_init**(`void`);

DESCRIPTION

This function can be used to set up a minimal paging environment. It first allocates and clears an initial page directory using `ptab_alloc` (see Section 10.9.7), sets `base_pdir_pa` to point to it (see Section 10.9.2), then direct-maps all known physical memory into this address space starting at linear address 0, allocating additional page tables as needed. Finally, this function enables the processor's paging mechanism, using the base page directory as the initial page directory.

The global variable `phys_mem_max` (see Section 10.11.2) is assumed to indicate the top of physical memory; all memory from 0 up to *at least* this address is mapped. The function actually rounds `phys_mem_max` up to the next 4MB superpage boundary, so that on Pentium and higher processors, all physical memory can be mapped using 4MB superpages even if known physical memory does not end exactly on a 4MB boundary. Note that `phys_mem_max` does not necessarily need to reflect all physical memory in the machine; for example, it is perfectly reasonable for the client OS to set it to some artificially lower value so that only that part of physical memory is direct-mapped.

On Pentium and higher processors, this function sets the `PSE` (page size extensions) bit in CR4 in addition to the `PG` (paging) bit, so that the 4MB page mappings used to map physical memory will work properly.

DEPENDENCIES

`base_pdir_pa`:	10.9.2
`ptab_alloc`:	10.9.7
`pdir_map_range`:	10.9.11
`base_cpuid`:	10.6.6
`paging_enable`:	10.5.8

10.9.2 `base_pdir_pa`: initial kernel page directory

SYNOPSIS

`#include <oskit/x86/base_paging.h>`

`extern oskit_addr_t` **base_pdir_pa**;

Description

This variable is initialized by `base_paging_init` (see Section 10.9.1) to contain the physical address of the base page directory. This is the value that should be loaded into the processor's page directory base register (CR3) in order to run in the linear address space defined by this page directory. (The base page directory is automatically activated in this way during initialization; the client OS only needs to load the CR3 register itself if it wants to switch among multiple linear address spaces.) The `pdir_find_pde` function (Section 10.9.3) and other related functions can be used to manipulate the page directory and its associated page tables.

Initially, the base page directory and its page tables directly map physical memory starting at linear address 0. The client OS is free to change the mappings after initialization, for example by adding new mappings outside of the physical address range, or by relocating the physical memory mappings to a different location in the linear address space as described in Section 10.6.3.

Most "real" operating systems will need to create other, separate page directories and associated page tables to represent different address spaces or protection domains. However, the base page directory may still be useful, e.g., as a template for initializing the common kernel portion of other page directories, or as a "kernel-only" address space for use by kernel tasks, etc.

10.9.3 `pdir_find_pde`: find an entry in a page directory given a linear address

Synopsis

`#include <oskit/x86/base_paging.h>`

`pd_entry_t` ***pdir_find_pde***(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*);

Description

This primitive macro uses the appropriate bits in linear address *la* (bits 22–31) to look up a particular entry in the specified page directory. Note that this function takes the *physical* address of a page directory, but returns a *kernel virtual* address (i.e., an ordinary pointer to the selected page directory entry).

Parameters

pdir_pa: Physical address of the page directory.

la: Linear address to be used to select a page directory entry.

Returns

Returns a pointer to the selected page directory entry.

Dependencies

`phystokv`: 10.6.2

10.9.4 `ptab_find_pte`: find an entry in a page table given a linear address

Synopsis

`#include <oskit/x86/base_paging.h>`

`pd_entry_t` ***ptab_find_pte***(`oskit_addr_t` *ptab_pa*, `oskit_addr_t` *la*);

Description

This macro uses the appropriate bits in *la* (bits 12–21) to look up a particular entry in the specified page table. This macro is just like `pdir_find_pde`, except that it selects an entry based on the page table index bits in the linear address rather than the page directory index bits (bits 22–31). Note that this function takes the ***physical*** address of a page table, but returns a ***kernel virtual*** address (an ordinary pointer).

Parameters

ptab_pa: Physical address of the page table.

la: Linear address to be used to select a page table entry.

Returns

Returns a pointer to the selected page table entry.

Dependencies

`phystokv`: 10.6.2

10.9.5 `pdir_find_pte`: look up a page table entry from a page directory

Synopsis

`#include <oskit/x86/base_paging.h>`

`pt_entry_t` ***pdir_find_pte***(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*);

Description

This function is a combination of `pdir_find_pde` and `ptab_find_pte`: it descends through *both* levels of the x86 page table hierarchy and finds the page table entry for the specified linear address.

This function assumes that if the page directory entry selected by bits 22–31 of *la* is valid (the `INTEL_PDE_VALID` bit is set), then that entry actually refers to a page table, and is *not* a 4MB page mapping. The caller must ensure that this is the case.

Parameters

pdir_pa: Physical address of the page directory.

la: Linear address to use to select the appropriate page directory and page table entries.

Returns

Returns a pointer to the selected page table entry, or NULL if there is no page table for this linear address.

Dependencies

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

10.9.6 pdir_get_pte: retrieve the contents of a page table entry

Synopsis

`#include <oskit/x86/base_paging.h>`

`pt_entry_t` **pdir_get_pte**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*);

Description

This function is a simple extension of `pdir_find_pte`: instead of returning the *address* of the selected page table entry, it returns the *contents* of the page table entry: i.e., the physical page frame in bits 12–31 and the associated `INTEL_PTE_*` flags in bits 0–11. If there is no page table in the page directory for the specified linear address, then this function returns 0, the same as if there *was* a page table but the selected page table entry was zero (invalid).

As with `pdir_find_pte`, this function assumes that if the page directory entry selected by bits 22–31 of *la* is valid (the `INTEL_PDE_VALID` bit is set), then that entry actually refers to a page table, and is *not* a 4MB page mapping.

Parameters

pdir_pa: Physical address of the page directory.

la: Linear address to use to select the appropriate page directory and page table entries.

Returns

Returns the selected page table entry, or zero if there is no page table for this linear address. Also returns zero if the selected page table entry exists but is zero.

Dependencies

`pdir_find_pte`: 10.9.5

10.9.7 ptab_alloc: allocate a page table page and clear it to zero

Synopsis

`#include <oskit/x86/base_paging.h>`

`int` **ptab_alloc**([out] `oskit_addr_t` **out_ptab_pa*);

Description

All of the following page mapping routines call this function to allocate new page tables as needed to create page mappings. It attempts to allocate a single page of physical memory, and if successful, returns 0 with the physical address of that page in `*out_ptab_pa`. The newly allocated page is cleared to all zeros by this function. If this function is unsuccessful, it returns nonzero.

The default implementation of this function assumes that the OSKit's minimal C library (`libc`) and list-based memory manager (`liblmm`) are being used to manage physical memory, and allocates page table pages from the `malloc_lmm` memory pool (see Section 9.5.1). However, in more complete OS environments, e.g., in which low physical memory conditions should trigger a page-out rather than failing immediately, this routine can be overridden to provide the desired behavior.

Parameters

out_ptab_pa: The address of a variable of type `oskit_addr_t` into which this function will deposit the physical address of the allocated page, if the allocation was successful.

Returns

Returns zero if the allocation was successful, or nonzero on failure.

Dependencies

`lmm_alloc_page`: 16.6.8

`malloc_lmm`: 9.5.1

`memset`: 9.4.18

`kvtophys`: 10.6.2

10.9.8 `ptab_free`: free a page table allocated using ptab_alloc

Synopsis

`#include <oskit/x86/base_paging.h>`

`void` **ptab_free**(`oskit_addr_t` *ptab_pa*);

Description

The page mapping and unmapping functions described in the following sections call this routine to free a page table that is no longer needed; thus, this function is the partner of `ptab_alloc` (see Section 10.9.7). The default implementation again assumes that the `malloc_lmm` memory pool is being used to manage physical memory. If the client OS overrides `ptab_alloc` to use a different allocation mechanism, it should also override `ptab_free` correspondingly.

Parameters

ptab_pa: The physical address of the page table page to free.

Dependencies

`lmm_free_page`: 16.6.10

10.9.9 `pdir_map_page`: map a 4KB page into a linear address space

Synopsis

`#include <oskit/x86/base_paging.h>`

`int` **pdir_map_page**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*, `pt_entry_t` *mapping*);

Description

This function creates a single 4KB page mapping in the linear address space represented by the specified page directory. If the page table covering the specified linear address does not exist (i.e., the selected page directory entry is invalid), then a new page table is allocated using `ptab_alloc` and inserted into the page directory before the actual page mapping is inserted into the page table. Any new page tables created by this function are mapped into the page directory with permissions `INTEL_PTE_USER | INTEL_PTE_WRITE`: full permissions are granted at the page directory level, although the specified *mapping* value, which is inserted into the selected page table entry, may restrict permissions at the individual page granularity.

This function assumes that if the page directory entry selected by bits 22–31 of *la* is valid (the `INTEL_PDE_VALID` bit is set), then that entry actually refers to a page table, and is *not* a 4MB page mapping. In other words, the caller should not attempt to create a 4KB page mapping in a part of the linear address space already covered by a valid 4MB superpage mapping. The caller must first unmap the 4MB superpage mapping, *then* map the 4KB page (which will cause a page table to be allocated). If the caller follows the guidelines described in Section 10.9, then this requirement should not be a problem.

PARAMETERS

pdir_pa: Physical address of the page directory acting as the root of the linear address space in which to make the requested page mapping.

la: Linear address at which to make the mapping. Only bits 12–31 are relevant to this function; bits 0–11 are ignored.

mapping: Contains the page table entry value to insert into the appropriate page table entry: the page frame number is in bits 12–31, and the `INTEL_PTE_*` flags are in bits 0–11. XXX The caller *must* include `INTEL_PTE_VALID`; other flags may be set according to the desired behavior. (To unmap pages, use `pdir_unmap_page` instead; see Section 10.9.10)

RETURNS

If all goes well and the mapping is successful, this function returns zero. If this function needed to allocate a new page table but the `ptab_alloc` function failed (returned nonzero), then this function passes back the return value from `ptab_alloc`.

DEPENDENCIES

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

`ptab_alloc`: 10.9.7

10.9.10 `pdir_unmap_page`: unmap a single 4KB page mapping

SYNOPSIS

`#include <oskit/x86/base_paging.h>`

`void` **pdir_unmap_page**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*);

DESCRIPTION

This function invalidates a single 4KB page mapping in the linear address space represented by the specified page directory. The *la* parameter should fall in a page previous mapped with `pdir_map_page`, otherwise the result of the call is undefined. XXX Is this overly restrictive?

This function assumes that if the page directory entry selected by bits 22–31 of *la* is valid (the `INTEL_PDE_VALID` bit is set), then that entry actually refers to a page table, and is *not* a 4MB page mapping. In other words, the caller should not attempt to destroy a 4KB page mapping in a part of the linear address space covered by a valid 4MB superpage mapping. Use `pmap_unmap_range` to remove a 4MB superpage mapping.

PARAMETERS

pdir_pa: Physical address of the page directory acting as the root of the linear address space from which the requested page mapping is to be removed.

la: Linear address contained in the page to be removed. Only bits 12–31 are relevant to this function; bits 0–11 are ignored.

Dependencies

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

`ptab_free`: 10.9.8

10.9.11 `pdir_map_range`: map a contiguous range of physical addresses

Synopsis

`#include <oskit/x86/base_paging.h>`

`int` **pdir_map_range**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*, `oskit_addr_t` *pa*, `oskit_size_t` *size*, `pt_entry_t` *mapping_bits*);

Description

This function maps a range of linear addresses in the linear address space represented by the specified page directory onto a contiguous range of physical addresses. The linear (source) address, physical (destination) address, and mapping size must be multiples of the 4KB architectural page size, but other than that no restrictions are imposed on the location or size of the mapping range. If the processor description in the global `base_cpuid` variable (see Section 10.6.6) indicates that page size extensions are available, and the physical and linear addresses are properly aligned, then this function maps as much of the range as possible using 4MB superpage mappings instead of 4KB page mappings. Where 4KB page mappings are needed, this function allocates new page tables as necessary using `ptab_alloc`. Any new page tables created by this function are mapped into the page directory with permissions `INTEL_PTE_USER | INTEL_PTE_WRITE`: full permissions are granted at the page directory level, although the `mapping_bits` may specify more restricted permissions for the actual page mappings.

This function assumes that no valid mappings already exist in the specified linear address range; if any mappings *do* exist, this function may not work properly. If the caller follows the guidelines described in Section 10.9, always unmapping previous mappings before creating new ones, then this requirement should not be a problem.

Parameters

pdir_pa: Physical address of the page directory acting as the root of the linear address space in which to make the requested mapping.

la: Starting linear address at which to make the mapping. Must be page-aligned.

pa: Starting physical address to map to. Must be page-aligned.

size: Size of the linear-to-physical mapping to create. Must be page-aligned.

mapping_bits: Permission bits to OR into each page or superpage mapping entry. The caller *must* include `INTEL_PTE_VALID`; other flags may be set according to the desired behavior. (To unmap ranges, use `pdir_unmap_range` instead; see Section 10.9.13)

Returns

If all goes well and the mapping is successful, this function returns zero. If this function needed to allocate a new page table but the `ptab_alloc` function failed (returned nonzero), then this function passes back the return value from `ptab_alloc`.

Dependencies

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

`ptab_alloc`: 10.9.7

`base_cpuid`: 10.6.6

10.9.12 `pdir_prot_range`: change the permissions on a mapped memory range

Synopsis

`#include <oskit/x86/base_paging.h>`

`void` **pdir_prot_range**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*, `oskit_size_t` *size*, `pt_entry_t` *new_mapping_bits*);

Description

This function can be used to modify the permissions and other attribute bits associated with a mapping range previously created with `pdir_map_range`. The *la* and *size* parameters must be *exactly* the same as those passed to the `pdir_map_range` used to create the mapping.

Parameters

pdir_pa: Physical address of the page directory acting as the root of the linear address space containing the mapping to modify.

la: Starting linear address of the mapping to modify. Must be exactly the same as the address specified to the `pdir_map_range` call used to create this mapping.

size: Size of the mapping to modify. Must be exactly the same as the size specified to the `pdir_map_range` call used to create this mapping.

new_mapping_bits: New permission flags to insert into each page or superpage mapping entry. The caller *must* include `INTEL_PTE_VALID`; other flags may be set according to the desired behavior. (To unmap ranges, use `pdir_unmap_range`; see Section 10.9.13)

Dependencies

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

10.9.13 `pdir_unmap_range`: remove a mapped range of linear addresses

Synopsis

`#include <oskit/x86/base_paging.h>`

`void` **pdir_unmap_range**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*, `oskit_size_t` *size*);

Description

This function removes a mapping range previously created using `pdir_map_range`. The *la* and *size* parameters must be *exactly* the same as those passed to the `pdir_map_range` used to create the mapping.

PARAMETERS

pdir_pa: Physical address of the page directory acting as the root of the linear address space containing the mapping to destroy.

la: Starting linear address of the mapping to destroy. Must be exactly the same as the address specified to the `pdir_map_range` call used to create this mapping.

size: Size of the mapping to destroy. Must be exactly the same as the size specified to the `pdir_map_range` call used to create this mapping.

DEPENDENCIES

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

10.9.14 pdir_clean_range: free unused page table pages in a page directory

SYNOPSIS

`#include <oskit/x86/base_paging.h>`

`void` **pdir_clean_range**(`oskit_addr_t` *pdir_pa*, `oskit_addr_t` *la*, `oskit_size_t` *size*);

DESCRIPTION

This function scans the portion of the given page directory covering the given address range, looking for associated page table pages that are unnecessary and frees them. “Unnecessary” page table pages are those which contain only entries for which `INTEL_PTE_VALID` is not set. These pages are freed with `ptab_free` and the corresponding page directory entries marked as invalid.

PARAMETERS

pdir_pa: Physical address of the page directory.

la: Starting linear address of the region to clean. Must be page-aligned.

size: Size (in bytes) of the region to clean. The value passed is rounded up to whole pages. Use “0” for *la* and “ (oskit_addr_t)0” for *size* to clean the entire page directory.

DEPENDENCIES

`pdir_find_pde`: 10.9.3

`ptab_find_pte`: 10.9.4

`ptab_free`: 10.9.8

10.9.15 pdir_dump: dump the contents of a page directory and all its page tables

SYNOPSIS

`#include <oskit/x86/base_paging.h>`

`void` **pdir_dump**(`oskit_addr_t` *pdir_pa*);

Description

This function is primarily intended for debugging purposes: it dumps the mappings described by the specified page directory and all associated page tables in a reasonably compact, human-readable form, using `printf`. 4MB superpage as well as 4KB page mappings are handled properly, and contiguous ranges of identical mappings referring to successive physical pages or superpages are collapsed into a single line for display purposes. The permissions and other page directory/page table entry flags are expanded out as human-readable flag names.

Parameters

pdir_pa: Physical address of the page directory describing the linear address space to dump.

Dependencies

`ptab_dump`: 10.9.16

`printf`: 9.6

`phystokv`: 10.6.2

10.9.16 `ptab_dump`: dump the contents of a page table

Synopsis

`#include <oskit/x86/base_paging.h>`

`void` **ptab_dump**(`oskit_addr_t` *ptab_pa*, `oskit_addr_t` *base_la*);

Description

This is primarily a helper function for `pdir_dump`, but it can also be used independently, to dump the contents of an individual page table. For output purposes, the page table is assumed to reside at *base_la* in "some" linear address space: in other words, this parameter provides the topmost ten bits in the linear addresses dumped by this routine. Contiguous ranges of identical mappings referring to successive physical pages are collapsed into a single line for display purposes. The permissions and other page directory/page table entry flags are expanded out as human-readable flag names.

Parameters

pdir_pa: Physical address of the page table to dump.

base_la: Linear address at which this page table resides, for purposes of displaying linear source addresses. Must be 4MB aligned.

Dependencies

`printf`: 9.6

`phystokv`: 10.6.2

10.10 X86 Base Environment: Protected-mode entry and exit

The full mode switching code is written and functional but not yet documented or integrated into the OSKit source tree.

10.11 X86 PC Base Environment: Physical Memory Management

The physical memory address space on PCs is divided into distinct regions which have certain attributes. The lowest 1MB of physical memory is "special" in that only it can be accessed from real mode. The lowest 16MB of physical memory is special in that only it can be accessed by the built-in DMA controller. On some PCs, there may be an additional boundary imposed by the motherboard. Memory above this boundary (e.g., 64MB) may not be cacheable by the L2 cache. The base environment uses the OSKit LMM library 16 to accommodate these differences. Physical memory is managed by a single LMM `malloc_lmm` with separate regions for each "type" of memory. Hence, LMM allocation requests can be made with appropriate flag values to obtain memory with the desired characteristics.

10.11.1 `phys_lmm.h`: Physical memory management for PCs

Synopsis

```
#include <oskit/x86/pc/phys_lmm.h>
```

Description

There are three priority values assigned to regions of physical memory as they are made available at boot time (via `lmm_add_region`). In increasing order of priority (i.e., increasing preference for allocation):

`LMM_PRI_1MB`: For physical memory below 1MB.

`LMM_PRI_16MB`: For physical memory below 16MB.

`LMM_PRI_HIGH`: For physical memory above 16MB.

These priorities prevent the simple memory allocation interfaces from handing out the more precious low-address memory. To enable savvy applications to explicitly allocate memory of a given type, the following flag values are assigned at boot time and can be passed to LMM allocation routines:

`LMMF_1MB`: Set on memory below 1MB; i.e., the `LMM_PRI_1MB` region.

`LMMF_16MB`: Set on memory below 16MB; i.e., the `LMM_PRI_1MB` and `LMM_PRI_16MB` regions.

Thus, if neither flag is set, memory can be allocated from any of the three regions with preference given to `LMM_PRI_HIGH`, followed by `LMM_PRI_16MB` and `LMM_PRI_1MB`. If just LMMF_16MB is set, memory can be allocated from either `LMM_PRI_16MB` or `LMM_PRI_1MB` in that order. If both flags are set, memory can only be allocated from the `LMM_PRI_1MB` region.

10.11.2 `phys_mem_max`: Highest physical memory address

Synopsis

```
#include <oskit/x86/pc/phys_lmm.h>
extern oskit_addr_t phys_mem_max;
```

Description

This variable records the highest physical memory address; i.e. the end address of the highest free block added to `malloc_lmm`. This is the highest address that the kernel should ever have to deal with.

Not all addresses between 0 and `phys_mem_max` are necessarily available for allocation, there may be holes in the available physical memory space.

10.11.3 `phys_lmm_init`: Initialize kernel physical memory LMM

SYNOPSIS

`#include <oskit/x86/pc/phys_lmm.h>`

`void` **phys_lmm_init**(`void`);

DESCRIPTION

This routine sets up `malloc_lmm` with three physical memory regions, one for each of the memory types described in `phys_lmm.h`. No actual memory is added to those regions. You can then call phys_lmm_add() to add memory to those regions. In the base environment, `base_multiboot_init_mem` handles this chore.

DEPENDENCIES

`malloc_lmm`: 9.5.1

`lmm_add_region`: 16.6.2

`phystokv`: 10.6.2

10.11.4 `phys_lmm_add`: Add memory to the kernel physical memory LMM

SYNOPSIS

`#include <oskit/x86/pc/phys_lmm.h>`

`void` **phys_lmm_add**(`oskit_addr_t` *min_pa*, `oskit_size_t` *size*);

DESCRIPTION

Add a chunk of physical memory to the appropriate region(s) on the `malloc_lmm`. The provided memory block may be arbitrarily aligned and may cross region boundaries (e.g. the 16MB boundary); it will be shrunken and split apart as necessary. If the address of the end of the block is greater than the current value in `phys_max_mem`, this address is recorded.

Note that `phys_lmm_add` takes a *physical* address, not a virtual address as the underlying LMM routines do. This routine will perform the conversion as needed with `phystokv`.

PARAMETERS

min_pa: Physical address of the start of the chunk being added.

size: Size in bytes of the chunk being added.

DEPENDENCIES

`malloc_lmm`: 9.5.1

`lmm_add_free`: 16.6.3

`phystokv`: 10.6.2

`phys_mem_max`: 10.11.2

10.12 X86 PC Base Environment: Interrupt Support

In the base environment, each hardware interrupt vector in the processor IDT points to a small assembly language stub that saves a standard trap frame (10.8.1), disables and acknowledges the hardware interrupt, and calls a designated high-level handler routine specified in the `base_irq_handlers` table (10.12.2). Initially, all the entries in this table point to `base_irq_default_handler` (10.12.5). Custom interrupt handlers can be installed by changing the appropriate entry in the table. The default action for all interrupts can be changed by overriding `base_irq_default_handler`.

The base environment also includes support for a single "software interrupt." A software interrupt is delivered after all pending hardware interrupts have been processed but before returning from the interrupt context. A software interrupt can be posted at any time with `base_irq_softint_request` (10.12.7) but will only be triggered upon return from a hardware interrupt; i.e., processing of a software interrupt requested from a non-interrupt context is deferred until a hardware interrupt occurs. The software interrupt handler is `base_irq_softint_handler` (10.12.8) which can be replaced by a custom version provided by the kernel.

10.12.1 `base_irq.h`: Hardware interrupt definitions for standard PCs

Synopsis

```
#include <oskit/x86/pc/base_irq.h>
```

Description

`BASE_IRQ_COUNT`: Number of interrupt request lines.

`BASE_IRQ_MASTER_BASE`: Default location in the IDT for programming the PIC.

`BASE_IRQ_SLAVE_BASE`: Default location in the IDT for programming the PIC.

`irq_master_base`: Variable storing the current master PIC interrupt vector base.

`irq_slave_base`: Variable storing the current slave PIC interrupt vector base.

`fill_irq_gate(irq_num, entry, selector, access)`: Fill the `base_idt` descriptor for the indicated IRQ with an interrupt gate containing the given entry, selector and access information.

10.12.2 `base_irq_handlers`: Array of handler routines for hardware interrupts

Synopsis

```
#include <oskit/x86/pc/base_irq.h>
void (*base_irq_handlers[BASE_IRQ_COUNT]) (struct trap_state *ts);
```

Description

Contains a function pointer for every hardware interrupt vector. By default, all entries in this table point to `base_irq_default_handler`. Custom interrupt handlers can be installed by changing the appropriate table entry.

Interrupt handlers can freely examine and modify the processor state (10.8.1) of the interrupted activity, e.g., to implement threads and preemption. On entry, the processor's IDT interrupt vector number is in `ts->trapno` and the hardware IRQ number that caused the interrupt is in `ts->err`.

Dependencies

`base_irq_default_handler`: 10.12.5

10.12.3 `base_irq_init`: Initialize hardware interrupts

SYNOPSIS

```
#include <oskit/x86/pc/base_irq.h>
void base_irq_init(void);
```

DESCRIPTION

Initializes the system to properly handle hardware interrupts. It loads the appropriate entries in the base IDT (10.7.4) with the gate descriptor information from `base_irq_inittab`, programs the PICs to the standard vector base addresses (see Section 10.12.1), and disables all interrupt request lines.

Processor interrupts must be disabled when this routine is called, they will be enabled upon return.

DEPENDENCIES

`base_idt`: 10.7.4

`base_irq_inittab`: 10.12.4

`gate_init`: 10.5.10

`pic_init`: 10.4.2

`pic_disable_all`: 10.4.2

`irq_master_base`: 10.12.1

`irq_slave_base`: 10.12.1

10.12.4 `base_irq_inittab`: initialization table for default interrupt entrypoints

SYNOPSIS

```
#include <oskit/x86/pc/base_irq.h>
extern struct gate_init_entry base_irq_inittab[];
```

DESCRIPTION

This gate initialization table (10.3.9) encapsulates the base environment's default interrupt entrypoint code. This module provides IDT entrypoints for all the standard PC hardware interrupt vectors; each entrypoint pushes a standard state frame on the stack (10.8.1), disables and acknowledges the hardware interrupt, and then calls the C handler function pointed to by the appropriate entry of the `base_irq_handlers` array (10.12.2). Upon return from the handler, the interrupt code checks for a pending software interrupt and dispatches to `base_irq_softint_handler`.

DEPENDENCIES

`base_irq_handlers`: 10.12.2

`base_irq_nest`: 10.12.6

`base_irq_softint_handler`: 10.12.8

10.12.5 `base_irq_default_handler`: default IRQ handler for unexpected interrupts

Synopsis

`#include <oskit/x86/pc/base_irq.h>`

`void` **base_irq_default_handler**(`struct trap_state *`*state*);

Description

This routine is the default handler for all interrupts in the base environment. It simply displays a warning message and returns.

It is expected that the client OS will override this default behavior for all interrupts it cares about, leaving this routine to be called only for unexpected interrupts.

Parameters

state: A pointer to the processor state at the time of the interrupt.

Dependencies

`struct trap_state`: 10.8.1

`printf`: 9.6

10.12.6 `base_irq_nest`: interrupt nesting counter and software interrupt flag

Synopsis

`#include <oskit/x86/pc/base_irq.h>`

`extern unsigned char` **base_irq_nest**;

Description

Hardware interrupt nesting counter, used to ensure that the software interrupt handler isn't called until all outstanding hardware interrupts have been processed. In addition, this variable also acts as the software interrupt pending flag: if the high bit is clear, a software interrupt is pending.

10.12.7 `base_irq_softint_request`: request a software interrupt

Synopsis

`#include <oskit/x86/pc/base_irq.h>`

`void` **base_irq_softint_request**(`void`);

Description

This routine requests a software interrupt and is typically called from a hardware interrupt handler to schedule lower priority processing.

After requesting a software interrupt, `base_irq_softint_handler` will be called when all hardware interrupt handlers have completed processing and `base_irq_nest` is zero. If an interrupt is scheduled from a non-interrupt context, the handler will not be called until the next hardware interrupt occurs and has been processed.

Only a single software interrupt may be pending at a time.

Dependencies

`base_irq_nest`: 10.12.6

10.12.8 `base_irq_softint_handler`: handler for software interrupts

Synopsis

`#include <oskit/x86/pc/base_irq.h>`

`void` **base_irq_softint_handler**(`struct trap_state` **state*);

Description

Software interrupt handler called by the interrupt entry/exit stub code when a software interrupt has been requested and needs to be run. The default implementation of this routine simply returns; to use software interrupts, the kernel must override it.

The handler is free to examine and modify the processor state in `state`.

Parameters

state: A pointer to the processor state at the time of the interrupt.

Dependencies

`struct trap_state`: 10.8.1

10.13 X86 PC Base Environment: Console Support

The base console environment allows "console" input and output using either the display and keyboard or a serial line. Additionally it allows a remote kernel debugging with GDB over a serial line. Selection of the display or serial port as console and which serial ports to use for the console and GDB are controlled by command line options or environment variables as described in this section.

The base console environment uses a simple polled interface for serial port input and output as well as for keyboard input. The video display output interface is a simple, dumb text terminal. See the appropriate sections for details.

In the base interface, all input via `stdin` (e.g., `getchar`, `scanf`) and all output via `stdout` or `stderr` (e.g., `putchar`, `printf`) use the console.

For simplicity, a set of vanilla console functions is provided that direct input and output to/from the appropriate device. For example, `console_putchar` will invoke `com_cons_putchar` if the console device is a serial port, or `direct_cons_putchar` if the console device is a display. Other vanilla console I/O routines include `console_getchar`, `console_puts`, and `console_putbytes` (a raw block output function that does not append a newline). All behave as expected. These routines are provided to so that higher level I/O code does not need to be concerned with which type of device is currenty the console. Both the minimal C library (section 9) and the FreeBSD C library (section 14) take advantage of this redirection.

10.13.1 `base_console.h`: definitions for base console support

Synopsis

`#include <oskit/x86/pc/base_console.h>`

Description

The following variable are used in the base_console code:

`serial_console`: Set non-zero if a serial port is being used as the console. Default value is zero, but may be turned on by either a command line option or the `CONS_COM` environment variable.

`cons_com_port`: If `serial_console` is non-zero, this variable indicates the COM port to use for console input and output. Default value is 1, but may be changed by setting the `CONS_COM` environment variable. Possible values are 1, 2, 3 or 4.

`enable_gdb`: If non-zero, enables the GDB trap handler; i.e. makes remote debugging possible. The default value is zero.

`gdb_com_port`: If `enable_gdb` is non-zero, this variable indicates the COM port to use for remote GDB interaction. The default value is 1 (the console and remote GDB can share the same serial port, but do not have to). Possible values are 1, 2, 3 or 4.

Refer to section 10.13.2 for more information on command line options and environment variables. See section 10.17 for more on remote GDB.

10.13.2 `base_console_init`: Initialize the base console

Synopsis

`#include <oskit/x86/base_console.h>`

`void` **base_console_init**(`int` *argc*, `char` `**`*argv*);

Description

This function parses the multiboot command line and optionally initializes the serial lines.

Command line options recognized by the base_console code include:

-f Enables "fast" serial ports. Sets the baud rate of the console and GDB serial ports to 115200.

-h Enables a serial line console on `cons_com_port`.

-d Enables remote GDB on `gdb_com_port`.

Environment variables recognized include:

`CONS_COM`: Serial port number (1, 2, 3 or 4) to use as the console. Sets `cons_com_port` to this value and sets `serial_console` non-zero.

`GDB_COM`: Serial port number (1, 2, 3 or 4) to use as the remote GDB interface. Sets `gdb_com_port` to this value and sets `enable_gdb` non-zero.

`BAUD`: Baud rate to use for both the console and GDB serial ports. Any of the standard values in `termios.h` (section 9.4.30) are valid.

Parameters

argc: Count of command line arguments in *argv*.

argv: Vector of command line arguments.

Dependencies

`getenv`: 9.4.17

`atoi`: 9.4.17

`base_cooked_termios`: 10.13.3

`base_raw_termios`: 10.13.4

`strcmp`: 9.4.18

`printf`: 9.6

`base_gdt_load`: 10.7.3

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`com_cons_init`: 10.13.8

`com_cons_flush`: 10.13.11

`com_cons_getchar`: 10.13.9

`com_cons_putchar`: 10.13.10

`gdb_pc_com_init`: 10.18.9

`gdb_serial_getchar`: 10.18.4

`gdb_serial_putchar`: 10.18.5

`gdb_serial_puts`: 10.18.6

`gdb_serial_exit`: 10.18.3

`direct_cons_getchar`: 10.13.5

`direct_cons_putchar`: 10.13.6

10.13.3 `base_cooked_termios`: Default `termios` setting for cooked-mode console

Synopsis

```
#include <termios.h>
extern struct termios base_cooked_termios;
```

Description

A POSIX `termios` structures with values suitable for a basic cooked-mode console tty. Used in the base environment when initializing a serial-port console in `com_cons_init`.

Dependencies

`termios.h`: 9.4.30

10.13.4 `base_raw_termios`: Default `termios` setting for raw-mode console

Synopsis

```
#include <termios.h>
extern struct termios base_raw_termios;
```

Description

A POSIX `termios` structures with values suitable for a basic raw-mode console tty. Used in the base environment when initializing a serial-port for remote GDB debuging in `com_cons_init`.

Dependencies

`termios.h`: 9.4.30

10.13.5 `direct_cons_getchar`: wait for and read a character from the keyboard

Synopsis

```
#include <oskit/x86/pc/direct_console.h>
int direct_cons_getchar(void);
```

Description

Read a character from the PC keyboard. If none is available, this routine loops polling the keyboard status register until a character is available.

Supports only a subset of the available key presses. In particular, only the shifted and unshifted printable ASCII characters along with Escape, Backspace, Tab, and Carriage return. It does not support the remaining control characters or multi-character (function) keys.

Returns

Returns the character read.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`inb`: 10.3.7

10.13.6 `direct_cons_putchar`: write a character to the video console

Synopsis

`#include <oskit/x86/pc/direct_console.h>`

`void` **direct_cons_putchar**(`unsigned char` *c*);

Description

Outputs the indicated character on the video console. Handles "\n" (newline), "\r" (carriage return), "\b" (backspace), and "\t" (tab) in addition to printable ASCII characters. Tabs are expanded to spaces (with stops are every 8 columns) and lines are automatically wrapped at 80 characters. Newline implies a carriage return.

Parameters

c: Character to be printed.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`phystokv`: 10.6.2

`outb_p`: 10.3.7

`memcpy`: 9.4.18

10.13.7 `direct_cons_trygetchar`: read an available character from the keyboard

Synopsis

`#include <oskit/x86/pc/direct_console.h>`

`int` **direct_cons_trygetchar**(`void`);

Description

Quick poll for an available input character. Returns a character or -1 if no character was available.

Due to the large delay between when a character is typed and when the scan code arrives at the keyboard controller (4-5 ms), there are a variety of situations in which this routine may return -1 even though a character has been typed:

- a valid scan code is in transit from the keyboard when called
- a key release scan code is received (from a previous key press)
- a SHIFT key press is received (shift state is updated however)
- a key press for a multi-character sequence is received (e.g., CTRL or a function key)

In other words, this routine never delays in an attempt to wait for the next scan code to arrive when one is not currently available. Hence the utility of this routine is questionable.

Returns

Returns a character if available, -1 otherwise.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`inb`: 10.3.7

10.13.8 `com_cons_init`: initialize a serial port

Synopsis

`#include <oskit/x86/pc/com_cons.h>`

`void` **com_cons_init**(`int` *port*, `struct termios *`*com_params*);

Description

This routine must be called once to initialize a COM port for use by other `com_cons` routines. The supplied `termios` structure indicates the baud rate and other settings. If com_params is zero, a default of 9600 baud, 8 bit characters, no parity, and 1 stop bit is used.

Parameters

port: COM port to initialize. Must be 1, 2, 3 or 4.

com_params: Pointer to a `termios` structures with the tty settings to use.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`base_cooked_termios`: 10.13.3

`inb`: 10.3.7

`outb`: 10.3.7

10.13.9 `com_cons_getchar`: wait for and read a character from a serial port

Synopsis

`#include <oskit/x86/pc/com_cons.h>`

`int` **com_cons_getchar**(`int` *port*);

Description

Read a character from the indicated serial port. If none is available, this routine loops polling the status register until a character is available.

Parameters

port: COM port to read from. Must be 1, 2, 3 or 4.

Returns

Returns the character read.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`inb`: 10.3.7

10.13.10 `com_cons_putchar`: write a character to a serial port

Synopsis

`#include <oskit/x86/pc/com_cons.h>`

`void` **com_cons_putchar**(`int` *port*, `int` *ch*);

Description

Outputs the indicated character on the specified serial port.

Parameters

port: COM port to write to. Must be 1, 2, 3 or 4.

ch: Character to be printed.

Dependencies

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

`inb`: 10.3.7

`outb`: 10.3.7

10.13.11 `com_cons_flush`: delay until all output is flushed on a serial line

Synopsis

`#include <oskit/x86/pc/com_cons.h>`

`void` **com_cons_flush**(`int` *port*);

Description

Waits until the transmit FIFOs are empty on the serial port specified. This is useful as it allows the programmer to know that the message sent out has been received. This is necessary before resetting the UART or changing settings if it is desirable for any data already "sent" to actually be transmitted.

Parameters

port: COM port to flush. Must be 1, 2, 3 or 4.

Dependencies

`inb`: 10.3.7

10.13.12 `com_cons_enable_receive_interrupt`: enable receive interrupts on a serial port

Synopsis

`#include <oskit/x86/pc/com_cons.h>`

`void` **com_cons_enable_receive_interrupt**(`int` *port*);

Description

Special function to enable receive character interrupts on a serial port.

Since the base COM console operates by polling, there is no need to handle serial interrupts in order to do basic I/O. However, if you want to be notified up when a character is received, call this function immediately after `com_cons_init`, and make sure the appropriate IDT entry is initialized properly.

For example, the serial debugging code for the PC COM port uses this so that the program can be woken up when the user presses the interrupt character (Ĉ) from the remote debugger.

Parameters

port: COM port to enable receive interrupts on. Must be 1, 2, 3 or 4.

Dependencies

`inb`: 10.3.7

10.14 X86 PC MultiBoot Startup

MultiBoot is a standardized interface between boot loaders and 32-bit operating systems on x86 PC platforms, which attempts to solve the traditional problem that every single operating system tends to come with its own boot loader or set of boot loaders which are completely incompatible with boot loaders written for any other operating system. The MultiBoot standard allows any MultiBoot-compliant operating system to be loaded from any MultiBoot-supporting boot loader. MultiBoot is also designed to provide advanced features needed by many modern operating systems, such as direct 32-bit protected-mode startup, and support for *boot modules*, which are arbitrary files loaded by the boot loader into physical memory along with the kernel and passed to the kernel on startup. These boot modules may be dynamically loadable device drivers, application program executables, files on an initial file system, or anything else the OS may need before it has full device access. The MultiBoot standard is already supported by several boot loaders and operating systems, and is gradually becoming more widespread. For details on the MultiBoot standard see Section 10.14.12.

The MultiBoot standard is separate from and largely independent of the OSKit. However, if MultiBoot is used, the toolkit can leverage it to provide a powerful, flexible, and extremely convenient method of booting custom operating systems that use the OSKit. The toolkit provides startup code which allows MultiBoot-compliant OS kernels to be built easily, and which handles the details of finding and managing physical memory on startup, interpreting the command line passed by the boot loader, finding and using boot modules, etc. If you use the OSKit's MultiBoot startup support, your kernel automatically inherits a complete, full-featured 32-bit protected-mode startup environment and the ability to use various existing boot loaders, without being constrained by the limitations of traditional OS-specific boot loaders.

10.14.1 Startup code organization

The MultiBoot startup code in the OSKit has two components. The first component is contained in the object file `multiboot.o`, installed by the toolkit in the *`prefix`*`/lib/oskit/` directory. This object file contains the actual MultiBoot header and entrypoint; it must be linked into the kernel as the very first object file, so that its contents will be at the very beginning of the resulting executable. (This object file takes the place of the `crt0.o` or `crt1.o` normally used when linking ordinary applications in a Unix-like system.) The second component is contained in the `libkern.a` library; it contains the rest of the MultiBoot startup code as well as various utility routines for the use of the client OS.

XXX diagram of MultiBoot kernel executable image

The toolkit's MultiBoot startup code will work when using either ELF or `a.out` format. ELF is the format recommended for kernel images by the MultiBoot standard; however, the `a.out` format is also supported through the use of some special header information embedded in the `multiboot.o` code linked at the very beginning of the kernel's text segment. This information allows the MultiBoot boot loader to determine the location and sizes of the kernel's text, data, and bss sections in the kernel executable without knowing the details of the particular `a.out` flavor in use (e.g., Linux, NetBSD, FreeBSD, Mach, VSTa, etc.), all of which are otherwise mutually incompatible.

10.14.2 Startup sequence

After the MultiBoot boot loader loads the kernel executable image, it searches through the beginning of the image for the MultiBoot header which provides important information about the OS being loaded. The boot loader performs its activities, then shuts itself down and jumps to the OS kernel entrypoint defined in the kernel's MultiBoot header. In one processor register the boot loader passes to the kernel the address of a *MultiBoot information structure*, containing various information passed from the boot loader to the OS, organized in a standardized format defined by the MultiBoot specification.

In the OSKit's MultiBoot startup code, the kernel entrypoint is a short code fragment in `multiboot.o` which sets up the initial stack and performs other minimal initialization so that ordinary 32-bit C code can be run safely.[3] This code fragment then calls the C function `multiboot_main`, with a pointer to the MultiBoot

[3]This file also turns all floppy drive motors off, since if we were booted from floppy the motor is most likely still on and can cause unnecessary wear on the floppy disk.

information structure as its argument. Normally, the `multiboot_main` function comes from `libkern.a`; it performs other high-level initialization to create a convenient, stable 32-bit environment, and then calls the familiar `main` routine, which the client OS must provide.

10.14.3 Memory model

Once the OS kernel receives control in its `main` routine, the processor has been set up in the base environment defined earlier in Section 10.6. The `base_gdt`, `base_idt`, and `base_tss` have been set up and activated, so that segmentation operations work and traps can be handled. Paging is disabled, and all kernel code and data segment descriptors are set up with an offset of zero, so that virtual addresses, linear addresses, and physical addresses are all the same. The client OS is free to change this memory layout later, e.g., by enabling paging and reorganizing the linear address space as described in Section 10.6.3.

As part of the initialization performed by `multiboot_main`, the OSKit's MultiBoot startup code uses information passed to the OS by the boot loader, describing the location and amount of physical memory available, to set up the `malloc_lmm` memory pool (see Section 9.5.1). This allows the OS kernel to allocate and manage physical memory using the normal C-language memory allocation mechanisms, as well as directly using the underlying LMM memory manager library functions. The physical memory placed on the `malloc_lmm` pool during initialization is guaranteed *not* to contain any of the data structures passed by the boot loader which the OS may need to use, such as the command line or the boot modules; this way, the kernel can freely allocate and use memory right from the start without worrying about accidentally "stepping on" boot loader data that it will need to access later on. In addition, the physical memory placed on the `malloc_lmm` is divided into the three separate regions defined in `phys_lmm.h` (see Section 10.11.1): one for low memory below 1MB, one for "DMA" memory below 16MB, and one for all physical memory above this line. This division allows the kernel to allocate "special" memory when needed for device access or for calls to real-mode BIOS routines, simply by specifying the appropriate flags in the LMM allocation calls.

10.14.4 Command-line arguments

The MultiBoot specification allows an arbitrary ASCII string to be passed from the boot loader to the OS as a "command line" for the OS to interpret as it sees fit. As passed from the boot loader to the OS, this is a single null-terminated ASCII string. However, the default MultiBoot initialization code provided by the OSKit performs some preprocessing of the command line before the actual OS receives control in its `main` routine. In particular, it parses the single command line string into an array of individual argument strings so that the arguments can be passed to the OS through the normal C-language `argc/argv` parameters to `main`. In addition, any command-line arguments containing an equals sign ('=') are added to the `environ` array rather than the `argv` array, effectively providing the OS with a minimal initial environment that can be specified by the user (through the boot loader) and examined by the OS using the normal `getenv` mechanism (see Section 9.4.17).

Note that this command-line preprocessing mechanism matches the kernel command-line conventions established by Linux, although it provides more convenience and flexibility to the OS by providing this information to the OS through standard C-language facilities, and by not restricting the "environment variables" to be comma-separated lists of numeric constants, as Linux does. This mechanism also provides *much* more flexibility than traditional BSD/Mach command-line mechanisms, in which the boot loader itself does most of the command-line parsing, and basically only passes a single fixed "flags" word to the OS.

10.14.5 Linking MultiBoot kernels

Since MultiBoot kernels initially run in physical memory, with paging disabled and segmentation effectively "neutralized," the kernel must be linked at an address within the range of physical memory present on typical PCs. Normally the best place to link the kernel is at 0x100000, or 1MB, which is the beginning of extended memory just beyond the real-mode ROM BIOS. Since the processor is already in 32-bit protected mode when the MultiBoot boot loader starts the OS, running above the 1MB "boundary" is not a problem. By linking at 1MB, the kernel has plenty of "room to grow," having essentially all extended memory available to it in one contiguous chunk.

In some cases, it may be preferable to link the kernel at a lower address, below the 1MB boundary, for example if the kernel needs to run on machines without any extended memory, or if the kernel contains code that needs to run in real mode. This is also allowed by the MultiBoot standard. However, note that the kernel should generally leave at least the first 0x500 bytes of physical memory untouched, since this area contains important BIOS data structures that will be needed if the kernel ever makes calls to the BIOS, or if it wants to glean information about the machine from this area such as hard disk configuration data.

10.14.6 `multiboot.h`: Definitions of MultiBoot structures and constants

Synopsis

`#include <oskit/x86/multiboot.h>`

Description

This header file is not specific to the MultiBoot startup code provided by the OSKit; it merely contains generic symbolic structure and constant definitions corresponding to the data structures specified in the MultiBoot specification. The following C structures are defined:

`struct multiboot_header`: Defines the MultiBoot header structure which is located near the beginning of all MultiBoot-compliant kernel executables.

`struct multiboot_info`: Defines the general information structure passed from the boot loader to the OS when control is passed to the OS.

`struct multiboot_module`: One of the elements of the `multiboot_info` structure is an optional array of boot modules which the boot loader may provide; each element of the boot module array is reflected by this structure.

`struct multiboot_addr_range`: Another optional component of the `multiboot_info` structure is a pointer to an array of address range descriptors, described by this structure, which define the layout of physical memory on the machine. (XXX name mismatch.)

For more information on these structures and the associated constants, see the `multiboot.h` header file and the MultiBoot specification.

XXX should move this to x86/pc/multiboot.h?

10.14.7 `boot_info`: MultiBoot information structure

Synopsis

`#include <oskit/x86/pc/base_multiboot.h>`

`extern struct multiboot_info` **boot_info**;

Description

The first thing that `multiboot_main` does on entry from the minimal startup code in `multiboot.o` is copy the MultiBoot information structure passed by the boot loader into a global variable in the kernel's bss segment. Copying the information structure this way allows it to be accessed more conveniently by the kernel, and makes it unnecessary for the memory initialization code (`base_multiboot_init_mem`; see Section 10.14.9) to carefully "step over" the information structure when determining what physical memory is available for general use.

After the OS has received control in its `main` routine, it is free to examine the `boot_info` structure and use it to locate other data passed by the boot loader, such as the boot modules. The client OS must *not* attempt to access the original copy of the information structure passed by the boot loader, since that copy of the structure may be overwritten as memory is dynamically allocated and used. However, this should not be a problem, since a pointer to the original copy of the `multiboot_info` structure is never even passed to the OS by the MultiBoot startup code; it is only accessible to the OS if it overrides the `multiboot_main` function.

10.14.8 `multiboot_main`: general MultiBoot initialization

Synopsis

`#include <oskit/x86/pc/base_multiboot.h>`

`void` **multiboot_main**(`oskit_addr_t` *boot_info_pa*);

Description

This is the first C-language function to run, invoked by the minimal startup code fragment in `multiboot.o`. The default implementation merely copies the MultiBoot information structure passed by the boot loader into the global variable `boot_info` (see Section 10.14.7), and then calls the following routines to set up the base environment and start the OS:

`base_cpu_setup`: Initializes the base GDT, IDT, and TSS, so that the processor's segmentation facilities can be used and processor traps can be handled.

`base_multiboot_init_mem`: Finds all physical memory available for general use and adds it to the `malloc_lmm` so that OS code can allocate memory dynamically.

`base_multiboot_init_cmdline`: Performs basic preprocessing on the command line string passed by the boot loader, splitting it up into standard C argument and environment variable lists.

`main`: This call is what invokes the actual OS code, using standard C-language startup conventions.

`exit`: As per C language conventions, if the `main` routine ever returns, `exit` is called immediately, using the return value from `main` as the exit code.

If the client OS does not wish some or all of the above to be performed, it may override the `multiboot_main` function with a version that does what it needs, or, alternatively, it may instead override the specific functions of interest called by `multiboot_main`.

Parameters

boot_info_pa: The physical address of the MultiBoot information structure as created and passed by the boot loader.

Returns

This function had better never return.

Dependencies

`phystokv`: 10.6.2

`boot_info`: 10.14.7

`base_cpu_setup`: 10.6.3

`base_multiboot_init_mem`: 10.14.9

`base_multiboot_init_cmdline`: 10.14.10

`exit`: 9.8.1

10.14.9 `base_multiboot_init_mem`: physical memory initialization

Synopsis

`#include <oskit/x86/pc/base_multiboot.h>`

`void` **base_multiboot_init_mem**(`void`);

Description

This function finds all physical memory available for general use and adds it to the `malloc_lmm` pool, as described in Section 10.14.3. It is normally called automatically during initialization by `multiboot_main` (see Section 10.14.8).

This function uses the lower and upper memory size fields in the MultiBoot information structure to determine the total amount of physical memory available; it then adds all of this memory to the `malloc_lmm` pool except for the following "special" areas:

- The first 0x500 bytes of physical memory are left untouched, since this area contains BIOS data structures which the OS might want to access (or the BIOS itself, if the OS makes any BIOS calls).
- The area from 0xa0000 to 0x100000 is the I/O and ROM area, and therefore does not contain usable physical memory.
- The memory occupied by the kernel itself is skipped, so that the kernel will not trash its own code, data, or bss.
- All interesting boot loader data structures, which can be found through the MultiBoot information structure, are skipped, so that the OS can examine them later. This includes the kernel command line, the boot module information array, the boot modules themselves, and the strings associated with the boot modules.

This function uses `phys_lmm_init` to initialize the `malloc_lmm`, and `phys_lmm_add` to add available physical memory to it (see Section 10.11.1); as a consequence, this causes the physical memory found to be split up automatically according to the three main functional "classes" of PC memory: low 1MB memory accessible to real-mode software, low 16MB memory accessible to the built-in DMA controller, and "all other" memory. This division allows the OS to allocate "special" memory when needed for device access or for calls to real-mode BIOS routines, simply by specifying the appropriate flags in the LMM allocation calls.

XXX currently doesn't use the memory range array.

Dependencies

`phystokv`: 10.6.2

`boot_info`: 10.14.7

`phys_lmm_init`: 10.11.3

`phys_lmm_add`: 10.11.4

`strlen`: 9.4.18

10.14.10 `base_multiboot_init_cmdline`: command-line preprocessing

Synopsis

`#include <oskit/x86/pc/base_multiboot.h>`

`void` **base_multiboot_init_cmdline**(`void`);

Description

This function breaks up the kernel command line string passed by the boot loader into independent C-language-compatible argument strings. Option strings are separated by any normal whitespace characters (spaces, tabs, newlines, etc.). In addition, strings containing an equals sign (‘`=`’) are added to the `environ` array rather than the `argv` array, effectively providing the OS with a minimal initial environment that can be specified by the user (through the boot loader) and examined by the OS using the normal `getenv` mechanism (see Section 9.4.17).

XXX example.

XXX currently no quoting support.

XXX currently just uses “kernel” as argv[0].

Dependencies

`phystokv`: 10.6.2

`strlen`: 9.4.18

`strtok`: 9.4.18

`malloc`: 9.5.2

`memcpy`: 9.4.18

`panic`: 9.8.3

10.14.11 `base_multiboot_find`: find a MultiBoot boot module by name

Synopsis

`#include <oskit/x86/pc/base_multiboot.h>`

`struct multiboot_module *`**base_multiboot_find**`(const char *`*string*`);`

Description

This is not an initialization function, but rather a utility function for the use of the client OS. Given a particular string, it searches the array of boot modules passed by the boot loader for a boot module with a matching string. This function can be easily used by the OS to locate specific boot modules by name.

If multiple boot modules have matching strings, then the first one found is returned. If any boot modules have *no* strings attached (no pun intended), then those boot modules will never be “found” by this function, although they can still be found by hunting through the boot module array manually.

Parameters

string: The string to match against the strings attached to the boot modules.

Returns

If successful, returns a pointer to the `multiboot_module` entry matched; from this structure, the actual boot module data can be found using the `mod_start` and `mod_end` elements, which contain the start and ending physical addresses of the boot module data, respectively.

If no matching boot module can be found, this function returns NULL.

Dependencies

`phystokv`: 10.6.2

`boot_info`: 10.14.7

`strcmp`: 9.4.18

10.14.12 Multiboot Specification

```
                  Excerpt from "MultiBoot Standard"
                             Version 0.6

                            March 29, 1996
     -----------------------------------------------------------------

(This contains the essential MultiBoot specification, omitting background
and related info found in ftp://flux.cs.utah.edu/flux/multiboot/.)

Contents
     * Terminology
     * Scope and Requirements
     * Details
     * Authors

     The following items are not part of the standards document,
     but are included for prospective OS and bootloader writers.
     * Example OS Code
     * Example Bootloader Code

     -----------------------------------------------------------------

Terminology

   Throughout this document, the term "boot loader" means whatever
   program or set of programs loads the image of the final operating
   system to be run on the machine. The boot loader may itself consist of
   several stages, but that is an implementation detail not relevant to
   this standard. Only the "final" stage of the boot loader - the stage
   that eventually transfers control to the OS - needs to follow the
   rules specified in this document in order to be "MultiBoot compliant";
   earlier boot loader stages can be designed in whatever way is most
   convenient.

   The term "OS image" is used to refer to the initial binary image that
   the boot loader loads into memory and transfers control to to start
   the OS. The OS image is typically an executable containing the OS
   kernel.

   The term "boot module" refers to other auxiliary files that the boot
   loader loads into memory along with the OS image, but does not
   interpret in any way other than passing their locations to the OS when
   it is invoked.

     -----------------------------------------------------------------
```

```
Scope and Requirements

  Architectures

   This standard is primarily targetted at PC's, since they are the most
   common and have the largest variety of OS's and boot loaders. However,
   to the extent that certain other architectures may need a boot
   standard and do not have one already, a variation of this standard,
   stripped of the x86-specific details, could be adopted for them as
   well.

  Operating systems

   This standard is targetted toward free 32-bit operating systems that
   can be fairly easily modified to support the standard without going
   through lots of bureaucratic rigmarole. The particular free OS's that
   this standard is being primarily designed for are Linux, FreeBSD,
   NetBSD, Mach, and VSTa. It is hoped that other emerging free OS's will
   adopt it from the start, and thus immediately be able to take
   advantage of existing boot loaders. It would be nice if commercial
   operating system vendors eventually adopted this standard as well, but
   that's probably a pipe dream.

  Boot sources

   It should be possible to write compliant boot loaders that load the OS
   image from a variety of sources, including floppy disk, hard disk, and
   across a network.

   Disk-based boot loaders may use a variety of techniques to find the
   relevant OS image and boot module data on disk, such as by
   interpretation of specific file systems (e.g. the BSD/Mach boot
   loader), using precalculated "block lists" (e.g. LILO), loading from a
   special "boot partition" (e.g. OS/2), or even loading from within
   another operating system (e.g. the VSTa boot code, which loads from
   DOS). Similarly, network-based boot loaders could use a variety of
   network hardware and protocols.

   It is hoped that boot loaders will be created that support multiple
   loading mechanisms, increasing their portability, robustness, and
   user-friendliness.

  Boot-time configuration

   It is often necessary for one reason or another for the user to be
   able to provide some configuration information to the OS dynamically
   at boot time. While this standard should not dictate how this
   configuration information is obtained by the boot loader, it should
   provide a standard means for the boot loader to pass such information
   to the OS.

  Convenience to the OS

   OS images should be easy to generate. Ideally, an OS image should
  simply be an ordinary 32-bit executable file in whatever file format
  the OS normally uses. It should be possible to 'nm' or disassemble OS
  images just like normal executables. Specialized tools should not be
  needed to create OS images in a "special" file format. If this means
  shifting some work from the OS to the boot loader, that is probably
  appropriate, because all the memory consumed by the boot loader will
  typically be made available again after the boot process is created,
  whereas every bit of code in the OS image typically has to remain in
  memory forever. The OS should not have to worry about getting into
  32-bit mode initially, because mode switching code generally needs to
  be in the boot loader anyway in order to load OS data above the 1MB
  boundary, and forcing the OS to do this makes creation of OS images
  much more difficult.

  Unfortunately, there is a horrendous variety of executable file
  formats even among free Unix-like PC-based OS's - generally a
  different format for each OS. Most of the relevant free OS's use some
  variant of a.out format, but some are moving to ELF. It is highly
  desirable for boot loaders not to have to be able to interpret all the
  different types of executable file formats in existence in order to
  load the OS image - otherwise the boot loader effectively becomes
  OS-specific again.

  This standard adopts a compromise solution to this problem. MultiBoot
  compliant boot images always either (a) are in ELF format, or (b)
  contain a "magic MultiBoot header", described below, which allows the
  boot loader to load the image without having to understand numerous
  a.out variants or other executable formats. This magic header does not
  need to be at the very beginning of the executable file, so kernel
  images can still conform to the local a.out format variant in addition
  to being MultiBoot compliant.

 Boot modules

  Many modern operating system kernels, such as those of VSTa and Mach,
  do not by themselves contain enough mechanism to get the system fully
  operational: they require the presence of additional software modules
  at boot time in order to access devices, mount file systems, etc.
  While these additional modules could be embedded in the main OS image
  along with the kernel itself, and the resulting image be split apart
  manually by the OS when it receives control, it is often more
  flexible, more space-efficient, and more convenient to the OS and user
  if the boot loader can load these additional modules independently in
  the first place.

  Thus, this standard should provide a standard method for a boot loader
  to indicate to the OS what auxiliary boot modules were loaded, and
  where they can be found. Boot loaders don't have to support multiple
  boot modules, but they are strongly encouraged to, because some OS's
  will be unable to boot without them.

    ----------------------------------------------------------------

Details
```

```
There are three main aspects of the boot-loader/OS image interface this
standard must specify:

     * The format of the OS image as seen by the boot loader.
     * The state of the machine when the boot loader starts the OS.
     * The format of the information passed by the boot loader to the OS.

  OS Image Format

An OS image is generally just an ordinary 32-bit executable file in the
standard format for that particular OS, except that it may be linked at a
non-default load address to avoid loading on top of the PC's I/O region or
other reserved areas, and of course it can't use shared libraries or other
fancy features. Initially, only images in a.out format are supported; ELF
support will probably later be specified in the standard.

Unfortunately, the exact meaning of the text, data, bss, and entry fields of
a.out headers tends to vary widely between different executable flavors, and it
is sometimes very difficult to distinguish one flavor from another (e.g. Linux
ZMAGIC executables and Mach ZMAGIC executables). Furthermore, there is no
simple, reliable way of determining at what address in memory the text segment
is supposed to start. Therefore, this standard requires that an additional
header, known as a 'multiboot_header', appear somewhere near the beginning of
the executable file. In general it should come "as early as possible", and is
typically embedded in the beginning of the text segment after the "real"
executable header. It _must_ be contained completely within the first 8192
bytes of the executable file, and must be longword (32-bit) aligned. These
rules allow the boot loader to find and synchronize with the text segment in
the a.out file without knowing beforehand the details of the a.out variant. The
layout of the header is as follows:

        +-------------------+
0       | magic: 0x1BADB002 |   (required)
4       | flags             |   (required)
8       | checksum          |   (required)
        +-------------------+
8       | header_addr       |   (present if flags[16] is set)
12      | load_addr         |   (present if flags[16] is set)
16      | load_end_addr     |   (present if flags[16] is set)
20      | bss_end_addr      |   (present if flags[16] is set)
24      | entry_addr        |   (present if flags[16] is set)
        +-------------------+

All fields are in little-endian byte order, of course. The first field is the
magic number identifying the header, which must be the hex value 0x1BADB002.

The flags field specifies features that the OS image requests or requires of
the boot loader. Bits 0-15 indicate requirements; if the boot loader sees any
of these bits set but doesn't understand the flag or can't fulfill the
requirements it indicates for some reason, it must notify the user and fail to
load the OS image. Bits 16-31 indicate optional features; if any bits in this
range are set but the boot loader doesn't understand them, it can simply ignore
them and proceed as usual. Naturally, all as-yet-undefined bits in the flags
```

word must be set to zero in OS images. This way, the flags fields serves for version control as well as simple feature selection.

If bit 0 in the flags word is set, then all boot modules loaded along with the OS must be aligned on page (4KB) boundaries. Some OS's expect to be able to map the pages containing boot modules directly into a paged address space during startup, and thus need the boot modules to be page-aligned.

If bit 1 in the flags word is set, then information on available memory via at least the 'mem_*' fields of the multiboot_info structure defined below must be included. If the bootloader is capable of passing a memory map (the 'mmap_*' fields) and one exists, then it must be included as well.

If bit 16 in the flags word is set, then the fields at offsets 8-24 in the multiboot_header are valid, and the boot loader should use them instead of the fields in the actual executable header to calculate where to load the OS image. This information does not need to be provided if the kernel image is in ELF format, but it should be provided if the images is in a.out format or in some other format. Compliant boot loaders must be able to load images that either are in ELF format or contain the load address information embedded in the multiboot_header; they may also directly support other executable formats, such as particular a.out variants, but are not required to.

All of the address fields enabled by flag bit 16 are physical addresses. The meaning of each is as follows:

* header_addr -- Contains the address corresponding to the beginning of the multiboot_header - the physical memory location at which the magic value is supposed to be loaded. This field serves to "synchronize" the mapping between OS image offsets and physical memory addresses.
* load_addr -- Contains the physical address of the beginning of the text segment. The offset in the OS image file at which to start loading is defined by the offset at which the header was found, minus (header_addr - load_addr). load_addr must be less than or equal to header_addr.
* load_end_addr -- Contains the physical address of the end of the data segment. (load_end_addr - load_addr) specifies how much data to load. This implies that the text and data segments must be consecutive in the OS image; this is true for existing a.out executable formats.
* bss_end_addr -- Contains the physical address of the end of the bss segment. The boot loader initializes this area to zero, and reserves the memory it occupies to avoid placing boot modules and other data relevant to the OS in that area.
* entry -- The physical address to which the boot loader should jump in order to start running the OS.

The checksum is a 32-bit unsigned value which, when added to the other required fields, must have a 32-bit unsigned sum of zero.

Machine State

When the boot loader invokes the 32-bit operating system, the machine must have the following state:

```
     * CS must be a 32-bit read/execute code segment with an offset of 0
       and a limit of 0xffffffff.
     * DS, ES, FS, GS, and SS must be a 32-bit read/write data segment
       with an offset of 0 and a limit of 0xffffffff.
     * The address 20 line must be usable for standard linear 32-bit
       addressing of memory (in standard PC hardware, it is wired to 0 at
       bootup, forcing addresses in the 1-2 MB range to be mapped to the
       0-1 MB range, 3-4 is mapped to 2-3, etc.).
     * Paging must be turned off.
     * The processor interrupt flag must be turned off.
     * EAX must contain the magic value 0x2BADB002; the presence of this
       value indicates to the OS that it was loaded by a
       MultiBoot-compliant boot loader (e.g. as opposed to another type
       of boot loader that the OS can also be loaded from).
     * EBX must contain the 32-bit physical address of the multiboot_info
       structure provided by the boot loader (see below).
```

All other processor registers and flag bits are undefined. This includes, in particular:

```
     * ESP: the 32-bit OS must create its own stack as soon as it needs
       one.
     * GDTR: Even though the segment registers are set up as described
       above, the GDTR may be invalid, so the OS must not load any
       segment registers (even just reloading the same values!) until it
       sets up its own GDT.
     * IDTR: The OS must leave interrupts disabled until it sets up its
       own IDT.
```

However, other machine state should be left by the boot loader in "normal working order", i.e. as initialized by the BIOS (or DOS, if that's what the boot loader runs from). In other words, the OS should be able to make BIOS calls and such after being loaded, as long as it does not overwrite the BIOS data structures before doing so. Also, the boot loader must leave the PIC programmed with the normal BIOS/DOS values, even if it changed them during the switch to 32-bit mode.

Boot Information Format

Upon entry to the OS, the EBX register contains the physical address of a 'multiboot_info' data structure, through which the boot loader communicates vital information to the OS. The OS can use or ignore any parts of the structure as it chooses; all information passed by the boot loader is advisory only.

The multiboot_info structure and its related substructures may be placed anywhere in memory by the boot loader (with the exception of the memory reserved for the kernel and boot modules, of course). It is the OS's responsibility to avoid overwriting this memory until it is done using it.

The format of the multiboot_info structure (as defined so far) follows:

```
        +-------------------+
0       | flags             |    (required)
        +-------------------+
4       | mem_lower         |    (present if flags[0] is set)
8       | mem_upper         |    (present if flags[0] is set)
        +-------------------+
12      | boot_device       |    (present if flags[1] is set)
        +-------------------+
16      | cmdline           |    (present if flags[2] is set)
        +-------------------+
20      | mods_count        |    (present if flags[3] is set)
24      | mods_addr         |    (present if flags[3] is set)
        +-------------------+
28 - 40 | syms              |    (present if flags[4] or flags[5] is set)
        +-------------------+
44      | mmap_length       |    (present if flags[6] is set)
48      | mmap_addr         |    (present if flags[6] is set)
        +-------------------+
```

The first longword indicates the presence and validity of other fields in the multiboot_info structure. All as-yet-undefined bits must be set to zero by the boot loader. Any set bits that the OS does not understand should be ignored. Thus, the flags field also functions as a version indicator, allowing the multiboot_info structure to be expanded in the future without breaking anything.

If bit 0 in the multiboot_info.flags word is set, then the 'mem_*' fields are valid. 'mem_lower' and 'mem_upper' indicate the amount of lower and upper memory, respectively, in kilobytes. Lower memory starts at address 0, and upper memory starts at address 1 megabyte. The maximum possible value for lower memory is 640 kilobytes. The value returned for upper memory is maximally the address of the first upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

If bit 1 in the multiboot_info.flags word is set, then the 'boot_device' field is valid, and indicates which BIOS disk device the boot loader loaded the OS from. If the OS was not loaded from a BIOS disk, then this field must not be present (bit 3 must be clear). The OS may use this field as a hint for determining its own "root" device, but is not required to. The boot_device field is layed out in four one-byte subfields as follows:

```
        +-------+-------+-------+-------+
        | drive | part1 | part2 | part3 |
        +-------+-------+-------+-------+
```

The first byte contains the BIOS drive number as understood by the BIOS INT 0x13 low-level disk interface: e.g. 0x00 for the first floppy disk or 0x80 for the first hard disk.

The three remaining bytes specify the boot partition. 'part1' specifies the "top-level" partition number, 'part2' specifies a "sub-partition" in the top-level partition, etc. Partition numbers always start from zero. Unused partition bytes must be set to 0xFF. For example, if the disk is partitioned using a simple one-level DOS partitioning scheme, then 'part1' contains the DOS

```
partition number, and 'part2' and 'part3' are both zero. As another example, if
a disk is partitioned first into DOS partitions, and then one of those DOS
partitions is subdivided into several BSD partitions using BSD's "disklabel"
strategy, then 'part1' contains the DOS partition number, 'part2' contains the
BSD sub-partition within that DOS partition, and 'part3' is 0xFF.

DOS extended partitions are indicated as partition numbers starting from 4 and
increasing, rather than as nested sub-partitions, even though the underlying
disk layout of extended partitions is hierarchical in nature. For example, if
the boot loader boots from the second extended partition on a disk partitioned
in conventional DOS style, then 'part1' will be 5, and 'part2' and 'part3' will
both be 0xFF.

If bit 2 of the flags longword is set, the 'cmdline' field is valid, and
contains the physical address of the the command line to be passed to the
kernel. The command line is a normal C-style null-terminated string.

If bit 3 of the flags is set, then the 'mods' fields indicate to the kernel
what boot modules were loaded along with the kernel image, and where they can
be found. 'mods_count' contains the number of modules loaded; 'mods_addr'
contains the physical address of the first module structure. 'mods_count' may
be zero, indicating no boot modules were loaded, even if bit 1 of 'flags' is
set. Each module structure is formatted as follows:

        +-------------------+
0       | mod_start         |
4       | mod_end           |
        +-------------------+
8       | string            |
        +-------------------+
12      | reserved (0)      |
        +-------------------+

The first two fields contain the start and end addresses of the boot module
itself. The 'string' field provides an arbitrary string to be associated with
that particular boot module; it is a null-terminated ASCII string, just like
the kernel command line. The 'string' field may be 0 if there is no string
associated with the module. Typically the string might be a command line (e.g.
if the OS treats boot modules as executable programs), or a pathname (e.g. if
the OS treats boot modules as files in a file system), but its exact use is
specific to the OS. The 'reserved' field must be set to 0 by the boot loader
and ignored by the OS.

NOTE: Bits 4 & 5 are mutually exclusive.

If bit 4 in the multiboot_info.flags word is set, then the following fields in
the multiboot_info structure starting at byte 28 are valid:

        +-------------------+
28      | tabsize           |
32      | strsize           |
36      | addr              |
40      | reserved (0)      |
        +-------------------+
These indicate where the symbol table from an a.out kernel image can be found.
'addr' is the physical address of the size (4-byte unsigned long) of an array
of a.out-format 'nlist' structures, followed immediately by the array itself,
then the size (4-byte unsigned long) of a set of null-terminated ASCII strings
(plus sizeof(unsigned long) in this case), and finally the set of strings
itself. 'tabsize' is equal to it's size parameter (found at the beginning of
the symbol section), and 'strsize' is equal to it's size parameter (found at
the beginning of the string section) of the following string table to which the
symbol table refers. Note that 'tabsize' may be 0, indicating no symbols, even
if bit 4 in the flags word is set.

If bit 5 in the multiboot_info.flags word is set, then the following fields in
the multiboot_info structure starting at byte 28 are valid:

        +-------------------+
28      | num               |
32      | size              |
36      | addr              |
40      | shndx             |
        +-------------------+

These indicate where the section header table from an ELF kernel is, the size
of each entry, number of entries, and the string table used as the index of
names. They correspond to the 'shdr_*' entries ('shdr_num', etc.) in the
Executable and Linkable Format (ELF) specification in the program header. All
sections are loaded, and the physical address fields of the elf section header
then refer to where the sections are in memory (refer to the i386 ELF
documentation for details as to how to read the section header(s)). Note that
'shdr_num' may be 0, indicating no symbols, even if bit 5 in the flags word is
set.

If bit 6 in the multiboot_info.flags word is set, then the 'mmap_*' fields are
valid, and indicate the address and length of a buffer containing a memory map
of the machine provided by the BIOS. 'mmap_addr' is the address, and
'mmap_length' is the total size of the buffer. The buffer consists of one or
more of the following size/structure pairs ('size' is really used for skipping
to the next pair):

        +-------------------+
-4      | size              |
        +-------------------+
0       | BaseAddrLow       |
4       | BaseAddrHigh      |
8       | LengthLow         |
12      | LengthHigh        |
16      | Type              |
        +-------------------+

where 'size' is the size of the associated structure in bytes, which can be
greater than the minimum of 20 bytes. 'BaseAddrLow' is the lower 32 bits of the
starting address, and 'BaseAddrHigh' is the upper 32 bits, for a total of a
64-bit starting address. 'LengthLow' is the lower 32 bits of the size of the
memory region in bytes, and 'LengthHigh' is the upper 32 bits, for a total of a
```

```
64-bit length. 'Type' is the variety of address range represented, where a
value of 1 indicates available RAM, and all other values currently indicated a
reserved area.

The map provided is guaranteed to list all standard RAM that should be
available for normal use.

  --------------------------------------------------------------------------

Authors

Bryan Ford
Flux Research Group
Dept. of Computer Science
University of Utah
Salt Lake City, UT 84112
multiboot@flux.cs.utah.edu
baford@cs.utah.edu

Erich Stefan Boleyn
924 S.W. 16th Ave, #202
Portland, OR, USA  97205
(503) 226-0741
erich@uruk.org

We would also like to thank the many other people have provided comments,
ideas, information, and other forms of support for our work.

  --------------------------------------------------------------------------

Example OS code can be found in the OSKit in the "kern/x86" directory and
in the oskit/x86/multiboot.h file.

  --------------------------------------------------------------------------

Example Bootloader Code (from Erich Boleyn) - The GRUB bootloader
project (http://www.uruk.org/grub) will be fully Multiboot-compliant,
supporting all required and optional features present in this
standard.  A final release has not been made, but the GRUB beta release
(which is quite stable) is available from ftp://ftp.uruk.org/public/grub/.
```

10.15 X86 PC Raw BIOS Startup

The BIOS startup code is written and functional but not yet documented or integrated into the OSKit source tree.

10.16 X86 PC DOS Startup

The DOS startup code is written and functional but not yet documented or integrated into the OSKit source tree.

10.17 Remote Kernel Debugging with GDB

In addition to the `libkern` functionality described above which is intended to facilitate implementing kernels, the library also provides complete, easy-to-use functionality to facilitate *debugging* kernels. The OSKit does not itself contain a complete kernel debugger (at least, not yet), but it contains extensive support for remote debugging using GDB, the GNU debugger. This remote debugging support allows you to run the debugger on one machine, and run the actual OS kernel being debugged on a different machine. The two machines can be of different architectures. A small "debugging stub" is linked into the OS kernel; this piece of code handles debugging-related traps and interrupts and communicates with the remote debugger, acting as a "slave" that simply interprets and obeys the debugger's commands.

This section describes remote debugging in general, applicable to any mechanism for communicating with the remote kernel (e.g., serial line or ethernet). The next section (10.18) describes kernel debugging support specific to the serial line mechanism (currently the only one implemented).

XXX diagram

One of the main advantages of remote debugging is that you can use a complete, full-featured source-level debugger, since it can run on a stable, well-established operating system such as Unix; a debugger running on the same machine as the kernel being debugged would necessarily have to be much smaller and simpler because of the lack of a stable underlying OS it can rely on. Another advantage is that remote debugging is less invasive: since most of the debugging code is on a different machine, and the remote debugging stub linked into the OS is much smaller than even a simple stand-alone debugger, there is much less that can "go wrong" with the debugging code when Strange Things start to happen due to subtle kernel bugs. The main disadvantage of remote debugging, of course, is that it requires at least two machines with an appropriate connection between them.

The GNU debugger, GDB, supports a variety of remote debugging protocols. The most common and well-supported is the serial-line protocol, which operates over an arbitrary serial line (typically null-modem) connection operating at any speed supported by the two machines involved. The serial-line debugging protocol supports a multitude of features such as multiple threads, signals, and data compression. GDB also supports an Ethernet-based remote debugging protocol and a variety of existing vendor- and OS-specific protocols.

Ths OS kit's GDB support has been tested with GDB versions 4.15 and 4.16; probably a version >= 4.15 is required.

10.17.1 Organization of remote GDB support code

The GDB remote debugging support provided by the OSKit is broken into two components: the protocol-independent component and the protocol-specific component. The protocol-independent component encapsulates all the processor architecture-specific code to handle processor traps and convert them into the "signals" understood by GDB, to convert saved state frames to and from GDB's standard representation for a given architecture, and to perform "safe" memory reads and writes on behalf of the remote user so that faulting accesses will terminate cleanly without causing recursive traps.

The protocol-specific component of the toolkit's remote GDB support encapsulates the code necessary to talk to the remote debugger using the appropriate protocol. Although this code is specific to a particular protocol, it is architecture-neutral. The OSKit currently supports only the standard serial-line protocol, although support for other protocols is planned (particularly the remote Ethernet debugging protocol) and should be easy to add.

10.17.2 Using the remote debugging code

If you are using the base environment's default trap handler, then activating the kernel debugger is extremely easy: it is simply necessary to call an appropriate initialization routine near the beginning of your kernel code; all subsequent traps that occur will be dispatched to the remote debugger. For example, on a PC, to activate serial-line debugging over COM1 using default serial parameters, simply make the call '`gdb_pc_com_init(1, 0)`'. Some example kernels are provided with the OSKit that demonstrate how to initialize and use the remote debugging facilities; see Section 1.6.1 for more information.

If you want a trap to occur *immediately* after initialization of the debugging mechanism, to transfer control to the remote debugger from the start and give you the opportunity to set breakpoints and such, simply invoke the `gdb_breakpoint` macro immediately after the call to initialize the remote debugger (see Section 10.17.11).

If your kernel uses its own trap entrypoint mechanisms or its own serial line communication code (e.g., "real" interrupt-driven serial device drivers instead of the simple polling code used by default by the toolkit), then you will have to write a small amount of "glue" code to interface the generic remote debugging support code in the toolkit with your specific OS mechanisms. However, this glue code should generally be extremely small and simple, and you can use the default implementations in the OSKit as templates to work from or use as examples.

10.17.3 Debugging address spaces other than the kernel's

Although the OSKit's remote debugging support code is most directly and obviously useful for debugging the OS kernel itself, most of the code does not assume that the kernel is the entity being debugged. In fact, it is quite straightforward to adapt the mechanism to allow remote debugging of other entities, such as user-level programs running on top of the kernel. To make the debugging stub operate on a different address space than the kernel's, it is simply necessary to override the `gdb_copyin` and `gdb_copyout` routines with alternate versions that transfer data to or from the appropriate address space. Operating systems that support a notion of user-level address spaces generally have some kind of "copyin" and "copyout" routines anyway to provide safe access to user address spaces; the replacement `gdb_copyin` and `gdb_copyout` routines can call those standard user space access routines. In addition, the trap handling mechanism may need to be set up so that only traps occurring in a particular context (e.g., within a particular user process or thread) will be dispatched to the remote debugger.

10.17.4 `gdb_state`: processor register state frame used by GDB

SYNOPSIS

```
#include <oskit/gdb.h>
struct gdb_state {
      ...;   /* architecture-specific definitions   */
};
```

DESCRIPTION

This structure represents the processor register state for the target architecture in the form in which GDB expects it. GDB uses a standard internal data structure for each processor architecture to represent the register state of a program being debugged, and most of GDB's architecture-neutral remote debugging protocols use this standard structure. The `gdb_state` structure defined by the OSKit is defined to match GDB's corresponding register state structure for each supported architecture.

10.17.5 `gdb_trap`: default trap handler for remote GDB debugging

SYNOPSIS

```
#include <oskit/gdb.h>
int gdb_trap(struct trap_state *trap_state);
```

DESCRIPTION

This function is intended to be installed as the kernel trap handler for all traps by setting each of the entries in the `base_trap_handlers` array to point to it (see Section 10.8.4), when remote GDB debugging is desired. (Alternatively, the client OS can use its own trap handlers which chain to `gdb_trap` when appropriate.) This function converts the contents of the `trap_state` structure saved by the base trap entrypoint code into the `gdb_state` structure used by GDB. It also converts the architecture-specific processor trap vector number into a suitable machine-independent signal number which can be interpreted by the remote debugger.

After converting the register state and trap vector appropriately, this function calls the appropriate protocol-specific GDB stub through the `gdb_signal` function pointer variable (see Section 10.17.9). Finally, it converts the final register state, possibly modified by the remote debugger, back into the original `trap_state` format and returns an appropriate success or failure code as described below.

On architectures that don't provide a way for the kernel to "validate" memory accesses before performing them, such as the x86, this function also provides support for "recovering" from faulting memory accesses during calls to `gdb_copyin` or `gdb_copyout` (see Sections 10.17.6 and 10.17.7). This is typically implemented using a "recovery pointer" which is set before a "safe" memory access and cleared afterwards; `gdb_trap` checks this recovery pointer, and if set, modifies the trap state appropriately and returns from the trap without invoking the protocol-specific GDB stub.

If the client OS uses its own trap entrypoint code which saves register state in a different format when handling traps, then the client OS will also need to override the `gdb_trap` function with a version that understands its custom saved state format.

PARAMETERS

trap_state: A pointer to the saved register state representing the processor state at the time the trap occurred. The saved state must be in the default format defined by the OSKit's base environment.

RETURNS

The `gdb_trap` function returns success (zero) when the remote debugger instructs the local stub to resume execution at the place it was stopped and "consume" the trap that caused the debugger to be invoked; this is the normal case.

This function returns failure (nonzero) if the remote debugger passed the same or a different signal back to the local GDB stub, instructing the local kernel to handle the trap (signal) itself. If the default trap entrypoint mechanism provided by the base environment in use, then this simply causes the kernel to panic with a register dump, since the default trap code does not know how to "handle" signals by itself. However, if the client OS uses its own trap entrypoint mechanism or interposes its own trap handler over `gdb_trap`, then it may wish to interpret a nonzero return code from `gdb_trap` as a request for the trap to be handled using the "normal" mechanism, (e.g., dispatched to the application being debugged).

DEPENDENCIES

`trap_state`:	10.8.1
`gdb_state`:	10.17.4
`gdb_signal`:	10.17.9
`gdb_trap_recover`:	10.17.8

10.17.6 `gdb_copyin`: safely read data from the subject's address space

SYNOPSIS

`#include <oskit/gdb.h>`

`int` **gdb_copyin**(`oskit_addr_t` *src_va*, `void` **dest_buf*, `oskit_size_t` *size*);

Description

The protocol-specific local GDB stub calls this function in order to read data in the address space of the program being debugged. The default implementation of this function provided by `libkern` assumes that the kernel itself is the program being debugged; thus, it acts basically like an ordinary `memcpy`. However, the client can override this function with a version that accesses a different address space, such as a user process's address space, in order to support remote debugging of entities other than the kernel.

If a fault occurs while trying to read the specified data, this function catches the fault cleanly and returns an error code rather than allowing a recursive trap to be dispatched to the debugger. This way, if the user of the debugger accidentally attempts to follow an invalid pointer or display unmapped or nonexistent memory, it will merely cause the debugger to report an error rather than making everything go haywire.

Parameters

src_va: The virtual address in the address space of the program being debugged (the kernel's address space, by default) from which to read data.

dest_buf: A pointer to the kernel buffer to copy data into. This buffer is provided by the caller, typically the local GDB stub,

size: The number of bytes of data to read into the destination buffer.

Returns

Returns zero if the transfer completed successfully, or nonzero if some or all of the source region is not accessible.

Dependencies

`gdb_trap_recover`: 10.17.8

10.17.7 `gdb_copyout`: safely write data into the subject's address space

Synopsis

`#include <oskit/gdb.h>`

`int` **gdb_copyout**(`const void *`*src_buf*, `oskit_addr_t` *dest_va*, `oskit_size_t` *size*);

Description

The protocol-specific local GDB stub calls this function in order to write data into the address space of the program being debugged. The default implementation of this function provided by `libkern` assumes that the kernel itself is the program being debugged; thus, it acts basically like an ordinary `memcpy`. However, the client can override this function with a version that accesses a different address space, such as a user process's address space, in order to support remote debugging of entities other than the kernel.

If a fault occurs while trying to write the specified data, this function catches the fault cleanly and returns an error code rather than allowing a recursive trap to be dispatched to the debugger. This way, if the user of the debugger accidentally attempts to write to unmapped or nonexistent memory, it will merely cause the debugger to report an error rather than making everything go haywire.

PARAMETERS

src_buf: A pointer to the kernel buffer containing the data to write.

dest_va: The virtual address in the address space of the program being debugged (the kernel's address space, by default) at which to write the data.

size: The number of bytes of data to transfer.

RETURNS

Returns zero if the transfer completed successfully, or nonzero if some or all of the destination region is not writable.

DEPENDENCIES

`gdb_trap_recover`: 10.17.8

10.17.8 `gdb_trap_recover`: recovery pointer for safe memory transfer routines

10.17.9 `gdb_signal`: vector to GDB trap/signal handler routine

SYNOPSIS

```
#include <oskit/gdb.h>
extern void (*gdb_signal)(int *inout_signo, struct gdb_state *inout_gdb_state);
```

DESCRIPTION

Before `gdb_trap` is called for the first time, this function pointer must be initialized to point to an appropriate GDB debugging stub, such as `gdb_serial_signal` (see Section 10.18.2). This function is called to notify the remote debugger that a relevant processor trap or interrupt has occurred, and to wait for further instructions from the remote debugger. When the function returns, execution will be resumed as described in Section 10.17.5.

PARAMETERS

inout_signo: On entry, the variable referenced by this pointer contains the signal number to transmit to the remote debugger. On return, this variable may have been modified to indicate what signal should be dispatched to the program being debugged. For example, if the variable is the same on return as on entry, then it means the remote debugger instructed the stub to "pass through" the signal to the application. If `*signo` is 0 on return from this function, it means the remote debugger has "consumed" the signal and execution of the subject program should be resumed immediately.

inout_gdb_state: On entry, this structure contains a snapshot of the processor state at the time the relevant trap or interrupt occurred. On return, the remote debugger may have modified this state; the new state should be used when resuming execution.

10.17.10 `gdb_set_trace_flag`: enable or disable single-stepping in a state frame

SYNOPSIS

```
#include <oskit/gdb.h>
void gdb_set_trace_flag(int trace_enable, [in/out] struct gdb_state *state);
```

DESCRIPTION

This architecture-specific function merely modifies the specified processor state structure to enable or disable single-stepping according to the *trace_enable* parameter. On architectures that have some kind of trace flag, this function simply sets or clears that flag as appropriate. On other architectures, this behavior is achieved through other means. This function is called by machine-independent remote debugging stubs such as `gdb_serial_signal` before resuming execution of the subject program, according to whether the remote debugger requested that the program "continue" or "step" one instruction.

PARAMETERS

trace_enable: True if single-stepping should be enabled, or false otherwise.

state: The state frame to modify.

10.17.11 `gdb_breakpoint`: macro to generate a manual instruction breakpoint

SYNOPSIS

```
#include <oskit/gdb.h>
void gdb_breakpoint(void);
```

DESCRIPTION

This is simply an architecture-specific macro which causes an instruction causing a breakpoint trap to be emitted at the corresponding location in the current function. This macro can be used to set "manual breakpoints" in program code, as well as to give control to the debugger at the very beginning of program execution as described in Section 10.17.2.

10.18 Serial-line Remote Debugging with GDB

The GDB serial-line debugging protocol is probably the most powerful and commonly-used remote debugging protocol supported by GDB; this is the only protocol for which the OSKit currently has direct support. The GDB serial-line debugging stub supplied with the OSKit is fully architecture-independent, and supports most of the major features of the GDB serial-line protocol.

For technical information on the remote serial-line GDB debugging protocol, or information on how to run and use the remote debugger itself, consult the appropriate sections of the GDB manual. This section merely describes how remote serial-line debugging is supported by the OSKit.

Note that source code for several example serial-line debugging stubs are supplied in the GDB distribution (`gdb/*-stub.c`); in fact, this code was used as a template and example for the OSKit's serial-line debugging stub. However, these stubs are highly machine-dependent and make many more assumptions about how they are used. For example, they assume that they have exclusive control of the processor's trap vector table, and are therefore only generally usable in an embedded environment where traps are *never* supposed to occur during normal operation and therefore all traps can be fielded directly by the debugger. In contrast, the serial-line debugging stub provided in the OSKit is much more generic and cleanly decomposed, and therefore should be usable in a much wider range of environments.

10.18.1 Redirecting console output to the remote debugger

If the machine on which the kernel is being debugged is truly "remote," e.g., in a different room or a completely different building, and you don't have easy access to the machine's "real" console, it is possible to make the kernel use the remote debugger as its "console" for printing status messages and such. To do this, simply write your kernel's "console" output functions (e.g., `putchar` and `puts`, if you're using the OSKit's minimal C library for console output routines such as `printf`) so that they call `gdb_serial_putchar` and `gdb_serial_puts`, described in Sections 10.18.5 and 10.18.6, respectively. The OSKit base console environment (section 10.13) does this as necessary.

This mechanism only works for console *output*: console input cannot be obtained from the remote debugger's console because the GDB serial-line debugging protocol does not currently support it. However, console input can be obtained "outside the protocol" as described in section 10.18.4.

10.18.2 `gdb_serial_signal`: primary event handler in the GDB stub

SYNOPSIS

`#include <oskit/gdb_serial.h>`

`void` **gdb_serial_signal**([in/out] `int` **signo*, [in/out] `struct gdb_state` **state*);

DESCRIPTION

This is the main trap/signal handler routine in the serial-line debugging stub; it should be called whenever a relevant processor trap occurs. This function notifies the remote debugger about the event that caused the processor to stop, and then waits for instructions from the remote debugger. The remote debugger may then cause the stub to perform various actions, such as examine memory, modify the register state, or kill the program being debugged. Eventually, the remote debugger will probably instruct the stub to resume execution, in which case this function returns with the signal number and trap state modified appropriately.

If this function receives a "kill" ('`k`') command from the remote debugger, then it breaks the remote debugging connection and then calls `panic` to reboot the machine. XXX may not be appropriate when debugging a user task; should call an intermediate function.

PARAMETERS

signo: On entry, the variable referenced by this pointer contains the signal number to transmit to the remote debugger. On return, this variable may have been modified to indicate what signal should be dispatched to the program being debugged. For example, if the variable is the same on return as on entry, then it means the remote debugger instructed the stub to "pass through" the signal to the application. If `*signo` is 0 on return from this function, it means the remote debugger has "consumed" the signal and execution of the subject program should be resumed immediately.

state: On entry, this structure contains a snapshot of the processor state at the time the relevant trap or interrupt occurred. On return, the remote debugger may have modified this state; the new state should be used when resuming execution.

Dependencies

`gdb_serial_send`: 10.18.8

`gdb_serial_recv`: 10.18.7

`gdb_copyin`: 10.17.6

`gdb_copyout`: 10.17.7

`gdb_set_trace_flag`: 10.17.10

`panic`: 9.8.3

10.18.3 `gdb_serial_exit`: notify the remote debugger that the subject is dead

Synopsis

`#include <oskit/gdb_serial.h>`

`void` **gdb_serial_exit**(`int` *exit_code*);

Description

This function sends a message to the remote debugger indicating that the program being debugged is terminating. This message causes the debugger to display an appropriate message on the debugger's console along with the *exit_code*, and causes it to break the connection (i.e., stop listening for further messages on the serial port). If no remote debugging connection is currently active, this function does nothing.

The client OS should typically call this function just before it reboots for any reason, so that the debugger does not hang indefinitely waiting for a response from a kernel that is no longer running. Alternatively, if the remote debugging facility is being used to debug a user-mode process running under the kernel, then this function should be called when *that process* terminates.

Note that despite its name, this function *does* return. It does not by itself cause the machine to "exit" or reboot or hang or whatever; it merely notifies the debugger that the subject program is *about to* terminate.

Parameters

exit_code: Exit code to pass back to the remote debugger. Typically this value is simply printed on the remote debugger's console.

Dependencies

`gdb_serial_send`: 10.18.8

`gdb_serial_recv`: 10.18.7

10.18.4 gdb_serial_getchar: input a character from the remote debugger's console

Synopsis

`#include <oskit/gdb_serial.h>`

`int` **gdb_serial_getchar**(`void`);

`static char inbuf[256];`

Description

Unfortunately, the GDB protocol doesn't support console input. However, we can simulate it with a rather horrible kludge: when the kernel first does a read from the console we take a breakpoint, allowing the user to fill an input buffer with a command such as:

call strcpy(inbuf, "hello˚")

The supplied characters will be returned from successive calls to `gdb_serial_getchar`, until `inbuf` is emptied, at which point we hit a breakpoint again.

Returns

Returns the next available character in the `inbuf` array.

Dependencies

`gdb_breakpoint`: 10.17.11

`base_critical_enter`: 10.2.5

`base_critical_leave`: 10.2.5

10.18.5 gdb_serial_putchar: output a character to the remote debugger's console

Synopsis

`#include <oskit/gdb_serial.h>`

`void` **gdb_serial_putchar**(`int` *ch*);

Description

If a remote debugging connection is currently active, this function sends the specified character to the remote debugger in a special "output" ('`O`') message which causes that character to be sent to the debugger's standard output. This allows the serial line used for remote debugging to double as a remote serial console, as described in Section 10.18.1.

Note that using `gdb_serial_putchar` by itself to print messages can be very inefficient, because a separate message is used for each character, and each of these messages must be acknowledged by the remote debugger before the next character can be sent. When possible, it is much faster to print strings of text using `gdb_serial_puts` (see Section 10.18.6). If you are using the implementation of `printf` in the OSKit's minimal C library (see Section 9.6), you can make this happen automatically by overriding `puts` with a version that calls `gdb_serial_puts` directly instead of calling `putchar` successively on each character.

If this function is called while no remote debugging connection is active, but the `gdb_serial_send` and `gdb_serial_receive` pointers are initialized to point to serial-line communication functions, then this function simply sends the specified character out the serial port using `gdb_serial_send`. This way, if the kernel attempts to print any messages before a connection has been established or after the connection has been dropped (e.g., by calling `gdb_serial_exit`), they won't confuse the debugger or cause the kernel to hang as they otherwise would, and they may be seen by the remote user if the serial port is being monitored at the time.

If the `gdb_serial_send` and `gdb_serial_receive` pointers are uninitialized (still NULL) when this function is called, it does nothing.

Parameters

ch: The character to send to the remote debugger's console.

Dependencies

`gdb_serial_send`: 10.18.8

`gdb_serial_recv`: 10.18.7

10.18.6 `gdb_serial_puts`: output a line to the remote debugger's console

Synopsis

`#include <oskit/gdb_serial.h>`

`void` **gdb_serial_puts**(`const char *`*s*);

Description

If a remote debugging connection is currently active, this function sends the specified string, followed by a newline character, to the remote debugger in a special "output" (`'O'`) message which causes the line to be sent to the debugger's standard output. This allows the serial line used for remote debugging to double as a remote serial console, as described in Section 10.18.1.

If this function is called while no remote debugging connection is active, but the `gdb_serial_send` and `gdb_serial_receive` pointers are initialized to point to serial-line communication functions, then this function simply sends the specified line out the serial port using `gdb_serial_send`. This way, if the kernel attempts to print any messages before a connection has been established or after the connection has been dropped (e.g., by calling `gdb_serial_exit`), they won't confuse the debugger or cause the kernel to hang as they otherwise would, and they may be seen by the remote user if the serial port is being monitored at the time.

If the `gdb_serial_send` and `gdb_serial_receive` pointers are uninitialized (still NULL) when this function is called, it does nothing.

Parameters

s: The string to send to the remote debugger's console. A newline is automatically appended to this string.

Dependencies

`gdb_serial_send`: 10.18.8

`gdb_serial_recv`: 10.18.7

10.18.7 `gdb_serial_recv`: vector to GDB serial line receive function

Synopsis

```
#include <oskit/gdb_serial.h>
int (*gdb_serial_recv)(void);
```

Description

Before the remote serial-line debugging stub can be used, this global variable must be initialized to point to a function to call to read a character from the serial port. The function should not return until a character has been received; the GDB stub has no notion of timeouts or interruptions.

Calling functions in the GDB serial-line debugging stub before this variable is initialized (i.e., while it is still null) is guaranteed to be harmless.

Returns

Returns the character received.

10.18.8 `gdb_serial_send`: vector to GDB serial line send function

Synopsis

```
#include <oskit/gdb_serial.h>
void (*gdb_serial_send)(int ch);
```

Description

Before the remote serial-line debugging stub can be used, this global variable must be initialized to point to a function to call to send out a character on the serial port.

Calling functions in the GDB serial-line debugging stub before this variable is initialized (i.e., while it is still null) is guaranteed to be harmless.

Returns

Returns the character received.

10.18.9 `gdb_pc_com_init`: X86 PC set up serial-line debugging over a COM port

Synopsis

```
#include <oskit/gdb.h>
void gdb_pc_com_init(int com_port, struct termios *com_params);
```

Description

This is a simple "wrapper" function which ties together all of the OSKit's remote debugging facilities to automatically create a complete remote debugging environment for a specific, typical configuration: namely, remote serial-line debugging on a PC through a COM port. This function can be used as-is if this configuration happens to suit your purposes, or it can be used as an example for setting up the debugging facilities for other configurations.

Specifically, this function does the following:

- Sets all entries in the `base_trap_handlers` array to point to `gdb_trap`. This establishes the GDB debugging trap handler as the basic handler used to handle all processor traps.

- Sets the `gdb_signal` variable to point to `gdb_serial_signal`. This "connects" the generic GDB debugging code to the serial-line debugging stub.
- Sets `gdb_serial_recv` to point to `com_cons_getchar`, and `gdb_serial_send` to point to `com_cons_putchar`. (Actually a wrapper that gives the port to those functions, as they now take a serial port as the first parameter). This connects the serial-line debugging stub to the simple polling PC COM-port console code.
- Initializes the specified COM port using the specified parameters (baud rate, etc.).
- Sets the hardware IRQ vector in the base IDT corresponding to the selected COM port to point to an interrupt handler that invokes the remote debugger with a "fake" SIGINT trap, and enables the serial port interrupt. This allows the remote user to interrupt the running kernel by pressing CTRL-C on the remote debugger's console, at least if the kernel is running with interrupts enabled.

Parameters

com_port: The COM port number through which to communicate: must be 1, 2, 3, or 4.

com_params: A pointer to a `termios` structure defining the required serial port communication parameters. If this parameter is NULL, the serial port is set up for 9600,8,N,1 by default.

Dependencies

`gdb_trap`: 10.17.5

`gdb_signal`: 10.17.9

`gdb_serial_signal`: 10.18.2

`gdb_serial_recv`: 10.18.7

`gdb_serial_send`: 10.18.8

`com_cons_init`: 10.13.8

`com_cons_getchar`: 10.13.9

`com_cons_putchar`: 10.13.10

`com_cons_enable_receive_interrupt`: 10.13.12

`base_idt`: 10.7.4

`base_raw_termios`: 10.13.4

10.19 Annotations

Kernel annotations are "markers" that can be placed in code or static data. Annotations are static and are collected into a special section of the object/executable file. How this section is created is object-file format specific and is normally handled by the default startup files (e.g, `crt0.o`).

Annotations are organized in tables which is sorted by a key value (typically the address being marked) at boot time via `anno_init`.

The basic annotation structures look like:

```
struct anno_table {
    struct   anno_entry *start;   /* first entry   */
    struct   anno_entry *end;     /* last entry    */
};
struct anno_entry {
    oskit_addr_t   val1;                /* lookup value              */
    oskit_addr_t   val2;                /* context dependent value   */
    oskit_addr_t   val3;                /* context dependent value   */
    struct         anno_table *table;   /* associated anno_table     */
};
```

Annotation tables contain a pointer to the first and last entries they contain. All entries in a table are contiguous and sorted by the key value.

Annotation entries specify the table they belong to, the key value used for lookups, and two uninterpreted values.

Though annotations can be structured arbitrarily, the OSKit supports two common kernel annotation uses: "trap tables" and "interrupt tables." These are described in the following `anno_trap` and `anno_intr` sections.

Currently, annotation support only works with the ELF binary file format (i.e., it does not work with `a.out`).

10.19.1 anno.h: X86 generic macros to place annotations in kernel code.

SYNOPSIS

```
#include <oskit/machine/anno.h>
ANNO_ENTRY(table, val1, val2, val3)
ANNO_TEXT(table, val2, val3)
```

DESCRIPTION

Contains architecture-dependent, assembly-code callable macros for placing annotations in kernel text and initialized data. No C-callable versions are included since most C compilers may reorder code making exact placement of annotations impossible.

`ANNO_ENTRY` creates a generic annotation entry associated with the indicated table and filled with the specified values.

`ANNO_TEXT` records an annotation in the given table for the current point in the text (code) segment. The current value of the program counter is placed in `val1`. The specified values for `val2` and `val3` are recorded.

10.19.2 anno_dump: dump all annotation tables

SYNOPSIS

```
#include <oskit/anno.h>
void anno_dump(void);
```

DESCRIPTION

Dumps, using `printf`, all registered annotation tables and entries. Should not be called before `anno_init`.

DEPENDENCIES

`printf`: 9.6

10.19.3 `anno_find_exact`: find annotation table exactly entry matching a value.

SYNOPSIS

`#include <oskit/anno.h>`

`struct anno_entry *`**anno_find_exact**(`struct anno_table *`*tab*, `oskit_addr_t` *val1*);

DESCRIPTION

Find an entry in the specified annotation table whose `val1` field exactly matches the specified value.

PARAMETERS

tab: annotation table to search

val1: value to search for

RETURNS

Returns a pointer to the `anno_entry` matching the value, or zero if no entry matches.

10.19.4 `anno_find_lower`: find greatest annotation table entry below a value.

SYNOPSIS

`#include <oskit/anno.h>`

`struct anno_entry *`**anno_find_lower**(`struct anno_table *`*tab*, `oskit_addr_t` *val1*);

DESCRIPTION

Find an entry in the specified annotation table with the largest `val1` field less than or equal to the specified value. If no entry has a lower or equal value, returns zero.

PARAMETERS

tab: annotation table to search

val1: value to search for

RETURNS

Returns a pointer to the appropriate `anno_entry`, or zero if no lower entry is found.

10.19.5 `anno_init`: initialize annotation tables and sort the entries.

SYNOPSIS

`#include <oskit/anno.h>`

`void` **anno_init**(`void`);

DESCRIPTION

This routine should be called once at program startup; it sorts all of the annotation entries appropriately and initializes the annotation tables they reside in.

10.19.6 `anno_intr`: X86 interrupt annotations

SYNOPSIS

`#include <oskit/anno.h>`

`struct anno_table` **anno_intr**;

ANNO_INTR(`routine, val3`)

`void` **anno_intr_handler**(`struct anno_entry *`*anno*, `struct trap_state *`*tstate*);

DESCRIPTION

The interrupt annotation table `anno_intr` contains entries which associate a handler function with ranges of addresses in the kernel. Each entry defines an action to be performed if an asynchronous exception occurs between the address associated with this entry and that associated with the following entry. When an interrupt occurs, the default OSKit interrupt handler (in `base_irq_inittab.S`) uses `anno_find_lower` to locate the correct annotation entry based on the instruction pointer at the time of the interrupt. This handler function is invoked *prior to* calling the standard interrupt handling function.

`ANNO_INTR` is a macro in `oskit/x86/anno.h`. It records an annotation in `anno_intr` for the current point in the code segment. The given *routine* and *val3* arguments are stored in the entry's `val2` and `val3` fields respectively. To disable interrupt redirection for a piece of code, place an `ANNO_INTR(0,0)` call before it.

The annotation interrupt handler function is called in the context of the interrupted thread with a pointer to the associated annotation entry and a pointer to the architecture-specific trap state for the thread. On return from the annotation handler, the actual interrupt handler is called.

An example of interrupt annotation usage in a kernel is an efficient alternative to using "spls" (i.e., raising processor priority) to protect critical sections from interrupts. By marking the critical section with an annotation entry, the kernel detects when an interrupt occurs within it and invokes an associated roll-back or roll-forward routine to back out of or complete the critical section before invoking the interrupt handler.

10.19.7 `anno_trap`: X86 trap annotations

SYNOPSIS

`#include <oskit/anno.h>`

`struct anno_table` **anno_trap**;

ANNO_TRAP(`routine, val3`)

`int` **anno_trap_handler**(`struct anno_entry *`*anno*, `struct trap_state *`*tstate*);

DESCRIPTION

The trap annotation table `anno_trap` contains entries which associate a handler function with specific addresses in the kernel. These addresses correspond to points where synchronous exceptions are expected to occur. When such an exception occurs, the default OSKit trap handler (in `base_trap_inittab.S`) uses `anno_find_exact` to locate an annotation entry based on the instruction pointer at the time of the fault. This handler function is invoked *instead of* the standard kernel trap handler in those instances.

`ANNO_TRAP` is a macro in `oskit/x86/anno.h`. It records an annotation in `anno_trap` for the current point in the code segment. The given *routine* and *val3* arguments are stored in the entry's `val2` and `val3` fields respectively.

The annotation trap handler function is called in the context of the faulting thread with a pointer to the matching annotation entry and a pointer to the architecture-specific trap state for the thread. If the handler returns zero, the OSKit trap handler will restore state from the trap state structure and resume execution. If it returns non-zero, it is considered a failed fault and the kernel will panic.

An example of trap annotation usage is a kernel `copyin` routine which must read data in the user's address space. Associating an annotation entry with the instruction which moves data from the user's address space enables the kernel to catch any access violation caused and reflect it to the user.

10.20 Boot Module Filesystem

The Boot Module (BMOD) filesystem is a memory-based filesystem exporting the OSKit filesystem interface. The initial contents of the BMOD filesystem are loaded from the Multiboot boot image. This allows an OSKit kernel to load a filesystem at boot time, possibly not even requiring an actual disk-based filesystem.

XXX multiboot strings are parsed to create hierarchical filesystem. XXX new BMODs may be added, multiboot BMODs may be modified or destroyed. XXX no guarantee on alignment of multiboot created BMOD files.

10.20.1 `oskit_bmod_init`: initialize BMOD filesystem

Synopsis

`#include <oskit/fs/bmodfs.h>`

`oskit_dir_t` ***oskit_bmod_init***(`void`);

Description

Initialize the BMOD filesystem.

Returns

Returns handle to the root of the BMOD filesystem.

10.20.2 `oskit_bmod_lock`: lock BMOD filesystem

Synopsis

`#include <oskit/fs/bmodfs.h>`

`void` **oskit_bmod_lock**(`void`);

Description

Lock the BMOD filesystem.

10.20.3 `oskit_bmod_unlock`: unlock BMOD filesystem

Synopsis

`#include <oskit/fs/bmodfs.h>`

`void` **oskit_bmod_unlock**(`void`);

Description

Unlock the BMOD filesystem.

10.20.4 `oskit_bmod_file_set_contents`: replace contents of a BMOD file

Synopsis

`#include <oskit/fs/bmodfs.h>`

`oskit_error_t` **oskit_bmod_file_set_contents**(`oskit_file_t` **file*, `void` **data*, `oskit_off_t` *size*, `oskit_off_t` *allocsize*, `oskit_bool_t` *can_sfree*, `oskit_bool_t` *inhibit_resize*);

Description

This function changes the indicated BMOD file to use the memory from [*data* - *data*+*size*-1] as its contents. *File* must be a regular BMOD file and not a directory.

Allocsize indicates the total amount of memory available for the file to use when growing and must be greater than or equal to *size*. If an attempt is made to grow the file to a size greater than *allocsize*, new memory will be allocated with `smemalign` and the file contents copied to the new memory.

If *inhibit_resize* is true, attempts to change the size of the file hereafter will fail with `OSKIT_EPERM`.

If *can_sfree* is true, `sfree` is called on the data buffer if the file grows beyond *allocsize*, is truncated to zero-length or is removed.

Parameters

file: File in the bmod filesystem whose contents are being replaced.

data: Pointer to memory to be used as the new file contents.

size: The new size of the file.

allocsize: Size of writable memory available for the file.

can_sfree: If true, indicates that the memory pointed to by *data* can be released with `sfree` when the file grows beyond *allocsize*, is truncated to zero-length, or is removed.

inhibit_resize: If true, fails any attempt to change the size of the file.

Returns

Returns zero on success, an error code otherwise.

Dependencies

`smemalign`: 9.5.10

`sfree`: 9.5.11

`memset`: 9.4.18

10.21 Signals

The signal handling facilities allow the client OS to provide compatibility with POSIX style signal handling semantics. The support provided is extremely basic and is intended to be used in conjunction with the OSKit's FreeBSD C library (see Section 14) by arranging for unexpected hardware traps to be converted into an appropriate signal and delivered through the C library. By default, the FreeBSD C library will not arrange for signals to be delivered unless the `oskit_init_libc` initialization routine is called (see Section 9.7.1). The exception are kernels linked with the POSIX threads module, which will *always* call the initialization routine.

10.21.1 `oskit_sendsig_init`: initialize kernel signal delivery

Synopsis

`#include <oskit/c/signal.h>`

`void` **oskit_sendsig_init**(`int` *(*deliver_function)*(int signo, int code, struct trap_state *ts));

Description

Initialize the kernel signal delivery mechanism, providing an upper level signal delivery routine. This delivery function will usually be an entrypoint in the C library that provides the appropriate signal semantics to the application. This entrypoint is responsible for converting the machine dependent trap state information into a suitable signal context structure, as defined by the API of the library in use. Since a pointer to the trap state structure is passed along, the callee is free to modify the machine state in way it wishes.

Dependencies

`oskit_init_libc`:	9.7.1
`oskit_init_libc`:	14.7.1

10.21.2 `oskit_sendsig`: deliver a signal

Synopsis

`#include <oskit/c/signal.h>`
`#include <oskit/x86/base_trap.h>`

`int` **oskit_sendsig**(`int` *signo*, `struct trap_state *`*state*);

Description

Propagate a machine trap to the signal handling entrypoint provided to `oskit_sendsig_init()` above. This routine is intended to be called by modules that have replaced a particular trap handler, and wish to propagate the trap to the application in the form of a signal. If the C library has not called `oskit_sendsig_init()`, the routine returns without doing anything.

Parameters

signo:	The signal number.
state:	A pointer to the trap state structure.

Returns

Returns non-zero if a C library handler has not been installed, and thus the signal could not be propagated.

10.21.3 `sendsig_trap_handler`: convert trap into a signal

SYNOPSIS

```
#include <oskit/c/signal.h>
#include <oskit/x86/base_trap.h>
```

`void` **oskit_sendsig**(`struct trap_state *`*state*);

DESCRIPTION

Convert the machine dependent trap state structure `state` (see Section 10.8.1) into a signal code, and pass that, along with the trap state, to the C library via `oskit_sendsig` above.

This routine is provided as a default trap handler that can be plugged into the `base_trap_handlers` array (see Section 10.8.4). Unexpected hardware traps are thus converted into signals and delivered to the application through the C library.

PARAMETERS

state: A pointer to the trap state structure.

Chapter 11

Symmetric Multiprocessing: `liboskit_smp.a`

11.1 Introduction

This library is designed to simplify the startup and use of multiprocessors. It defines a common interface to multiprocessor machines that is fairly platform independent.

Combined with the spin-locks provided in `libkern`, it is possible to implement a complete symmetric multiprocessor (SMP) based system using the OSKit code.

There is currently one machine-dependent interface, `smp_apic_ack` for the x86.

11.2 Supported Systems

Currently, SMP support is only provided for Intel x86 systems conforming to the Intel Multiprocessor Specification.

11.2.1 Intel x86

Systems which fully comply to the Intel MultiProcessing Specification (IMPS) should be supported. Since some of the code is based on Linux 2.0, some features (such as dual I/O APICs) are not fully supported. The APIC (Advanced Programmable Interrupt Controller) is not yet used for general interrupt delivery. Instead, all hardware interrupts are sent to the BootStrap Processor (BSP).

If a machine works with Linux 2.0 it should work with the OSKit; however, testing has been limited to a few dual-processor machines.

The SMP code must be compiled with a compiler that supports *.code16* for full functionality. The smp library will compile without it, but it will only support a single processor.

Inter-processor interrupts (IPIs) are implemented. These are currently the only interrupts received by the Application Processors (APs). IPIs allow the client OS to implement TLB-shoot-down and reschedule requests.

It is important to note that if more than one processor wishes to run in "user mode," that the per-processor data structures in `libkern` (such as `base_tss`, `base_idt`, and `base_gdt`) will have to be made per-processor.

The OSKit code has not been tested with more than two processors. Success (and failure) reports for systems with three or more processors would be appreciated.

`smp_apic_ack` mentions a potential pitfall with Intel x86 SMPs. If more than one processor tries to send an IPI to a target processor, or if a processor sends multiple IPIs without waiting for them to be processed, IPIs can get lost. It is up to the programmer to deal with this limitation.

11.2.2 External dependencies

The SMP library assumes that the base environment is usable. It starts up the Application Processors on the kernel support library's "base" data structures. It is possible (in fact required in many cases) to reload per-processor copies.

The following are symbols from the kernel support library required by the SMP library:

Dependencies

: base_gdt 10.7.1
: base_idt 10.7.4
: base_tss_load 10.7.8
: boot_info 10.14.7
: phys_mem_va 10.6.2

The LMM library is used to allocate pages of memory below 1MB. This requires the symbols:

Dependencies

: lmm_alloc_page 16.6.8
: malloc_lmm 9.5.1

These minimal C library symbols are pulled in by the SMP support code:

Dependencies

: panic 9.8.3
: printf 9.6

This library provides SMP-safe implementations for:

Dependencies

: base_critical_enter 10.2.5
: base_critical_leave 10.2.5

11.3 API reference

11.3.1 `smp_init`: Initializes the SMP startup code

Synopsis

```
#include <oskit/smp.h>
int smp_init(void);
```

Description

This function does the initial setup for the SMP support. It should be called before any other SMP library routines are used. It identifies the processors and gets them ready and waiting in a busy-loop for a "go" from the boot processor.

Note that success *does not* necessarily mean the system has multiple processors. Rather, failure indicates that the machine does not support multiple processors. `smp_get_num_cpus` should be used to determine the number of CPUs present.

Don't call this more than once…yet.

Returns

It returns 0 on success (SMP-capable system is found). E_SMP_NO_CONFIG is returned on non-IMPS-compliant x86 machines.

11.3.2 `smp_find_cur_cpu`: Return the processor ID of the current processor.

Synopsis

`#include <oskit/smp.h>`

`int` **smp_find_cur_cpu**(`void`);

Description

This function returns a unique (per-processor) integer representing the current processor. Note that the numbers are *not* guaranteed to be sequential or starting from 0, although that may be a common case.

On the x86, these numbers correspond to the processor's APIC ID, which is set by the hardware. However, these are to be treated as logical processor numbers since the smp library may do a transformation in the future.

Returns

The processor's ID.

11.3.3 `smp_find_cpu`: Return the next processor ID

Synopsis

`#include <oskit/smp.h>`

`int` **smp_find_cpu**(`int` *first*);

Description

Given a number *first*, it returns the first processor ID such that the ID is greater than or equal to that number.

In order to be assured of finding all the CPUs, the initial call should be made with an argument of 0 and subsequent calls should be made with one more than the previously returned value.

This is designed to be used as an iterator function for the client OS to determine which processor numbers are present.

Parameters

first: The processor number at which to start searching.

Returns

Returns E_SMP_NO_PROC if there are no more processors, otherwise the ID of the next processor.

11.3.4 smp_start_cpu: Starts a processor running a specified function

Synopsis

`#include <oskit/smp.h>`

`void` **smp_start_cpu**(`int` *processor_id*, `void` *(*func)*(void *data), void *data, void *stack_ptr);

Description

This releases the specified processor to start running a function with the specified stack.

Results are undefined if:

1. the processor indicated does not exist,
2. a processor attempts to start itself,
3. any processor is started more than once, or
4. any of the parameters is invalid.

`smp_find_cur_cpu` can be used to prevent calling `smp_start_cpu` on yourself. This function must be called for each processor started up by `smp_init`; if the processor is not used, then `func` should execute the halt instruction immediately.

It is up to the user to verify that the processor is started up correctly.

Parameters

processor_id: The ID of a processor found by the startup code.

func: A function pointer to be called by the processor after it has set up its stack.

data: A pointer to some structure that is placed on that stack before `func` is called.

stack_ptr: The stack pointer to be used by the processor. This should point to the top of the stack to be used by the processor, and should be large enough for `func`'s requirements.

11.3.5 smp_get_num_cpus: Returns the total number of processors

Synopsis

`#include <oskit/smp.h>`

`int` **smp_get_num_cpus**(`void`);

Description

This returns the number of processors that exist.

Returns

The number of processors that have been found. In a non-SMP-capable system, this will always return one.

11.3.6 smp_map_range: Request the OS map physical memory

Synopsis

`#include <oskit/smp.h>`

`oskit_addr_t` **smp_map_range**(`oskit_addr_t` *start*, `oskit_size_t` *size*);

Description

This function is a hook provided by the host OS to allow the SMP library to request physical memory be mapped into its virtual address space. This is called by `smp_init_paging`.

Note that this could be implemented using `osenv_mem_map_phys`.

Returns

The virtual address where the physical pages are mapped. Returns zero if unable to map the memory.

11.3.7 `smp_init_paging`: Tell the SMP code that paging is being enabled

Synopsis

`#include <oskit/smp.h>`

`int` **smp_init_paging**(`void`);

Description

This routine is called by the OS when it is ready to turn on paging. This call causes the SMP library to make call-backs to the OS to map the regions that are SMP-specific. On Intel x86 processors, this means the APICS.

Returns

Zero on success, non-zero on failure.

11.3.8 `smp_message_pass`: Send an inter-processor interrupt to another CPU

Synopsis

`#include <oskit/smp.h>`

`void` **smp_message_pass**(`int` *cpunum*);

Description

This causes the target processor to run its interrupt handler for the IPI vector, if the appropriate entry of `smp_message_pass_enable` has been set to non-zero by that processor. A processor should only modify its own `smp_message_pass_enable` entry after it is ready to start receiving IPIs.

This call offers very limited functionality. The expectation is that the OS writer will implement the desired functionality on top of this primitive.

11.3.9 `smp_message_pass_enable`:

Synopsis

smp_message_pass_enable[CPUID]

Description

This array contains an entry for each processor. If a processor is ready to start receiving inter-processor interrupts, it should set smp_message_pass_enable[smp_find_cur_cpu()] to non-zero. This is used internally by the SMP library to prevent interrupts from being delivered before the processor has set up enough state to receive them.

11.3.10 smp_apic_ack: X86 acknowledge an inter-processor interrupt

Synopsis

```
#include <oskit/x86/smp.h>
void smp_apic_ack(void);
```

Description

This routine ACKs the local APIC. The APIC must be ACKed before returning from the IPI handler. Due to limitations in the APIC design, IPIs can be lost if sent too closely together, as the APIC only handles two outstanding requests.

Chapter 12

Kernel Device Driver Support: `liboskit_dev.a`

This chapter is extremely incomplete; it is basically only a bare skeleton.

12.1 Introduction

This library provides default implementations of various functions needed by device drivers under the OSKit device driver framework. These default implementations can be used by the host OS, if appropriate, to make it easier to adopt the driver framework. The facilities provided include:

- Hardware resource management and tracking functions to allocate and free IRQs, I/O ports, DMA channels, etc.
- Device namespace management
- Memory allocation for device drivers
- Data buffer management

XXX: oskit_dev_init() call this to init libdev
XXX: oskit_dev_probe() call this after init to probe for devices

12.2 Device Registration

XXX Builds a hardware tree. An example hardware tree is shown in Figure 12.1.

Roughly... the library is initialized through a call to `oskit_dev_init`. It first does auto-configuration by calling the initialization and probe routines of the different driver sets. After auto-configuration, it builds a device tree representing the topology of the machine. While building the tree, it also organizes the drivers into "driver sets." A driver set consists of driver that share a common set of properties. After initialization, the library is ready to perform I/O requests for the OS.

`void` **oskit_dev_init**(`void`);

This function initializes the library.

12.3 Naming

To be done.

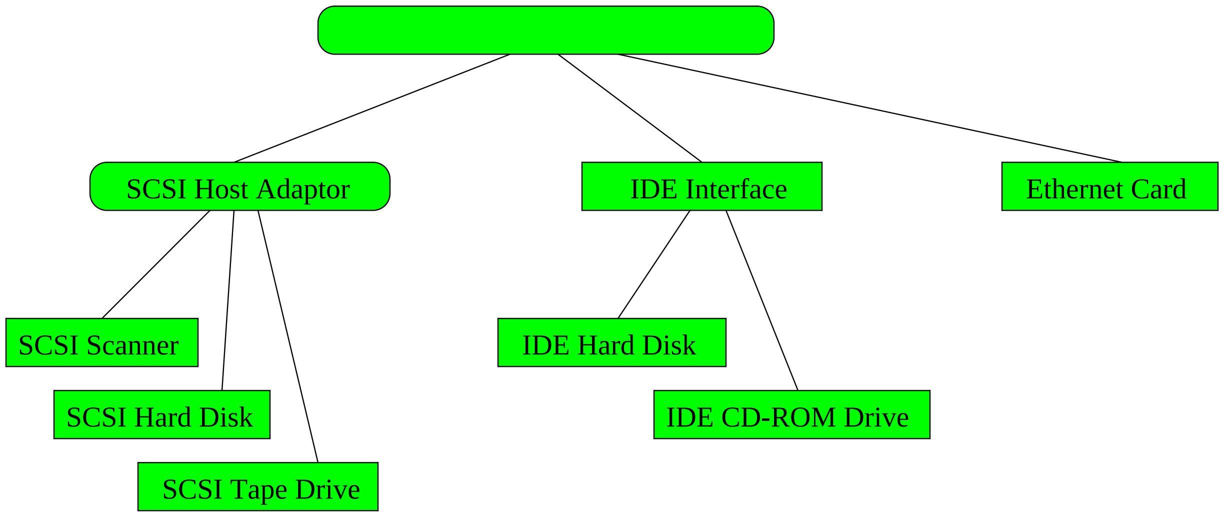

Figure 12.1: Example Hardware Tree

12.4 Memory Allocation

Default implementation uses the LMM.

12.5 Buffer Management

Provides a “simple buffer” implementation, in which buffers are simply regions of physically- and virtually-contiguous physical memory.

12.6 Processor Bus Resource Management

XXX to allocate and free IRQs, I/O ports, DMA channels, etc.

Part IV

Component Libraries

Chapter 13

POSIX Interface Library: `liboskit_posix.a`

13.1 Introduction

The POSIX library adds support for what a POSIX conformant system would typically implement as system calls. These POSIX operations are mapped to the corresponding OSKit COM interfaces. Both the minimal C library (Section 9) and the FreeBSD C library (Section 14) rely on the POSIX library to provide the necessary system level operations. For example, `fopen` in the C library will chain to `open` in the POSIX library, which in turn will chain to the appropriate `oskit_dir` and `oskit_file` COM operations. All of the pathname operations, file descriptor bookkeeping, locking, and other details normally carried out in a "kernel" implementation of a system call interface, are handled by the POSIX library. Alternatively, the POSIX library bridges differences between the COM interfaces and the functions as defined by POSIX.

Since almost all of the functions and definitions provided by the POSIX library implement well-known, well-defined ANSI and POSIX C library interfaces which are amply documented elsewhere, we do not attempt to describe the purpose and behavior of each function in this chapter. Instead, only specfic peculiarities, such as implementation interdependencies and side effects, are described here.

The following set of functions are implemented, and correspond to their POSIX.1 equivalents: accept, access, bind, chdir, chmod, chown, chroot, close, connect, creat, dup, dup2, fchdir, fchmod, fchown, fcntl, fpathconf, fstat, fsync, ftruncate, getpagesize, getpeername, getsockname, getsockopt, gettimeofday, getumask, ioctl, lchown, link, listen, lseek, lstat, mkdir, mkfifo, mknod, open, pathconf, pipe, read, readlink, readv, recv, recvfrom, rename, rmdir, select, send, sendto, setitimer, setsockopt, shutdown, sigaction, socket, socketpair, stat, symlink, truncate, umask, unlink, uname, utime, utimes, write, and writev.

13.2 Modified Functions

These functions are not fully implemented, and return an error condition if called: adjtime, getdirentries, sbrk, flock.

13.2.1 getdtablesize: get descriptor table size

The `getdtablesize` function returns a constant value, even though there is no limit on the number of open file descriptors. This function is provided for backwards comptability with older BSD system call interfaces.

13.2.2 mmap, munmap, mprotect: map files into memory

`mmap` is extremely limited in its capabilities. Anonymous memory requests are satisfied using malloc. The combination of `MAP_PRIVATE` and `PROT_WRITE` is not supported. Beyond that, the underlying file or device must provide the `oskit_openfile` COM interface.

13.2.3 getpid: get process id

`getpid` always returns zero since there is no concept of "process" in a standalone OSKit application.

13.2.4 gettimeofday: get current time

Timing functions such as `gettimeofday` are mapped to the (currently undocumented) `oskit_clock` COM interface. In essence, this interface is a very natural adaptation of the POSIX.1 real-time extensions. The system time is initialized with the `set_system_clock` routine (see Section 13.3.3).

13.3 Initialization Functions

The POSIX library exports a number of initializtion routines that the application should call when appropriate. In the `fopen` example above, the root filesystem must have been initialized by a call to `fs_init` in the application startup code. The next few sections describe the initialization routines that are provided.

13.3.1 `fs_init`: Provide a root directory defining the file system namespace

SYNOPSIS

`#include <oskit/c/fs.h>`

`oskit_error_t` **fs_init**(`oskit_dir_t` **root*);

DESCRIPTION

Provide a root directory defining the file system namespace. The current directory will initially also be this root directory. Note that the call is equivalent to `fs_mount("/", root)`.

A typical call sequence in a standalone application that uses the BMOD (section 10.20) file system might be as follows:

```
fs_init(oskit_bmod_init());
```

PARAMETERS

root: A valid `oskit_dir` instance. A reference to *root* is acquired.

RETURNS

Returns zero on success, or `OSKIT_EINVAL` if *root* is NULL.

13.3.2 `fs_release`: Release root and current directory references

SYNOPSIS

`#include <oskit/c/fs.h>`

`void` **fs_release**(`void`);

DESCRIPTION

This call releases the root and current directory references, leaving the file system module in its original uninitialized state.

13.3.3 `set_system_clock`: initialize clock support

SYNOPSIS

`#include <oskit/c/sys/time.h>`

`void` **set_system_clock**(`struct oskit_clock` **clock*);

Description

This function provides the C library with a clock device. A typical call sequence in a standalone application might be as follows:

```
#define LOCAL_TO_GMT(t) (t)->tv_sec += secondswest

void start_clock()
{
        oskit_timespec_t time;
        /* use fdev's default clock device */
        oskit_clock_t    *clock = oskit_clock_init();

        oskit_rtc_get(&time);  /* read rtc */
        LOCAL_TO_GMT(&time); /* adjust for local time */
        oskit_clock_settime(clock, &time); /* set time */

        set_system_clock(clock);
}
```

Parameters

clock: A pointer to an instance implementing the `oskit_clock` interface.

13.4 Extended API functions

13.4.1 `fs_mount`, `fs_unmount`: Compose file system name spaces

SYNOPSIS

```
#include <oskit/c/fs.h>
```

`oskit_error_t` **fs_mount**(`const char` **path*, `oskit_file_t` **subtree*);
`oskit_error_t` **fs_unmount**(`const char` **path*);

DESCRIPTION

BSD-like mount and unmount functions which the client can use to build its file system namespace out of multiple file systems.

Note that the underlying `oskit_dir` COM interface doesn't support mount points, so crossing mount points while traversing the file system space is implemented in the C library function doing the lookup (`fs_lookup`).

PARAMETERS

path: A valid pathname in the current file system space where the file system should be added or removed.

subtree: The root of the file system to be added. A reference to *subtree* is acquired.

RETURNS

Returns zero on success, or an appropriate error code.

13.4.2 `_exit`: terminate normally

DESCRIPTION

`_exit`, which terminates the calling process in Unix, calls `oskit_libc_exit` with the exit status code. `oskit_libc_exit` is declared as `void (*oskit_libc_exit)(int)`. It is initialized to a function which loops infinitely. Other OSKit libraries and user libraries can set this function pointer at will accordingly.

That OSKit kernel library will initialize this function pointer with a function that performs necessary cleanup and reboots the machine; if you set `oskit_libc_exit`, be sure to save and invoke that function if that behavior is desired.

Chapter 14

FreeBSD C Library: `liboskit_freebsd_c.a`

14.1 Introduction

The FreeBSD C library is provided as an alternative to the OSKit's minimal C library (see Section 9) so that more sophisicated applications can be built. It is derived from version 2.2.2. In addition to the standard single threaded version of the library, a multi threaded version is also built which relies on the pthread library (see Section 19) to supply the locking primitives. Both of these libraries can be found in the lib directory as oskit_freebsd_c.a and oskit_freebsd_c_r.a. In order to link with the FreeBSD C library, the application must be compiled against the FreeBSD C header files. Example kernels that are built with the FreeBSD libraries can be found in the examples/extended and examples/threads directories.

The following sections briefly describe the OSKit's implementation of the FreeBSD C library. Not all of the library is built since some parts do not make sense in the OSKit's basic environment. Those functions are listed below, as well as a list of the extended initialization functions.

14.2 posix Interface

Like the minimal C library, the FreeBSD C library depends on the POSIX library (see Section 13) to provide mappings to the appropriate OSKit COM interfaces. For example, `fopen` in the C library will chain to `open` in the POSIX library, which in turn will chain to the appropriate `oskit_dir` and `oskit_file` COM operations. Applications that link with the FreeBSDC library must also link with the COM library (but not the POSIX library since that is included as part of the FreeBSD C library archive file). Further, certain initialization routines in the POSIX library may need to be called; refer to Section 13.3 for details. A multi-threaded version of the POSIX library is also provided for applications that link with the multi-threaded version of the FreeBSDC library.

14.3 Malloc Support

The FreeBSD malloc has been completely replaced with the OSKit's basic memory allocator. Please refer to to Section 9.5 for a description of the OSKit's allocator interface. *This is a temporary measure; a future release will include a more traditional "fast" memory allocator.*

14.4 Signal Support

Rudimentary signal support is provided in both the single and multi-threaded versions of the library. As part of the C library initialization, a delivery handler is provided to the kernel library that is used to pass up hardware exceptions (see Section 10.21). Assuming the application has made the necessary calls to `sigaction` to arrange for catching signals, an exception causes the delivery function to be invoked, which converts the machine trap state into a more standard `sigcontext` structure, and passes that to the application via the signal handler. The application can freely modify the sigcontext structure; the sigcontext is copied back into the trap state when the handler returns, which then becomes the new machine state. Use caution! Note that the default action for *all* signals is to call panic and reboot the machine. Any hardware exception that that results in a signal that is blocked, also generates a panic and reboot.

14.5 Missing Functionality

Not all of the FreeBSD C library has been compiled. In some cases, the functions missing simply cannot be implemented in the OSKit's basic environment. In other cases, they are on the yet to be done list, and will eventually be added. The list of the missing functions follows is:

All of the external data representation (xdr) functions, all of the remote procedure call (rpc) functions, gethostid, sethostid, getwd, killpg, setpgrp, setrgid, setruid, sigvec, sigpause, catopen, catclose, catgets, clock, confstr, crypt, ctermid, daemon, devname, errlst, execve, execl, execlp, execle, exect, execv, execvp, getfsent, getfsspec, getfsfile, setfsent, endfsent, getbootfile, getbsize, cgetent, cgetset, cgetmatch, cgetcap, cgetnum, cgetstr, cgetustr, cgetfirst, cgetnext, cgetclose getcwd, getdomainname, getgrent, getgrnam, getgrgid, setgroupent, setgrent, endgrent, getlogin, getmntinfo, getnetgrent, innetgr, setnetgrent, endnetgrent getosreldate, getpass, getpwent, getpwnam, getpwuid, setpassent, setpwent, endpwent, getttyent, getttynam, setttyent, endttyent, getusershell, setusershell, endusershell, getvfsbyname, getvfsbytype, getvfsent, setvfsent, endvfsent, vfsisloadable, vfsload, glob, globfree, initgroups, msgctl, msgget, msgrcv, msgsnd, nice, nlist, ntp_gettime, pause, popen, psignal, user_from_uid, group_from_gid, scandir, seekdir, semconfig, semctl, semget, semop, setdomainname, sethostname, longjmperror, setmode, getmode, shmat, shmctl, shmdt, shmget, siginterrupt, siglist, sleep, sysconf, sysctl, times, timezone, ttyname, ttyslot, ualarm, unvis, usleep, valloc, vis, wait, wait3, and waitpid.

14.6 errno.h

The symbolic constants defined in `errno.h` have been redefined with the corresponding symbols defined in `oskit/error.h` (see 4.6.2), which are the error codes used through the OSKit's COM interfaces; this way, error codes from arbitrary OSKit components can be used directly as `errno` values at least by programs that use the FreeBSD C library. The main disadvantage of using COM error codes as `errno` values is that, since they don't start from around 0 like typical Unix errno values, it's impossible to provide a traditional Unix-style `sys_errlist` table for them. However, they are fully compatible with the `strerror` and `perror` routines.

14.7 Library Initialization

14.7.1 `oskit_init_libc`: Initialize the FreeBSD C library

Synopsis

`void` **oskit_init_libc**(`void`);

Description

`oskit_init_libc` allows for internal initializatons to be done. This routine *must* be called when the operating system is initialized, typically at the beginning of the `main` program.

Chapter 15

FreeBSD Math Library: `liboskit_freebsd_m.a`

15.1 Introduction

The OSKit's math library provides the traditional UNIX "math library" functions as required by the POSIX.1 and X/Open CAE specifications. These functions are required by a number of non-trivial applications, such as a Java or SR runtime system. The library itself is taken directly from the FreeBSD source tree (`/usr/src/lib/msun`), though it was developed by Sun Microsystems.

15.1.1 Architecture Dependencies

The library supports both big or little endian architectures as well as multiple standards (with respect to how exceptions are handled and reported). The OSKit library also includes the i387-optimized versions of routines that were added by FreeBSD. The OSKit version is also built for "multi-standard" support with IEEE as the default.

15.1.2 External Dependencies

The file `k_standard.c` requires either `write` to file descriptor 2 or an `fputs` and `fflush` using `stderr`.

15.1.3 Caveats

The OSKit math library is largely untested.

There is currently no other floating point support in the OSKit. Most importantly, there is no floating point emulation code to allow math functions to run on systems without hardware FPUs. There is also no support for "context switching" floating point state. The default `setjmp` and `longjmp` calls do not save and restore floating point registers, nor does the default exception handler. Thus, any multi-threaded floating point application using a thread package built on top of these mechanisms would not work correctly. Finally, the minimal C library contains no functions for conversion or printing of floating point numbers.

15.2 Functions

Following is a list of the functions supported by the library. Since these functions and their implementations are fully standard, they are not described in detail here; refer to the ISO C and Unix standards for more information.

`acos,asin,atan,atan2`: arc cosine, sine, tangent function

`acosh,asinh,atanh`: inverse hyperbolic cosine, sine, tangent functions

`cbrt`: cube root function

`ceil`: ceiling value function

`cos,sin,tan`: cosine, sine, tangent functions

`cosh,sinh,tanh`: hyperbolic cosine, sine, tangent functions

`erf,erfc`: error and complementary error functions

`exp,expm1`: exponential function

`fabs`: absolute value function

`floor`: floor value function

`fmod`: floating point remainder function

`frexp`: extract mantissa and exponent from double precision number

`gamma,lgamma`: log gamma functions

`hypot`: Euclidean distance function

`ilogb`: returns exponent part of a floating point number

`isnan`: test for NaN

`j0,j1,jn`: Bessel functions of the first kind

`ldexp`: load exponent of a floating point number

`log,log1p`: natural logarithm functions

`log10`: base 10 logarithm function

`logb`: radix-independent exponent function

`modf`: decompose floating point number

`nextafter`: return next representable double-precision floating point number

`pow`: power function

`remainder`: floating point remainder function

`rint`: round to nearest integral value

`scalb`: load exponent of a radix-independent floating point number

`sqrt`: square root function

`y0,y1,yn`: Bessel functions of the second kind

Chapter 16

List-based Memory Manager: liboskit_lmm.a

16.1 Introduction

The list-based memory manager is a component that provides simple but extremely generic and flexible memory management services. It provides functionality at a lower level than typical ANSI C `malloc`-style memory allocation mechanisms.[1] For example, the LMM does not keep track of the sizes of allocated memory blocks; that job is left to the client of the LMM library or other high-level memory allocation mechanisms. (For example, the default version of `malloc()` provided by the minimal C library, described in Section 9.5.2, is implemented on top of the LMM.)

The LMM attempts to make as few assumptions as possible about the environment in which it runs and the use to which it is put. For example, it does not assume that all allocatable "heap" memory is contained in one large continuous range of virtual addresses, as is the case in typical Unix process environments. Similarly, it does not assume that the heap can be expanded on demand (although the LMM can certainly be used in situations in which the heap *is* expandable). It does not assume that it is OK to "waste" pages on the assumption that they will never be assigned "real" physical memory unless they are actually touched. It does not assume that there is only one "type" of memory, or that all allocatable memory in the program should be managed as a single heap. Thus, the LMM is suited for use in a wide variety of environments, and can be used for both physical and virtual memory management.

The LMM has the following main features:

- Very efficient use of memory. At most fourteen bytes are wasted in a given allocation (because of alignment restrictions); there is *no* memory overhead for properly-aligned allocations.

- Support for allocating memory with specific alignment properties. Memory can be allocated at any given power-of-two boundary, or at an arbitrary offset beyond a specified power-of-two boundary. Allocation requests can also be constrained to specific address ranges or even exact addresses.

- Support for allocations of memory of a specific "type." For example, on the PC architecture, sometimes memory needs to be allocated specifically from the first 16MB of physical memory, or from the first 1MB of memory.

- Support for a concept of *allocation priority*, which allows certain memory regions to be preferred over others for allocation purposes.

- The LMM is pure and does not use any global variables; thus, different LMM pools are completely independent of each other and can be accessed concurrently without synchronization. Section 2.2 describes the pure execution environment supported by the LMM in more detail.

[1]The LMM is designed quite closely along the lines of the Amiga operating system's low-level memory management system.

- Extremely flexible management of the memory pool. LMM pools can be grown or shrunk at any time, under the complete control of the caller. The client can also "map" the free memory pool, locating free memory blocks without allocating them.

Some of the LMM's (potential) disadvantages with respect to more conventional memory allocators are:

- It requires the caller to remember the size of each allocated block, and pass its size back as a parameter to `lmm_free`. Thus, a `malloc` implemented on top of this memory manager would have to remember the size of each block somewhere.
- Since the LMM uses sequential searches through linked lists, allocations are not as blazingly fast as in packages that maintain separate free lists for different sizes of memory blocks. However, performance is still generally acceptable for many purposes, and higher-level code is always free to cache allocated blocks of commonly used sizes if extremely high-performance memory allocation is needed. (For example, a `malloc` package built on top of the LMM could do this.)
- The LMM does not know how to "grow" the free list automatically (e.g. by calling `sbrk()` or some equivalent); if it runs out of memory, the allocation simply fails. If the LMM is to be used in the context of a growable heap, an appropriate grow-and-retry mechanism must be provided at a higher level.
- In order to avoid making the LMM dependent on threading mechanisms, it does not contain any internal synchronization code. The LMM can be used in multithreaded environments, but the calling code must explicitly serialize execution while invoking LMM operations on a particular LMM heap. However, LMM operations on different heaps are fully independent and do not need to be synchronized with each other.

16.2 Memory regions

The LMM maintains a concept of a *memory region*, represented by the data type `lmm_region_t`, which represents a range of memory addresses within which free memory blocks may be located. Multiple memory regions can be attached to a single LMM pool, with different attributes attached to each region.

The attributes attached to memory regions include a set of caller-defined flags, which typically represent fundamental properties of the memory described by the region (i.e., the ways in which the region can be used), and a caller-specified *allocation priority*, which allows the caller to specify that some regions are to be preferred over others for satisfying allocation requests.

It is not necessary for all the memory addresses covered by a region to actually refer to valid memory locations; the LMM will only ever attempt to access subsections of a region that are explicitly added to the free memory pool using `lmm_add_free`. Thus, for example, it is perfectly acceptable to create a single region covering all virtual addresses from 0 to `(oskit_addr_t)-1`, as long as only the memory areas that are actually valid and usable are added to the free pool with `lmm_add_free`.

The LMM assumes that if more than one region is attached to an LMM pool, the address ranges of those regions do not overlap each other. Furthermore, the end address of each region must be larger than the start address, using unsigned arithmetic: a region must not "wrap around" the top of the address space to the bottom. These restrictions are not generally an issue, but can be of importance in some situations such as when running on the x86 with funny segment layouts.

16.2.1 Region flags

The region flags, of type `lmm_flags_t`, generally indicate certain *features* or *capabilities* of a particular range of memory. Allocation requests can specify a mask of flag bits that indicate which region(s) the allocation may be made from. For each flag bit set in the allocation request, the corresponding bit *must* be set in the region in order for the region to be considered for satisfying the allocation.

For example, on PCs, the lowest 1MB of physical memory is "special" in that only it can be accessed from real mode, and the lowest 16MB of physical memory is special in that only it can be accessed by the built-in

DMA controller. Thus, typical behavior on a PC would be to create three LMM regions: one covering the lowest 1MB of physical memory, one covering the next 15MB, and one covering all other physical memory. The first region would have the "1MB memory" and "16MB memory" bits set in its associated flags word, the second region would have only the "16MB memory" bit set, and the third region would have neither. Normal allocations would be done with a flags word of zero, which allows the allocation to be satisfied from any region, but, for example, allocations of DMA buffers would be done with the "16MB memory" flag set, which will force the LMM to allocate from either the first or second region. (In fact, this is the default arrangement used by the `libkern` library when setting up physical memory for an OS running on a PC; see Section 10.11 for more details.)

16.2.2 Allocation priority

The second attribute associated with each region, the *allocation priority*, indicates in what order the regions should be searched for free memory to satisfy memory allocation requests. Regions with a higher allocation priority value are preferred over regions with a lower priority.

Allocation priorities are typically useful in two situations. First, one section of a machine's physical memory may provide faster access than other regions for some reason, for example because it is directly connected to the processor rather than connected over a slower bus of some kind. (For example, the Amiga has what is known as "fast" memory, which typically supports faster access because it does not contend with ongoing DMA activity in the system.) In this case, if it is not likely that all available memory will be needed, the memory region describing the faster memory might be given higher priority so that the LMM will allocate from it whenever possible.

Alternatively, it can be useful to give a region a *lower* priority because it is in some way more "precious" than other memory, and should be conserved by satisfying normal allocation requests from other regions whenever possible. For example, on the PC, it makes sense to give 16MB memory a lower priority than "high" memory, and 1MB memory a still lower priority; this will decrease the likelihood of using up precious "special" memory for normal allocation requests which just need any type of memory, and causing memory shortages when special memory really *is* needed.

16.3 Example use

To make an LMM pool ready for use, a client generally proceeds in three stages:

1. Initialize the LMM pool, using `lmm_init`.
2. Add one or more memory regions to the LMM, using `lmm_add_region`.
3. Add some free memory to the pool, using `lmm_add_free`. (The free memory added should overlap at least one of the regions added in step 2; otherwise it will simply be thrown away.)

Here is an example initialization sequence that sets up an LMM pool for use in a Unix-like environment, using an (initially) 1MB memory pool to service allocations. It uses only one region, which covers all possible memory addresses; this allows additional free memory areas to be added to the pool later regardless of where they happen to be located.

```
#include <oskit/lmm.h>

lmm_t lmm;
lmm_region_t region;

int setup_lmm()
{
  unsigned mem_size = 1024*1024;
  char *mem = sbrk(mem_size);
  if (mem == (char*)-1)
```

```
    return -1;

  lmm_init(&lmm);
  lmm_add_region(&lmm, &region, (void*)0, (oskit_size_t)-1, 0, 0);
  lmm_add_free(&lmm, mem, mem_size);

  return 0;
}
```

After the LMM pool is set up properly, memory blocks can be allocated from it using any of the `lmm_alloc` functions described in the reference section below, and returned to the memory pool using the `lmm_free` function.

16.4 Restrictions and guarantees

This section describes some of the important restrictions the LMM places on its use. Many of these are restrictions one would expect to be present; however, they are listed here anyway in order to make them explicit and to make it more clear in what situations the LMM can and can't be used.

As mentioned previously, the LMM implements no internal synchronization mechanisms, so if it is used in a multithreaded environment, the caller must explicitly serialize execution when performing operations on a particular LMM pool.

If a client uses multiple LMM memory pools, then each pool must manage disjoint blocks of memory. In other words, a particular chunk of memory must never be present on two or more LMM pools at once. However, as long as the actual memory blocks in different pools are disjoint, the overall memory *regions* managed by the pools can overlap. For example, it is OK if pages 1 and 3 are managed by one LMM pool and page 2 is managed by another, as long as none of those pages are managed by two LMM pools at once.

The LMM uses the memory it manages as storage space for free list information. This means that the LMM is not suitable for managing memory that cannot be accessed directly using normal C pointer arithmetic in the local address space, or memory with special access semantics, such as flash memory. In such a situation, you must use a memory management system that stores free memory metadata separately from the free memory itself.

The LMM guarantees that it will not use any memory other than the memory explicitly given to it for its use through the `lmm_init`, `lmm_add_region`, and `lmm_add_free` calls. This implies that no "destructor" functions need to be provided by the library in order to destroy LMM pools, regions, or free lists: an LMM pool can be "destroyed" by the caller simply by overwriting or reinitializing the memory with something else. Of course, it is up to the caller to ensure that no attempts are made to use an LMM pool that has been destroyed.

16.5 Sanity checking

When the OSKit is compiled with debugging enabled (`--enable-debug`), a fairly large number of sanity checks are compiled into the LMM library to help detect memory list corruption bugs and such. Assertion failures in the LMM library can indicate bugs either in the LMM itself or in the application using it (e.g., freeing blocks twice, overwriting allocated buffers, etc.). In practice such assertion failures usually tend to be caused by the application, since the LMM library itself is quite well-tested and stable. For additional help in debugging memory management errors in applications that use the C-standard malloc/free interfaces, the OSKit's `memdebug` library can be used as well (see Section 20).

Note that the sanity checks in the LMM library are likely to slow down the library considerably under normal use, so it may be a good idea to turn off this debugging support when linking the LMM into "stable" versions of a program.

16.6 API reference

The following sections describe the functions exported by the LMM in detail. All of these functions, as well as the types necessary to use them, are defined in the header file `<oskit/lmm.h>`.

16.6.1 `lmm_init`: initialize an LMM pool

SYNOPSIS

> `#include <oskit/lmm.h>`
>
> `void` **lmm_init**(`lmm_t` **lmm*);

DESCRIPTION

> This function initializes an LMM pool. The caller must provide a pointer to an `lmm_t` structure, which is typically (but doesn't have to be) statically allocated; the LMM system uses this structure to keep track of the state of the LMM pool. In subsequent LMM operations, the caller must pass back a pointer to the same `lmm` structure, which acts as a handle for the LMM pool.
>
> Note that the LMM pool initially contains no regions or free memory; thus, immediate attempts to allocate memory from it will fail. The caller must register one or more memory regions using `lmm_add_region`, and then add some free memory to the pool using `lmm_add_free`, before the LMM pool will become useful for servicing allocations.

PARAMETERS

> *lmm*: A pointer to an uninitialized structure of type `lmm_t` which is to be used to represent the LMM pool.

16.6.2 `lmm_add_region`: register a memory region in an LMM pool

SYNOPSIS

> `#include <oskit/lmm.h>`
>
> `void` **lmm_add_region**(`lmm_t` **lmm*, `lmm_region_t` **region*, `void` **addr*, `oskit_size_t` *size*, `lmm_flags_t` *flags*, `lmm_pri_t` *pri*);

DESCRIPTION

> This function attaches a new memory region to an LMM pool. The region describes a contiguous range of addresses with specific attributes, in which free memory management may need to be done.
>
> The caller must supply a structure of type `lmm_region_t` in which the LMM can store critical state for the region. This structure must remain available for the exclusive use of the LMM for the entire remaining lifetime of the LMM pool to which it is attached. However, the contents of the structure is opaque; client code should not examine or modify its contents directly.
>
> The caller must only ensure that if multiple regions are attached to a single LMM pool, they refer to disjoint address ranges.
>
> Note that this routine does not actually make any free memory available; it merely registers a range of addresses in which free memory *might* be made available later. Typically this call is followed by one or more calls to `lmm_add_free`, which actually adds memory blocks to the pool's free memory list.
>
> The act of registering a new region does not cause any of the memory described by that region to be accessed or modified in any way by the LMM; only the `lmm_region_t` structure itself is

modified at this point. The LMM will only access and modify memory that is explicitly added to the free list using `lmm_add_free`. This means, for example, that it is safe to create a single region with a base of 0 and a size of `(oskit_size_t)-1`, regardless of what parts of that address range actually contain valid memory.

See Section 16.2 for general information on memory regions.

PARAMETERS

lmm: The LMM pool to which the region should be added.

region: A pointer to a structure in which the LMM maintains the critical state representing the region. The initial contents of the structure don't matter; however, the structure must remain available and untouched for the remaining lifetime of the LMM pool to which it is attached.

addr: The start address of the region to add. Different regions attached to a single LMM pool must cover disjoint areas of memory.

size: The size of the region to add. Must be greater than zero, but no more than `(oskit_addr_t)-1 - addr`; in other words, the region must not wrap around past the end of the address space.

flags: The attribute flags to be associated with the region. Allocation requests will be satisfied from this region only if all of the flags specified in the allocation request are also present in the region's flags word.

pri: The allocation priority for the region, as a signed integer. Higher priority regions will be preferred over lower priority regions for satisfying allocations.

16.6.3 `lmm_add_free`: add a block of free memory to an LMM pool

SYNOPSIS

`#include <oskit/lmm.h>`

`void` **lmm_add_free**(`lmm_t *`*lmm*, `void *`*block*, `oskit_size_t` *size*);

DESCRIPTION

This routine declares a range of memory to be available for allocation, and attaches that memory to the specified LMM pool. The memory block will be made available to satisfy subsequent allocation requests.

The caller can specify a block of any size and alignment, as long as the block does not wrap around the end of the address space. The LMM may discard a few bytes at the beginning and end of the block in order to enforce internal alignment constraints; however, the LMM will never touch memory *outside* the specified block (unless, of course, that memory is part of another free block).

If the block's beginning or end happens to coincide exactly with the beginning or end of a block already on the free list, then the LMM will merge the new block with the existing one. Of course, the block may be further subdivided or merged later as memory is allocated from the pool and returned to it.

The new free block will automatically be associated with whatever region it happens to fall in. If the block crosses the boundary between two regions, then it is automatically split between the regions. If part of the block does not fall within *any* region, then that part of the block is simply ignored and forgotten about. (By extension, if the entire block does not overlap any region, the entire block is dropped on the floor.)

Parameters

lmm: The LMM pool to add the free memory to.

block: The start address of the memory block to add. There are no alignment restrictions.

size: The size of the block to add, in bytes. There are no alignment restrictions, but the size must not be so large as to wrap around the end of the address space.

16.6.4 `lmm_remove_free`: remove a block of memory from an LMM pool

Synopsis

`#include <oskit/lmm.h>`

`void` **lmm_remove_free**(`lmm_t *`*lmm*, `void *`*block*, `oskit_size_t` *size*);

Description

This routine is complementary to `lmm_add_free`: it removes all free memory blocks in a specified address range from an LMM memory pool. After this call completes, unless the caller subsequently adds memory in this range back onto the LMM pool using `lmm_add_free` or `lmm_free` it is guaranteed that no subsequent memory allocation will return a memory block that overlaps the specified range.

The address range specified to this routine does not actually all have to be on the free list. If the address range contains several smaller free memory blocks, then all of those free blocks are removed from the pool without touching or affecting any memory parts of the address range that *weren't* in the free memory list. Similarly, if a free block crosses the beginning or end of the range, then the free block is "clipped" so that the part previously extending into the address range is removed and thrown away.

One use for this routine is to reserve a specific piece of memory for some special purpose, and ensure that no subsequent allocations use that region. For example, the example MultiBoot boot loaders in the OSKit use this routine to reserve the address range that will eventually be occupied by the OS executable being loaded, ensuring that none of the information structures to be passed to the OS will overlap with the final position of its executable image.

This routine works by finding all the free memory in the given range and allocating it. This means that if blocks were allocated in that range *before* the `lmm_remove_free` call and then freed afterwords, then they will be candidates for future allocations.

Parameters

lmm: The LMM pool from which to remove free memory.

block: The start address of the range in which to remove all free memory.

size: The size of the address range.

16.6.5 `lmm_alloc`: allocate memory

Synopsis

`#include <oskit/lmm.h>`

`void *`**lmm_alloc**(`lmm_t *`*lmm*, `oskit_size_t` *size*, `lmm_flags_t` *flags*);

Description

This is the primary routine used to allocate memory from an LMM pool. It searches for a free memory block of the specified size and with the specified memory type requirements (indicated by the *flags* argument), and returns a pointer to the allocated memory block. If no memory block of sufficient size and proper type can be found, then this function returns NULL instead.

Note that unlike with *malloc*, the caller must keep track of the size of allocated blocks in order to allow them to be freed properly later.

Parameters

lmm: The memory pool from which to allocate.

size: The number of contiguous bytes of memory needed.

flags: The memory type required for this allocation. For each bit set in the *flags* parameter, the corresponding bit in a region's flags word must also be set in order for the region to be considered for allocation. If the *flags* parameter is zero, memory will be allocated from any region.

Returns

Returns a pointer to the memory block allocated, or NULL if no sufficiently large block of the correct type is available. The returned memory block will be at least doubleword aligned, but no other alignment properties are guaranteed by this routine.

16.6.6 `lmm_alloc_aligned`: allocate memory with a specific alignment

Synopsis

`#include <oskit/lmm.h>`

`void *`**lmm_alloc_aligned**(`lmm_t *`*lmm*, `oskit_size_t` *size*, `lmm_flags_t` *flags*, `int` *align_bits*, `oskit_addr_t` *align_ofs*);

Description

This routine allocates a memory block with specific alignment constraints. It works like `lmm_alloc`, except that it enforces the rule that the lowest *align_bits* bits of the address of the returned block must match the lowest *align_bits* of *align_ofs*. In other words, *align_bits* specifies an alignment boundary as a power of two, and *align_ofs* specifies an offset from “natural” alignment. If no memory block with the proper requirements can be found, then this function returns NULL instead.

Parameters

lmm: The memory pool from which to allocate.

size: The number of contiguous bytes of memory needed.

flags: The memory type required for this allocation. For each bit set in the *flags* parameter, the corresponding bit in a region's flags word must also be set in order for the region to be considered for allocation. If the *flags* parameter is zero, memory will be allocated from any region.

align_bits: The number of low bits of the returned memory block address that must match the corresponding bits in *align_ofs*.

align_ofs: The required offset from natural power-of-two alignment. If *align_ofs* is zero, then the returned memory block will be naturally aligned on a 2^{align_bits} boundary.

Returns

Returns a pointer to the memory block allocated, or NULL if no memory block satisfying the specified requirements can be found.

16.6.7 `lmm_alloc_gen`: allocate memory with general constraints

Synopsis

`#include <oskit/lmm.h>`

`void *`**lmm_alloc_gen**(`lmm_t *`*lmm*, `oskit_size_t` *size*, `lmm_flags_t` *flags*, `int` *align_bits*, `oskit_addr_t` *align_ofs*, `oskit_addr_t` *in_min*, `oskit_size_t` *in_size*);

Description

This routine allocates a memory block meeting various alignment and address constraints. It works like `lmm_alloc_aligned`, except that as an additional constraint, the returned memory block must fit entirely in the address range specified by the *in_min* and *in_size* parameters.

If *in_size* is equal to *size*, then memory will only be allocated if a block can be found at *exactly* the address specified by *in_min*; i.e. the returned pointer will either be *in_min* or NULL.

Parameters

lmm: The memory pool from which to allocate.

size: The number of contiguous bytes of memory needed.

flags: The memory type required for this allocation. For each bit set in the *flags* parameter, the corresponding bit in a region's flags word must also be set in order for the region to be considered for allocation. If the *flags* parameter is zero, memory will be allocated from any region.

align_bits: The number of low bits of the returned memory block address that must match the corresponding bits in *align_ofs*.

align_ofs: The required offset from natural power-of-two alignment. If *align_ofs* is zero, then the returned memory block will be naturally aligned on a 2^{align_bits} boundary.

in_min: Start address of the address range in which to search for a free block. The returned memory block, if found, will have an address no lower than *in_min*.

in_size: Size of the address range in which to search for the free block. The returned memory block, if found, will fit entirely within this address range, so that $mem\_block + size <= in\_min + in\_size$.

Returns

Returns a pointer to the memory block allocated, or NULL if no memory block satisfying all of the specified requirements can be found.

16.6.8 `lmm_alloc_page`: allocate a page of memory

Synopsis

`#include <oskit/lmm.h>`

`void *`**lmm_alloc_page**(`lmm_t *`*lmm*, `lmm_flags_t` *flags*);

Description

This routine allocates a memory block that is exactly one minimum-size hardware page in size, and is naturally aligned to a page boundary. The same effect can be achieved by calling `lmm_alloc_aligned` with appropriate parameters; this routine merely provides a simpler interface for this extremely common action.

Parameters

lmm: The memory pool from which to allocate.

flags: The memory type required for this allocation. For each bit set in the *flags* parameter, the corresponding bit in a region's flags word must also be set in order for the region to be considered for allocation. If the *flags* parameter is zero, memory will be allocated from any region.

Returns

Returns a pointer to the memory page allocated, or NULL if no naturally-aligned page can be found.

16.6.9 `lmm_free`: free previously-allocated memory

Synopsis

`#include <oskit/lmm.h>`

`void` **lmm_free**(`lmm_t` **lmm*, `void` **block*, `oskit_size_t` *size*);

Description

This routine is used to return a memory block allocated with one of the above `lmm_alloc` functions to the LMM pool from which it was allocated.

Parameters

lmm: The memory pool from which the memory block was allocated.

block: A pointer to the memory block to free, as returned by one of the `lmm_alloc` functions.

size: The size of the memory block to free, as specified to the allocation function when the block was allocated.

16.6.10 `lmm_free_page`: free a page allocated with lmm_alloc_page

Synopsis

`#include <oskit/lmm.h>`

`void` **lmm_free_page**(`lmm_t` **lmm*, `void` **block*);

Description

This routine simply calls `lmm_free` with `PAGE_SIZE` as the *size* argument, providing a companion to `lmm_alloc_page`.

Parameters

lmm: The memory pool from which the page was allocated.

block: A pointer to the page to free, as returned by the `lmm_alloc_page` function.

16.6.11 `lmm_avail`: find the amount of free memory in an LMM pool

Synopsis

`#include <oskit/lmm.h>`

`oskit_size_t` **lmm_avail**(`lmm_t` **lmm*, `lmm_flags_t` **flags*);

Description

This routine returns the number of bytes of free memory currently exist in the specified LMM memory pool of a certain memory type, specified by the *flags* argument.

Note that the returned value does not imply that a block of that size can be allocated; due to fragmentation it may only be possible to allocate memory in significantly smaller chunks.

Parameters

lmm: The LMM pool in which to tally free memory.

flags: The memory type to determine the availability of. Only memory regions whose flags words contain all the bits set in the *flags* parameter will be considered in counting available memory. If *flags* is zero, then *all* free memory in the LMM pool will be counted.

Returns

Returns the number of bytes of free memory available of the requested memory type.

16.6.12 `lmm_find_free`: scan a memory pool for free blocks

Synopsis

`#include <oskit/lmm.h>`

`void` **lmm_find_free**(`lmm_t` **lmm*, [in/out] `oskit_addr_t` **inout_addr*, [out] `oskit_size_t` **out_size*, [out] `lmm_flags_t` **out_flags*);

Description

This routine can be used to locate free memory blocks in an LMM pool. It searches the pool for free memory starting at the address specified in **inout_addr*, and returns a description of the lowest block of available memory starting at at least this address. The address and size of the next block found are returned in **inout_addr* and **out_size*, respectively, and the memory type flags associated with the region in which the block was found are returned in **out_flags*. If no further free memory can be found above the specified address, then this routine returns with **out_size* set to zero.

If the specified **inout_addr* points into the middle of a free block, then a description of the *remainder* of the block is returned, i.e. the part of the block starting at **inout_addr* and extending to the end of the free block.

This routine does not actually cause any memory to be allocated; it merely reports on available memory blocks. The caller must not actually attempt to use or modify any reported blocks without allocating them first. The caller can allocate a block reported by this routine using `lmm_alloc_gen`, using its *in_min* and *in_size* parameters to constrain the address of the allocated block to exactly the address reported by `lmm_find_free`. If this allocation is done immediately after the call to `lmm_find_free`, without any intervening memory allocations, then the allocation is guaranteed to succeed. However, any intervening memory allocation operations will effectively invalidate the information returned by this routine, and a subsequent attempt to allocate the reported block may fail.

Parameters

lmm: The LMM pool in which to search for free memory.

inout_addr: On entry, the value pointed to by this parameter must be the address at which to start searching for free memory. On return, it contains the start address of the next free block actually found.

out_size: On return, the value pointed to by this parameter contains the size of the next free memory block found, or zero if no more free blocks could be located.

out_flags: On return, the value pointed to by this parameter contains the flags word associated with the region in which the next free memory block was found.

16.6.13 `lmm_dump`: display the free memory list in an LMM pool

Synopsis

`#include <oskit/lmm.h>`

`void` **lmm_dump**(`lmm_t` **lmm*);

Description

This routine is primarily used for debugging the LMM and the code that uses it. It scans through the LMM pool and calls `printf` to display each attached memory region and all the blocks of free memory currently contained in each.

Chapter 17

Address Map Manager: `liboskit_amm.a`

17.1 Introduction

The *Address Map Manager* (AMM) library manages collections of resources where each element of a collection has a name (address) and some set of attributes. These collections are described by *address maps*. Examples of resources which might be managed by address maps include:

- Swap space. An OS might use a disk partition as backing store for memory. Here the names are disk block (sector) numbers and the single attribute of a block is either "free" or "allocated."
- A process address space. An OS keeps track of which portions of a process's virtual address space are allocated and what access permissions are allowed to the allocated regions. Here the names are virtual addresses or virtual page numbers and the attributes include "read," "write," and "execute" as well as "allocated" or "free."

Logically, an address map is an array of attribute values indexed by address. However, such an implementation would be impractical for all but the smallest address ranges so the AMM coalesces ranges of contiguous addresses with identical attributes and describes each such range with an *address map entry*. Hence address maps are collections of map entries with every possible address contained in exactly one entry.

The AMM library includes routines to create and destroy address maps, lookup addresses within a map, modify the attributes of addresses within a map, and iterate over all entries in a map. The library is responsible for maintaining a consistent and efficient representation of the map. The complete set of routines is described in section 17.8.

The AMM is a *pure* component: it uses no static or global variables, so clients can freely make concurrent AMM calls on different address maps without synchronization. However, in interruptible or multithreaded environments, the client is responsible for synchronizing calls that manipulate a single AMM pool. Section 2.2 describes the pure execution environment supported by the AMM in more detail.

17.2 Addresses and attributes

The AMM attempts to be general by making very few assumptions about addresses or attributes. An address is defined as an arbitrary unsigned integer of some reasonable size (typically 32 or 64 bits). The value of the integer is not interpreted by AMM in any way. The only assumption that the AMM makes about addresses is that two addresses are considered *contiguous* (i.e., represent adjacent pieces of a resource) if their integer values are consecutive. This is an intuitive assumption and not likely to be a restriction for many (any?) uses.

Attributes are represented using an opaque flag word (again, an integer of reasonable size). The only assumption that AMM makes about attributes is that the attributes of two addresses are *identical* if their flag words have the same integer value. This again is a fairly obvious assumption. Note that though the flag word appears to limit the number of possible attributes to 32 or 64 single-bit values, the user can actually define arbitrary attributes for an address by associating additional state with the address. Section 17.5 explains how this is done.

17.3 Address maps and entries

Each address map is represented by an `amm_t` structure. This structure is allocated by the user and initialized once at startup via a library routine. After initialization, a pointer to the structure is used as an opaque handle for all subsequent AMM calls.

Many of the library routines take map entries as parameters or pass them back as return values. Thus the AMM defines an opaque `amm_entry_t` type and includes routines to obtain the starting and ending addresses, size and attributes associated with the entry.

Implementing an address map as a collection of map entries requires that the AMM be able to *split* and *join* map entries as the attributes of addresses change. An entry is split into two entries when some sub-range of the entry changes attributes. Analogously, two entries which are adjacent may be joined when the attributes of one change to be identical to those of the other.

The AMM library provides hooks to allow the user to associate additional state with map entries. These hooks include explicit map entry parameters to some routines as well as callback interfaces to handle allocation, deallocation, splitting and joining map entries. Section 17.5 contains complete details.

17.4 Simple interface

Though the AMM is general enough to allow the user to associate arbitrary attributes with address map entries, one of the most common uses of address maps involves keeping track of which elements of a resource are free and which are available.

The AMM library provides an interface to make this "simple" use more convenient and, to some extent, more efficient. It is important to note that the simple interfaces are just wrappers for the generic interfaces described in Section 17.5. They are not a separate implementation. Thus, simple and generic routines can be called for the same map.

In the simple interface, the AMM pre-defines three attribute values: `AMM_ALLOCATED`, `AMM_FREE`, and `AMM_RESERVED`, and provides a set of routines that knows their semantics. Addresses that are `AMM_ALLOCATED` represent resources that are in use and unavailable until freed. Addresses that are `AMM_FREE` represent resources that are available for allocation. `AMM_RESERVED` addresses represent "out of range" resources that can not be allocated or freed.

`amm_init` performs the one-time initialization of an address map including restricting the "valid" range of addresses. Addresses within the specified range are marked as free and all others marked as reserved. `amm_reserve` allows additional areas of the address map to be reserved. This can be used to represent "holes" within an otherwise contiguous address map. `amm_allocate` allocates a range of a given size from the valid portion of the address map. This routine includes a "hint" address indicating where to start searching for free space in which to locate the range. Finally, `amm_deallocate` can be used to free a range of allocated space.

This set of routines is sufficient to implement basic resource management similar to that provided by UNIX resource maps. For example:

```
#include <oskit/amm.h>
#include <assert.h>

struct simple_rmap {
    amm_t amm;
};

void simple_rminit(struct simple_rmap *smap, oskit_addr_t saddr, oskit_addr_t eaddr)
{
    assert(saddr != 0);    /* zero is used to indicate allocation errors */
    amm_init(&smap->amm, saddr, eaddr);
}

oskit_addr_t simple_rmalloc(struct simple_rmap *smap, oskit_size_t size)
{
    oskit_addr_t addr = 0;

    return amm_allocate(&smap->amm, &addr, size, 0) ? 0 : addr;
}

void simple_rmfree(struct simple_rmap *smap, oskit_addr_t addr, oskit_size_t size)
{
    (void)amm_deallocate(&smap->amm, addr, size);
}
```

Three additional routines enable the simple interface to be used for another common OS usage: managing process/task address spaces. `amm_protect` changes the attributes associated with a range of the map. Attributes are restricted to whatever can be described in the flag word and should not clash with the `AMM_ALLOCATED`, `AMM_FREE`, and `AMM_RESERVED` bits, but there are typically only a small number of address space protection bits anyway so this isn't likely to be a problem. `amm_find_addr` looks up an address, returning a pointer to the entry containing the address. A final function, `amm_iterate` takes a function pointer and calls that function for every entry in the map, passing the map and map entry pointers as arguments. This allows the user to traverse the map, examining each entry in turn.

When using the simple AMM interface, map entries are allocated with `malloc`, deallocated with `free`, and the default library routines are used to split and join entries based on the attributes in the flag word.

17.5 Generic interface

The more general AMM interface routines provide the same basic capability as the simple interface. `amm_init_gen` provides the one-time initialization of a map, `amm_find_gen` locates an address range in a map, `amm_modify` changes the attributes of an address range, and `amm_iterate_gen` allows a user-provided function to be called for selected entries in a map.

The primary differences between the two interfaces are that the generic interface allows fine-grained control over the selection of addresses and attributes within a map and it permits user-directed allocation and management of address map entries.

Fine-grained control of address selection enables the user to specify exact alignment and offset criteria when attempting to find a range of addresses in a map using `amm_find_gen`. For example, an address space manager can allocate arbitrary-sized, page-aligned address ranges using this technique.

Similarly, fine-grained control of attribute selection enables inexact attribute matches when locating an address range using `amm_find_gen`. In the address space manager example, this would allow an address space deallocation routine to match any allocated map entry regardless of its additional protection bits. These mask and match parameters are also used in `amm_iterate_gen` to selectively iterate over entries in a map.

User-directed allocation and management of entries allows the user to embed the standard `amm_entry_t` structure in a larger application specific structure thus associating additional state with each entry. Allocation control is addressed in two ways. First, is by providing explicit map entry parameters to `amm_init_gen` and `amm_modify`, the two routines that result in an entry being added to a map. In this way, the caller can pass in the wrapped `amm_entry_t` structure to use. Second, as part of the `amm_init_gen` call, the user can register a routine to be called when a map entry is to be split. Since entry splitting results in one or two new entries being created, this routine allows the user to provide the wrapped entries to use.

User-directed management is attained through the above allocation hooks as well as additional initialization-time registered callback routines for joining and deallocation of map entries. The join routine permits the user to determine if two entries which are adjacent in address and have equivalent attributes flags can be coalesced into a single entry. If they can be joined, this routine is responsible for merging the extended attribute information and returning a new entry with this state. The deallocation routine enables user control over situations where the AMM library destroys map entries, either left over entries from a join operation or entries that are completely replaced by a single entry in `amm_modify`. This routine is responsible for freeing the extended attribute information.

17.6 Generic interface example

As an example of a non-trivial use of AMM, consider a Fluke address space manager which maintains a map to describe the address space in which threads run. In Fluke, an address space is populated with memory by mapping memory from other address spaces into the space using kernel-managed mapping objects. Hence each allocated area of the address space would be described by a composite entry consisting of an `amm_entry_t` structure and a Fluke mapping object:

```
#include <oskit/amm.h>
#include <fluke/mapping.h>

struct as {
    amm_t map;
    ...
}

struct as_entry {
    amm_entry_t entry;
    fluke_mapping_t mapping;
};
```

Note that unallocated (`AMM_FREE`) entries don't need to have mapping objects associated with them and could just be standard `amm_entry_t` structures. Thus, a user-provided entry allocation routine can create and return the appropriate structure depending on whether it is for an allocated or free entry. In the case of allocated map entries, the Fluke mapping object can be created at this time. Similarly, a user-provided deallocation routine can free an entry appropriately, destroying mapping objects as necessary:

```
amm_entry_t *as_entry_alloc(amm_t *amm, oskit_addr_t addr, oskit_size_t size, int flags)
{
    struct as_entry *aentry;

    if (flags == AMM_FREE)
        return malloc(sizeof(amm_entry_t));
    if ((aentry = malloc(sizeof *aentry)) == 0)
        return 0;
    if (fluke_mapping_create(&aentry->mapping)) {
        free(aentry);
        return 0;
    }
    return &aentry->entry;
}

void as_entry_free(amm_t *amm, amm_entry_t *entry)
{
    struct as_entry *aentry;

    if (amm_entry_flags(entry) == AMM_FREE)
        free(sizeof *entry);
    else {
        aentry = (struct as_entry *)entry;
        fluke_mapping_destroy(&aentry->mapping);
        free(sizeof *aentry);
    }
}
```

Address space is allocated, freed, or the protections changed, using `amm_modify`. Here the address space manager first creates an entry of the correct type, creating and initializing the mapping object as necessary. It then calls `amm_modify` with that entry:

```
int as_allocate(struct as *map, oskit_addr_t addr, oskit_size_t size, int prot)
{
    struct as_entry *aentry;
    int rc;

    /* check range to ensure it is available, etc. */
    ...

    aentry = (struct as_entry *)as_entry_alloc(&map->amm, addr, size, prot|AMM_ALLOCATED);

    /* setup Fluke mapping state */
    ...
    fluke_mapping_set_state(&aentry->mapping, ...);

    rc = amm_modify(&map->amm, addr, size, prot|AMM_ALLOCATED, &aentry->entry);
    ...
}

int as_deallocate(struct as *map, oskit_addr_t addr, oskit_size_t size)
{
    int rc;

    /* as_entry_free will destroy all Fluke mappings */
    rc = amm_modify(&map->amm, addr, size, AMM_FREE, 0);
    ...
}

int as_protect(struct as *map, oskit_addr_t addr, oskit_size_t size, int prot)
{
    struct as_entry *aentry;
    int rc;

    /* check range to ensure it is allocated, etc. */
    ...

    aentry = (struct as_entry *)as_entry_alloc(&map->amm, addr, size, prot|AMM_ALLOCATED);

    /* setup Fluke mapping state */
    ...
    fluke_mapping_set_state(&aentry->mapping, ...);

    rc = amm_modify(&map->amm, addr, size, prot|AMM_ALLOCATED, &aentry->entry);
    ...
}
```

Note that since AMM_FREE entries are just standard `amm_entry_t` structures, it is not necessary to pass `amm_modify` an explicit entry parameter when freeing address space. In this situation, `amm_modify` will call the user-provided allocation routine which will allocate a basic map entry structure based on the fact that the flag parameter is `AMM_FREE`. This works since no user initialization of the `amm_entry_t` is required. In general, the parameters passed to the allocation routine do not provide sufficient information to initialize user-extended attributes and thus these entries must be initialized and passed to `amm_modify`.

In `amm_modify`, if the modification results in an entry being split, the user-provided split routine is called with the entry and an address at which the entry is to be broken. The split routine will create a new entry of the appropriate type, creating and initializing a Fluke mapping object if necessary. It then adjusts any Fluke mapping object associated with the existing entry to reflect the split and returns both objects.

After isolating the range identified by the address and size parameters, `amm_modify` discards it by calling the user-provided deallocation routine for every entry within the range. The deallocation routine will destroy any Fluke mapping object associated with an entry.

Once the address range has been cleared, `amm_modify` inserts the user-provided entry in the map. The newly inserted entry may now be compatible with one or both of its neighbors in which case the user-provided join routine is called (possibly twice) with the entries to join as parameters. If the entries can be merged, the join routine modifies one of the existing entries to cover the joined range and returns a pointer to that entry. Note that, even though the map entry addresses and attributes are compatible to join, the user-provided join routine may choose not to join them. In this example, it is possible that the source addresses of the two mapping objects are not adjacent and hence the two mappings cannot be combined into one. Thus, the join function will fail and the two entries will remain separate.

The following code illustrates the split and join functions described:

```
int as_entry_split(amm_t *amm, amm_entry_t *entry, oskit_addr_t addr,
                   amm_entry_t **head, amm_entry_t **tail)
{
    amm_entry_t *nentry;
    struct as_entry *aentry;
    int flags = amm_entry_flags(entry);

    nentry = as_entry_alloc(amm, addr, amm_entry_end(entry) - addr, flags);
    if (nentry == 0)
        return ENOMEM;
    *head = entry;
    *tail = nentry;
    if (flags == AMM_FREE)
        return 0;

    /* Modify existing Fluke mapping for first half of range */
    aentry = (struct as_entry *)entry;
    ...
    fluke_mapping_set_state(&aentry->mapping, ...);

    /* Setup new Fluke mapping for last half of range */
    aentry = (struct as_entry *)nentry;
    ...
    fluke_mapping_set_state(&aentry->mapping, ...);

    return 0;
}

int as_entry_join(amm_t *amm, amm_entry_t *head, amm_entry_t *tail, amm_entry_t **new)
{
    struct as_entry *aentryh, *aentryt;
    int flags = amm_entry_flags(head);

    *new = head;

    /* Basic entries are always joined */
    if (flags == AMM_FREE)
        return 0;

    /* Sources of mappings must be adjacent to join */
    aentryh = (struct as_entry *)head;
    aentryt = (struct as_entry *)tail;
    if (mapping_source(&aentryh->mapping) + amm_entry_size(entry) !=
        mapping_source(&aentryt->mapping))
        return 1;

    /* Collapse range of two Fluke mappings into head mapping */
    ...
    fluke_mapping_set_state(&aentryh->mapping, ...);

    /* Caller will deallocate tail mapping via as_entry_free */
    return 0;
}
```

Finally, the address space manager may want to perform some operation on only selected parts of the address space. For example, assume it wants to write-protect some arbitrary subset of the address space. Write protecting should only be done to the parts of the address space which are actually allocated (i.e., not free or reserved) and, for efficiency, only on those parts which currently allow write access (i.e., include

FLUKE_PROT_WRITE in their attributes). Here it could use `amm_iterate_gen` to process the map matching only those entries which are allocated and have write permission. `Amm_modify` can then be used on those entries to write-protect them as follows:

```
int as_write_protect(struct as *map, oskit_addr_t addr, oskit_size_t size)
{
    return amm_iterate_gen(&map->amm, as_wp_func, 0, addr, size,
                           AMM_ALLOCATED|FLUKE_PROT_WRITE,
                           AMM_ALLOCATED|FLUKE_PROT_WRITE);
}

int as_wp_func(amm_t *amm, amm_entry_t *entry, void *arg)
{
    struct as_entry *aentry = (struct as_entry *)entry;

    /* Tweak permission of Fluke mapping to remove write permission */
    ...
    fluke_mapping_set_state(&aentry->mapping, ...);

    /* Modify the existing AMM entry, removing write permission */
    rc = amm_modify(&map->amm, amm_entry_start(entry), amm_entry_size(entry),
                    amm_entry_flags(entry) & ~FLUKE_PROT_WRITE, entry);
    ...

    return 0;
}
```

17.7 External dependencies

The AMM library requires only four external routines. AMM uses `smalloc` and `sfree` (Section 9.5) to allocate and free entries for maps which don't specify an allocator (see `amm_init_gen`). The `amm_dump` routine uses `printf` (Section 9.6) to generate output. Various routines use `panic` (Section 9.8.3) when an internal consistency check fails.

17.8 API reference

The following sections describe the functions exported by the AMM in detail. All of these functions, as well as the types necessary to use them, are defined in the header file `<oskit/amm.h>`.

17.8.1 `amm_alloc_func`: Allocate an AMM map entry (user-provided callout)

Synopsis

`#include <oskit/amm.h>`

`amm_entry_t` ***amm_alloc_func***(`amm_t` **amm*, `oskit_addr_t` *addr*, `oskit_size_t` *size*, `int` *flags*);

Description

User-provided function called whenever an AMM entry needs to be allocated in the given map *amm*. The allocation function for an AMM is set at initialization time by passing a pointer to it as a parameter to `amm_init_gen`.

The parameters to `amm_alloc_func` provide information about the entry being created; i.e., the entry will cover the range [*addr* - *addr*+*size*-1] and have the attributes specified in *flags*.

If the map is using extended map entries, this routine should initialize the extended portion of the entry. No initialization of the AMM-private portion is necessary.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the range being created.

size: Size of the range being created.

flags: Attributes of the range being created.

Returns

Returns a pointer to the uninitialized, AMM-private part of the allocated entry, or zero if no memory can be allocated.

Related Information

`amm_init_gen`

17.8.2 `amm_allocate`: Allocate an address range in an AMM (simple interface)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_allocate**(`amm_t` **amm*, [in/out] `oskit_addr_t` **addrp*, `oskit_size_t` *size*, `int` *prot*);

Description

Looks for a range of the indicated size with flags `AMM_FREE` and modifies it to have the attributes `AMM_ALLOCATED`|*prot*.

On call, **addrp* specifies a hint address at which to start searching for a range of the desired size. The search will progress toward higher addresses from that point. If no range is found before the maximum possible address the search "wraps around," starting from the lowest address and searching forward until it reaches the original hint address. If no free range of sufficient size is found, `ENOMEM` is returned.

`Amm_allocate` is a simplified interface to `amm_modify` intended to be used with `amm_init`, `amm_deallocate`, `amm_protect`, and `amm_reserve`.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addrp: On call, a pointer to the address at which to start searching. On return, the address chosen.

size: Size of the desired range.

prot: Additional attribute flags to associate with the range.

Returns

Returns zero if successful, an error code otherwise.

Related Information

`amm_deallocate`, `amm_init`, `amm_modify`, `amm_protect`, `amm_reserve`

17.8.3 `amm_deallocate`: Deallocate an address range in an AMM (simple interface)

SYNOPSIS

`#include <oskit/amm.h>`

`int` **amm_deallocate**(`amm_t *`*amm*, `oskit_addr_t` *addr*, `oskit_size_t` *size*);

DESCRIPTION

Marks a range of address space as `AMM_FREE`. Only pieces of the range marked as `AMM_ALLOCATED` (e.g., allocated with `amm_allocate`) are "freed," all other regions are ignored.

`Amm_Deallocate` is a simplified interface to `amm_modify` intended to be used with `amm_init`, `amm_allocate`, `amm_protect`, and `amm_reserve`.

PARAMETERS

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the range.

size: Size of the range.

RETURNS

Returns zero if successful, an error code otherwise.

RELATED INFORMATION

`amm_allocate`, `amm_init`, `amm_modify`, `amm_protect`, `amm_reserve`

17.8.4 `amm_destroy`: Destroy an AMM

SYNOPSIS

`#include <oskit/amm.h>`

`void` **amm_destroy**(`amm_t *`*amm*);

DESCRIPTION

Free all the address map entries associated with the map *amm*. The user-provided free function is called for every entry in the map. If no free function is associated with the map, the standard libc `free` function is used.

PARAMETERS

amm: A pointer to the `amm_t` structure representing the address map.

RELATED INFORMATION

`amm_free_func`

17.8.5 `amm_dump`: display the AMM-private data for every entry in an AMM

SYNOPSIS

`#include <oskit/amm.h>`

`void` **amm_dump**(`amm_t *`*amm*);

Description

This routine is primarily used for debugging the AMM and the code that uses it. It scans through the AMM and calls `printf` to display the AMM-private data for each entry in it.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

17.8.6 `amm_entry_`*field*: Accessor macros for AMM-private data members

Synopsis

`#include <oskit/amm.h>`

`oskit_addr_t` **amm_entry_start**(`amm_entry_t` **entry*);

`oskit_addr_t` **amm_entry_end**(`amm_entry_t` **entry*);

`oskit_size_t` **amm_entry_size**(`amm_entry_t` **entry*);

`int` **amm_entry_flags**(`amm_entry_t` **entry*);

Description

Macros provided to access AMM-private data members. Currently defined are macros to return the starting and ending virtual addresses of the entry as well as the size and attributes of the range covered by the entry.

Parameters

entry: A pointer to a valid `amm_entry_t` structure.

Returns

Returns a data member of the appropriate type.

17.8.7 `amm_find_addr`: Locate the map entry containing a specific address

Synopsis

`#include <oskit/amm.h>`

`amm_entry_t` ***amm_find_addr**(`amm_t` **amm*, `oskit_addr_t` *addr*);

Description

Locates the map entry describing the given address in the map *amm* and returns a pointer to it. Since AMM maps contain every possible address in some entry, this routine will always succeed; i.e., it will always return a valid `amm_entry_t` pointer.

AMM-private fields of the returned entry can be queried with the amm_entry_*field* macros.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Address to locate.

RETURNS

Returns a pointer to the entry containing the desired address.

17.8.8 `amm_find_gen`: Locate a map entry matching specified criteria

SYNOPSIS

`#include <oskit/amm.h>`

`amm_entry_t` ***amm_find_gen**(`amm_t` **amm*, [in/out] `oskit_addr_t` **addrp*, `oskit_size_t` *size*, `int` *flags*, `int` *flagmask*, `int` *align_bits*, `oskit_addr_t` *align_off*, `int` *find_flags*);

DESCRIPTION

Returns a pointer to a map entry in the map *amm* with the indicated attributes which contains an address range of the given size and alignment. If no range can be found, `amm_find_gen` returns zero.

On call, **addrp* contains a "hint" at which to start searching for the range. On a successful return, **addrp* contains the address chosen; i.e., [**addrp* - **addrp*+*size*-1] is the desired range. This address may *not* be the same as the start address of the chosen map entry.

Flags and *flagmask* specify the attributes that the returned range must match. Only entries which satisfy ((entry->flags & *flagmask*) == *flags*) are considered when looking for a range.

Align_bits and *align_off* specify the alignment of the returned range. *Align_bits* specifies an alignment boundary as a power of two, and *align_ofs* specifies an offset from "natural" alignment; i.e. the lowest *align_bits* bits of the returned address must match the lowest *align_bits* of *align_ofs*. For example, align_bits == 12 and align_ofs == 8 would return a range starting 8 bytes past a 4096 byte boundary.

Find_flags can be used to modify the behavior of the lookup:

`AMM_EXACT_ADDR`. Range must start at the specified address. If that address is unsuitable, `amm_find_gen` returns zero.

`AMM_FORWARD`. Search forward from the hint address looking for a match. This is the default behavior.

`AMM_BACKWARD`. Search backward from the hint address looking for a match. **Not implemented.**

`AMM_FIRSTFIT`. Return the first entry found that contains a suitable range. This is the default behavior.

`AMM_BESTFIT`. Of all entries containing a suitable range, return the entry which is the closest fit. **Not implemented.**

PARAMETERS

amm: A pointer to the `amm_t` structure representing the address map.

addrp: On call, pointer to a search hint address. On return, the actual address found.

size: Size of the desired address range.

flags: Attribute flags that an entry must possess after masking with *flagmask*.

flagmask: Attribute mask to bitwise-AND with when matching an entry.

align_bits: The number of low bits of the returned address that must match the corresponding bits in *align_ofs*.

align_ofs: The required offset from natural power-of-two alignment. If *align_ofs* is zero, then the returned address will be naturally aligned on a 2^{align_bits} boundary.

find_flags: Flags modifying the behavior of the address space search.

Returns

Returns a pointer to the entry containing the desired range, or zero if no suitable range could be found.

17.8.9 `amm_free_func`: Free an AMM map entry (user-provided callout)

Synopsis

`#include <oskit/amm.h>`

`void` **amm_free_func**(`amm_t *`*amm*, `amm_entry_t *`*entry*);

Description

User-provided function called whenever an AMM entry needs to be deallocated from the given map *amm*. The free function for an AMM is set at initialization time by passing a pointer to it as a parameter to `amm_init_gen`.

The range and attributes of the entry can be obtained using the amm_entry_*field* macros.

If the map is using extended map entries, this routine should clean up any map-private data before deallocating the entry.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

entry: The entry to be destroyed.

Related Information

`amm_entry_`*field*, `amm_init_gen`

17.8.10 `amm_init`: initialize an address map (simple interface)

Synopsis

`#include <oskit/amm.h>`

`void` **amm_init**(`amm_t *`*amm*, `oskit_addr_t` *lo*, `oskit_addr_t` *hi*);

Description

This function initializes an address map as it would be used in most "simple" applications. The caller must provide a pointer to an `amm_t` structure; the AMM system uses this structure to keep track of the state of the address map. In subsequent AMM operations, the caller must pass a pointer to the same `amm_t` structure, which acts as a handle for the address map.

The address range [*lo* - *hi*-1] forms the valid area of the map. A single map entry is created for that range with attribute `AMM_FREE` so that all addresses within the range are eligible for allocation with `amm_allocate`. If necessary, entries are created for the ranges [`AMM_MINADDR` - *lo*-1] and [*hi* - `AMM_MAXADDR`] with attribute `AMM_RESERVED` so that addresses within those ranges are ignored by other simple interface routines.

`Amm_Init` is a simplified interface to `amm_init_gen` intended to be used with `amm_allocate`, `amm_deallocate`, `amm_protect` and `amm_reserve`.

Parameters

amm: A pointer to an uninitialized structure of type `amm_t` which is to be used to represent the address map.

lo: The first address to be marked `AMM_FREE`.

hi: The last address + 1 to be marked `AMM_FREE`.

Related Information

`amm_allocate`, `amm_deallocate`, `amm_modify`, `amm_protect`, `amm_reserve`

17.8.11 `amm_init_gen`: initialize an address map

Synopsis

```
#include <oskit/amm.h>
```

`void` **amm_init_gen**(`amm_t *`*amm*, `int` *flags*, `amm_entry_t *`*entry*, `amm_entry_t *`*(*amm_alloc_func)*(), void (*amm_free_func)(), int (*amm_split_func)(), int (*amm_join_func)());

Description

This function initializes an address map. The caller must provide a pointer to an `amm_t` structure; the AMM system uses this structure to keep track of the state of the address map. In subsequent AMM operations, the caller must pass a pointer to the same `amm_t` structure, which acts as a handle for the address map.

The map is initialized to contain a single entry describing the maximum possible address range [`AMM_MINADDR` - `AMM_MAXADDR`] and have the attributes specified in *flags*. If the *entry* parameter is non-zero, it is used as the initial entry. This allows the caller to allocate a structure larger than the basic `amm_entry_t` and store additional attribute data in the extended structure. If the caller supplies such an entry they must have initialized any caller-private data in that entry, but `amm_init_gen` will initialize the AMM-private part (the actual `amm_entry_t`). If *entry* is zero, a standard entry will be allocated using the default or caller-provided entry allocation routine (described below).

The four function pointer parameters permit the caller to customize AMM entry management on a per-AMM basis. If non-zero, *amm_alloc_func* and *amm_free_func* specify routines that the AMM library will callout to whenever an AMM entry is to be created or destroyed. If zero, `malloc` and `free` are used to manage basic `amm_entry_t` structures.

If non-zero, *amm_split_func* and *amm_join_func* specify routines that the AMM library will callout to whenever an AMM entry needs to be split or two entries need to be joined. If zero, default split and join calls are used. Split and join calls only occur as a side-effect of an `amm_modify` call.

When using extended AMM structures, the caller needs to provide free, split and join functions. The alloc function is not strictly necessary since the two AMM functions which create entries, `amm_init_gen` and `amm_modify`, have explicit entry parameters, and the third function which can create an entry, `amm_split_func`, will be caller-provided. The allocation hook is primarily provided to allow the caller control over the placement of AMM entry storage.

Parameters

amm: A pointer to an uninitialized structure of type `amm_t` which is to be used to represent the address map.

flags: Initial attribute flags to assign to the entry representing the entire address range.

entry: Initial entry to associate with the map.

amm_alloc_func: If non-zero, the function called whenever the AMM library needs to allocate a new map entry.

amm_free_func: If non-zero, the function called whenever the AMM library needs to destroy a map entry.

amm_split_func: If non-zero, the function called whenever the AMM library needs to split an existing map entry into two entries.

amm_join_func: If non-zero, the function called whenever the AMM library needs to join adjacent map entries.

Related Information

`amm_alloc_func`, `amm_free_func`, `amm_join_func`, `amm_split_func`

17.8.12 `amm_iterate`: Call a user-defined function for every entry in an AMM (simple interface)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_iterate**(`amm_t *`*amm*, `int` *(*amm_iterate_func)*(), void *arg);

Description

Calls a user-provided function *amm_iterate_func* for every entry of *amm*.

Arg is an opaque value which is passed to every instance of `amm_iterate_func` along with *amm* and the entry itself.

`amm_iterate` continues until the function has been called for all entries in the AMM or until one instance of the function returns non-zero. In the latter case, that non-zero value will be returned from `amm_iterate`.

Since the iteration function may modify or even destroy the entry passed in, `amm_iterate` uses the following technique for locating the "next" entry. At the beginning of each iteration, `amm_iterate` records the last address covered by the current entry. After the specified iteration function returns, `amm_find_addr` is called with this address to "relocate" the current entry. From this entry, `amm_iterate` derives the next entry.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

amm_iterate_func: Function to be called for every entry.

arg: Argument to be passed to every instance of the iteration function.

Returns

Returns zero if `amm_iterate_func` returned zero for all entries. Returns the first non-zero value returned from any `amm_iterate_func` call.

Related Information

`amm_iterate_func`

17.8.13 `amm_iterate_func`: Function to call with every AMM entry (user-provided callout)

SYNOPSIS

`#include <oskit/amm.h>`

`int` **amm_iterate_func**(`amm_t *`*amm*, `amm_entry_t *`*entry*, `void *`*arg*);

DESCRIPTION

Function called successively by `amm_iterate` and `amm_iterate_gen` with each selected entry from the map `amm`. The iteration function may modify or destroy the entry passed in.

If this function returns non-zero, the iterator will stop and `amm_iterate` or `amm_iterate_gen` will return that non-zero value. Returning zero will continue the iteration.

PARAMETERS

amm: A pointer to the `amm_t` structure representing the address map.

entry: The selected entry in the map.

arg: The opaque argument provided to `amm_iterate` and `amm_iterate_gen` and passed to each instance of `amm_iterate_func`.

RETURNS

Should return zero if the iteration is to continue, non-zero otherwise.

RELATED INFORMATION

`amm_iterate`, `amm_iterate_gen`

17.8.14 `amm_iterate_gen`: Call a user-defined function for select entries in an AMM

SYNOPSIS

`#include <oskit/amm.h>`

`int` **amm_iterate_gen**(`amm_t *`*amm*, `int` *(*amm_iterate_func)*(), void *arg, oskit_addr_t addr, oskit_size_t size, int flags, int flagmask);

DESCRIPTION

Calls a user-provided function *amm_iterate_func* for every entry of *amm* which falls partially or completely in the range [*addr* - *addr*+*size*-1] and matches the given attribute criteria.

Arg is an opaque value which is passed to every instance of `amm_iterate_func` (along with *amm* and the entry itself).

Flags and *flagmask* specify the attributes that an entry in the range must match for `amm_iterate_func` to be invoked. Only entries with ((entry->flags & *flagmask*) == *flags*) are considered.

`amm_iterate_gen` continues until the function has been called for all appropriate entries in the range or until one instance of the function returns non-zero. In the latter case, that non-zero value will be returned from `amm_iterate_gen`.

Since the iteration function may modify or even destroy the entry passed in, `amm_iterate_gen` uses the following technique for locating the "next" entry. At the beginning of each iteration, `amm_iterate_gen` records the last address covered by the current entry. After the specified iteration function returns, `amm_find_addr` is called with this address to "relocate" the current entry. From this entry, `amm_iterate_gen` derives the next entry.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

amm_iterate_func: Function to be called for every matching entry.

arg: Argument to be passed to every instance of the iteration function.

addr: Address at which to start iterating.

size: Size of the desired range over which to iterate.

flags: Attribute flags that an entry must possess after masking with *flagmask*.

flagmask: Attribute mask to bitwise-AND with when matching an entry.

Returns

Returns zero if `amm_iterate_func` returned zero for all entries. Returns the first non-zero value returned from any `amm_iterate_func` call.

Related Information

`amm_iterate_func`

17.8.15 `amm_join_func`: Join two adjacent map entries (user-provided callout)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_join_func**(`amm_t` **amm*, `amm_entry_t` **head*, `amm_entry_t` **tail*, [out] `amm_entry_t` ***new*);

Description

User-provided function called whenever the AMM thinks that two map entries for the map *amm* can be joined, based on comparison of the their flag words. The join function for an AMM is set at initialization time by passing a pointer to it as a parameter to `amm_init_gen`.

Head and *tail* are the two entries to join. If the join is successful, a pointer to the joined entry is returned in *new*. The returned entry may be one of the two entries passed in or it may be an entirely new entry. The AMM will call the entry free function for any "left-over" entries on return from a successful join call.

This routine is responsible for merging the map-private attributes of the two entries if they can be joined.

If the map-private attributes of the two entries are incompatible, the call should return non-zero to prevent the caller from reflecting the join in the map. Failure to join two entries is not an error, and the return code will not be propagated up through the call chain.

`amm_join_func` is only called by `amm_modify`.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

head,tail: The two entries to be joined.

new: A pointer to the new, joined entry.

Returns

Returns zero if the join was successful, non-zero if not.

Related Information

`amm_init_gen`, `amm_modify`

17.8.16 `amm_modify`: Modify the attributes of an address range

Synopsis

`#include <oskit/amm.h>`

`int` **amm_modify**(`amm_t` **amm*, `oskit_addr_t` *addr*, `oskit_size_t` *size*, `int` *flags*, `amm_entry_t` **entry*);

Description

Creates a new map entry in *amm* describing the range [*addr* - *addr*+*size*-1] with the attributes indicated in *flags*.

Any existing map entries wholly within the range are deleted, any that partly overlap the range are split as necessary. After adding the new entry, the AMM may attempt to join it with adjacent already-existing entries if the flag words are compatible.

If *entry* is zero, a standard `amm_entry_t` structure is allocated for the new range. If *entry* is non-zero, the caller must have already allocated and initialized any extra attribute data in the extended entry. In either case, the AMM will initialize the private part of the new entry, including setting its attribute flags to *flags*.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the range being modified.

size: Size of the range being modified.

flags: New attributes for the range being modified.

entry: If non-zero, the entry structure to use in the map.

Returns

Returns zero if the modification was successful, non-zero if an entry split failed.

Related Information

`amm_alloc_func`, `amm_free_func`, `amm_join_func`, `amm_split_func`

17.8.17 `amm_protect`: Modify the attribute flags of an address range in an AMM (simple interface)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_protect**(`amm_t` **amm*, `oskit_addr_t` *addr*, `oskit_size_t` *size*, `int` *prot*);

Description

Modifies the attribute flags associated with with all `AMM_ALLOCATED` entries within the specified address range. The resulting attributes are `AMM_ALLOCATED`|*prot*. `AMM_RESERVED` and `AMM_FREE` areas within the range are ignored.

`Amm_Protect` is a simplified interface to `amm_modify` intended to be used with `amm_init`, `amm_allocate`, `amm_deallocate`, and `amm_reserve`.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the desired range.

size: Size of the desired range.

prot: New attribute flags to associate with the range.

Returns

Returns zero if successful, an error code otherwise.

Related Information

`amm_allocate`, `amm_deallocate`, `amm_init`, `amm_modify`, `amm_reserve`

17.8.18 `amm_reserve`: Mark as unavailable an address range in an AMM (simple interface)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_reserve**(`amm_t *`*amm*, `oskit_addr_t *`*addr*, `oskit_size_t` *size*);

Description

Mark the specified address range as `AMM_RESERVED`. All entries within the range are effected regardless of existing attributes.

`Amm_Reserve` is a simplified interface to `amm_modify` intended to be used with `amm_init`, `amm_allocate`, `amm_deallocate`, and `amm_protect`.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the desired range.

size: Size of the desired range.

Returns

Returns zero if successful, an error code otherwise.

Related Information

`amm_allocate`, `amm_deallocate`, `amm_init`, `amm_modify`, `amm_protect`

17.8.19 `amm_select`: Returns an entry describing an address range exactly

Synopsis

`#include <oskit/amm.h>`

`amm_entry_t *`**amm_select**(`amm_t *`*amm*, `oskit_addr_t` *addr*, `oskit_size_t` *size*);

Description

Return a map entry in *amm* describing the range [*addr* - *addr*+*size*-1]. If either the start or end address is contained within an entry, the entry is split to create one starting or ending at the desired address.

Note that the desired range may still be described by multiple entries. `Amm_select` only guarantees that there is an entry starting at *addr* and an entry ending at *addr*+*size*-1.

This function returns a pointer to the selected entry. In the event that the desired range is described by multiple entries, `amm_select` returns the first entry. Successive entries may be obtained using `amm_find_addr` using the end address of the current entry (`amm_entry_end`).

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

addr: Start address of the desired range.

size: Size of the desired range.

Returns

Returns a pointer to the first entry describing the range.

Related Information

`amm_split_func`, `amm_find_addr`, `amm_entry_`*`field`*

17.8.20 `amm_split_func`: Split a map entry into two entries (user-provided callout)

Synopsis

`#include <oskit/amm.h>`

`int` **amm_split_func**(`amm_t *`*amm*, `amm_entry_t *`*entry*, `oskit_addr_t` *split_addr*, [out] `amm_entry_t **`*head*, [out] `amm_entry_t **`*tail*);

Description

User-provided function called whenever the AMM needs to split an entry in map *amm* due to conflicting flags. The split function for an AMM is set at initialization time by passing a pointer to it as a parameter to `amm_init_gen`.

Entry is the entry to be split and *split_addr* is the address at which to split it. If the split is successful, *head* and *tail* are pointers to the resulting entries describing the ranges [`amm_entry_start`(*entry*) - *split_addr*-1] and [*split_addr* - `amm_entry_end`(*entry*)-1] respectively. Both may be entirely new entries allocated in this routine, or one may point to the modified original entry. The AMM will call the entry free function for the original entry if it is not one of the returned values.

This routine is responsible for initializing the map-private attributes of the resulting new entries.

If the split cannot be done, e.g., due to lack of resources, a non-zero value indicating the error should be returned. A non-zero return value is propagated on to whoever performed the action which triggered the split request.

`amm_split_func` is only called by `amm_modify`.

Parameters

amm: A pointer to the `amm_t` structure representing the address map.

entry: The map entry to be split.

split_addr: The address at which to split the entry.

head,tail: Pointers to the the resulting new entries.

Returns

Returns zero if the split was successful, non-zero if not.

Related Information

`amm_init_gen`

Chapter 18

Simple Virtual Memory: liboskit_svm.a

18.1 Introduction

The Simple Virtual Memory (SVM) component provides very simple virtual memory management routines that are somewhat more application friendly than the extremely basic support provided by the kernel library (see Section 10.9.11). Applications are able to allocate large contiguous blocks of virtual memory that are backed by physical memory. Applications may also control the page level protection of memory. In addition, there is optional pageout support that allows the application to allocate more virtual memory than physical memory on machines where a disk swap partition is available. The SVM component is thread safe, although only a single virtual memory context is provided; all threads share the same set of page tables.

The SVM manager makes use of the list-based memory manager (LMM) (see Section 16), the address map manager (AMM) (see Section 17), and the page directory support in the kernel library. The LMM is used to to control physical memory, while the AMM is used to control the virtual memory mappings. The kernel paging support handles the details of manipulating the low level page tables. As a result, the SVM manager is very simple in its construction.

Although on the surface it might appear that the SVM provides generalized VM support, nothing could be further from the truth. What is provided is a means to allocate memory in the range above existing physical memory, and map those ranges to physical pages. With paging enabled, the application is able to use more virtual memory than physical memory. It should be noted that the kernel remains where it was initially loaded, and that unused pages of physical memory are left accessible by the application.

18.2 API reference

18.2.1 `svm_init`: initialize SVM system

Synopsis

`#include <oskit/svm/svm.h>`

`void` **svm_init**(`oskit_absio_t` **pager_absio*);

Description

This function initializes the SVM system, turning on base paging support (see Section 10.9.1), and optionally configuring pageout support. The `pager_absio` argument is optional, and if specified should be a device suitable for use as the swap area for the pager. `pager_absio` may also be a `oskit_blkio_t`; the pager will query the object to determine which type it is. Only a single paging area is supported. The initialization code will create an initial set of page tables that maps all of physical memory as readable and writable, except for kernel text which is mapped read-only. A stack redzone is also created, although stack overflows are fatal since there is not enough support to allow recovery. The address range above the end of physical memory is mapped as invalid so that accesses result in a page fault trap (instead of silently returning bogus data).

In the case of a multi-threaded kernel, `pager_absio` must be a properly wrapped object (see Section 19.4). The current multi-threaded locking strategy is extremely simple; a single lock protects the entire SVM module.

PARAMETERS

pager_absio: An oskit_absio_t * or oskit_blkio_t * that is suitable for use as the swap area.

18.2.2 `svm_alloc`: allocate a region of virtual memory

SYNOPSIS

`#include <oskit/svm/svm.h>`

`int` **svm_alloc**(`oskit_addr_t *`*addr*, `oskit_size_t` *length*, `int` *prot*, `int` *flags*);

DESCRIPTION

Allocate a region of virtual memory, returning the base address of the new region in `addr`. The region is `length` bytes in size, and is initialized to the page level protection specified by `prot`. The size of the allocation must be an integral number of pages. The caller can optionally specify the (page aligned) base address at which to place the region by providing a non-zero value in `addr`. The actual base address might differ if the system cannot place the region at that address. Alternatively, if the `flags` value contains `SVM_ALLOC_FIXED`, and the region cannot be placed at the requested address, the allocation will fail and return an error code.

PARAMETERS

addr: The location in which to store the base address of the new region. Also used to provide an optional address.

length: The size of the new region in bytes. Must be an integral number of pages.

prot: Page level protection of the new region, composed of `SVM_PROT_READ` and `SVM_PROT_WRITE`.

flags: Optional flags.

RETURNS

Returns zero on success. Returns `OSKIT_E_INVALIDARG` if either the base address or the size of the allocation is not page aligned. Returns `OSKIT_E_OUTOFMEMORY` if the region cannpt be assigned to the fixed location requested by the caller.

RELATED INFORMATION

`svm_dealloc`, `svm_protect`

18.2.3 `svm_dealloc`: deallocate a region of virtual memory

SYNOPSIS

`#include <oskit/svm/svm.h>`

`int` **svm_dealloc**(`oskit_addr_t` *addr*, `oskit_size_t` *length*);

Description

Deallocate a range of memory that was previously allocated with `svm_alloc`. The range starts at `addr`, and is `length` bytes in size. The base address must be page aligned, and the length must be an integral number of pages. The range may be a subset of a previously allocated range; only that subset is deallocated.

Parameters

addr: The address of the region to deallocate.

length: The size in bytes of the region to deallocate.

Returns

Returns zero on success. Returns `OSKIT_E_INVALIDARG` if either the base address or the size of the allocation is not page aligned, or if the range is not within an existing allocation.

Related Information

`svm_alloc`, `svm_protect`

18.2.4 `svm_protect`: control the protection of a region of virtual memory

Synopsis

`#include <oskit/svm/svm.h>`

`int` **svm_protect**(`oskit_addr_t` *addr*, `oskit_size_t` *length*, `int` *prot*);

Description

Change the page level protection on a region of memory. The region begins at `addr` and extends for `length` bytes. The base address must be page aligned, and the length must be an integral number of pages. The page level protection of each page in the region is set to `prot`. Unlike `svm_dealloc`, this routine may called on any region of memory, not just regions that were allocated with `svm_alloc`.

Parameters

addr: The address of the region.

length: The size in bytes of the region.

prot: Page level protection of the new region, composed of `SVM_PROT_READ` and `SVM_PROT_WRITE`.

Returns

Returns zero on success. Returns `OSKIT_E_INVALIDARG` if either the base address or the size of the allocation is not page aligned.

Related Information

`svm_alloc`, `svm_dealloc`

Chapter 19

POSIX Threads: `liboskit_threads.a`

19.1 Introduction

This chapter describes the POSIX threads module and associated support for writing multithreaded kernels. At present, threads support is very new and not every combination of components is known to work; see Section 19.2 for a more detailed description of what has been tested. Section 19.3 describes the application program interface for the core POSIX threads module, while Section 19.4 contains a discussion of how the threads system interacts with the device driver framework.

19.2 Examples and Caveats

The sample kernels in the `examples/threads` directory (see Section 1.6.1), contain several sample kernels demonstrating the use of the POSIX threads module.

- `dphils`: A computational example that tests basic POSIX threads operations such as thread creation, mutexes, and conditions. Solves the classic Dining Philosophers problem.
- `disktest`: A contrived disk thrashing program that tests the interaction between POSIX threads and the NetBSD filesystem (see section 26). A number of threads are created, where each one creates and copies files in varying block sizes.
- `disknet`: Another contrived program that builds on the disk thrashing program above. Also tested is the interaction bewteen POSIX threads and the BSD network interface. Half of the threads created thrash the disk and the other half connect to a server process and send and receive data blocks. This program achieves reasonable interleaving of work.
- `http_proxy`: A simplified HTTP proxy daemon that tests the interaction between POSIX threads and the BSD network interface. For each new connection request, three threads are created to manage that connection and forward data between the client and the server.

This small set of test programs clearly does not test every possible combination of components. A larger set of test program is in the works. In addition, not all of the thread-safe adaptors are implemented, so some components cannot be used in a multithreaded environment. *For now, the* POSIX *threads module should be used with caution.* Note that these examples are compiled and linked against the multithreaded version of the FreeBSD C library (see Section 14), rather than the minimal C library (Section 9).

19.3 POSIX Threads Reference

As with most POSIX threads implementations, this one is slightly different than others. This section briefly covers the specific interfaces, but does not describe the semantics of each interface function in great detail.

The reader is advised to consult the POSIX documentation for a more complete description. All of these functions, as well as the types necessary to use them, are defined in the header file `<oskit/threads/pthread.h>`.

19.3.1 `pthread.h`: Thread constants and data structures

DESCRIPTION

This header file defines the following standard symbols.

`PRIORITY_MIN`: Lowest possible thread scheduling priority.

`PRIORITY_NORMAL`: Default thread scheduling priority.

`PRIORITY_MAX`: Highest possible thread scheduling priority.

`SCHED_FIFO`: The "first in first out" thread scheduling policy.

`SCHED_RR`: The "round robin" thread scheduling policy.

`PTHREAD_STACK_MIN`: The minumum allowed stack size.

`PTHREAD_CREATE_JOINABLE`: Thread attribute; thread is created joinable.

`PTHREAD_CREATE_DETACHED`: Thread attribute; thread is created detached.

`PTHREAD_PRIO_NONE`: Mutex attribute; mutex does not do priority inheritance.

`PTHREAD_PRIO_INHERIT`: Mutex attribute; mutex does priority inheritance.

`PTHREAD_MUTEX_NORMAL`: Mutex attribute; normal error checking, no recursion.

`PTHREAD_MUTEX_ERRORCHECK`: Mutex attribute; extra error checking, no recursion.

`PTHREAD_MUTEX_RECURSIVE`: Mutex attribute; normal error checking, recursion allowed.

`PTHREAD_MUTEX_DEFAULT`: Mutex attribute; normal error checking, no recursion.

`PTHREAD_CANCEL_ENABLE`: Cancelation state; Cancelation state is enabled.

`PTHREAD_CANCEL_DISABLE`: Cancelation state; Cancelation state is disabled.

`PTHREAD_CANCEL_DEFERRED`: Cancelation type; Cancelation type deferred,

`PTHREAD_CANCEL_ASYNCHRONOUS`: Cancelation type; Cancelation type is asynchronous.

`PTHREAD_CANCELED`: The exit status returned by pthread_join for a canceled thread.

`pthread_t`: Thread identifier type definition.

`pthread_mutex_t`: Mutex type definition.

`pthread_cond_t`: Condition variable type definition.

`pthread_attr_t`: Thread attributes type definition.

`pthread_attr_default`: Default thread attributes object.

`pthread_mutexattr_t`: Mutex attributes type definition.

`pthread_mutexattr_default`: Default mutex attributes object.

`pthread_condattr_t`: Condition variable attributes type definition.

`pthread_condattr_default`: Default condition variable attributes object.

`sched_param_t`: Type definition for the `pthread_setschedparam` interface function.

19.3.2 `pthread_init`: Initialize the threads system

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`void` **pthread_init**(`int` *preemptible*);

Description

This function initializes the threads system. It should be called as the first function in the application's main program function.

When `pthread_init` returns, the caller is now running within the main thread, although on the same stack as when called. One or more idle threads have also been created, and are running at low priority. At this point, the application is free to use any of the pthread interface functions described in this section.

Parameters

preemptible: A boolean value specifying whether the threads system should use preemption based scheduling. When preemption based scheduling is not used, it is up to the application to yield the processor using `sched_yield` as necessary.

19.3.3 `pthread_attr_init`: Initialize a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_init**(`pthread_attr_t` `*`*attr*);

Description

Initialize a thread attributes object for use with `pthread_create`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

Returns

Returns zero on success.

Related Information

`pthread_create`, `pthread_attr_setprio`, `pthread_attr_setstacksize`

19.3.4 `pthread_attr_setdetachstate`: Set the detach state in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setdetachstate**(`pthread_attr_t` `*`*attr*, `int` *detachstate*);

Description

Set the thread detach state in a previously initialized threads attribute object, for use with `pthread_create`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

detachstate: Either PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.

Returns

Returns zero on success. Returns EINVAL if `detachstate` is invalid.

Related Information

`pthread_create`, `pthread_attr_init`

19.3.5 `pthread_attr_setprio`: Set the priority in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setprio**(`pthread_attr_t` **attr*, `int` *pri*);

Description

Set the priority value in a previously initialized threads attribute object, for use with `pthread_create`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

pri: A value between PRIORITY_MIN and PRIORITY_MAX.

Returns

Returns zero on success. Returns EINVAL if `priority` is outside the range of PRIORITY_MIN to PRIORITY_MAX.

Related Information

`pthread_create`, `pthread_attr_init`, `pthread_attr_setstacksize`

19.3.6 `pthread_attr_setstackaddr`: Set the stack address in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setstackaddr**(`pthread_attr_t` **attr*, `oskit_u32_t` *stackaddr*);

Description

Set the stack address in a previously initialized threads attribute object, for use with `pthread_create`. The new thread will be created using the provided stack. It is necessary to call `pthread_attr_setstacksize()` if the size is not `PTHREAD_STACK_MIN`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

stackaddr: The address of the stack.

Returns

Returns zero on success.

Related Information

`pthread_create`, `pthread_attr_init`, `pthread_attr_setstacksize`

19.3.7 `pthread_attr_setguardsize`: Set the stack guard size in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setguardsize**(`pthread_attr_t *`*attr*, `oskit_size_t` *guardsize*);

Description

Set the stack guard size in a previously initialized threads attribute object, for use with `pthread_create`. This much extra space will be allocated at the end of the stack and set as a redzone to catch stack overflow. The guard size is rounded up to a multiple of the native page size. Stack guards are not created for stacks provided with `pthread_attr_setstackaddr`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

guardsize: A reasonable stack guard size.

Returns

Returns zero on success.

Related Information

`pthread_create`, `pthread_attr_init`, `pthread_attr_setstackaddr`

19.3.8 `pthread_attr_setstacksize`: Set the stack size in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setstacksize**(`pthread_attr_t *`*attr*, `oskit_size_t` *stacksize*);

Description

Set the stack size in a previously initialized threads attribute object, for use with `pthread_create`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

stacksize: A reasonable stack size.

Returns

Returns zero on success. Returns EINVAL if `stacksize` is less than PTHREAD_STACK_MIN.

Related Information

`pthread_create`, `pthread_attr_init`, `pthread_attr_setprio`

19.3.9 `pthread_attr_setschedpolicy`: Set the scheduling policy in a thread attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_attr_setschedpolicy**(`pthread_attr_t` **attr*, `int` *policy*);

Description

Set the scheduling policy in a previously initialized threads attribute object, for use with `pthread_create`.

Parameters

attr: A pointer to the `pthread_attr_t` object representing the attributes for a thread creation.

policy: Either SCHED_FIFO or SCHED_RR.

Returns

Returns zero on success. Returns EINVAL if `policy` is invalid.

Related Information

`pthread_create`, `pthread_attr_init`

19.3.10 `pthread_mutexattr_init`: Initialize a mutex attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutexattr_init**(`pthread_mutexattr_t` **attr*);

Description

Initialize an mutex attributes object for use with `pthread_mutex_init`.

Parameters

attr: A pointer to the `pthread_mutexattr_t` object representing the attributes for a mutex initialization.

Returns

Returns zero on success.

Related Information

`pthread_mutex_init`, `pthread_mutex_setprotocol`

19.3.11 `pthread_mutexattr_setprotocol`: Set the protocol attribute of a mutex attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutexattr_setprotocol**(`pthread_mutexattr_t *`*attr*, `int` *protocol*);

Description

Set the protocol in a previously initialized mutex attribute object. When a mutex is created with the protocol PTHREAD_PRIO_INHERIT, threads that blocked on the mutex will result in a transfer of priority from higher to lower priority threads.

Parameters

attr: A pointer to the `pthread_mutexattr_t` object representing the attributes for a mutex initialization.

protocol: Either PTHREAD_PRIO_NONE or PTHREAD_PRIO_INHERIT.

Returns

Returns zero on success.

Related Information

`pthread_mutex_init`

19.3.12 `pthread_mutexattr_settype`: Set the type attribute of a mutex attributes object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutexattr_settype**(`pthread_mutexattr_t *`*attr*, `int` *type*);

Description

Set the type in a previously initialized mutex attribute object. PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, and PTHREAD_MUTEX_DEFAULT are equivalent. PTHREAD_MUTEX_REC
allows a mutex to be recursively locked.

Parameters

attr: A pointer to the `pthread_mutexattr_t` object representing the attributes for a mutex initialization.

type: One of PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_DEFAUL
or PTHREAD_MUTEX_RECURSIVE.

Returns

Returns zero on success.

Related Information

`pthread_mutex_init`

19.3.13 `pthread_condattr_init`: Initialize a condition attributes object

Synopsis

```
#include <oskit/threads/pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
```

Description

Initialize an condition variable attributes object for use with `pthread_cond_init`.

Parameters

attr: A pointer to the `pthread_condattr_t` object representing the attributes for a condition variable initialization.

Returns

Returns zero on success.

Related Information

`pthread_cond_init`

19.3.14 `pthread_cancel`: Cancel a running thread

Synopsis

```
#include <oskit/threads/pthread.h>
int pthread_cancel(pthread_t tid);
```

Description

Cancel the thread specified by `tid`. The thread is marked for cancellation, but because of scheduling and device delays, might not be acted upon until some future time.

Parameters

tid: The thread identifier of the thread to be canceled.

Returns

Returns zero on success. EINVAL if `tid` specifies an invalid thread.

Related Information

`pthread_create`, `pthread_sleep`

19.3.15 `pthread_cleanup_push`: Push a cancellation cleanup handler routine onto the calling thread's cancellation cleanup stack

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`void` **pthread_cleanup_push**(`void` *(*routine)*(void *), void *arg);

DESCRIPTION

Push a cancellation cleanup handler routine onto the calling thread's cancellation cleanup stack. When requested, the cleanup `routine` will be popped from the cancellation stack, and invoked with the argument `arg`.

PARAMETERS

routine: The cleanup handler routine.

arg: The argument to pass to the cleanup handler routine.

RELATED INFORMATION

`pthread_cancel`, `pthread_cleaup_pop`

19.3.16 `pthread_setcancelstate`: Set the cancelation state

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`void` **pthread_setcancelstate**(`int` *state*, `int` **oldstate*);

DESCRIPTION

Set the cancel `state` for the current thread, returning the old state in `oldstate`. Valid states are either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`. This routine is async-cancel safe.

PARAMETERS

state: New cancel state.

oldstate: Location in which to place the original cancel state.

RELATED INFORMATION

`pthread_cancel`, `pthread_setcanceltype`

19.3.17 `pthread_setcanceltype`: Set the cancelation type

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`void` **pthread_setcanceltype**(`int` *type*, `int` **oldtype*);

DESCRIPTION

Set the cancel `type` for the current thread, returning the old type in `oldtype`. Valid types are either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. This routine is async-cancel safe.

PARAMETERS

type: New cancel type.

oldtype: Location in which to place the original cancel type.

RELATED INFORMATION

`pthread_cancel`, `pthread_setcancelstate`

19.3.18 `pthread_testcancel`: Check for a cancelation point

SYNOPSIS

```
#include <oskit/threads/pthread.h>
void pthread_testcancel(void);
```

DESCRIPTION

Test whether a cancelation is pending, and deliver the cancelation if the cancel state is `PTHREAD_CANCEL_ENABLED`.

RELATED INFORMATION

`pthread_cancel`, `pthread_setcancelstate`

19.3.19 `pthread_cond_broadcast`: Wakeup all threads waiting on a condition variable

SYNOPSIS

```
#include <oskit/threads/pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

DESCRIPTION

Wakeup all threads waiting on a condition variable.

PARAMETERS

cond: A pointer to the condition variable object.

RETURNS

Returns zero on success.

RELATED INFORMATION

`pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`

19.3.20 `pthread_cond_destroy`: Destroy a condition variable

SYNOPSIS

```
#include <oskit/threads/pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

Description

Destroy a condition variable object. The condition variable should be unused, with no threads waiting for it. The memory for the object is left intact; it is up to the caller to deallocate it.

Parameters

cond: A pointer to the condition variable object.

Returns

Returns zero on success. EINVAL if there are threads still waiting.

Related Information

`pthread_cond_init`

19.3.21 `pthread_cond_init`: Initialize a condition variable

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_cond_init**(`pthread_cond_t` **cond*, `pthread_condattr_t` **attr*);

Description

Initialize a condition variable object, using the provided condition attributes object. The attributes object may be a NULL pointer, in which case `pthread_condattr_default` is used.

Parameters

cond: A pointer to the condition variable object.

attr: A pointer to the condition variable attributes object.

Returns

Returns zero on success.

Related Information

`pthread_cond_destroy`

19.3.22 `pthread_cond_signal`: Wakeup one thread waiting on a condition variable

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_cond_signal**(`pthread_cond_t` **cond*);

Description

Wakeup the highest priority thread waiting on a condition variable.

Parameters

cond: A pointer to the condition variable object.

Returns

Returns zero on success.

Related Information

`pthread_cond_wait`, `pthread_cond_broadcast`

19.3.23 `pthread_cond_wait`: Wait on a condition variable

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_cond_wait**(`pthread_cond_t` **cond*, `pthread_mutex_t` **mutex*);

Description

The current thread is made to wait until the condition variable is signaled or broadcast. The mutex is released prior to waiting, and reacquired before returning.

Parameters

cond: A pointer to the condition variable object.

mutex: A pointer to the mutex object.

Returns

Returns zero on success.

Related Information

`pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_timedwait`

19.3.24 `pthread_cond_timedwait`: Wait on a condition variable with timeout

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_cond_timedwait**(`pthread_cond_t` **cond*, `pthread_mutex_t` **mutex*, `oskit_timespec_t` **abstime*);

Description

The current thread is made to wait until the condition variable is signaled or broadcast, or until the timeout expires. The mutex is released prior to waiting, and reacquired before returning. The timeout is given as an absolute time in the future that bounds the wait.

Parameters

cond: A pointer to the condition variable object.

mutex: A pointer to the mutex object.

abstime: A pointer to an oskit_timespec structure.

RETURNS

Returns zero on success. Returns ETIMEDOUT if the timeout expires.

RELATED INFORMATION

`pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_wait`

19.3.25 `pthread_create`: Create a new thread and start it running

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`int` **pthread_create**(`pthread_t` **tid*, `const` *pthread_attr_t *attr*, `void` *(*function)*(void *), void *arg);

DESCRIPTION

Create a new thread and schedule it to run. The thread is created using the attributes object `attr`, which specifies the initial priority, stack size, and detach state. If a NULL attributes object is provided, a system default attributes object is used instead, specifying that the thread is detachable, has priority PRIORITY_NORMAL, and with a reasonable stack size.

This call returns immediately, with the thread id stored in the location given by `tid`. This thread id should be saved if the application wishes to manipulate the thread's state at some future time.

The new thread is scheduled to run. When the thread starts up, it will call `void (*function)(void *arg)`.

PARAMETERS

tid: A pointer to the location where the thread id should be stored.

attr: A pointer to the thread creation attributes object.

function: The initial function to call when the thread first starts.

arg: The argument to the initial function.

RETURNS

Returns zero on success, storing the tid of the new thread into `*tid`.

RELATED INFORMATION

`pthread_join`, `pthread_detach`, `pthread_exit`

19.3.26 `pthread_detach`: Detach a thread from its parent

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`int` **pthread_detach**(`pthread_t` *tid*);

DESCRIPTION

The thread specified by `tid` is detached from its parent. If the thread has already exited, its resources are released.

Parameters

tid: The thread id of the thread being detached.

Returns

Returns zero on success. EINVAL if `tid` refers to a non-existent thread.

Related Information

`pthread_join`, `pthread_create`, `pthread_exit`

19.3.27 `pthread_exit`: Terminate a thread with status

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_exit**(`void *`*status*);

Description

The current thread is terminated, with its status value made available to the parent using `pthread_join`.

Parameters

status: The exit status.

Returns

This function does not return.

Related Information

`pthread_join`, `pthread_create`, `pthread_detach`

19.3.28 `pthread_join`: Join with a target thread

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_join**(`pthread_t` *tid*, `void **`*status*);

Description

The current thread indicates that it would like to join with the target thread specified by `tid`. If the target thread has already terminated, its exit status is provided immediately to the caller. If the target thread has not yet exited, the caller is made to wait. Once the target has exited, all of the threads waiting to join with it are woken up, and the target's exit status provided to each.

Parameters

tid: The thread id of the thread being joined with.

status: A pointer to a location where the target's exit status is placed.

RETURNS

Returns zero on success, storing the target's exit status in `*status`. EINVAL if `tid` refers to a non-existent thread.

RELATED INFORMATION

`pthread_join`, `pthread_create`, `pthread_detach`

19.3.29 `pthread_key_create`: Create a thread-specific data key

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`int` **pthread_key_create**(`pthread_ket_t` **key*, `void` *(*destructor)*(void *));

DESCRIPTION

Create a thread-specific key for use with `pthread_setspecific`. If specified, the destructor is called on any non-NULL key/value pair when a thread exits.

PARAMETERS

key: Address where the new key value should be stored.

destructor: Pointer to the destructor function, which may be NULL.

RETURNS

Returns zero on success, and stores the new key value at `*key`. Returns EAGAIN if the are no more keys available.

RELATED INFORMATION

`pthread_key_delete`, `pthread_setspecific`, `pthread_getspecific`

19.3.30 `pthread_key_delete`: Delete a thread-specific data key

SYNOPSIS

`#include <oskit/threads/pthread.h>`

`int` **pthread_key_delete**(`pthread_ket_t` **key*);

DESCRIPTION

Delete the thread-specific key. Attempts to use a key via `pthread_setspecific` or `pthread_getspecific` after it has been deleted is undefined.

PARAMETERS

key: The key that should be deleted.

RETURNS

Returns zero on success. Returns EINVAL if `key` refers to an invalid key.

Related Information

`pthread_key_create`, `pthread_setspecific`, `pthread_getspecific`

19.3.31 `pthread_setspecific`: Set a thread-specific data value

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_setspecific**(`pthread_ket_t` *key*, `const` *void *value*);

Description

Associate a new thread-specific value with the specified key.

Parameters

key: The key that should be set.

value: The new value to associate with the key.

Returns

Returns zero on success. Returns EINVAL if `key` refers to an invalid key.

Related Information

`pthread_key_create`, `pthread_key_delete`, `pthread_getspecific`

19.3.32 `pthread_getspecific`: Set a thread-specific data value

Synopsis

`#include <oskit/threads/pthread.h>`

`void *`**pthread_setspecific**(`pthread_ket_t` *key*);

Description

Get the thread-specific value associated the specified key.

Parameters

key: The key for the value that should be retrieved.

Returns

Returns the value of the key. Errors always return zero.

Related Information

`pthread_key_create`, `pthread_key_delete`, `pthread_setspecific`

19.3.33 `pthread_mutex_init`: Initialize a mutex object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutex_init**(`pthread_mutex_t *`*m*, `pthread_mutexattr_t *`*attr*);

Description

Initialize a mutex object, using the provided mutex attributes object. The attributes object may be a NULL pointer, in which case `pthread_mutexattr_default` is used.

Parameters

mutex: A pointer to the mutex object.

attr: A pointer to the mutex attributes object.

Returns

Returns zero on success.

Related Information

`pthread_mutex_destroy`, `pthread_mutex_lock`, `pthread_mutex_unlock`

19.3.34 `pthread_mutex_destroy`: Destroy a mutex object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutex_destroy**(`pthread_mutex_t *`*m*);

Description

The mutex object is destroyed, although the memory for the object is not deallocated. The mutex must not be held.

Parameters

mutex: A pointer to the mutex object.

Returns

Returns zero on success. Returns EBUSY if the mutex is still held.

Related Information

`pthread_mutex_init`

19.3.35 `pthread_mutex_lock`: Lock a unlocked mutex object

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_mutex_lock**(`pthread_mutex_t *`*m*);

Description

Lock a mutex object. If the mutex is currently locked, the thread waits (is suspended) for the mutex to become available.

Parameters

mutex: A pointer to the mutex object.

Returns

Returns zero on success.

Related Information

`pthread_mutex_init`, `pthread_mutex_unlock`, `pthread_mutex_trylock`

19.3.36 `pthread_mutex_trylock`: Attempt to lock a unlocked mutex object

Synopsis

```
#include <oskit/threads/pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *m);
```

Description

Attempt to lock a mutex object. This function always returns immediately.

Parameters

mutex: A pointer to the mutex object.

Returns

Returns zero on success. Returns EBUSY if the mutex object is locked.

Related Information

`pthread_mutex_init`, `pthread_mutex_unlock`, `pthread_mutex_lock`

19.3.37 `pthread_mutex_unlock`: Unlock a mutex object

Synopsis

```
#include <oskit/threads/pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *m);
```

Description

Unlock a mutex object. If there other threads waiting to acquire the mutex, the highest priority thread is woken up and granted the mutex.

Parameters

mutex: A pointer to the mutex object.

Returns

Returns zero on success.

Related Information

`pthread_mutex_init`, `pthread_mutex_trylock`, `pthread_mutex_lock`

19.3.38 `pthread_resume`: Resume a suspended thread

Synopsis

`#include <oskit/threads/pthread.h>`
`int` **pthread_resume**(`pthread_t` *tid*);

Description

Resume a thread that has been suspended with `pthread_suspend`.

Parameters

tid: The thread identifier of the thread to be resumed.

Returns

Returns zero on success. EINVAL if `tid` specifies an invalid thread.

Related Information

`pthread_create`, `pthread_suspend`

19.3.39 `pthread_self`: Return the thread identifier of the current thread

Synopsis

`#include <oskit/threads/pthread.h>`
`pthread_t` **pthread_self**(`void`);

Description

Return the thread identifier of the current thread.

Returns

Returns the thread identifier.

Related Information

`pthread_create`

19.3.40 `pthread_setprio`: Change the priority of a thread

Synopsis

`#include <oskit/threads/pthread.h>`
`int` **pthread_setprio**(`pthread_t` *tid*, `int` *newpri*);

Description

Change the priority of a thread. If the change causes a thread to have a higher priority than the currently running thread, a reschedule operation is performed.

Parameters

tid: The thread identifier of the thread whose priority should be changed.

newpri: The new priority, which must be from PRIORITY_MIN to PRIORITY_MAX.

Returns

Returns zero on success. EINVAL if `tid` specifies an invalid thread or `newpri` specifies an invalid priority.

Related Information

`pthread_create`, `sched_yield`, `pthread_setschedparam`

19.3.41 `pthread_setschedparam`: Set the scheduling parameters for a thread

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_setschedparam**(`pthread_t` *tid*, `int` *policy*, `const` *struct sched_param *param*);

Description

Change the scheduling parameters for a thread. The thread's scheduling policy and priority are changed. If the change causes a thread to have a higher priority than the currently running thread, a reschedule operation is performed.

Parameters

tid: The thread identifier of the thread whose scheduling parameters should be changed.

policy: The new scheduling policy, as defined in `pthread.h`

param: A pointer to the `sched_param_t` object representing the new scheduling parameters.

Returns

Returns zero on success. EINVAL if `tid` specifies an invalid thread or `policy` specifies an invalid policy.

Related Information

`pthread_create`, `sched_yield`, `pthread_setprio`

19.3.42 `pthread_sleep`: Sleep for an interval of time

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_sleep**(`oskit_s64_t` *milliseconds*);

Description

The calling thread is put to sleep for the number of milliseconds specified. The thread cannot be woken up until the given amount of time has passed, although the thread may be canceled with `pthread_cancel`

Parameters

milliseconds: The number of milliseconds the thread should sleep for.

Returns

Returns zero on success.

Related Information

`pthread_cancel`

19.3.43 `pthread_suspend`: Suspend a thread

Synopsis

`#include <oskit/threads/pthread.h>`

`int` **pthread_suspend**(`pthread_t` *tid*);

Description

Suspend the specified thread indefinitely. The thread may be resumed with `pthread_resume` or it may be canceled with `pthread_cancel`.

Parameters

tid: The thread identifier of the thread to be suspended

Returns

Returns zero on success. EINVAL if `tid` specifies an invalid thread.

Related Information

`pthread_resume`

19.3.44 `sched_yield`: Yield the processor

Synopsis

`#include <oskit/threads/pthread.h>`

`void` **sched_yield**(`void`);

Description

The calling thread voluntarily yields the processor. The highest priority thread is chosen for execution.

Related Information

`pthread_setprio`, `pthread_setschedparam`

19.3.45 `osenv_process_lock`: Lock the process lock

Synopsis

```
#include <oskit/threads/pthread.h>
void osenv_process_lock(void);
```

Description

Attempt to lock the process lock. If the lock cannot be immediately granted, the thread is put to sleep until it can be. The process lock is provided so that the client operating system can protect the device driver framework from concurrent execution. It is expected than any entry into the device framework will first take the process lock. If the thread executing inside the device driver framework blocks by calling `osenv_sleep`, the process lock will be released so that another thread may enter it safely. When the thread is woken up later, it will take the process lock again before returning from the sleep.

Attempts to recursively lock the process lock will result in a panic. This is intended as a debugging measure to prevent indiscriminate nesting of components that try to take the lock.

Related Information

`osenv_process_unlock`, `osenv_sleep`, `osenv_wakeup`

19.3.46 `osenv_process_unlock`: Unlock the process lock

Synopsis

```
#include <oskit/threads/pthread.h>
void osenv_process_unlock(void);
```

Description

Release the process lock. If another thread is waiting to lock the process lock, it will be woken up. The process lock is provided so that the client operating system can protect the device driver framework from concurrent execution.

Related Information

`osenv_process_lock`, `osenv_sleep`, `osenv_wakeup`

19.4 Thread-safe Adaptors

To facilitate the use of the device driver framework within a multithreaded client operating system, a number of *adaptors* are provided. An adaptor acts as COM interface wrapper on another COM interface. Adaptors are intended to provide thread-safety with respect to the device driver framework. The thread system is expected to provide an implementation of a *process lock* that is used to prevent concurrent execution inside the device driver framework. An adaptor method simply takes the process lock, calls the appropriate method in the underlying COM interface, and then releases the process lock when the method returns. If a thread blocks inside a device driver (`osenv_sleep`), the process lock is released at that time, allowing another thread to enter the driver set. When the original thread is woken up, it will reacquire the process lock before being allowed to return from the sleep. Thus, only one thread is allowed to operate inside the driver set at a time.

Implementationally, an adaptor is a COM interface that maintains a reference to the original, non thread-safe COM interface. Operations using the adaptor behave just like the original, invoking the corresponding method in the original. It should be noted that the query, addref, and release methods all operate on the adaptor itself. When the last reference to an adaptor is released, the reference to the underlying COM interface is released. As an example, consider the `oskit_dir_t` adaptor as it is used when mounting the root filesystem in a multithreaded client operating system. In order to provide a thread-safe implementation to the C library, the root directory that is passed to `fs_init` is first wrapped up in a thread-safe adaptor. All subsequent references to the corresponding filesystem go through the adaptor, and are thus thread-safe. A sample code fragment follows:

```
#include <oskit/c/fs.h>
#include <oskit/com/wrapper.h>
#include <oskit/threads/pthread.h>

oskit_error_t
mountroot(oskit_dir_t *fsroot)
{
    oskit_dir_t    *wrappedroot;
    oskit_error_t  err;

    rc = oskit_wrap_dir(fsroot,
                (void (*)(void *))osenv_process_lock,
                (void (*)(void *))osenv_process_unlock,
                0, &wrappedroot);
    if (rc)
        return rc;

    /* Don't need the root anymore, the wrapper has a ref. */
    oskit_dir_release(fsroot);

    return fs_init(wrappedroot);
}
```

The adaptor prototypes are found in `<oskit/com/wrapper.h>`, and have a common format. Each one takes the COM interface to be wrapped up, and returns the adaptor. Additional arguments are the process lock and unlock routines, as well as an optional cookie to be passed to the lock and unlock routines. It should be noted that the process lock is specific to the thread implementation, and thus the adaptor interface is intended to be as generic as possible. For the `pthread` interface, the process lock does not need a cookie value.

19.4.1 `oskit_wrap_socket`: Wrap an `oskit_socket` in a thread-safe adaptor

SYNOPSIS

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_socket**(`struct` *oskit_socket *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_socket **out);

DESCRIPTION

Create and return an `oskit_socket` thread-safe adaptor.

PARAMETERS

in: The `oskit_socket` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_socket` adaptor COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.2 `oskit_wrap_stream`: Wrap an `oskit_stream` in a thread-safe adaptor

SYNOPSIS

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_stream**(`struct` *oskit_stream *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_stream **out);

DESCRIPTION

Create and return an `oskit_dir` thread-safe adaptor.

PARAMETERS

in: The `oskit_stream` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_stream` adaptor COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.3 `oskit_wrap_asyncio`: Wrap an `oskit_asyncio` in a thread-safe adaptor

SYNOPSIS

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_asyncio**(`struct` *oskit_asyncio *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_asyncio **out);

DESCRIPTION

Create and return an `oskit_dir` thread-safe adaptor.

PARAMETERS

in: The `oskit_asyncio` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_asyncio` adaptor COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.4 `oskit_wrap_sockio`: Wrap an `oskit_sockio` in a thread-safe adaptor

SYNOPSIS

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_sockio**(`struct` *oskit_sockio *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_sockio **out);

DESCRIPTION

Create and return an `oskit_dir` thread-safe adaptor.

PARAMETERS

in: The `oskit_sockio` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_sockio` adaptor COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.5 `oskit_wrap_posixio`: Wrap an `oskit_posixio` in a thread-safe adaptor

SYNOPSIS

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_posixio**(`struct` *oskit_posixio *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_posixio **out);

DESCRIPTION

Create and return an `oskit_dir` thread-safe adaptor.

PARAMETERS

in: The `oskit_posixio` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_posixio` adaptor COM interface.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.6 `oskit_wrap_file`: Wrap an `oskit_file` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_file**(`struct` *oskit_file *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_file **out);

Description

Create and return an `oskit_dir` thread-safe adaptor.

Parameters

in: The `oskit_file` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_file` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.7 `oskit_wrap_dir`: Wrap an `oskit_dir` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_dir**(`struct` *oskit_dir *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_dir **out);

Description

Create and return an `oskit_dir` thread-safe adaptor.

Parameters

in: The `oskit_dir` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_dir` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.8 `oskit_wrap_filesystem`: Wrap an `oskit_filesystem` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_filesystem**(`struct` *oskit_filesystem *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_filesystem **out);

Description

Create and return an `oskit_dir` thread-safe adaptor.

Parameters

in: The `oskit_filesystem` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_filesystem` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.9 `oskit_wrap_openfile`: Wrap an `oskit_openfile` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_openfile**(`struct` *oskit_openfile *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_openfile **out);

Description

Create and return an `oskit_dir` thread-safe adaptor.

Parameters

in: The `oskit_openfile` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_openfile` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.10 `oskit_wrap_blkio`: Wrap an `oskit_blkio` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_blkio**(`struct` *oskit_blkio *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_blkio **out);

Description

Create and return an `oskit_blkio` thread-safe adaptor.

Parameters

in: The `oskit_blkio` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_blkio` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.4.11 `oskit_wrap_absio`: Wrap an `oskit_absio` in a thread-safe adaptor

Synopsis

`#include <oskit/com/wrapper.h>`

`oskit_error_t` **oskit_wrap_absio**(`struct` *oskit_absio *in*, `void` *(*lock)*(void *), void (*unlock)(void *), void *cookie, struct oskit_absio **out);

Description

Create and return an `oskit_absio` thread-safe adaptor.

Parameters

in: The `oskit_absio` COM interface to be wrapped.

lock: The process lock routine.

unlock: The process unlock routine.

cookie: A cookie to be passed to the lock and unlock routines.

out: The `oskit_absio` adaptor COM interface.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.5 InterThread Communication

This section describes the “interthread” communication primitives provided by the pthread library.

19.5.1 `oskit_ipc_send`: Send a message to another thread

SYNOPSIS

```
#include <oskit/threads/pthread.h>
#include <oskit/threads/ipc.h>
```

`oskit_error_t` **oskit_ipc_send**(`pthread_t` *dst*, `void` **msg*, `oskit_size_t` *msg_size*, `oskit_s32_t` *timeout*);

DESCRIPTION

Send a message to another thread. The destination thread is specified by its `pthread_t`. The sending thread blocks until the receiving thread notices the message and actually initiates a receive operation for it. Control returns to the caller only when the receiver has initiated the receive.

The timeout value is currently ignored.

PARAMETERS

dst: The `pthread_t` of the destination thread.

msg: The message buffer.

msg_size: The size of the message, in bytes.

timeout: A timeout value. Currently ignored.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.5.2 `oskit_ipc_recv`: Receive a message from a specific thread

SYNOPSIS

```
#include <oskit/threads/pthread.h>
#include <oskit/threads/ipc.h>
```

`oskit_error_t` **oskit_ipc_recv**(`pthread_t` *src*, `void` **msg*, `oskit_size_t` *msg_size*, `oskit_size_t` **actual*, `oskit_s32_t` *timeout*);

DESCRIPTION

Receive a message from another thread. The sending thread is specified by its `pthread_t`. If the specified sending thread has not attempted to send a message to current thread, the thread is blocked until such time as the sender initiates a send operation to the current thread. However, if the sender is blocked trying to send a message to the current thread, the message is immediately received and the sender is woken up.

The timeout value is either zero or non-zero. A zero value means do not wait, but simply check to see if a message from the sender is pending. A non-zero value means wait forever.

PARAMETERS

src: The `pthread_t` of the sending thread.

msg: The message buffer.

msg_size: The size of the message buffer, in bytes.

actual: The location in which to place the number of bytes received.

timeout: A timeout value. Currently only zero and non-zero values are legal. zero means no wait, non-zero means wait forever.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.5.3 `oskit_ipc_wait`: Receive a message from any thread

Synopsis

`#include <oskit/threads/pthread.h>`
`#include <oskit/threads/ipc.h>`

`oskit_error_t` **oskit_ipc_wait**(`pthread_t` **src*, `void` **msg*, `oskit_size_t` *msg_size*, `oskit_size_t` **actual*, `oskit_s32_t` *timeout*);

Description

This function operates identically to `oskit_ipc_recv`, except that the sending thread does not need to be a specific thread. The first thread that attempts to send to the current thread will succeed. The `pthread_t` of that thread is returned to the caller in `src`.

Parameters

src: The location in which to place the `pthread_t` of the sending thread.

msg: The message buffer.

msg_size: The size of the message buffer, in bytes.

actual: The location in which to place the number of bytes received.

timeout: A timeout value. Currently only zero and non-zero values are legal. zero means no wait, non-zero means wait forever.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.5.4 `oskit_ipc_call`: make a synchronous IPC call to another thread

Synopsis

`#include <oskit/threads/pthread.h>`
`#include <oskit/threads/ipc.h>`

`oskit_error_t` **oskit_ipc_call**(`pthread_t` *dst*, `void` **sendmsg*, `oskit_size_t` *sendmsg_size*, `void` **recvmsg*, `oskit_size_t` *recvmsg_size*, `oskit_size_t` **actual*, `oskit_s32_t` *timeout*);

Description

Make a synchronous IPC call to another thread, and wait for a reply. The destination thread is specified by its `pthread_t`. The sending thread is blocked until the receiving thread replies to the IPC using `oskit_ipc_reply`. The send buffer and the reply buffer are specified separately, with the actual number bytes contained in the reply returned in the location pointed to by `actual`.

PARAMETERS

dst: The `pthread_t` of the destination thread.

sendmsg: The message buffer to send.

sendmsg_size: The size of the send message buffer, in bytes.

recvmsg: The message receive buffer.

recvmsg_size: The size of the receive message buffer, in bytes.

actual: The location in which to place the number of bytes contained in the reply message.

timeout: A timeout value. Currently ignored.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

19.5.5 `oskit_ipc_reply`: reply to a synchronous IPC invocation

SYNOPSIS

```
#include <oskit/threads/pthread.h>
#include <oskit/threads/ipc.h>
```

`oskit_error_t` **oskit_ipc_reply**(`pthread_t` *src*, `void` **msg*, `oskit_size_t` *msg_size*);

DESCRIPTION

Reply to a synchronous IPC invocation made with `oskit_ipc_call`. The destination thread is specified by its `pthread_t`, and it must be blocked in a call operation, waiting for the reply message. If the destination thread is canceled before the reply is made, this call with return OSKIT_ECANCELED.

PARAMETERS

dst: The `pthread_t` of the destination thread.

msg: The message buffer.

msg_size: The size of the message, in bytes.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 20

Memory Debugging Utilities: `liboskit_memdebug.a`

20.1 Introduction

The Memory Debug Utilities Library is a set of functions which replace the standard OSKit memory allocation functions, see Section 9.5, of the minimal C library. The replacement routines detect problems with memory allocation, and can print out file and line information, along with a back-trace to the offending allocation.

All of the standard functions are covered: `malloc`, `memalign`, `calloc`, `realloc`, `free`, and `smalloc`, `smemalign`, and `sfree`.

To use the library, just include `-lmemdebug` on the linker command line *before* the standard C library (or wherever it is the standard allocation routines are coming from).

The memdebug library implements a fence-post style `malloc` debug library. It detects the following problems:

- **Over-runs and under-runs.** Over-runs and under-runs of allocated memory blocks are detected by "fence-posts" at each end of every allocated block of memory.
- **Allocation/release style mismatches.** Mismatches between `malloc` style and `smalloc` style allocations and the respective `free` function are detected. This type of error is corrected by the library and only a warning is printed.
- **Memory use after it is `free`'d.** Memory is wiped to a recognizable (nonzero) bit pattern on allocation and when it is freed, to force bugs to show up when memory is used after it is freed. (See below for which values are used where.)
- **Incorrect size passed to `sfree`.** The `sfree` size is checked against that used when the block is created.
- **`free` called on bad blocks.** Freeing of blocks that were never allocated or were already released is detected.

Whenever a problem is encountered a back-trace (in the form of program counter values) is dumped (back-tracing from the *allocation* of the memory). File and line number information from where the allocation call was made are also printed (if available). If the failure was detected in a call to `free`, the file and line of that call are printed. This is called a "bogosity dump."

When correctable errors are detected (e.g., `sfree`'ing a `malloc`'d block, or `sfree`'ing with the wrong size block). the correct thing will be done, and the program will continue as normal (except for the bogosity dump).

Note that file and line number information is only available if you're using the macro wrappers for the allocators defined in `malloc_debug.h`. The call stack trace is always available.

One of the shortcomings of the library is that errors are only detected during explicit calls into the library, and not at the time that they happen. The `memdebug_sweep` function will check the validity of all allocated blocks, and by judiciously sprinkling calls throughout your code you can narrow down memory trashing problems. Similarly, the `memdebug_ptrchk` function will run a sanity check on a single pointer. Both functions, when printing "bogosity dumps" will also print the file and line at which they were called.

To help detect leaks of unfreed memory, use `memdebug_mark` and `memdebug_check`. `memdebug_mark` tags all allocated blocks, and then `memdebug_check` will check for untagged blocks. In this way, you can "mark" all blocks as okay and at a later point when all memory allocated after the mark should have been released, insert a "check". The library will print a bogosity dump for any allocation that is untagged.

To help detect accesses after memory is released, or accesses to uninitialized memory, the library sets all bytes of an allocation to:

- `0xaa` after an `smalloc`-style allocation.
- `0xbb` after a `malloc`-style allocation.
- `0xdd` after `free`.
- `0xee` after `sfree`.

20.1.1 Memdebug Library Configuration

There are several configuration options in the library-private `memdebug.h` header file. The `NO_MEM_FATAL` `#define` controls whether errors in an allocation are fatal (via `panic`) or if they return 0. The `#define` `ALLOW_MORALLY_QUESTIONABLE_PRACTICE` controls the library's handling of `malloc(0)` and `free(NULL)`. While both of these constructs are technically legal, they usually signal errors in the caller; the option merely controls whether a message is printed or not. The `MALLOC_0_RET_NULL` option controls the behavior of `malloc(0)`, either returning `NULL` or returning a valid, unique (per-allocation) pointer.

20.1.2 Memdebug Library Internals

The default `malloc` and the provided `memdebug_alloc` *can have different policies.* `memdebug_alloc` uses the LMM library directly to provide memory, in the same way as the default `libc malloc`. But `malloc` can be replaced, and `memdebug_alloc` will not change.

When allocating memory on small alignment boundaries, those boundaries will actually be bumped up to the alignment necessary for the leading fence-post of the allocation. Thus, when running under memdebug data may be aligned at a larger granularity than when running without memdebug.

All of the routines use `memdebug_printf` to print all output. This function should always be defined such that it guarantees that it will never cause any memory to be allocated. You should override this if you cannot guarantee that `vfprintf` calls will not allocate memory.

The allocation management routines all call `mem_lock` and `mem_unlock` to protect access to the global `malloc_lmm`. See the minimal C library's section on Memory Allocation (Section 9.5) for more information on these functions.

20.1.3 External Dependencies

The memdebug library uses several functions, and one global variable that it does not define. Outside of memory allocation primitives, it uses `panic` for flagging internal consistency failures, and `memset` for wiping swaths of memory. The default implementation of `memdebug_printf` requires `vprintf`.

For memory allocation primitives, the memdebug library depends on `lmm_alloc_aligned` and `lmm_free` from the LMM library (see Section 16). The same LMM pool, `malloc_lmm`, used by the default implementations of `malloc`, et al is used by the memdebug library. Additionally, calls to `mem_lock` and `mem_unlock` are made around accesses to `malloc_lmm`. A call to `morecore` is made when any allocation function returns 0. (`morecore`, `mem_lock`, `mem_unlock`, and the `malloc_lmm` are described in more detail in the Memory Allocation section of the Minimal C Library chapter, Section 9.5.)

20.2 Debugging versions of standard routines

The functions listed below are defined as macros in the header file `oskit/memdebug.h`, they are also defined as simple wrappers in the library. The macro versions provide the library with file and line number information.

They are drop-in replacements for the allocation functions described in Section 9.5.

malloc: `void *`**malloc**(`size_t` *size*);

realloc: `void *`**realloc**(`void *`*buf*, `size_t` *new_size*);

calloc: `void *`**calloc**(`size_t` *nelt*, `size_t` *elt_size*);

memalign: `void *`**memalign**(`size_t` *alignment*, `size_t` *size*);

free: `void` **free**(`void *`*buf*);

smalloc: `void *`**smalloc**(`size_t` *size*);

smemalign: `void *`**smemalign**(`size_t` *alignment*, `size_t` *size*);

sfree: `void` **sfree**(`void *`*buf*, `size_t` *size*);

20.3 Additional Debugging Utilities

These routines provide additional features useful for tracking down memory leaks and dynamic memory corruption.

memdebug_mark: Mark all currently allocated blocks

memdebug_check: Look for blocks allocated since mark that haven't been freed

memdebug_ptrchk: Check validity of a pointer's fence-posts

memdebug_sweep: Check validity of all allocated block's fence-posts

These routines are internal to the memdebug library, but may be worth overriding in your system.

memdebug_printf: A standard printf-style routine that can be guaranteed to not allocate any memory.

memdebug_bogosity: Dumps information about an allocation block when an error in the block is detected.

memdebug_store_backtrace: Stores a back-trace (the call-stack) in a provided buffer.

20.3.1 `memdebug_mark`: Mark all currently allocated blocks.

SYNOPSIS

```
#include <oskit/memdebug.h>
void memdebug_mark(void);
```

DESCRIPTION

This function walks the list of all allocated objects and "marks" them. This is useful so that you can determine what was allocated before a certain point in your program.

Objects only have one bit to keep track of marks, so calling `memdebug_mark` more than once may not have the effect you would like.

RELATED INFORMATION

`memdebug_sweep`

20.3.2 `memdebug_check`: Look for blocks allocated since mark that haven't been freed.

SYNOPSIS

```
#include <oskit/memdebug.h>
void memdebug_check(void);
```

DESCRIPTION

This functions walks the list of all allocated blocks and for each block that is *not* marked (by `memdebug_mark`, it prints a bogosity dump.

For example, at the beginning of a server loop call `memdebug_mark`, then when the server loop is about to iterate, call `memdebug_check` to make sure that the loop didn't leave any allocated objects lying about.

Related Information

`memdebug_bogosity`, `memdebug_mark`

20.3.3 `memdebug_ptrchk`: Check validity of a pointer's fence-posts

Synopsis

```
#include <oskit/memdebug.h>
int memdebug_ptrcheck(void* ptr);
```

Description

This function runs a host of sanity checks on a given pointer. Of course, these only work if the pointer, `ptr` is one returned by a memdebug-wrapped allocator. For any errors a bogosity dump is printed.

Parameters

ptr: A pointer to a memory block allocated by some memdebug wrapped allocator.

Returns

Returns -1 if the fence posts are trashed so badly that the information in them cannot be trusted. Returns 1 if there was a problem detected but it is not "fatal". Returns 0 if everything is A-okay.

Related Information

`memdebug_bogosity`

20.3.4 `memdebug_sweep`: Check validity of all allocated block's fence-posts

Synopsis

```
#include <oskit/memdebug.h>
void memdebug_sweep(void);
```

Description

This function walks the list of all allocated blocks and calls `memdebug_ptrchk` on each entry.

Related Information

`memdebug_ptrchk`

20.3.5 `memdebug_printf`: A printf-style routine guaranteed not to allocate memory

Synopsis

```
#include <oskit/memdebug.h>
int memdebug_printf(const char *fmt, ...);
```

Description

Works just like standard libc `printf`, but this function must be guaranteed to not allocate any memory while it runs.

Parameters

fmt: The standard `printf` format string.

...: The standard `printf` arguments for the specific format string.

Returns

Returns the standard `printf` return value.

20.3.6 `memdebug_bogosity`: Prints a memdebug bogosity message

Synopsis

`#include <oskit/memdebug.h>`

`void` **memdebug_bogosity**(`memdebug_mhead` **head*);

Description

Prints a bogosity dump given the first fence-post of an allocation. Uses `memdebug_printf` for all output.

This routine is called by all others in the library to dump information about an allocation.

Parameters

head: The head fence-post for the given allocation. Contains the back-trace, file and line number information, and allocation-style information.

Related Information

`memdebug_printf`

20.3.7 `memdebug_store_backtrace`: Stores call-stack trace in provided buffer

Synopsis

`#include <oskit/memdebug.h>`

`void` **memdebug_store_backtrace**(`unsigned` **backtrace*, `int` *max_len*);

Description

Stores a machine-specific back-trace in the provided buffer. In conjunction with the object code and the `nm` utility, the back-trace can provide a function call stack.

Parameters

backtrace: A buffer of at least `max_len unsigned ints`.

max_len: Size of back-trace buffer.

Chapter 21

Profiling Support: `liboskit_gprof.a`

21.1 Introduction

The gprof program and associated libc and kernel routines provide a mechanism to produce an execution profile of an oskit-based kernel. The gprof program is linked right into the kernel, performing its data reduction and analysis just before the kernel exits and producing output to the console device. See gprof(1) for more information.

21.2 Caveats

The application to be profiled must be called "`a.out`".

It is expected that the current behavior of generating ASCII output to the console device will be changed in the future. For example, the interface might be modified to allow specification of an `oskit_stream` object (Section 4) to which the binary "gmon.out" data would be written. Typically, this object would refer to a persistent file or a network connection with another machine.

See Section 21.4 for line-by-line instructions on using gprof in the OSKit, including some non-obvious linking magic.

21.3 API reference

21.3.1 `profil`: Enable, disable, or change statistical sampling

Synopsis

`#include <oskit/c/sys/gmon.h>`

`#include <oskit/c/sys/profile.h>`

`int` **profil**(`char *`*samples*, `int` *size*, `int` *offset*, `int` *scale*);

Description

This function enables or disables the statistical sampling of the program counter for the kernel. If profiling is enabled, at RTC clock tick (see below), the program counter is recorded in the `samples` buffer. This function is most frequently called by `moncontrol()`.

Parameters

samples: A buffer containing `size` bytes which is divided into a number of bins. A bin represents a range of addresses in which the PC was found when the profiling sample was taken.

size: The size in bytes of the samples array.

offset: The lowest address at which PC samples should be taken. In the oskit, this defaults to the location of the `_start` symbol.

scale: The scale determines the granularity of the bins. A scale of 65536 means each bin gets 2 bytes of address range. A scale of 32768 gives 4 byte, etc. A scale value of 0 disables profiling.

Returns

Returns 0 if all is OK, or -1 on error. Sets `errno` to the reason for failure.

21.3.2 `moncontrol`: enable or disable profiling

Synopsis

`#include <oskit/c/sys/gmon.h>`

`#include <oskit/c/sys/profile.h>`

`void` **moncontrol**(`int` *mode*);

Description

If `mode` is non-zero (true), enables profiling. If `mode` is zero, disables profiling.

Parameters

mode: Determines if profiling should be enabled or disabled

21.3.3 `monstartup`: Start profiling for the first time

Synopsis

`#include <oskit/c/sys/gmon.h>`

`#include <oskit/c/sys/profile.h>`

`void` **monstartup**(`unsigned` *long *lowpc*, `unsigned` *long *highpc*);

Description

`monstartup` initiates profiling of the kernel; it should only be called once. Note that by default, `monstartup` is called by `base_multiboot_main` when profiling is enabled with `configure`. If you wish to delay profiling until a later time, disable the monstartup call in `base_multiboot_main`, and place your own call to `monstartup` later in your code.

Parameters

lowpc: The lowest address for which statistics should be collected. Usually the location of the `_start` symbol.

highpc: The highest address for which statistics should be collected. Usually the location of the `etext` (end of text segment) symbol.

21.4 Using gprof

1. Configure your sources with –enable-profiling
2. When you link your program, link against:
   (a) the `_p` versions of all libraries you would normally use
   (b) the `.po` versions of all .o files you would use *except* `crtn.o` and `multiboot.o` (if you use them)
   (c) Insert "`-loskit_gprof -loskit_kern_p -loskit_c_p`" immediately after the existing "`-loskit_kern_p -loskit_c_p`." That's right, *another* instance of the kern and C libs. If you use the FreeBSD C library, do the analogous thing. If the above doesn't work, try including the libs more times (yes, this is bogus).
   (d) Be sure to include the following libraries:
      - `oskit_dev_p`
      - `oskit_lmm_p`
3. Run `mkbsdimage` *multiboot_kernel multiboot_kernel*:`a.out`
   or
   `mkmbimage` *multiboot_kernel multiboot_kernel*:`a.out`

   This step is necessary so gprof can access the kernel's symbol table via the bmodfs.
4. Run the kernel (if created as above, it would be named `Image`).
5. Profiling output will be spit out at its exit.

21.5 Files

- gprof/* The gprof directory contains the files necessary for the gprof program itself.
- libc/gmon/gmon.c contains moncontrol() and monstartup().
- libc/gmon/mcount.c contains the C-language _mcount routine.
- libc/x86/mcount_md.S contains the asm routines which are linked into functions compiled with -pg (mcount for C-language functions). The __mcount routine must be called manually by your assembly functions, though this call is handled automatically if you use the ENTRY() macro.
- oskit/c/sys/gmon.h
- oskit/c/sysprofile.h The profiling headers.
- kern/x86/pc/profil.c The profil() system call (architecture-dependent).

21.6 Changing parameters and other FAQs

21.6.1 The sampling rate

The default sampling rate is 8192 Hz, using the RTC as the source of the sampling interrupts. You can adjust this by modifying one #define in gmon.h:
    Redefine #define PROFHZ xxxx to the sampling rate.
    The rate you select must be a power of 2 between 128 and 8192.

21.6.2 How can I temporarily disable gprof's output while still linking it in?

in base_console.c, change int enable_gprof = 1; to = 0.

21.6.3 Why isn't there a command line option for it?

The FreeBSD boot manager won't pass in the -p flag.

21.6.4 Why don't my assembly routines register properly with mcount?

You need to hand-code stubs for them which call __mcount. Sorry. The compiler only autogenerates the _mcount stubs for C routines. The call to __mcount *is* performed for you if you use the oskit ENTRY() macro.

21.6.5 Why is the call graph wrong when a routine was called from an assembly function?

If you don't use one of the oskit ENTRY macros, then your function's symbol may not be declared properly. If you want to do it by hand, then declare the symbol:

```
.globl symbol_name
.type  symbol_name,@function
```

Note that this is taken care of for you by the macros in asm.h if you simply declare a function with ENTRY(x) or NON_GPROF_ENTRY(x).

21.6.6 What will gprof break?

Gprof takes over the RTC (irq 8). If you have code which uses the oskit interrupt request mechanism to grab irq 8, it won't work. If your code just steals irq 8 by replacing the interrupt handler for it, you'll break gprof.

Gprof installs some atexit handlers for the kernel 'main'. These are installed in `base_multiboot_main.c`.

Chapter 22

Disk Partition Interpreter: liboskit_diskpart.a

22.1 Introduction

The OSKit includes code that understands the various partitioning schemes used to divide disk drives into smaller pieces for use by filesystems. This code enables the use of various (possibly nested) partitioning schemes in an easy manner without requiring knowledge of which partitioning scheme was used, or how these partitioning schemes work. E.g., you don't need to understand or know the format of a VTOC to use the partitioning, as the library does all of it for you.

22.2 Supported Partitioning Schemes

Supported partitioning schemes are:

- BSD Disklabels
- IBM-PC BIOS/DOS partitions (including logical)
- VTOC labels (Mach).
- OMRON and DEC label support based on old Mach code is provided, although it is completely untested.

22.3 Example Use

22.3.1 Reading the partition table

This shows how the partitioning information can be extracted in user-mode (running under Unix). In the kernel, it would likely be necessary to pass a `driver_info` structure to a device-specific read function. In this case, driver_info is simply a filename string.

```
/* This is the testing program for the partitioning code. */
#include <oskit/diskpart/diskpart.h>
#include <stdio.h>
#include <fcntl.h>

#define FILENAME "/dev/sd0c"

/* We pass in a fixed-size table; this defines how big we want it. */
```

```
#define MAX_PARTS 30
diskpart_t part_array[MAX_PARTS];

/*
 * In this case, we are defining the disk size to be 10000 sectors.
 * Normally, this would be the number of physical sectors on the
 * disk.  If the ‘disk’ is a ‘file’, it would be better to get the
 * equivalent number of sectors from the file size.
 * This is only used to fill in the whole-drive partition entry.
 */
#define DISK_SIZE 10000

/*
 * This is the function pointer I pass to the partition code
 * to read sectors on the drive.
 */
int my_read_fun(void *driver_info, int sector, char *buf);


int
main(int argc, char *argv[])
{
        int numparts;
        char *filename;

        if (argc == 2)
                filename = argv[1];
        else
                filename = FILENAME;

        /* call the partition code */
        numparts = diskpart_get_partition(filename, my_read_fun, part_array,
                MAX_PARTS, DISK_SIZE);

        printf("%d partitions found\n",numparts);
        /* diskpart_dump(part_array, 0); */
}

static int
my_read_fun(void *driver_info, int sector, char *buf)
{
        char *filename = driver_info;

        int fd = open(filename, O_RDONLY, 0775);

        lseek(fd, SECTOR_SIZE * sector, SEEK_SET);
        read(fd, buf, SECTOR_SIZE);
        close(fd);

        /* Should bzero the result if read error occurs */
        return(0);
}
```

22.3.2 Using Partition Information

The routine `diskpart_lookup_bsd_compat` is an example of how the old partition naming can be used even with the new nested structure. This takes two integers representing the slice and partition. The behavior is intended to be similar to `diskpart_lookup_bsd_string` (below), using integers as parameters.

While this 'hack' allows two levels of nesting (slice and partition), it is not general enough to support arbitrary nesting. Arbitrary nesting support is most easily achieved by passing string names to a lookup function which can follow the structure down the partition specifications. For example, 'sd0eab' would be used to specify the second partition in the first partition inside the fifth top-level partition on the first SCSI disk. Since the lookup routine doesn't need to know about the disk, 'eab' would be the partition name passed to the lookup routine. This naming scheme would work well as long as there are not more than 26 partitions at any nesting layer.

`diskpart_lookup_bsd_string` does a string lookup using the FreeBSD style slice names. FreeBSD considers the DOS partitioning to be slices. A slice can contain a BSD disklabel, and if it does, then partitions can be inside the slice. If the third DOS partition contains a disklabel, then 's3a' would be partition 'a' inside the disklabel. The slice name without a partition would mean the entire slice. Note also that 'a' would alias to partition 'a' in the first BSD slice. If there is no BSD slice, then 'a' would be aliased to 's1' instead. However, to avoid confusion, if slice-naming is used, aliases should only be used to point inside a BSD slice.

22.4 Restrictions

This is a list of known restrictions/limitations of the partitioning library.

22.4.1 Endian

The partitioning code only recognizes labels created with the same endian-ness as the machine it is running on. While it is quite possible to detect an endian conflict and interpret the information in the label, the information stored in the partitions will probably not be very useful, as most filesystems expect the numeric representations to remain constant.

22.4.2 Nesting

Strict nesting, in which a child is not allowed to extend outside the parent, is not enforced, or even checked by the library. This allows greater flexibility in the use of nested partitions, while also placing greater responsibility on the user's shoulders to ensure that the partition information on the disk is correct. Enforcement of strict nesting, should it be desired, is left to the user.

Due to previous constraints, the search routine does not yet do a recursive search for all possible nestings, although all 'sensible' ones are searched manually. This is a change that will be incorporated as soon as nesting of this type exists and it can be utilized by something.

22.4.3 Lookup

A general lookup routine is not yet part of the library. The `diskpart_lookup` routine is only able to do one layer of nesting. More general support may be added in the future, or it may be left to the user to determine a naming scheme to access the subpartitions.

Also, the lookup routines currently assume a sector size of 512 bytes.

22.5 API reference

22.5.1 `diskpart_get_partition`: initialize an array of partition entries

Synopsis

```
#include <oskit/diskpart/diskpart.h>
```

`int` **diskpart_get_partition**(`void` **driver_info*, `int` *(*diskpart_read_func)*(), struct diskpart *array, int array_size, int disk_size);

Description

This function initializes an array of `struct diskpart` entries. The caller must provide a pointer to a `struct diskpart` array, and a function to read the disk.

Parameters

driver_info: A pointer to an initialized structure of user-defined type which is passed unmodified to `diskpart_read_func`.

diskpart_read_func: A function pointer provided by the user which can read a sector given `driver_info`.

array: Array of `struct diskpart`.

array_size: integer containing the number of allocated entries in the array.

disk_size: Size of the whole disk, in sectors.

Returns

Returns an integer count of the number of partition entries that were filled by the library. If there were more partitions found than space available, this will be `array_size`. Empty partitions (unused entries in a BSD disklabel, for example) occupy an entry the same as 'used' entries.

For example, a PC-DOS partition with a single filled entry would still report 4 partitions, as that is the size of the DOS partition table.

Related Information

`diskpart_read_func`

22.5.2 `diskpart_read_func`: read a disk sector (user-provided callout)

Synopsis

`#include <oskit/diskpart/diskpart.h>`

`int` **diskpart_read_func**(`void` **driver_info*, `int` *sector*, `char` **buf*);

Description

This function is called from `diskpart_get_partition` and `diskpart_get_`*`type`* whenever they need to read data from the target disk.

Parameters

driver_info: The parameter passed to `diskpart_get_partition` and `diskpart_get_`*`type`*. Used to pass data through the diskpart library to this read routine.

sector: The sector to read.

buf: Memory location where the sector should be read in to. The buffer must be at least `SECTOR_SIZE` bytes.

Returns

Returns zero on success, non-zero to indicate an error.

22.5.3 `diskpart_blkio_get_partition`: initialize an array of partition entries

Synopsis

`#include <oskit/diskpart/diskpart.h>`

`int` **diskpart_blkio_get_partition**(`oskit_blkio_t` **block_io*, `struct diskpart` **array*, `int` *array_size*);

Description

This function initializes an array of `struct diskpart` entries. The caller must provide a pointer to a `struct diskpart` array.

This function is a version of `diskpart_get_partition` using an OSKit "Block I/O" interface in place of an explicit callback function.

Parameters

block_io: An `oskit_blkio_t` that represents the disk whose partitions we are interested in.

array: Array of `struct diskpart`.

array_size: integer containing the number of allocated entries in the array.

Returns

Returns an integer count of the number of partition entries that were filled by the library. If there were more partitions found than space available, this will be `array_size`. Empty partitions (unused entries in a BSD disklabel, for example) occupy an entry the same as 'used' entries.

For example, a PC-DOS partition with a single filled entry would still report 4 partitions, as that is the size of the DOS partition table.

Related Information

The OSKit Block I/O Interface (section 5.3).

22.5.4 `diskpart_fill_entry`: initialize a single partition entry

Synopsis

`#include <oskit/diskpart/diskpart.h>`

`void` **diskpart_fill_entry**(`struct diskpart` **array*, `int` *start*, `int` *size*, `struct diskpart` **subs*, `int` *nsubs*, `short` *type*, `short` *fsys*);

Description

This function initializes a single partition entry.

Parameters

array: Pointer to the `struct diskpart` entry to be filled

start: Starting sector on the disk for the partition.

size: Number of sectors in the partition.

subs: Pointer to its first child partition.

nsubs: Number of sub-partitions.

type: Partition type, as defined in diskpart.h

fsys: Filesystem in the partition (if known), as defined in diskpart.h

22.5.5 `diskpart_dump`: print a partition entry to stdout

Synopsis

`#include <oskit/diskpart/diskpart.h>`

`void` **diskpart_dump**(`struct diskpart *`*array*, `int` *level*);

Description

This function prints a partition entry with indentation and labeling corresponding to its nesting level. It also recursively prints any child partitions on separate lines, with level+1.

This provides valuable diagnostic messages for debugging disk or filesystem problems.

Parameters

array: A pointer to the first entry to be printed. It and any sub-partitions are printed.

level: int representing current level. This controls indentation and naming of the output. `diskpart_dump` called with the root `struct diskpart` entry and 0 will print the entire table.

Returns

Returns nothing, but does write to stdout.

22.5.6 `diskpart_lookup_bsd_compat`: search for a partition entry

Synopsis

`#include <oskit/diskpart/diskpart.h>`

`struct diskpart *`**diskpart_lookup_bsd_compat**(`struct diskpart *`*array*, `short` *slice*, `short` *part*);

Description

This function is a *sample* lookup routine which finds a partition given a slice number and partition number.

This demonstrates how a two-level naming scheme can be implemented using integers. This was first used in Mach 4 (UK22) to provide support for FreeBSD slices as well as backwards-compatibility with previous naming methods.

Parameters

array: This should be the pointer to the start of the array.

slice: Slice 0 is used as a 'compatibility slice', in that it is aliased to a BSD partition, if it exists. This allows users to not specify the slice for compatibility.

part: Partition 0 is used to represent the whole slice, and Partition 0, Slice 0 is the whole drive.

Returns

Returns a pointer to the corresponding partition entry, or zero if it is invalid.

22.5.7 `diskpart_lookup_bsd_string`: search for a partition entry

SYNOPSIS

`#include <oskit/diskpart/diskpart.h>`

`struct diskpart *`**diskpart_lookup_bsd_string**(`struct diskpart *`*array*, `char *`*name*);

DESCRIPTION

This function is a *sample* lookup routine which finds a partition given a FreeBSD style slice string. If no slice number is given, it defaults to the first BSD partition, and then to the whole disk if no BSD partition is found.

PARAMETERS

array: This should be the pointer to the start of the array.

name: A case-insensitive, NULL-terminated, ASCII string containing an optional Slice specifier followed by an optional partition. [s<num>][<part>], where part is a valid partition in the BSD slice specified by num (or default).

RETURNS

Returns a pointer to the corresponding partition entry, or zero if it is invalid.

22.5.8 `diskpart_blkio_lookup_bsd_string`: search for a partition entry

SYNOPSIS

`#include <oskit/diskpart/diskpart.h>`

`struct diskpart *`**diskpart_blkio_lookup_bsd_string**(`struct diskpart *`*array*, `char *`*name*, `oskit_blkio_t *`*block_io*, [out] `oskit_blkio_t **`*out_block_io*);

DESCRIPTION

This is similar to (and uses) `diskpart_lookup_bsd_string` but returns an OSKit "Block I/O" interface for the partition; i.e., operations on the returned `oskit_blkio_t` are restricted to the bounds of the partition.

PARAMETERS

array: This should be the pointer to the start of the array.

name: A case-insensitive, NULL-terminated, ASCII string containing an optional Slice specifier followed by an optional partition. [s<num>][<part>], where part is a valid partition in the BSD slice specified by num (or default).

block_io: The `oskit_blkio_t` whose partitions we are interested in.

out_block_io: A pointer to the new `oskit_blkio_t`.

RETURNS

Returns a pointer to the corresponding partition entry, or zero if it is invalid.

RELATED INFORMATION

`diskpart_lookup_bsd_string`, the OSKit Block I/O Interface (section 5.3).

22.5.9 `diskpart_get_`*type*: Search for *type* type partitions

SYNOPSIS

`#include <oskit/diskpart/diskpart.h>`

`int` **diskpart_get_*type***(`struct diskpart *`*array*, `char *`*buf*, `int` *start*, `void *`*driver_info*, `int` *(*diskpart_read_func)*(), int max_part);

DESCRIPTION

This function finds *type* type partitions if they are on the disk. These routines would not normally be invoked directly. However, the API is documented here so that `diskpart_lookup` can be extended easily for future or additional labeling schemes.

Currently defined functions are: `pcbios, disklabel, vtoc, dec,` and `omron`.

They should return immediately if `diskpart_read_func` returns non-zero, and return that error code.

PARAMETERS

array: Pointer to the start of preallocated storage.

buf: Pointer to a sector-sized scratch area.

start: Offset from start of disk the partition starts.

driver_info: A pointer to an initialized structure of user-defined type which is passed unmodified to `diskpart_read_func`.

diskpart_read_func: A function pointer provided by the user which can read a sector given `driver_info`.

max_part: Maximum number of partition entries that can be filled. This will generally be equal to the number of pre-allocated entries that are available.

RETURNS

Returns the number of partition entries of that type found. If none were found, it returns 0.

If the return value is equal to `max_part` then it is possible that there were more partitions than space for them. It is up to the user to ensure that adequate storage is passed to `diskpart_get_partitions`.

RELATED INFORMATION

`diskpart_read_func`

Chapter 23

File System Reader: `liboskit_fsread.a`

23.1 Introduction

The `fsread` library provides simple read-only access to Linux ext2fs, BSD FFS, and MINIX filesystems and is useful for small programs, such as boot loaders, that need to read files from the local disk but don't have the space for or need the features of the larger OSKit filesystem libraries.

Typically this library is used with the disk partitioning library (Section 22) and one of the driver libraries.

23.2 External dependencies

This depends on several memory and string routines from the OSKit C library, more specifically it depends on

- `free`, `malloc`
- `memcpy`, `memmove`, `memset`
- `oskit_blkio_iid`, `oskit_iunknown_iid`
- `strcmp`, `strcpy`, `strncpy`

23.3 Limitations

- Absolute symbolic links are interpreted relative to the root of the FS, since the FS readers have no notion of a "global root."
- For ext2fs, the "sb" mount option is not supported, the super block is assumed to be at block 1.
- The MINIX support has not been tested in the OSKit. but worked in a previous incarnation in Mach.

23.4 API reference

Each of these functions takes an `oskit_blkio_t` (Section 5.3) interface to the underlying device and a pathname relative to the root directory of the file system, and if the specified file can be found, returns an `oskit_blkio_t` object that can be used to read from that file. The `blkio` object returned will have a block size of 1, meaning that there are no alignment restrictions on file reads. The `blkio` object passed, representing the underlying device, can have a block size greater than 1, but if it is larger than the file system's block size, file system interpretation will fail. Also, any absolute symlinks followed during the open will be interpreted as if this is the root file system.

23.4.1 `fsread_open`: Open a file on various filesystems

SYNOPSIS

`#include <oskit/fs/read.h>`

`oskit_error_t` **fsread_open**(`oskit_blkio_t *`*device*, `const char *`*path*, [out] `oskit_blkio_t **`*out_file*);

DESCRIPTION

Tries to open a file named by *path* in the filesystem on *device*. If successful, returns a `blkio` into *out_file* that can be used to read the file.

This function is just a wrapper that calls the various filesystem-specific `fsread` functions, failing if none of them recognize the filesystem.

PARAMETERS

device: An `oskit_blkio_t` (Section 5.3) representing a device containing a filesystem.

path: A pathname indicating an existing file to open. This pathname is taken relative to the root of the filesystem

out_file: Upon success, this is set to an `oskit_blkio_t` that can be used to read from the file.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

23.4.2 `fsread_`*FSTYPE*`_open`: Open a file on the *FSTYPE* filesystem

SYNOPSIS

`#include <oskit/fs/read.h>`

`oskit_error_t` **fsread_*FSTYPE*_open**(`oskit_blkio_t *`*device*, `const char *`*path*, [out] `oskit_blkio_t **`*out_file*);

DESCRIPTION

Tries to open a file named by *path* in the *FSTYPE* filesystem on *device*. If successful, returns a `blkio` into *out_file* that can be used to read the file.

FSTYPE can be one of `ext2`, `ffs`, `minix`.

PARAMETERS

device: An `oskit_blkio_t` (Section 5.3) representing a device containing a *FSTYPE* filesystem.

path: A pathname indicating an existing file to open. This pathname is taken relative to the root of the filesystem

out_file: Upon success, this is set to an `oskit_blkio_t` that can be used to read from the file.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 24

Executable Program Interpreter: liboskit_exec.a

The OSKit provides a small library that can recognize and load program executables in a variety of formats. It is analogous to the GNU Binary File Descriptor (BFD) library, except that it only supports loading linked program executables rather than general reading and writing of all types of object files. For this reason, it is much smaller and simpler than BFD.

Furthermore, as with the other OSKit components, the executable interpreter library is designed to be as generic and environment-independent as possible, so that it can readily be used in any situation in which it is useful. For example, the library does not directly do any memory allocation; it operates purely using memory provided to it explicitly. Furthermore, it does not make any assumptions about how a program's code and data are to be written into the proper target address space; instead it uses generic callback functions for this purpose. All of the library functions are pure, not containing or relying on any global shared state.

All of the executable loading functions take pointers to two callback functions as parameters; the library calls these functions, which the client OS must provide, to load data from the executable and map or copy it into the address space of the program being loaded. Since all loading is done through these callback functions, the OSKit's executable interpreter code can be used to load executables either into the same address space as the program currently running (e.g., loading a kernel from a boot loader) or into a different address space. The prototypes and semantics of these callback functions are defined below, in Section 24.2.

24.1 Header Files

This section describes the header files provided with the OSKit's executable interpreter library. For normal use of the library, only `exec.h` is needed; however, the other headers are provided as a convenience for clients that need to do their own executable interpretation.

24.1.1 `exec.h`: definitions for executable interpreter functions

Description

This header file contains all the function prototypes for the executable loading functions described below, and all other symbol definitions required to use the executable program interpreter functions. Ths `exec.h` header file also defines the following error codes, returned from the executable loading functions:

`EX_NOT_EXECUTABLE`: Indicates that the file is not in any recognized executable format.

`EX_WRONG_ARCH`: Indicates that the file is in a recognized executable file format, but it is an executable for a different processor architecture.

`EX_CORRUPT`: Indicates that the file appears to be an executable file in a recognized format, but something is seriously wrong with it (e.g., the file was truncated or mangled).

`EX_BAD_LAYOUT`: Indicates that something is wrong with the memory or file image layout described by the executable file.

24.1.2 `a.out.h`: semi-standard `a.out` file format definitions

Description

This header file defines a set of structures and symbols describing `a.out`-format object and executable files. Since the `a.out` format is extremely nonstandard and varies widely even across different operating systems for the same processor architecture, this header file only provides a minimal, "least-common-denominator" set of definitions that applies to all the `a.out` variants we know of. Therefore, actually interpreting `a.out` files requires considerably more information than is provided in this header file; for more information, see the source code for the OSKit's `a.out` interpreter, in `exec/x86/aout.c`.

An `a.out` file contains a simple fixed-size header, represented by the following structure:

```
struct exec {
    unsigned long a_magic;    /* Magic number              */
    unsigned long a_text;     /* Size of text segment      */
    unsigned long a_data;     /* Size of initialized data  */
    unsigned long a_bss;      /* Size of uninitialized data */
    unsigned long a_syms;     /* Size of symbol table      */
    unsigned long a_entry;    /* Entry point               */
    unsigned long a_trsize;   /* Size of text relocation   */
    unsigned long a_drsize;   /* Size of data relocation   */
};
```

The `a_magic` field typically contain one of the following traditional magic numbers:

`OMAGIC`: Used for relocatable object files (`.o`'s).

`NMAGIC`: Originally used for executable files before demand-loading; current systems generally no longer use this.

`ZMAGIC`: This is the standard magic number for demand-loadable executable files; however, the exact meaning of this magic number varies from system to system.

`QMAGIC`: An alternate demand-loadable format, in which the `a.out` header itself is actually part of the text segment.

In addition, this header defines the `nlist` structure which describes the format of `a.out` symbol table entries:

```
struct nlist {
    long       n_strx;         /* Offset of symbol name in the string table    */
    unsigned   char n_type;    /* Symbol/relocation type                        */
    char       n_other;        /* Miscellaneous info                            */
    short      n_desc;         /* Miscellaneous info                            */
    unsigned   long n_value;   /* Symbol value                                  */
};
```

24.1.3 `elf.h`: standard 32-bit ELF file format definitions

Description

This header file contains a number of structure and symbol definitions describing the data structures used in ELF files. Since these names and their meanings are fairly well standardized, they are not described here; instead, see the ELF specification for details.

24.2 Types

This section describes the types and other symbols which the client OS must interact with in order to use the executable interpreter library. These symbols are defined in `oskit/exec/exec.h`.

24.2.1 `exec_read_func_t`: executable file reader callback

SYNOPSIS

`#include <oskit/exec/exec.h>`

`typedef int` **exec_read_func_t**(`void` **handle*, `oskit_addr_t` *file_ofs*, `void` **buf*, `oskit_size_t` *size*, [out] `oskit_size_t` **actual*);

DESCRIPTION

This type describes the function prototype of the `read` callback function which the client OS must supply to the executable interpreter; it is used by the executable interpreter library to read "metadata" from the executable file such as the executable file's header (as opposed to the actual executable data itself). It is basically analogous in purpose and semantics to the standard POSIX `read` function.

PARAMETERS

handle: This is simply the opaque pointer value originally passed by the client in the call to the executable interpreter; the client's callbacks typically use it to locate any state relevant to the executable being loaded. The actual use or meaning of this parameter is completely opaque to the executable interpreter library.

file_ofs: This parameter indicates the offset in the file at which to start reading.

buf: The buffer into which to read data.

size: The maximum amount of data to read from the file. Less than this much data may be read if end-of-file is reached during the read.

actual: The client callback returns the amount of data actually read in this parameter. It should be equal to the requested size unless the end-of-file was reached.

RETURNS

Returns 0 on success, or an error code on failure. The error code may be either one of the `EX_` error codes defined in `exec.h`, or it may be a caller-defined error code, which the executable interpreter code will simply pass directly back through to the original caller.

24.2.2 `exec_read_exec_func_t`: executable file reader callback

SYNOPSIS

`#include <oskit/exec/exec.h>`

`typedef int` **exec_read_exec_func_t**(`void` **handle*, `oskit_addr_t` *file_ofs*, `oskit_size_t` *file_size*, `oskit_addr_t` *mem_addr*, `oskit_size_t` *mem_size*, `exec_sectype_t` *section_type*);

Description

This type describes the function prototype of the `read_exec` callback function which the client OS must supply to the executable interpreter; it is used by the executable interpreter library to read actual executable code and data from the executable file to be copied or mapped into the loaded program's image. It is also used to indicate to the client where debugging information can be found in the executable, and what format it is in. The executable interpreter generally calls this function once for each "section" it finds in the executable file, indicating where in the executable file to load or map from and where in the resulting program image to copy or map to. The actual executable data itself never actually "passes through" the generic executable interpreter itself; instead, the interpreter merely "directs" the loading process, giving the client OS ultimate flexibility in the way the loading is performed. In fact, the client's callback function does not even necessarily need to "load" the executable: for example, if the client merely wants to determine the memory layout described by the executable file, it can provide a callback that does not actually load anything but instead just records the information passed by the executable interpreter.

Note that not all sections in an executable file are necessarily relevant to the loaded program image itself: for example, the executable interpreter also calls this callback when it encounters debug sections that the client may be interested in. Therefore, to avoid choking on such sections, the client's implementation of this callback function should always check the *section_type* parameter and ignore sections for which `EXEC_SECTYPE_ALLOC` is not set and it doesn't otherwise know how to deal with.

Parameters

handle: This is simply the opaque pointer value originally passed by the client in the call to the executable interpreter; the client's callbacks typically use it to locate any state relevant to the executable being loaded. The actual use or meaning of this parameter is completely opaque to the executable interpreter library.

file_ofs: This parameter indicates the offset in the file at which the section's data begins. This is only valid for sections that *have* file data: for example, for BSS sections, which are allocated but not loaded, this parameter is undefined.

file_size: Size of the section's data in the executable file, or zero for sections that have no file data, such as BSS sections.

mem_addr: The address in the loaded program's address space at which this section should be loaded. This address is found in or deduced from the executable file's metadata, and generally indicates the address for which this section of the program was linked. For sections that are not allocated in the program image (sections without the `EXEC_SECTYPE_ALLOC` flag), this parameter is undefined and should be ignored.

mem_size: The amount of memory to allocate for this section in the loaded program's address space. This is usually equal to *file_size*, but may be larger, in which case the remaining portion of the section past the end of the data actually loaded from the file must be initialized to zero.

section_type: Indicates the type of this section; it is a mask of the flag bits described below in Section 24.2.3.

Returns

Returns 0 on success, or an error code on failure. The error code may be either one of the `EX_` error codes defined in `exec.h`, or it may be a caller-defined error code, which the executable interpreter code will simply pass directly back through to the original caller.

24.2.3 `exec_sectype_t`: section type flags word

Synopsis

```
#include <oskit/exec/exec.h>

typedef int exec_sectype_t;
```

Description

The following flag definitions describe the `section_type` value that the executable program interpreter library passes back to the client in the `read_exec` callback:

`EXEC_SECTYPE_READ`: Indicates that the pages into which this section is loaded should be given read permission.

`EXEC_SECTYPE_WRITE`: Indicates that the pages into which this section is loaded should be given write permission.

`EXEC_SECTYPE_EXECUTE`: Indicates that the pages into which this section is loaded should be given execute permission.

`EXEC_SECTYPE_PROT_MASK`: This is a bit mask containing the above three bits; it can be used to isolate the protection information in the section type flags word.

`EXEC_SECTYPE_ALLOC`: This bit indicates that this section's virtual address range is valid and the corresponding region should be allocated in the loaded program's virtual address space. Most normal program sections include both `EXEC_SECTYPE_ALLOC` and `EXEC_SECTYPE_LOAD`, indicating that the section should be allocated and loaded. However, a section with only `EXEC_SECTYPE_ALLOC` corresponds to an uninitialized data or "BSS" section: the section's address range is allocated and zero-filled. Sections that don't include `EXEC_SECTYPE_ALLOC` typically contain debugging, symbol table, relocations, or other information not normally loaded with the program.

`EXEC_SECTYPE_LOAD`: This bit indicates that at least some of the section's virtual address range should be initialized by loading it from the executable program image, as specified in the `file_ofs` and `file_size` parameters to the `read_exec` function above. Implies `EXEC_SECTYPE_ALLOC`.

`EXEC_SECTYPE_DEBUG`: Indicates that this section contains debugging information, such as symbol table or line number information. The specific type of debugging information it contains is indicated by other bits defined below.

`EXEC_SECTYPE_AOUT_SYMTAB`: Indicates that this section is the symbol table section from an `a.out` executable. Implies `EXEC_SECTYPE_DEBUG`.

`EXEC_SECTYPE_AOUT_STRTAB`: Indicates that this section is the string table section from an `a.out` executable. Implies `EXEC_SECTYPE_DEBUG`.

24.2.4 `exec_info_t`: executable information structure

Synopsis

```
#include <oskit/exec/exec.h>

struct exec_info {
    exec_format_t  format;   /* Executable file format   */
    oskit_addr_t   entry;    /* Entrypoint address       */
    oskit_addr_t   init_dp;  /* Initial data pointer     */
}; typedef struct exec_info exec_info_t;
```

DESCRIPTION

Each of the executable interpreter functions described below fills in a caller-provided structure of this type after successfully loading an executable. This structure contains miscellaneous information about the executable: in particular, information needed to actually start the program running.

`format`: The file format in which the executable was expressed.

`entry`: The entrypoint address of the executable, which is where it should start running.

`init_dp`: This value is only relevant on some architectures (and in particular *not* the x86); it is a secondary address, typically loaded into another processor register when the program is started and used as a "data pointer" for accessing global data.

24.3 Function Reference

This section describes the actual functions exported to the client OS by the executable interpreter library, `liboskit_exec.a`.

24.3.1 `exec_load`: detect the type of an executable file and load it

Synopsis

`#include <oskit/exec/exec.h>`

`int` **exec_load**(`exec_read_func_t` **read*, `exec_read_exec_func_t` **read_exec*, `void` **handle*, [out] `exec_info_t` **info*);

Description

This is the primary entrypoint to the executable interpreter: it automatically detects the type of an executable file and loads it using the specified callback functions. This function simply calls, in succession, each of the format-specific executable loader functions that apply to the architecture for which the OSKit was compiled, until one succeeds or returns an error other than `EX_NOT_EXECUTABLE`.

Parameters

read: Specifies the metadata reader callback function, as described in Section 24.2.1.

read_exec: Specifies the executable data reader callback function, as described in Section 24.2.2.

handle: An opaque pointer value which the executable interpreter simply passes through to the callback functions.

info: A pointer to an `exec_info` structure which the executable interpreter will fill with information about the loaded executable.

Returns

Returns 0 on success, or an error code on failure. The error code may be either one of the `EX_` error codes defined in `exec.h`, or it may be a caller-defined error code returned by one of the callback functions and passed through to the client.

24.3.2 `exec_load_elf`: load a 32-bit ELF executable file

Synopsis

`#include <oskit/exec/exec.h>`

`int` **exec_load_elf**(`exec_read_func_t` **read*, `exec_read_exec_func_t` **read_exec*, `void` **handle*, [out] `exec_info_t` **info*);

Description

This function recognizes, interprets, and loads executables in the ELF (Executable and Linkable File) format.

PARAMETERS

read: Specifies the metadata reader callback function, as described in Section 24.2.1.

read_exec: Specifies the executable data reader callback function, as described in Section 24.2.2.

handle: An opaque pointer value which the executable interpreter simply passes through to the callback functions.

info: A pointer to an `exec_info` structure which the executable interpreter will fill with information about the loaded executable.

RETURNS

Returns 0 on success, or an error code on failure. The error code may be either one of the `EX_` error codes defined in `exec.h`, or it may be a caller-defined error code returned by one of the callback functions and passed through to the client.

24.3.3 `exec_load_aout`: load an `a.out`-format executable file

SYNOPSIS

`#include <oskit/exec/exec.h>`

`int` **exec_load_aout**(`exec_read_func_t` **read*, `exec_read_exec_func_t` **read_exec*, `void` **handle*, [out] `exec_info_t` **info*);

DESCRIPTION

This function recognizes, interprets, and loads executables in the traditional Unix `a.out` file format. Unfortunately, there are many variants of the `a.out` format, even on a single processor architecture, each with similar but incompatible interpretations. Furthermore, there is no completely reliable and unambiguous way to differentiate between many of these formats: they often use the same magic numbers even though they have very different meanings. However, by using some hairy but fairly reliable heuristics, tthe OSKit's `a.out` loader for the x86 architecture simultaneously supports Linux, NetBSD, FreeBSD, and Mach executables, in the `OMAGIC`, `NMAGIC`, `QMAGIC`, and several `ZMAGIC` variants. Thus, at least for executables linked on one of these systems, the OSKit's loader should "just work." Nevertheless, the `a.out` format is very outdated at best, and we strongly recommend anyone using the OSKit to use a more modern and powerful executable format such as ELF.

PARAMETERS

read: Specifies the metadata reader callback function, as described in Section 24.2.1.

read_exec: Specifies the executable data reader callback function, as described in Section 24.2.2.

handle: An opaque pointer value which the executable interpreter simply passes through to the callback functions.

info: A pointer to an `exec_info` structure which the executable interpreter will fill with information about the loaded executable.

RETURNS

Returns 0 on success, or an error code on failure. The error code may be either one of the `EX_` error codes defined in `exec.h`, or it may be a caller-defined error code returned by one of the callback functions and passed through to the client.

Chapter 25

Linux File Systems: `liboskit_linux_fs.a`

The Linux filesystem library consists of the Linux virtual and real filesystem code along with glue code to export the OSKit filesystem interface (See Chapter 7).

The header file `<oskit/fs/linux_filesystems.h>` determines which of the real Linux real filesystems, e.g. `ext2`, `iso9660`, are compiled into `liboskit_linux_fs.a`. All the filesystems listed in that file will compile, but only `ext2`, `msdos`, `vfat`, and `iso9660` have been tested.

The Linux filesystem library provides two additional interfaces:

`fs_linux_init`: Initialize the Linux fs library.

`fs_linux_mount`: Mount a filesystem via the Linux fs library

25.0.4 `fs_linux_init`: Initialize the Linux fs library

Synopsis

```
#include <oskit/fs/linux.h>
oskit_error_t fs_linux_init(void);
```

Direction

client OS $\rightarrow$ filesystem library

Description

This function initializes the Linux fs library, and must be invoked prior to the first call to `fs_linux_mount`. This function only needs to be invoked once by the client operating system.

All filesystems listed in `<oskit/fs/linux_filesystems.h>` are initialized.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

25.0.5 `fs_linux_mount`: Mount a filesystem via the Linux fs library

Synopsis

```
#include <oskit/fs/linux.h>
oskit_error_t fs_linux_mount(oskit_blkio_t *bio, oskit_u32_t flags, oskit_filesystem_t
**out_fs);
```

Direction

client OS → filesystem library

Description

This function looks in the partition described by `bio` for a filesystem superblock, calls the corresponding filesystem mount code, and returns a handle to an `oskit_filesystem_t` for this filesystem.

This function may be used multiple times by a client operating system to mount multiple file systems.

Note that this function does not graft the filesystem into a namespace; `oskit_dir_reparent` or other layers may be used for that purpose.

Typically, this interface is not exported to clients, and is only used by the client operating system during initialization.

This function is a wrapper for Linux's `mount_root`.

Parameters

bio: Describes the partition containing a filesytem. Can be the whole disk like that returned from `oskit_linux_block_open`, or a subset of one like what is given by `diskpart_blkio_lookup_bsd_string`.

flags: The mount flags. These are formed by or'ing the following values:

OSKIT_FS_RDONLY	Read only filesystem
OSKIT_FS_NOEXEC	Can't exec from filesystem
OSKIT_FS_NOSUID	Don't honor setuid bits on fs
OSKIT_FS_NODEV	Don't interpret special files

out_fs: Upon success, is set to point to an `oskit_filesystem_t` for this filesystem.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 26

NetBSD File Systems: `liboskit_netbsd_fs.a`

The NetBSD filesystem library consists of the NetBSD filesystem code and a collection of glue code which encapsulates the NetBSD code within the OSKit filesystem framework.

The NetBSD filesystem library provides two additional interfaces:

`fs_netbsd_init`: Initialize the NetBSD fs library.

`fs_netbsd_mount`: Mount a filesystem via the NetBSD fs library

26.0.6 `fs_netbsd_init`: Initialize the NetBSD fs library

SYNOPSIS

`#include <oskit/fs/netbsd.h>`

`oskit_error_t` **fs_netbsd_init**(`void`);

DIRECTION

client OS → filesystem library

DESCRIPTION

This function initializes the NetBSD fs library, and must be invoked prior to the first call to `fs_netbsd_mount`. This function only needs to be invoked once by the client operating system.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

26.0.7 `fs_netbsd_mount`: Mount a filesystem via the Netbsd fs library

SYNOPSIS

`#include <oskit/fs/netbsd.h>`

`oskit_error_t` **fs_netbsd_mount**(`oskit_blkio_t` **bio*, `oskit_u32_t` *flags*, `oskit_filesystem_t` ***out_fs*);

Direction

client OS → filesystem library

Description

This function mounts a FFS filesystem from the partition described by `bio`, and returns a handle to an `oskit_filesystem_t` for this filesystem.

This function may be used multiple times by a client operating system to mount multiple file systems.

Note that this function does not graft the filesystem into a namespace; `oskit_dir_reparent` or other layers may be used for that purpose.

Typically, this interface is not exported to clients, and is only used by the client operating system during initialization.

Parameters

bio: Describes the partition containing a filesytem. Can be the whole disk like that returned from `oskit_linux_block_open`, or a subset of one like what is given by `diskpart_blkio_lookup_bsd_string`.

flags: The mount flags. These are formed by or'ing the following values:

OSKIT_FS_RDONLY Read only filesystem

OSKIT_FS_NOEXEC Can't exec from filesystem

OSKIT_FS_NOSUID Don't honor setuid bits on fs

OSKIT_FS_NODEV Don't interpret special files

out_fs: Upon success, is set to point to an `oskit_filesystem_t` for this filesystem.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 27

FreeBSD Networking: liboskit_freebsd_net.a

A note of caution. The design of the networking framework is nowhere near where it should be. However, we, the OSKit developers, felt that networking is important and decided to release what we have.

The current implementation has been tested with only the most frequently used calls for TCP and UDP over a single ethernet interface, with a default router on the subnet to which the interface is connected. XXX We have done a simple router test with two interfaces.

27.1 Introduction

This chapter describes the use and implementation of the FreeBSD TCP/IP networking stack.

Note: The file `unsupported/start_network.c` has example code of how to quickly start up your network. Then you may use the `socket` et al functions in the C library, as demonstrated in `socket_bsd.c` in the example directory. The file `socket_com.c` contains the same example but uses the COM interfaces without C library support.

Limitation/pecularity of the current implementation:

We have not yet implemented "principals" as described in the filesystem framework. All operations run with full privileges.

`oskit_socket_t` instances created by the networking stack do not currently implement the following methods[1]: `getsockopt`, `recvmsg/sendmsg`

Also, the local loopback interface does not work because it is not properly set up. If you try to connect to `127.0.0.1` or to the local IP address, you'll see a division by zero trap in `freebsd/src/sys/netinet/ip_output.c:302` because the `if_mtu` field is uninitialized. Required fix is to call the required "ifconfig" functions for the loopback interface.

[1]An `OSKIT_E_NOTIMPL` error code will be returned.

27.2 Header Files

27.2.1 `freebsd.h`: definitions for the FreeBSD-derived networking code

Synopsis

```
#include <oskit/net/freebsd.h>
```

Description

Contains function definitions for the functions needed to initialize and use the FreeBSD networking stack.

It defines some convenience functions to initialize the code, to set up an interface and to establish a default route. It defines `struct oskit_freebsd_net_ether_if` which is **opaque** to OS clients.

27.3 Interfaces

27.3.1 oskit_freebsd_net_init: initialize the networking code

SYNOPSIS

`#include <oskit/net/freebsd.h>`

`oskit_error_t` **oskit_freebsd_net_init**([out] *oskit_socket_factory_t **outfact*);

DIRECTION

OS → Network stack

DESCRIPTION

This function initializes the FreeBSD networking code.

PARAMETERS

outfact: **outfact* will contain a `oskit_socket_factory_t *` which can be used according to the specifications in the OSKit socket_factory COM interface in Chapter 8.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Limitation/pecularity of the current implementation:

`oskit_freebsd_net_init` will always succeed, and may panic otherwise.

27.3.2 oskit_freebsd_net_open_ether_if: find and open an ethernet interface

SYNOPSIS

`#include <oskit/net/freebsd.h>`

`oskit_error_t` **oskit_freebsd_net_open_ether_if**(*struct oskit_etherdev *dev*, [out] `struct` *oskit_freebsd_net_ether_if ** out_eif*);

DIRECTION

OS → Network stack

DESCRIPTION

This function is a convenience function to open an ethernet device and create the necessary `oskit_netio_t` instances to "connect" the `netio` device drivers to the protocol stack.

Note: The code uses the following internal structure to keep track of an ethernet interface, defined in `oskit/net/freebsd.h`

```
struct oskit_freebsd_net_ether_if {
    oskit_etherdev_t  *dev;                                  /* ethernet device                    */
    oskit_netio_t     *send_nio;                             /* netio for sending packets          */
    oskit_netio_t     *recv_nio;                             /* netio for receiving packets        */
    oskit_devinfo_t   info;                                  /* device info                        */
    unsigned          char haddr[OSKIT_ETHERDEV_ADDR_SIZE];  /* MAC address                        */
    struct            ifnet *ifp;                            /* actual interface seen by BSD code  */
};
```

ifp is the actual interface as seen by the BSD code. *recv_nio* points to the `netio` COM interface receiving packets from the ethernet device *dev* and passing them to the BSD networking code. *send_nio* is the `netio` used by the code to send packets. *haddr* contains the MAC address, and *info* the device info associated with *dev*.

PARAMETERS

dev: The ethernet device to be used with the interface. Note that the FreeBSD net library will take this reference over.

out_eif: **out_eif* points to an `oskit_freebsd_net_ether_if` structure on success.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

27.3.3 `oskit_freebsd_net_open_first_ether_if`: find and open first ethernet interface

SYNOPSIS

`#include <oskit/net/freebsd.h>`

`oskit_error_t` **oskit_freebsd_net_open_first_ether_if**([out] *struct oskit_freebsd_net_ether_if ** out_eif*);

DIRECTION

OS → Network stack

DESCRIPTION

This function is a convenience function to find and open the first ethernet device[2] and to create an associated `oskit_freebsd_net_ether_if` structure.
Limitation/pecularity of the current implementation:

It leaks references to other ethernet devices, if any.

PARAMETERS

out_eif: **out_eif* points to an `oskit_freebsd_net_ether_if` structure on success.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

27.3.4 `oskit_freebsd_net_close_ether_if`: close an ethernet interface

SYNOPSIS

`#include <oskit/net/freebsd.h>`

`void` **oskit_freebsd_net_close_ether_if**(`struct` *oskit_freebsd_net_ether_if *eif*);

DIRECTION

OS → Network stack

[2]according to `osenv_device_lookup` using an `oskit_etherdev_iid`

Description

The function closes the interface and frees the oskit_freebsd_net_ether_if structure.

This is currently done by releasing the two `netio` instances and the `oskit_etherdev_t` instance in `struct oskit_freebsd_net_ether_if`.

Parameters

eif: Interface to be closed.

27.3.5 `oskit_freebsd_net_ifconfig`: configure an interface

Synopsis

`#include <oskit/net/freebsd.h>`

`oskit_error_t` **oskit_freebsd_net_ifconfig**(*struct oskit_freebsd_net_ether_if *eif*, `char *`*name*, `char *`*ipaddr*, `char *`*netmask*);

Direction

OS → Network stack

Description

This is a temporary convenience function which does the setup usually performed by FreeBSD's `ifconfig(8)` command. This function is equivalent to the following command:

```
ifconfig de0 inet 155.99.214.164 link2 netmask 255.255.255.0
```

with `155.99.214.164` being the IP address to be used by the ethernet interface, `255.255.255.0` the netmask of the local subnet, and `de0` the (arbitrary) name of the interface.

Parameters

eif: A structure describing the physical interface as returned by `oskit_freebsd_net_open_ether_if`.

name: The name of the interface. Should be a 3 byte string of the `"abn"` where `a` and `b` are letters and `n` is a number. Use different names for different interfaces.

ipaddr: The address to be used by the interface in `"xxx.xxx.xxx.xxx"` notation.

netmask: The netmask of the subnet to be used by the interface in `"xxx.xxx.xxx.xxx"` notation.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

27.3.6 `oskit_freebsd_net_add_default_route`: set a default route

Synopsis

`#include <oskit/net/freebsd.h>`

`oskit_error_t` **oskit_freebsd_net_add_default_route**(`char *`*gateway*);

Direction

OS → Network stack

Description

This function sets a default route.

Limitation/pecularity of the current implementation:

Take a look at the implementation in `freebsd/net/bsdnet_add_default_route.c`.

Parameters

gateway: The IP address of the default gateway to be set in `"xxx.xxx.xxx.xxx"` notation.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 28

BOOTP Support: `liboskit_bootp.a`

28.1 Introduction

The OSKit provides a small library that allows OSKit clients to contact a BOOTP server to obtain the information necessary to configure their network parameters, such as IP address or hostname. The implementation is based on RFC 1048/1533.

28.2 External Dependencies

The current implementation requires the user to provide an implementation of the `oskit_etherdev` interface. Such interfaces are supported by the OSKit's device driver components. It also requires access to the system clock. The library obtains access to the system clock by calling `oskit_clock_init` It releases its reference after the BOOTP protocol has finished. Additionally, `bootp_dump` uses `printf`.

28.3 API reference

The following sections describe the functions exported by the BOOTP library in detail. All of these functions, as well as the types necessary to use them, are declared in the header file `<oskit/net/bootp.h>`.

28.3.1 bootp_net_info: BOOTP protocol information structures

SYNOPSIS

```
struct bootp_addr_array {
        struct in_addr *addr;   /* array of addresses */
        int             len;    /* number of addresses */
};

struct bootp_net_info {
        oskit_u32_t    flags;           /* which fields are valid */
        struct in_addr ip;              /* client IP address */
        struct in_addr netmask;         /* subnet mask */
        struct in_addr server;          /* server that replied  */
        struct bootp_addr_array gateway;        /* gateways */
        struct bootp_addr_array dns_server;     /* DNS server address  */
        struct bootp_addr_array time_server;    /* time server address  */
        struct bootp_addr_array log_server;     /* log server address  */
        struct bootp_addr_array lpr_server;     /* LPR server address  */
        oskit_s32_t    time_offset;     /* offset from UTC */
        char *hostname;                 /* client hostname */
        char *server_name;              /* name of replying server */
        char *bootfile_name;            /* boot file name */
        char *domainname;               /* domain name */
        unsigned char server_hwaddr[ETHER_ADDR_SIZE];   /* server address */
};
```

DESCRIPTION

The first field of the `struct bootp_net_info` structure, *flags*, denotes which of the other fields are valid. *flags* is an ORed combination of the flag values below. Each flag corresponds to a field in the structure with the same name (in lower case).

All other fields are of one of three types:

1. An IP address encoded as a `struct in_addr`,
2. A string encoded as a `char *`,
3. A list of IP addresses encoded in a `struct bootp_addr_array`.

The following table gives an overview of the flags that are currently supported and the types of the corresponding fields.

Field	Type	Function
BOOTP_NET_IP	IP address	IP address
BOOTP_NET_NETMASK	IP address	netmask
BOOTP_NET_GATEWAY	IP address	gateway
BOOTP_NET_SERVER	IP address	server that answered BOOTP request
BOOTP_NET_DNS_SERVER	List of IP addrs	domain name servers
BOOTP_NET_TIME_SERVER	List of IP addrs	time servers
BOOTP_NET_LOG_SERVER	List of IP addrs	log servers
BOOTP_NET_LPR_SERVER	List of IP addrs	LPR servers
BOOTP_NET_TIME_OFFSET	unsigned int	see below
BOOTP_NET_HOSTNAME	string	hostname
BOOTP_NET_SERVER_NAME	string	name of the BOOTP server
BOOTP_NET_BOOTFILE_NAME	string	bootfile name
BOOTP_NET_DOMAINNAME	string	DNS domain name
BOOTP_NET_SERVER_ADDR	unsigned char[6]	Ethernet MAC address of BOOTP server

The *time_offset* field specifies the time offset of the local subnet in seconds from Coordinated Universal Time (UTC). It is a signed 32-bit integer, positive numbers indicate west of the Prime Meridian.

28.3.2 `bootp_gen`: Generate a BOOTP protocol request

SYNOPSIS

`#include <oskit/net/bootp.h>`

`int` **bootp_gen**(`oskit_etherdev_t` **dev*, [in/out] `struct bootp_net_info` **info*, `int` *retries*, `int` *timeout*);

DESCRIPTION

This function broadcasts *retries* BOOTP request packets, waiting *timeout* milliseconds for a reply.

The only field of *info* that is used as input for the request is the *ip* field corresponding to `BOOTP_NET_IP`. See RFC 1048 for more explanation. Users should set this field if they know their IP address; `BOOTP_NET_IP` needs to be set in *flags* if this is the case.

Lists of IP addresses and strings are dynamically allocated as needed, users of `boopt_gen` should pass the *info* struct to `bootp_free` to deallocate them.

PARAMETERS

dev: A pointer to an `oskit_etherdev` device interface.

info: The BOOTP info to be used.

retries: Number of BOOTP request packets that are sent.

timeout: Timeout in milliseconds.

RETURNS

Returns zero on success, `OSKIT_ETIMEDOUT` if the operation timed out, or another error code as specified in `<oskit/error.h>`

RELATED INFORMATION

`bootp_net_info`

28.3.3 `bootp`: Generate a BOOTP protocol request (simple interface)

SYNOPSIS

`#include <oskit/net/bootp.h>`

`int` **bootp**(`oskit_etherdev_t` **dev*, [in/out] `struct bootp_net_info` **info*);

DESCRIPTION

This function performs a BOOTP request with a timeout of 200 milliseconds (1/5 of a second) with five retries using `bootp_gen`.

The only field of *info* that is used as input for the request is the *ip* field corresponding to `BOOTP_NET_IP`. See RFC 1048 for more explanation. Users should set this field if they know their IP address; `BOOTP_NET_IP` needs to be set in *flags* if this is the case.

Lists of IP addresses and strings are dynamically allocated as needed, users of `boopt_gen` should pass the *info* struct to `bootp_free` to deallocate them.

Parameters

dev: A pointer to an `oskit_etherdev` device interface.

info: The BOOTP info to be used.

Returns

Returns zero on success, or an error code specified in `<oskit/error.h>` on error.

Related Information

`bootp_net_info`

28.3.4 `bootp_free`: Free the result of a BOOTP request

Synopsis

```
#include <oskit/net/bootp.h>
void bootp_free([in/out] struct bootp_net_info *info);
```

Description

The function frees data structures that were allocated during a call to `bootp_gen`.

Parameters

info: The BOOTP info to be freed.

Related Information

`bootp_gen`, `bootp_net_info`

28.3.5 `bootp_dump`: Dump the result of a BOOTP via printf

Synopsis

```
#include <oskit/net/bootp.h>
void bootp_dump(struct bootp_net_info *info);
```

Description

This function prints the contents stored in a `bootp_net_info` structure to via `printf`.

Parameters

info: The BOOTP info to be printed.

Related Information

`bootp_net_info`, `printf`

Chapter 29

HPFQ: Hierarchical Network Link Sharing: `liboskit_hpfq.a`

29.1 Introduction

This short section outlines the Hierarchical Packet Fair Queuing (H-PFQ) network link-sharing implementation in the OSKit. The actual H-PFQ algorithm implemented is called H-WF^{2}Q and is described in the SIGCOMM96 paper by Bennet and Zhang. A working understanding of this paper would be useful in understanding the use of this library.

Be aware that this is a first-cut implementation and is not thoroughly tested nor tuned.

This library allows the user to hierarchically schedule outgoing traffic from different sessions through a single link. Each session is guaranteed a percentage of its parent's bandwidth, relative to its siblings. This is done by creating a scheduling tree where interior nodes correspond to one-level PFQ schedulers, and leaf nodes corresond to `oskit_netio` objects for the sessions they represent (see Section 5.5). The root node corresponds to the physical link being shared. The definition of *session* is up to the user, who controls what is sent to the leaf node `oskit_netio`'s. Typical session types are real-time traffic, traffic from different organizations, or protocol types. Note, however, that this more general issue of *packet classification* is not part of this library.

Currently, only one link can be managed at a time, because of the `oskit_pfq_root` global variable. This will be fixed in a future version.

29.2 Configuration

The H-PFQ library depends on certain modifications to the OSKit Linux device driver set (Section 30) that are enabled only when the OSKit is configured with the `--enable-hpfq` configure option. This configure option enables H-PFQ-specific code in the Linux driver set.

This library will *not* work correctly with an improperly configured Linux driver set. Similarly, a Linux driver set configured with `--enable-hpfq` will only work correctly for non-H-PFQ applications if `oskit_pfq_root` is `NULL`.

29.3 Usage

The basic procedure of using this library is to first create a scheduling hierarchy according to the user's needs, and then to retrieve the `oskit_netio`'s from the leaf nodes for use by the various sessions. The sessions can then send on these `oskit_netio`'s and the data will flow to the root according to the policies and allocations in place.

There are no restrictions on the format of the data sent to the leaf `oskit_netio`'s, but it must be what the `oskit_netio` corresponding to the root expects. In the common case of `oskit_netio`'s created by the

Linux driver library, this data will simply be Ethernet frames.

The creation of the hierarchy is done by first creating a root node and setting the global variables `oskit_pfq_root` and `oskit_pfq_reset_path` appropriately (see Section 29.4). Then various intermediate and/or leaf nodes are created and attached to the root with appropriate share values. This process is then repeated as needed for the children of any intermediate nodes.

In this library, share values are floating point numbers that represent a percentage of the parent's bandwidth allocated to the child. For example, a child with share value 0.45 is guaranteed 45% of the parent's bandwidth when the child has data to send, assuming the parent has not over-subscribed its bandwidth.

On a given level of the hierarchy, only the relative differences between share values is important, however for simplicity it is recommended that share values on a given level add up to 1.

A more subtle implication of this relative-differences fact, is that parents can over-subscribe their bandwidth to their children. More specifically, there is no guarantee that a session with a share value of, say 50% will actually receive that amount of the parent's bandwidth. To see this, consider the case of an intermediate node with two children, each allocated 50% of the bandwidth. Another child may be added with a share value of 50%, but it will in reality only receive 33%. This is more generally termed a problem of *admission control*, and is not currently dealt with in this library.

29.4 API reference

The following sections describe the functions exported by the H-PFQ library in detail. All of these functions, as well as the types necessary to use them, are declared in the header file `<oskit/hpfq.h>`.

This API reference is split into two parts. The first part describes the external requirements for the library and the actual functions exported, which are basically constructors for `pfq_sched` and `pfq_leaf` objects. The second part describes the `pfq_sched` and `pfq_leaf` COM interfaces.

29.5 External Requirements and Constructors

This section describes the external requirements of the library and the actual functions exported, which consist of "constructor" functions to create `pfq_sched` and `pfq_leaf` COM objects.

29.5.1 `oskit_pfq_root`: the root node scheduler

SYNOPSIS

```
#include <oskit/hpfq.h>
extern pfq_sched_t *oskit_pfq_root;
```

DESCRIPTION

This variable is not directly used by the H-PFQ library but is used to communicate between the Linux driver set and the program using the H-PFQ library.

The client of the H-PFQ library is responsible for setting this variable to point to the root node of its scheduling hierarchy before any sessions attempt to send to their respective leaf `oskit_netio` objects.

If this variable is set to the `NULL` value, then the Linux driver library will not call back to the H-PFQ code and thus will behave like a Linux driver set not configured for H-PFQ.

Note that this implies that only one link can be managed at a time. This will be fixed in a future version.

29.5.2 `oskit_pfq_reset_path`: pointer to the `reset_path` function

Synopsis

```
#include <oskit/hpfq.h>
extern void (*oskit_pfq_reset_path)(pfq_sched_t *);
```

Description

This variable is not directly used by the H-PFQ library but is used to communicate between the Linux driver set and the program using the H-PFQ library.

The client of the H-PFQ library is responsible for setting this variable to point to the `pfq_reset_path` function before any sessions attempt to send to their respective leaf `oskit_netio` objects.

29.5.3 `pfq_sff_create_root`: create a root node implementing SFF

Synopsis

```
#include <oskit/hpfq.h>
oskit_error_t pfq_sff_create_root(oskit_netio_t *link, pfq_sched_t **out_sched);
```

Description

Creates a root PFQ node implementing the Smallest Finish time First (SFF) scheduling policy.

Parameters

link: The link that the scheduling tree intends to manage.

out_sched: A pointer to the `pfq_sched` object representing the root of the hierarchy. This can be then used with future `pfq_sched` methods to add children, etc.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.5.4 `pfq_ssf_create_root`: create a root node implementing SSF

Synopsis

```
#include <oskit/hpfq.h>
oskit_error_t pfq_ssf_create_root(oskit_netio_t *link, pfq_sched_t **out_sched);
```

Description

Creates a root PFQ node implementing the Smallest Start time First (SSF) scheduling policy.

Parameters

link: The link that the scheduling tree intends to manage.

out_sched: A pointer to the `pfq_sched` object representing the root of the hierarchy. This can be then used with future `pfq_sched` methods to add children, etc.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.5.5 `pfq_sff_create`: create an intermediate node implementing SFF

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_sff_create**(`pfq_sched_t **`*out_sched*);

Description

Creates an intermediate PFQ node implementing the Smallest Finish time First (SFF) scheduling policy.

Parameters

out_sched: A pointer to the `pfq_sched` object representing the the created intermediate node. This can be then used with future `pfq_sched` methods to add children, etc.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.5.6 `pfq_ssf_create`: create an intermediate node implementing SSF

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_ssf_create**(`pfq_sched_t **`*out_sched*);

Description

Creates an intermediate PFQ node implementing the Smallest Start time First (SSF) scheduling policy.

Parameters

out_sched: A pointer to the `pfq_sched` object representing the the created intermediate node. This can be then used with future `pfq_sched` methods to add children, etc.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.5.7 `pfq_leaf_create`: create a leaf node

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_leaf_create**(`pfq_leaf_t **`*out_leaf*);

Description

Create a leaf PFQ node.

The `oskit_netio` that can be used by the session corresponding to this leaf can be retreived by calling `pfq_leaf_get_netio`, described elsewhere in this document.

Parameters

out_leaf: A pointer to the `pfq_leaf` object representing the the created intermediate node. This can be then used with future `pfq_leaf` methods to set the share value, etc.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.6 `pfq_sched`: Interface to PFQ Schedulers

This section describes the `pfq_sched` COM interface to PFQ scheduler objects. Note that the `child` parameter to these methods is declared as a `pfq_sched`. However, `pfq_leaf` inherits from `pfq_sched` and thus may be used as a `child` parameter when suitably cast.

The `pfq_sched` interface inherits from `IUnknown` and has the following additional methods:

`add_child`: Add a child to this node

`remove_child`: Remove a child from this node

`set_share`: Set the bandwidth share given to this node

29.6.1 `pfq_sched_add_child`: add a child to a root or intermediate node

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_sched_add_child**(`pfq_sched_t` **sched*, `pfq_sched_t` **child*, `float` *share*);

Description

This method attaches a child `pfq_sched` object to a parent and assigns it an initial share of the parent. The share can be later adjusted with the `set_share` method if needed.

Parameters

sched: The parent `pfq_sched` object.

child: The child being added.

share: The initial share value of the child. See Section 29.3 for details on share values.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.6.2 `pfq_sched_remove_child`: remove a child from a root or intermediate node

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_sched_remove_child**(`pfq_sched_t` **sched*, `pfq_sched_t` **child*);

Description

This method removes a child from a parent `pfq_sched` object.

Parameters

sched: The parent `pfq_sched` object losing a child.

child: The child to be orphaned.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.6.3 `pfq_sched_set_share`: allocate a percentage of the parent's bandwidth

SYNOPSIS

```
#include <oskit/hpfq.h>
oskit_error_t pfq_sched_set_share(pfq_sched_t *sched, pfq_sched_t *child, float share);
```

DESCRIPTION

This method adjusts the share value of a `pfq_sched` object. See Section 29.3 for details on share values.

PARAMETERS

sched: The parent `pfq_sched` object.

child: The child getting their share adjusted.

share: The new share value of the child.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.7 `pfq_leaf`: Interface to PFQ Leaf Nodes

This section describes the `pfq_leaf` COM interface to PFQ leaf objects.

The `pfq_leaf` interface inherits from `pfq_sched` and the following additional method:

`get_netio`: Get the `oskit_netio` corresponding to this leaf

Note that since `pfq_leaf` inherits from `pfq_sched`, it may be used in place of a `pfq_sched` object when suitably cast.

29.7.1 `pfq_leaf_add_child`: add a child to a root or intermediate node

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_leaf_add_child**(`pfq_leaf_t` **sched*, `pfq_sched_t` **child*, `float` *share*);

Description

This does not make sense for leaf nodes and is thus not implemented.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

29.7.2 `pfq_leaf_remove_child`: remove a child from a root or intermediate node

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_leaf_remove_child**(`pfq_leaf_t` **sched*, `pfq_sched_t` **child*);

Description

This does not make sense for leaf nodes and is thus not implemented.

29.7.3 `pfq_leaf_set_share`: allocate a percentage of the parent's bandwidth

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_leaf_set_share**(`pfq_leaf_t` **sched*, `pfq_sched_t` **child*, `float` *share*);

Description

This does not make sense for leaf nodes and is thus not implemented.

29.7.4 `pfq_leaf_get_netio`: get the `oskit_netio` corresonding to this leaf

Synopsis

`#include <oskit/hpfq.h>`

`oskit_error_t` **pfq_leaf_get_netio**(`pfq_leaf_t` **leaf*, `oskit_netio_t` ***out_netio*);

Description

Retrieves a pointer to an `oskit_netio` that can be used to send data by the session corresponding to this leaf.

Parameters

leaf: The leaf who's `oskit_netio` is of interest.

out_netio: A pointer to the `oskit_netio` object that can be used to send data by the session corresponding to this leaf.

Returns

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

Chapter 30

Linux Driver Set: `liboskit_linux_dev.a`

The Linux device driver library consists of various native Linux device drivers coupled with glue code to export the OSKit interfaces such as blkio, netio, and bufio (See Chapter 5). See the source file `linux/dev/README` for a list of devices and their status.

The header files `oskit/dev/linux_ethernet.h` and `oskit/dev/linux_scsi.h` determine which network and SCSI drivers are compiled into `liboskit_linux_dev.a`. Those files also influence driver probing; see the `oskit_linux_init` routines below.

30.1 Initialization and Registration

There are several ways to initalize this library. One can either initialize all the compiled-in drivers (`oskit_linux_init_devs`, initialize a specific class of drivers (`oskit_linux_init_ethernet`), or initialize specific drivers (e.g., `oskit_linux_init_scsi_ncr53c8xx`).

These initialization functions initialize various glue code and register the appropriate device(s) in the device tree, to be probed with `oskit_dev_probe`.

30.1.1 `oskit_linux_init_devs`: Initialize and register all known drivers

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_devs(void);
```

Description

This function initializes and registers all known drivers. The known drivers are: the IDE disk driver, and all drivers listed in the `<oskit/dev/linux_ethernet.h>` and `<oskit/dev/linux_scsi.h>` files.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.2 `oskit_linux_init_net`: Initialize and register known network drivers

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_net(void);
```

Description

This function initializes and registers all available network drivers. Currently this means Ethernet drivers only, but in the future there may be other network drivers supported such as Myrinet. The known Ethernet drivers are listed in the `<oskit/dev/linux_ethernet.h>` file.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.3 `oskit_linux_init_ethernet`: Initialize and register known Ethernet network drivers

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_ethernet(void);
```

Description

This function initializes and registers all available Ethernet network drivers. The known Ethernet drivers are listed in the `<oskit/dev/linux_ethernet.h>` file.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.4 `oskit_linux_init_scsi`: Initialize and register known SCSI disk drivers

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_scsi(void);
```

Description

This function initializes and registers all available SCSI disk drivers. The known SCSI drivers are listed in the `<oskit/dev/linux_scsi.h>` file.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.5 `oskit_linux_init_ide`: Initialize and register known IDE disk drivers

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_ide(void);
```

Description

This function initializes and registers all available IDE disk drivers. There is currently only one IDE driver.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.6 `oskit_linux_init_scsi_`*`name`*: Initialize and register a specific SCSI disk driver

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_scsi_name(void);
```

Description

This function initializes and registers a specific SCSI disk driver. The *name* must be one from the `name` field of the drivers listed in the `<oskit/dev/linux_scsi.h>` file.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.1.7 `oskit_linux_init_ethernet_`*`name`*: Initialize and register a specific Ethernet network driver

Synopsis

```
#include <oskit/dev/linux.h>
void oskit_linux_init_ethernet_name(void);
```

Description

This function initializes and registers a specific Ethernet network driver. The *name* must be one from the `name` field of the drivers listed in the `<oskit/dev/linux_ethernet.h>` file.

Once drivers are registered, their devices may be probed via `oskit_dev_probe`.

30.2 Obtaining object references

Once the desired drivers are initialized, registered, and probed, one can obtain references to their `blkio`, `netio`, etc interfaces (See Chapter 5) two different ways.

The first way is to look them up via their Linux name, e.g., "sd0" for a SCSI disk, or "eth0" for a Ethernet device. This is described here as it is specific to Linux.

The second, and preferred, way is to use `osenv_device_lookup` to find a detected device with the desired interface, such as `oskit_etherdev_iid` (See Chapter 12).

30.2.1 `oskit_linux_block_open`: Open a disk given its Linux name

SYNOPSIS

`#include <oskit/dev/linux.h>`

`oskit_error_t` **oskit_linux_block_open**(`const` *char *name*, `unsigned` *flags*, [out] `oskit_blkio_t` `**`*out_io*);

DESCRIPTION

This function takes a Linux name of a disk, e.g., "sd0" or "wd1", and returns an `oskit_blkio_t` that can be used to access the device.

The `oskit_blkio` interface is described in Chapter 5.

PARAMETERS

name: The Linux name of the device e.g., "sd0" or "wd1".

flags: Formed by or'ing the following values:

OSKIT_DEV_OPEN_READ
OSKIT_DEV_OPEN_WRITE
OSKIT_DEV_OPEN_ALL

out_io: Upon success, is set to point to an `oskit_blkio_t` that can be used to access the device.

RETURNS

Returns 0 on success, or an error code specified in `<oskit/error.h>`, on error.

30.2.2 `oskit_linux_block_open_kdev`: Open a disk given its Linux kdev

30.2.3 `oskit_linux_netdev_find`: Open a netcard given its Linux name

30.2.4 `oskit_linux_net_open`: Open a netcard given its Linux name

The rest of this chapter is very incomplete. Some of the internal details of the Linux driver emulation are described, but not the aspects relevant for typical use of the library.

30.3 Introduction

XXX

Much of the data here on Linux device driver internals is out-of-date with respect to the newer device drivers that are now part of the OSKit. This section documents drivers from Linux 1.3.6.8 or earlier; the current OSKit drivers are from Linux 2.0.29, so parts of this section are likely no longer correct.

XXX Library can be used either as one component or can be used to produce many separate components, depending on how it is used.

30.4 Partially-compliant Drivers

There are a number of assumptions made by *some* drivers: if a given assumption is not met by the OS using the framework, then the drivers that make the assumption will not work, but other drivers may still be usable. The specific assumptions made by each partially-compliant driver are listed in a table in the appropriate section below; here is a summary of the assumptions some of the drivers make:

- Kernel memory can be allocated from interrupt handlers.
- Drivers can allocate contiguous chunks of physical memory larger than one page.
- (x86) Drivers can allocate memory specifically in the low 16MB of memory accessible to the PC's built-in DMA controller.
- Drivers can sleep uninterruptibly.
- Drivers can access the clock timer and DMA registers directly.
- "Poll-and-Yield:" polls for short periods of time and yields the CPU without explicitly going to sleep.

30.5 Internals

The following sections document all the variables and functions that Linux drivers can refer to. These variables and functions are provided by the glue code supplied as part of the library, so this information should not be needed for normal use of the library under the device driver framework. However, they are documented here for the benefit of those working on this library or upgrading it to new versions of the Linux drivers, or for those who wish to "short-cut" through the framework directly to the Linux device drivers in some situations, e.g., for performance reasons.

30.5.1 Namespace Management Rules

For an outline of our namespace management conventions, see Section 4.7.2 in our SOSP paper, http://www.cs.utah.edu/projects/flux/papers.html#SOSKIT.

30.5.2 Variables

`current`: This is a global variable that points to the state for the current process. It is mostly used by drivers to set or clear the interruptible state of the process.

`jiffies`: Many Linux device drivers depend on a global variable called `jiffies`, which in Linux contains a clock tick counter that is incremented by one at each 10-millisecond (100Hz) clock tick. The device drivers typically read this counter while polling a device during a (hopefully short) interrupt-enabled busy-wait loop. Although a few drivers take the global clock frequency symbol `HZ` into account when determining timeout values and such, most of the drivers just used hard-coded values when using the `jiffies` counter for timeouts, and therefore assume that `jiffies` increments "about" 100 times per second.

`irq2dev_map`: This variable is an array of pointers to network device structures. The array is indexed by the interrupt request line (IRQ) number. Linux network drivers use it in interrupt handlers to find the interrupting network device given the IRQ number passed to them by the kernel.

`blk_dev`: This variable is an array of "struct blk_dev_struct" structures. It is indexed by the major device number. Each element contains the I/O request queue and a pointer to the I/O request function in the driver. The kernel queues I/O requests on the request queue, and calls the request function to process the queue.

`blk_size`: This variable is an array of pointers to integers. It is indexed by the major device number. The subarray is indexed by the minor device number. Each cell of the subarray contains the size of the device in 1024 byte units. The subarray pointer can be NULL, in which case, the kernel does not check the size and range of an I/O request for the device.

`blksize_size`: This variable is an array of pointers to integers. It is indexed by the major device number. The subarray is indexed by the minor device number. Each cell of the subarray contains the block size of the device in bytes. The subarray can be NULL, in which case, the kernel uses the global definition BLOCK_SIZE (currently 1024 bytes) in its calculations.

`hardsect_size`: This variable is an array of pointers to integers. It is indexed by the major device number. The subarray is indexed by the minor device number. Each cell of the subarray contains the hardware sector size of the device in bytes. If the subarray is NULL, the kernel uses 512 bytes in its calculations.

`read_ahead`: This variable is an array of integers indexed by the major device number. It specifies how many sectors of read-ahead the kernel should perform on the device. The drivers only initialize the values in this array; the Linux kernel block buffer code is the actual user of these values.

`wait_for_request`: The Linux kernel uses a static array of I/O request structures. When all I/O request structures are in use, a process sleeps on this variable. When a driver finishes an I/O request and frees the I/O request structure, it performs a wake up on this variable.

`EISA_bus`: If this variable is non-zero, it indicates that the machine has an EISA bus. It is initialized bye the Linux kernel prior to device configuration.

`high_memory`: This variable contains the address of the last byte of physical memory plus one. It is initialized by the Linux kernel prior to device configuration.

`intr_count`: This variable gets incremented on entry to an interrupt handler, and decremented on exit. Its purpose is let driver code determine if it was called from an interrupt handler.

`kstat`: This variable contains Linux kernel statistics counters. Linux drivers increment various fields in it when certain events occur.

`tq_timer`: Linux has a notion of "bottom half" handlers. These handlers have a higher priority than any user level process but lower priority than hardware interrupts. They are analogous to software interrupts in BSD. Linux checks if any "bottom half" handlers need to be run when it is returning to user mode. Linux provides a number of lists of such handlers that are scheduled on the occurrence of specific events. `tq_timer` is one such list. On every clock interrupt, Linux checks if any handlers are on this list, and if there are, immediately schedules the handlers to run.

`timer_active`: This integer variable indicates which of the timers in `timer_table` (described below) are active. A bit is set if the timer is active, otherwise it is clear.

`timer_table`: This variable is an array of "struct timer_struct" elements. The array is index by global constants defined in ¡linux/timer.h¿. Each element contains the duration of timeout, and a pointer to a function that will be invoked when the timer expires.

`system_utsname`: This variable holds the Linux version number. Some drivers check the kernel version to account for feature differences between different kernel releases.

30.5.3 Functions

`autoirq_setup`: `int` **autoirq_setup**(`int` *waittime*);

This function is called by a driver to set up for probing IRQs. The function attaches a handler on each available IRQ, waits for *waittime* ticks, and returns a bit mask of IRQs available IRQs. The driver should then force the device to generate an interrupt.

`autoirq_report`: `int` **autoirq_report**(`int` *waittime*);

This function is called by a driver after it has programmed the device to generate an interrupt. The function waits *waittime* ticks, and returns the IRQ number on which the device interrupted. If no interrupt occurred, 0 is returned.

`register_blkdev`: `int` **register_blkdev**(`unsigned` *major*, `const` *char *name*, `struct` *file_operations *fops*);

This function registers a driver for the major number *major*. When an access is made to a device with the specified major number, the kernel accesses the driver through the operations vector *fops*. The function returns 0 on success, non-zero otherwise.

`unregister_blkdev`: `int` **unregister_blkdev**(`unsigned` *major*, `const` *char *name*);

This function removes the association between a driver and the major number *major*, previously established by `register_blkdev`. The function returns 0 on success, non-zero otherwise.

`getblk`: `struct buffer_head *`**getblk**(`kdev_t` *dev*, `int` *block*, `int` *size*);

This function is called by a driver to allocate a buffer *size* bytes in length and associate it with device *dev*, and block number *block*.

`brelse`: `void` **brelse**(`struct` *buffer_head *bh*);

This function frees the buffer *bh*, previously allocated by `getblk`.

`bread`: `struct buffer_head *`**bread**(`kdev_t` *dev*, `int` *block*, `int` *size*);

This function allocates a buffer *size* bytes in length, and fills it with data from device *dev*, starting at block number *block*.

`block_write`: `int` **block_write**(`struct` *inode *inode*, `struct` *file *file*, `const` *char *buf*, `int` *count*);

This function is the default implementation of file write. It is used by most of the Linux block drivers. The function writes *count* bytes of data to the device specified by *i_rdev* field of *inode*, starting at byte offset specified by *f_pos* of *file*, from the buffer *buf*. The function returns 0 for success, non-zero otherwise.

`block_read`: `int` **block_read**(`struct` *inode *inode*, `struct` *file *file*, `const` *char *buf*, `int` *count*);

This function is the default implementation of file read. It is used by most of the Linux block drivers. The function reads *count* bytes of data from the device specified by *i_rdev* field of *inode*, starting at byte offset specified by *f_pos* field of *file*, into the buffer *buf*. The function returns 0 for success, non-zero otherwise.

`check_disk_change`: `int` **check_disk_change**(`kdev_t` *dev*);

This function checks if media has been removed or changed in a removable medium device specified by *dev*. It does so by invoking the *check_media_change* function in the driver's file operations vector. If a change has occurred, it calls the driver's *revalidate* function to validate the new media. The function returns 0 if no medium change has occurred, non-zero otherwise.

`request_dma`: `int` **request_dma**(`unsigned` *drq*, `const` *char *name*);

This function allocates the DMA request line *drq* for the calling driver. It returns 0 on success, non-zero otherwise.

`free_dma`: `void` **free_dma**(`unsigned` *drq*);

This function frees the DMA request line *drq* previously allocated by `request_dma`.

`disable_irq`: `void` **disable_irq**(`unsigned` *irq*);

This function masks the interrupt request line *irq* at the interrupt controller.

`enable_irq`: `void` **enable_irq**(`unsigned` *irq*);

This function unmasks the interrupt request line *irq* at the interrupt controller.

`request_irq`: `int` **request_irq**(`unsigned int` *irq*, `void` *(*handler*)(int, struct), unsigned long flags, const char *device);

This function allocates the interrupt request line *irq*, and attach the interrupt handler *handler* to it. It returns 0 on success, non-zero otherwise.

`free_irq`: `void` **free_irq**(`unsigned` *int irq*);

This function frees the interrupt request line *irq*, previously allocated by `request_irq`.

`kmalloc`: `void` ***kmalloc**(`unsigned int` *size*, `int` *priority*);

This function allocates *size* bytes memory. The *priority* argument is a set of bitfields defined as follows:

`GFP_BUFFER`: Not used by the drivers.

`GFP_ATOMIC`: Caller cannot sleep.

`GFP_USER`: Not used by the drivers.

`GFP_KERNEL`: Memory must be physically contiguous.

`GFP_NOBUFFER`: Not used by the drivers.

`GFP_NFS`: Not used by the drivers.

`GFP_DMA`: Memory must be usable by the DMA controller. This means, on the x86, it must be below 16 MB, and it must not cross a 64K boundary. This flag implies GFP_KERNEL.

`kfree`: `void` **kfree**(`void` **p*);

This function frees the memory *p* previously allocated by `kmalloc`.

`vmalloc`: `void` ***vmalloc**(`unsigned` *long size*);

This function allocates *size* bytes of memory in kernel virtual space that need not have underlying contiguous physical memory.

`check_region`: `int` **check_region**(`unsigned` *port*, `unsigned` *size*);

Check if the I/O address space region starting at *port* and *size* bytes in length, is available for use. Returns 0 if region is free, non-zero otherwise.

`request_region`: `void` **request_region**(`unsigned` *port*, `unsigned` *size*, `const` *char *name*);

Allocate the I/O address space region starting at *port* and *size* bytes in length. It is the caller's responsibility to make sure the region is free by calling `check_region`, prior to calling this routine.

`release_region`: `void` **release_region**(`unsigned` *port*, `unsigned` *size*);

Free the I/O address space region starting at *port* and *size* bytes in length, previously allocated by `request_region`.

`add_wait_queue`: `void` **add_wait_queue**(`struct` *wait_queue **q*, `stuct` *wait_queue *wait*);

Add the wait element *wait* to the wait queue *q*.

`remove_wait_queue`: `void` **remove_wait_queue**(`struct` *wait_queue **q*, `struct` *wait_queue *wait*);

Remove the wait element *wait* from the wait queue *q*.

`down`: `void` **down**(`struct` *semaphore *sem*);

Perform a down operation on the semaphore *sem*. The caller blocks if the value of the semaphore is less than or equal to 0.

`sleep_on`: `void` **sleep_on**(`struct` *wait_queue **q*, `int` *interruptible*);

Add the caller to the wait queue *q*, and block it. If *interruptible* flag is non-zero, the caller can be woken up from its sleep by a signal.

`wake_up`: `void` **wake_up**(`struct` *wait_queue **q*);

Wake up anyone waiting on the wait queue *q*.

`wait_on_buffer`: `void` **wait_on_buffer**(`struct` *buffer_head *bh*);

Put the caller to sleep, waiting on the buffer *bh*. Called by drivers to wait for I/O completion on the buffer.

`schedule`: `void` **schedule**(`void`);

Call the scheduler to pick the next task to run.

`add_timer`: `void` **add_timer**(`struct` *timer_list *timer*);

Schedule a time out. The length of the time out and function to be called on timer expiry are specified in *timer*.

`del_timer`: `int` **del_timer**(`struct` *timer_list *timer*);

Cancel the time out *timer*.

30.5.4 Directory Structure

The `linux` subdirectory in the OSKit source tree is organized as follows. The top-level `linux/dev` directory contains all the glue code implemented by the Flux project to squash the Linux drivers into the OSKit driver framework. `linux/fs` contains our glue for Linux filesystems, and `linux/shared` contains glue used by both components. In general, everything *except* the code in the `linux/src` directory was written by us, whereas everything under `linux/src` comes verbatim from Linux. Each of the subdirectories of `linux/src` corresponds to the identically named subdirectories of in the Linux kernel source tree.

Of course, there are a few necessary deviations from this rule: a few of the Linux header and source files are slightly modified, and a few of the Linux header files (but no source files) were completely replaced. The header files that were heavily modified include:

`linux/src/include/linux/sched.h`: Linux task and scheduling declarations

30.6 Block device drivers

30.7 Network drivers

Things drivers may want to do that make emulation difficult:

- Call the 16-bit BIOS.
- Use the system DMA controller.

Name	Description	$V = P$	`jiffies`	`P+Y`	`current`	
cmd640.c	CMD640 IDE Chipset					
floppy	Floppy drive	*	*	*	*	
ide-cd.c	IDE CDROM		*	*	*	
ide.c	IDE Disk					
rz1000.c	RZ1000 IDE Chipset					
sd.c	SCSI disk		*			
sr.c	SCSI CDROM					
triton.c	Triton IDE Chipset	*				

Table 30.1: Linux block device drivers

- Assume kernel virtual addresses are equivalent to physical addresses.
- Assume kernel virtual addresses can be mapped to physical addresses merely by adding a constant offset.
- Implement timeouts by busy-waiting on a global clock-tick counter.
- Busy-wait for interrupts. XXX This means that the OS *must* allow interrupts during execution of process-level driver code, and not just when all process-level activity is blocked.

30.8 SCSI drivers

The Linux SCSI driver set includes both the low-level SCSI host adapter drivers and the high-level SCSI drivers for generic SCSI disks, tapes, etc.

Name	Description	$V = P$	`jiffies`	`P+Y`	`current`	
3c501.c	3Com 3c501 ethernet		*			
3c503.c	NS8390 ethernet		*			
3c505.c	3Com Etherlink Plus (3C505)		*			
3c507.c	3Com EtherLink16		*			
3c509.c	3c509 EtherLink3 ethernet		*			
3c59x.c	3Com 3c590/3c595 "Vortex"		*			
ac3200.c	Ansel Comm. EISA ethernet		*			
apricot.c	Apricot	*	*			
at1700.c	Allied Telesis AT1700		*			
atp.c	Attached (pocket) ethernet		*			
de4x5.c	DEC DE425/434/435/500		*			
de600.c	D-link DE-600		*			
de620.c	D-link DE-620		*			
depca.c	DEC DEPCA & EtherWORKS		*			
e2100.c	Cabletron E2100		*			
eepro.c	Intel EtherExpress Pro/10		*			
eexpress.c	Intel EtherExpress		*			
eth16i.c	ICL EtherTeak 16i & 32		*			
ewrk3.c	DEC EtherWORKS 3		*			
hp-plus.c	HP PCLAN/plus		*			
hp.c	HP LAN		*			
hp100.c	HP10/100VG ANY LAN		*			
lance.c	AMD LANCE	*	*			
ne.c	Novell NE2000		*			
ni52.c	NI5210 (i82586 chip)		*			
ni65.c	NI6510 (am7990 'lance' chip)	*	*			
seeq8005.c	SEEQ 8005		*			
sk_g16.c	Schneider & Koch G16		*			
smc-ultra.c	SMC Ultra		*			
tulip.c	DEC 21040	*	*			
wavelan.c	AT&T GIS (NCR) WaveLAN		*			
wd.c	Western Digital WD80x3		*			
znet.c	Zenith Z-Note		*			

Table 30.2: Linux network drivers

Name	Description	$V = P$	`jiffies`	`P+Y`	`current`	
53c7,8xx.c	NCR 53C7x0, 53C8x0	*	*			
AM53C974.c	AM53/79C974 (PCscsi)	*				
BusLogic.c	BusLogic MultiMaster adapters	*	*			
NCR53c406a.c	NCR53c406a	*	*			
advansys.c	AdvanSys SCSI Adapters	*	*			
aha152x.c	Adaptec AHA-152x		*			
aha1542.c	Adaptec AHA-1542	*	*			
aha1740.c	Adaptec AHA-1740	*				
aic7xxx.c	Adaptec AIC7xxx	*	*			
eata.c	EATA 2.x DMA host adapters		*			
eata_dma.c	EATA/DMA host adapters	*	*			
eata_pio.c	EATA/PIO host adapters		*			
fdomain.c	Future Domain TMC-16x0		*			
in2000.c	Always IN 2000		*			
NCR5380.c	Generic NCR5380	*	*	*		
pas16.c	Pro Audio Spectrum/Studio 16					
qlogic.c	Qlogic FAS408		*			
scsi.c	SCSI middle layer	*	*	*	*	
scsi_debug.c	SCSI debug layer		*			
seagate.c	ST01,ST02, TMC-885		*			
t128.c	Trantor T128/128F/228					
u14-34f.c	UltraStor 14F/34F	*	*			
ultrastor.c	UltraStor 14F/24F/34F	*				
wd7000.c	WD-7000	*	*			

Table 30.3: Linux SCSI drivers

Chapter 31

FreeBSD Driver Set: `liboskit_freebsd_dev.a`

31.1 Introduction

This chapter is woefully incomplete. The OSKit FreeBSD device library provides an infrastructure for using unmodified FreeBSD 2.1.7 device drivers.

31.2 Supported Devices

The FreeBSD device library currently supports only the system console and a few ISA-based serial port interfaces all exporting the `oskit_ttystream` interface. Only the system console and PS/2 mouse have been tested.

Following is a list of supported drivers. The tag is the name used by the device library to refer to the devices (see Section 12.3 for details on device naming).

sc	PC system console.
sio	PC serial port.
cx	ISA bus Cronyx-Sigma serial port adapter.
cy	ISA bus Cyclades Cyclom-Y serial board.
rc	ISA bus RISCom/8 serial board.
si	ISA bus Specialix serial line multiplexor.
mse	Bus mouse.
psm	PS/2 mouse.

31.3 Header Files

31.3.1 `freebsd.h`: common device driver framework definitions

Synopsis

```
#include <oskit/dev/freebsd.h>
```

Description

Contains common definitions and function prototypes for the OSKit's FreeBSD device interfaces described in the next section.

31.4 Interfaces

31.4.1 `oskit_freebsd_init`: Initialize and FreeBSD device driver support package

Synopsis

`#include <oskit/dev/freebsd.h>`

`void` **oskit_freebsd_init**(`void`);

Direction

OS → Component

Description

Initializes the support code for FreeBSD device drivers.

Currently the `oskit_freebsd_init_`*`driver`* routines take care of invoking any required freebsd device initialization functions, including this one. This may change in the future.

31.4.2 `oskit_freebsd_init_devs`: Initialize and register all FreeBSD device drivers

Synopsis

`#include <oskit/dev/freebsd.h>`

`void` **oskit_freebsd_init_devs**(`void`);

Direction

OS → Component

Description

Initialize and register all available FreeBSD device drivers.

Warning messages will be printed with `osenv_log` for drivers which cannot be initialized but the initialization process will continue.

Related Information

`osenv_log`

31.4.3 `oskit_freebsd_init_isa`: Initialize and register all FreeBSD ISA bus device drivers

Synopsis

`#include <oskit/dev/freebsd.h>`

`void` **oskit_freebsd_init_isa**(`void`);

Direction

OS → Component

DESCRIPTION

Initialize and register all available FreeBSD ISA bus device drivers. See `<oskit/dev/freebsd_isa.h>` for the currently available devices.

Warning messages will be printed with `osenv_log` for drivers which cannot be initialized but the initialization process will continue.

Currently the `oskit_freebsd_init_`*`driver`* routines take care of invoking any required freebsd device initialization functions, including this one. This may change in the future.

RELATED INFORMATION

`osenv_log`

31.4.4 `oskit_freebsd_init_`*`driver`*: Initialize and register a single FreeBSD device driver

SYNOPSIS

`#include <oskit/dev/freebsd.h>`

`oskit_error_t` **oskit_freebsd_init_*driver***(`void`);

DIRECTION

OS $\rightarrow$ Component

DESCRIPTION

Initialize a single FreeBSD device driver. Possible values for *driver* are listed in Section 31.2.

RETURNS

Returns 0 on success, an error code specified in `<oskit/dev/error.h>` on error.

31.5 "Back door" Interfaces

"Back door" interfaces are intended for users which have some builtin knowledge of FreeBSD internals and want to convert that knowledge to interface-level equivalents.

31.5.1 `oskit_freebsd_chardev_open`: Open a character device using a FreeBSD major/minor device value

SYNOPSIS

`#include <oskit/dev/freebsd.h>`

`oskit_error_t` **oskit_freebsd_chardev_open**(`int` *major*, `int` *minor*, `int` *flags*, [out] `oskit_ttystream_t **`*tty_stream*);

DIRECTION

OS → Component

DESCRIPTION

Opens a character device given a FreeBSD major and minor device value. Returns a pointer to an `oskit_ttystream_t` interface as though `oskit_ttydev_open` was called on an OSKit `oskit_ttydev_t` interface.

PARAMETERS

major: Major device number. In FreeBSD, this is the index of the device in the character device switch.

minor: Minor device number. In FreeBSD, the interpretation of the minor device number is device specific.

flags: POSIX `open` flags.

tty_stream: Returned `oskit_ttystream_t` interface.

RETURNS

Returns 0 on success, an error from `<oskit/dev/error.h>` otherwise.

31.5.2 `oskit_freebsd_xlate_errno`: Translate a FreeBSD error number

SYNOPSIS

`#include <oskit/dev/freebsd.h>`

`oskit_error_t` **oskit_freebsd_xlate_errno**(`int` *freebsd_error*);

DIRECTION

OS → Component

DESCRIPTION

Translates a FreeBSD error number into an OSKit error number.

PARAMETERS

freebsd_error: The FreeBSD error code to be translated.

Returns

Returns an equivalent error value from `<oskit/dev/error.h>`, or `OSKIT_E_UNEXPECTED` if there is no suitable translation.

Chapter 32

WIMPi Window Manager: `liboskit_wimpi.a`

32.1 Introduction

WIMP is a hierarchical windowing system based on Bellcore's MGR. The University of Arizona's Scout project produced WIMP from MGR; the "WIMPi" OSKit component is a modified version of Arizona's work.

32.2 So how do I use this?

Due to the convoluted path the code has gone through, the programming interfaces for MGR, WIMP and WIMPi bear only a superficial resemblance to each other at this point. Many of the functions directly correspond to their counterparts in the Scout winMgr interface, however. For more information on those, see in the OSKit source tree `wimp/scoutdoc/{winmgr.h,wimp.tex}` from the Scout distribution.

To use wimpi, you must first initialize it by calling `wimpi_initialize`. Once you have initialized all the windows for your application and are ready to enter the event loop, you will then need to call `wimpi_main_loop` with the `wimpiSession` variable you got from calling `wimpi_initialize`.

You will also need to have handlers to send input data to wimp, and to handle wimp events when they happen. These functions are set by calling the `wimp_set_input_handler` and `wimp_set_event_handler` functions.

You can also look at `examples/x86/wimpirun.c` for more information.

32.3 Dependencies

WIMPi currently depends on the video_svgalib library; see its "Dependencies" section for more information.

32.4 API reference

32.4.1 `wimpi_initialize`: Initialize the wimpi code

Synopsis

```
#include <oskit/wimpi.h>
```

`wimpiSession` **wimpi_initialize**(`void`);

32.4.2 `wimpi_main_loop`: Start main wimpi event loop

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_main_loop**(`wimpiSession` *session*);

32.4.3 `wimpi_create_toplevel`: Create a top level wimpi window

Synopsis

`#include <oskit/wimpi.h>`

`wimpiToplevelWindow` **wimpi_create_toplevel**(`wimpiSession` *session*, `bool` *multi*, `int` *width*, `int` *height*, `int` *x*, `int` *y*, `char*` *title*, `bool` *mapped*, `void*` *data*);

32.4.4 `wimpi_destroy_toplevel`: Destroy a top level wimpi window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_destroy_toplevel**(`wimpiToplevelWindow` *w*);

32.4.5 `wimpi_kbd_input`: Send keyboard input to wimpi

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_kbd_input**(`wimpiSession` *session*, `unsigned` *char c*);

32.4.6 `wimpi_mouse_input`: Send mouse input to wimpi

Synopsis

`#include <oskit/wimpi.h>`

`long` **wimpi_mouse_input**(`wimpiSession` *s*, `char` *but_state*, `int` *dx,int dy*);

32.4.7 `wimpi_set_event_handler`: Register a callback function for event handling

Synopsis

`#include <oskit/wimpi.h>`

`wimpiEventHandler` **wimpi_set_event_handler**(`wimpiSession` *session*, `wimpiEventHandler` *proc*);

32.4.8 `wimpi_set_input_routine`: Register a callback function for input to wimpi

Synopsis

`#include <oskit/wimpi.h>`

`wimpiInputRoutine` **wimpi_set_input_routine**(`wimpiSession` *session*, `wimpiInputRoutine` *proc*);

32.4.9 `wimpi_send_expose_event`: Send and expose event to a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_send_expose_event** (`WINDOW` **win*);

32.4.10 `wimpi_send_mouse_event`: Send a mouse event to a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_send_mouse_event** (`WINDOW` **win*, `wimpiEventType` *type*, `int` *button*);

32.4.11 `wimpi_send_move_resize_event`: Send a move/resize event to a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_send_move_resize_event** (`WINDOW` **win*);

32.4.12 `wimpi_send_destroy_event`: Send a destroy event to a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_send_destroy_event** (`WINDOW` **win*);

The following functions have direct counterparts in the scout winMgr interface `#include <scout/winmgr.h>`:

32.4.13 `wimpi_create_window`: Create a sub window

Synopsis

`#include <oskit/wimpi.h>`

`wimpiWindow` **wimpi_create_window** (`wimpiWindow` *parent*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*, `int` *color*, `int` *has_border*);

32.4.14 `wimpi_destroy_window`: Destroy a sub window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_destroy_window** (`wimpiWindow` *w*);

32.4.15 `wimpi_map_window`: Map a sub window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_map_window** (`wimpiWindow` *w*);

Description

Make a window visible on the screen.

32.4.16 `wimpi_unmap_window`: Unmap a sub window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_unmap_window** (`wimpiWindow` *w*);

Description

Remove a window from the screen.

32.4.17 `wimpi_raise_window`: Raise a sub window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_raise_window** (`wimpiWindow` *w*);

32.4.18 `wimpi_lower_window`: Lower a sub window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_lower_window** (`wimpiWindow` *w*);

32.4.19 `wimpi_set_window_background`: Set a window's background color

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_set_window_background** (`wimpiWindow` *w*, `int` *color*);

32.4.20 `wimpi_set_foreground`: Set a window's foreground color

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_set_foreground** (`wimpiWindow` *w*, `int` *color*);

32.4.21 `wimpi_clear_area`: Clear a section of a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_clear_area** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*, `bool` *exposures*);

32.4.22 `wimpi_move_resize_window`: Move/resize a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_move_resize_window** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*);

32.4.23 `wimpi_fill_rectangle`: Draw a filled rectangular area in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_fill_rectangle** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*);

32.4.24 `wimpi_draw_string`: Draw a string in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_draw_string** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `char` **string*, `int` *length*);

32.4.25 `wimpi_draw_line`: Draw a line in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_draw_line** (`wimpiWindow` *w*, `int` *x0*, `int` *y0*, `int` *x1*, `int` *y1*);

32.4.26 `wimpi_draw_arc`: Draw an arc in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_draw_arc** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*, `int` *angle1*, `int` *angle2*);

32.4.27 `wimpi_draw_ellipse`: Draw an ellipse in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_draw_ellipse** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*);

32.4.28 `wimpi_draw_rectangle`: Draw a rectangle in a window

Synopsis

`#include <oskit/wimpi.h>`

`void` **wimpi_draw_rectangle** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `int` *wide*, `int` *high*);

32.4.29 `wimpi_put_image`: Blit an image into a window

SYNOPSIS

`#include <oskit/wimpi.h>`

`void` **wimpi_put_image** (`wimpiWindow` *w*, `int` *x*, `int` *y*, `void` **data*, `int` *wide*, `int` *high*, `unsigned` *depth*);

32.4.30 `wimpi_copy_area`: Copy a rectangular region in a window

SYNOPSIS

`#include <oskit/wimpi.h>`

`void` **wimpi_copy_area** (`wimpiWindow` *w*, `int` *src_x*, `int` *src_y*, `int` *width*, `int` *height*, `int` *dst_x*, `int` *dst_y*);

32.4.31 `wimpi_set_window_title`: Set the title of a window

SYNOPSIS

`#include <oskit/wimpi.h>`

`void` **wimpi_set_window_title** (`wimpiWindow` *w*, `char` **title*);

32.4.32 `wimpi_make_child_window`: Make a child window

SYNOPSIS

`#include <oskit/wimpi.h>`

`void` **wimpi_make_child_window** (`wimpiToplevelWindow` *w*, `Window` *parent*, `wimpiToplev elWindow` *child*);

Chapter 33

Video Support: `liboskit_video.a`

33.1 Introduction

Video support is currently a three-way mishmash and it's incomplete, but it provides useful services for some popular boards.

The first way provides the XFree86 S3 driver and its normal XFree86 interface, along with a few other functions.

The second way provides the VGA portion of the well known "Svgalib" library for Linux.

The third way for OSKit kernels to access video (and the keyboard and mouse) is via X11 client support. Through this support OSKit kernels can interact (like any other X client) with remote machines running an X server. In the `x11/client` source directory we provide patches against both the stock XFree86 X11R6.3 and XConsortium X11R6.3 distributions, along with helper functions specific to the OSKit. See the README's there.

33.2 X11 S3 Video Library

The x11video library contains a video driver supporting S3 video cards. This includes all video hardware supported by the XF86_S3 server (see http://www.xfree86.org for more information).

33.2.1 So how do I use this?

The driver currently requires a configuration (XF86Config) file. A sample file is included in the x11/video directory. You'll need to make sure that it's readable as /etc/XF86Config. This can either be done through the BMOD (section 10.20) filesystem, or by using the FreeBSD or Linux filesystem components.

This file is a subset of a standard XF86Config file and can only contain the Monitor, Device and Screen sections. The easiest way to get things running is when you already have an XF86Config file for that computer/monitor configuration. In that case you can just strip out everything from it but the sections listed above. Otherwise you'll need to make one from scratch, or from a sample config file from an XFree86 distribution.

Currently the x11video driver only uses the default (first) entry in the Screen section. Unlike XFree86, there is no way to change or specify color depths or resolution at runtime.

33.2.2 Dependencies

x11video depends on `oskit_startup`, `oskit_unsupp`, `oskit_dev`, `oskit_kern`, `oskit_freebsd_m`, `oskit_c`, and `oskit_lmm`.

33.2.3 API reference

33.2.4 `s3_init_framebuffer`: Initializes the s3 video code

Synopsis

```
#include <oskit/video/s3.h>
oskit_fb_t *s3_init_framebuffer(void);
```

33.2.5 `s3_cmap_write`: Write a colormap entry

Synopsis

```
#include <oskit/video/s3.h>
int s3_cmap_write(oskit_cmap_entry_t *c);
```

33.2.6 `s3_cmap_read`: Read a colormap entry

Synopsis

```
#include <oskit/video/s3.h>
int s3_cmap_read(oskit_cmap_entry_t *c);
```

33.2.7 `s3_cmap_fg_index`: Return the colormap index for the foreground color

Synopsis

```
#include <oskit/video/s3.h>
int s3_cmap_fg_index(void);
```

33.3 Svgalib Video Library

The video_svgalib library contains the VGA portion of the well known "svgalib" library. It does not support the keyboard, joystick, mouse or gl functionality of svgalib.

This library and its interface may change or disappear in the future, but it's useful for now.

33.3.1 So how do I use this?

The interface is the same as the VGA interface for Svgalib. The extensive man pages for those functions are in `oskit/video/svgalib/doc`, handily formatted for you. Svgalib programs that use only the VGA interface should work with little change: you should only have to add code to do OSKit initialization.

For an example of how to convert an svgalib program, you can look at `examples/x86/svgalib.c`, which is one of the standard svgalib example programs converted to run on the oskit.

33.3.2 Dependencies

video_svgalib depends on `oskit_startup`, `oskit_dev`, `oskit_kern`, `oskit_freebsd_m`, `oskit_c`, and `oskit_lmm`.

Part V

Utilities

Chapter 34

Network Booting

34.1 Introduction

NetBoot is a small MultiBoot-compliant operating system, and example of OSKit use, that provides one service: fast booting of other MultiBoot-compliant operating systems across the network while itself remaining resident in order to regain control after the target OS exits. This avoids going back to the BIOS to boot the next kernel and allows a reboot cycle that is often an order of magnitude faster than normal.

When NetBoot is booted it prompts for the name of an OS to fetch and boot. The booted OS is passed a special command line flag indicating a return address that it can use to return control back to NetBoot upon exiting. Therefore NetBoot could be thought of as a crude batch-processing operating system.

NetBoot is intended to be used as a kernel development tool, not as a mechanism for implementing diskless workstations, although it can conceivably be used either way.

Note: the netboot described here is not to be confused with the FreeBSD netboot EPROM code of the same name. They share some code but perform different functions.

34.2 Implementation Issues and Requirements

NetBoot is built as a MultiBoot-compliant operating system; therefore to boot it with LILO or the BSD boot program, an appropriate image must be made with `mklinuximage` or `mkbsdimage`, respectively (see Section 1.6.2).

NetBoot boots MultiBoot-compliant operating systems such as the example programs that come with the OSKit. If the desired OS to boot needs MultiBoot boot-modules, they and the OS can be combined into one MultiBoot image via the `mkmbimage` script included with the OSKit.

NetBoot requires a BOOTP server to be running on the local network in order to obtain the IP address, gateway address, netmask, and hostname of the host it runs on. If no BOOTP server responds when NetBoot is booted, it will ask to retry or exit.

The files that NetBoot fetches and boots must reside in a directory that is NFS exported to the host running NetBoot. In the future, NetBoot may support other protocols such as TFTP.

The OS that NetBoot will not know about all of memory. This is because NetBoot stashes itself and some other things at the top of memory and then lies to the booted OS about where the top is. This is to allow the booted OS to return control to NetBoot upon exit; avoiding the time-consuming process of rebooting the machine. There is currently no way to disable this feature.

34.3 Using NetBoot

When NetBoot is booted it will print something like the following:

```
NetBoot metakernel v2.4
```
. . . various startup output, driver probes, etc. . .

```
Type "help" for help.

NetBoot> _
```

At the `NetBoot>` prompt one can boot another OS, get help, or quit.

34.3.1 Booting Another OS

If NetBoot is given a pseudo URL-style name at the prompt it will fetch that file and boot it.
The format of the name is as follows:

hostname:*path* [*args*]

Where:

hostname is either an IP-address or a name of a host from which to get the OS. Currently the hostname lookup is fake and depends on hardwired names in the NetBoot code. The OSKit includes resolver code but that code depends on the OSKit BSD socket package, which NetBoot currently does not use.

path is the path to the desired OS. This directory must be NFS-exported to the machine running NetBoot or the fetch will fail.

args are optional command line arguments to pass to the booted kernel.

Two args, `-h` and `-f`, are handled as toggles. These args are checked for by the default OSKit console startup code and determine if the serial console will be used (`-h`) and if it runs at 115200 baud (`-f`). Therefore, if NetBoot was booted with either of these args it will pass them to the booted OS, assuming it wants to use the same console. However, the OS to fetch may be specified with either of these args and they will be removed from the default argument list.

Another arg is placed in the booted OS's argument string before booting. NetBoot passes a flag of the form "`-retaddr` *hex*" to the booted OS so it can return to this address if it wants to return control to NetBoot. This is normally done by the booted OS's `_exit` routine.

34.3.2 Getting Help

Typing "help" at the NetBoot prompt will give some basic usage help.

34.3.3 Quitting

Typing "quit" or "exit" at the NetBoot prompt will make NetBoot exit.

Part VI

The Legal Stuff

Chapter 35

Copyrights and Licenses

35.1 Copyrights & Licenses

The OSKit is free software, also known as "open source" software. The majority of the OSKit is covered by the standard GNU Public License (GPL), found in the file "COPYING," which basically allows free use, modification, and redistribution, as long as the source to the OSKit and any code linked against it is made freely available. If this appears too constraining, alternate licensing terms for much of the OSKit may be explored by contacting csl-dist@cs.utah.edu or calling +1-801-585-3271. Reproduction of the documentation is limited to non-commercial uses, including academic, research, evaluation, and personal use. For alternate terms, contacts are the same as above.

Because much of the code in the OSKit was "donated" by external projects there are a plethora of additional copyrights and licences, but their requirements are straightforward. Some parts of the OSKit are additionally covered by BSD/Mach/X11-style licenses which require acknowledgement and sometimes a notice in any advertising. Finally, the separate Wimpi code is covered by the Bellcore MGR license, which restricts commercial use.

The individual files describe the copyright and licensing restrictions. In all cases the BSD/Mach/X11-style restrictions are no more than inclusion of a name or notice in the accompanying documentation of the program that uses the code. For example, when linking in code from the FreeBSD project that is copyrighted by the Regents of the University of California, accompanying documentation must include the copyright and license notice, while advertising materials using the library must include the acknowledgement "This product includes software developed by the University of California, Berkeley and its contributors." Other BSD-style license have similar requirements, but for a different copyright holder.

To help clarify matters, the Acknowledgements section, below, describes which parts of the OSKit are covered by which license(s).

35.2 Contributors

While much of the code in the OSKit was developed outside the University of Utah, by large free operating systems projects—most notable are the FreeBSD, Linux, Mach and NetBSD projects—the OSKit proper was developed by members of the University of Utah's Flux Research Group, including (in alphabetical order): Chris Alfeld, Dave Andersen, Godmar Back, Greg Benson (U.C. Davis on location at Utah), Steve Clawson, Bryan Ford, Shantanu Goel, Mike Hibler, Jay Lepreau, Roland McGrath, Bart Robinson, Steve Smalley (on location at Utah), Leigh Stoller, Sai Susarla, Patrick Tullmann, and Kevin Van Maren. While many of these made huge contributions, Bryan Ford stands out for conceiving the OSKit and doing most of the early development and architecture. The majority of the OSKit work was done under DARPA support, which we gratefully acknowledge. Finally, the GNU build tool chain was essential, as usual.

35.3 Acknowledgements

Libraries that contain source code licensed under some variation of the BSD-style license generally contain the restriction that the copyright holders of that code be acknowledged in documentation accompanying the program. Because not all users of the OSKit need to acknowledge all of the multitudinous authors of the available code in the OSKit, we have broken down the acknowledgement requirements on a per-library basis. The remainder of this chapter lists each library and the license requirements of linking in that library. Note that some licenses only show up in files that you may not need or use in that library.

We have done our best to make this accurate and to include notices as required, but have probably missed things. If so, let us know. The source code is the final authority.

35.3.1 liboskit_diskpart.a

35.3.2 liboskit_exec.a

35.3.3 liboskit_freebsd_{dev,net,m}.a

....

....

....

35.3.4 liboskit_kern.a

....

....

35.3.5 liboskit_libc.a

35.3.6 liboskit_wimp.a

35.3.7 Various OSKit header files

`oskit/page.h`,
`oskit/queue.h`,
`oskit/diskpart/dec.h`,
`oskit/diskpart/vtoc.h`,
`oskit/diskpart/pcbios.h`,
`oskit/diskpart/omron.h`,
`oskit/x86/asm.h`,
`oskit/x86/base_trap.h`,
`oskit/x86/eflags.h`,
`oskit/x86/fp_reg.h`,
`oskit/x86/paging.h`,
`oskit/x86/pio.h`,
`oskit/x86/proc_reg.h`,
`oskit/x86/seg.h`,
`oskit/x86/spin_lock.h`,
`oskit/x86/trap.h`,
`oskit/x86/tss.h`,
`oskit/x86/c/setjmp.h`,
`oskit/x86/c/stdarg.h`,
`oskit/x86/pc/keyboard.h`,
`oskit/x86/pc/pic.h`:

....

oskit/c/netdb.h,
oskit/c/resolv.h,
oskit/c/signal.h,
oskit/c/sys/mount.h,
oskit/c/netinet/in.h,
oskit/c/arpa/nameser.h,
oskit/c/arpa/inet.h,
oskit/diskpart/disklabel.h,
oskit/x86/c/float.h,
oskit/x86/c/limits.h,
oskit/x86/pc/isa.h,
oskit/x86/pc/pit.h,
oskit/x86/pc/rtc.h:

oskit/exec/elf.h:

35.3.8 liboskit_netbsd_fs.a

Index