BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M7

An Operating System Development Environment

by
Robert Boyer

Technical Report CS-91-28

An Operating System Development Environment

Rob Boyer
Brown University
Advisor: Tom Doeppner

April 29, 1991

Contents

1 Introduction and project goals 1

1.1 Structure of this document 1

1.2 Terminology 2

2 Machine model 3

2.1 Instruction set 3

2.2 Execution modes and stacks 3

2.3 Machine registers 4

2.4 Devices and interrupts 4

2.5 The system control block (SCB) 5

2.6 Memory Management 5

2.6.1 Real versus Virtual mode 6

2.6.2 Physical memory, pages, and page frames 6

2.6.3 Memory management registers 6

2.6.4 Page protection 7

2.6.5 Additional paging support 7

2.7 Machine support for contexts 7

3 Low-level components of the prototype operating system 8

3.1 POS initialization routine 8

3.2 Interrupt service routines, trap, and exception handlers 8

3.3 Data structures 9

3.4 Process contexts 9

4 A UNIX-based implementation 10

4.1 Design overview 10

4.1.1 Device simulation and interrupts 10

4.1.2 Traps and exceptions 11

4.1.3 Context switching 11

4.2 Current status and limitations 12

5 An OSF/1-based implementation 13

5.1 Task structure . . . 13

5.1.1 The Sim task . . . 14

5.1.2 Process-context task(s) . . . 15

5.1.3 Intertask communication . . . 15

5.2 Virtual memory implementation . . . 17

5.2.1 The pmap module . . . 17

5.2.2 Address space layout . . . 17

5.2.3 Enabling virtual memory . . . 19

5.2.4 Use of the Mach external pager feature . . . 19

5.2.5 Non-resident page access . . . 20

5.2.6 Responding to page faults . . . 20

5.2.7 Altering and flushing page mapping tables . . . 22

5.2.8 "Referenced" and "Modified" flags . . . 23

5.2.9 User-mode code and data . . . 24

5.2.10 Switching between user and supervisor modes . . . 24

5.3 Simulating device I/O . . . 25

5.3.1 DMA support . . . 25

5.4 Traps, exceptions, and interrupts . . . 26

5.4.1 Manipulating the POS thread's state . . . 26

5.4.2 Trap handling . . . 26

5.4.3 Exception handling . . . 27

5.4.4 Delivering an Interrupt . . . 27

5.4.5 `SIMrei()` and `SIMsetipl()` . . . 27

5.5 Support for multiple process contexts . . . 28

5.5.1 Context creation . . . 28

5.5.2 Context switching . . . 29

5.6 Task and thread synchronization . . . 29

5.7 Startup and boot sequence . . . 29

5.8 Current status and limitations . . . 31

6 Conclusion **32**

6.1 Possibilities for further research 32

A Summary of the simulator interface **33**

A.1 Accessing privileged processor registers 33

A.1.1 Retrieving the contents of the Processor Status Register 33

A.1.2 `SIMsetipl` 33

A.1.3 `SIMsetscb` 33

A.1.4 PIR – pending interrupt register 33

A.1.5 MMRs – memory management registers 34

A.2 Virtual memory interface 34

A.2.1 Enabling virtual memory 34

A.2.2 Flushing page tables 34

A.3 Device interface 34

A.3.1 Terminal device 35

A.3.2 Disk device 37

A.3.3 Clock device 38

A.4 Traps, exceptions, and interrupts 38

A.4.1 Traps 38

A.4.2 Exceptions 39

A.4.3 Interrupts and interrupt handlers 39

A.4.4 The `SIMrei` instruction 40

A.4.5 Context manipulation 40

A.4.6 Context initialization 41

A.4.7 Process context switching 41

A.4.8 Altering the user-mode context 41

B OSF/1 implementation notes **43**

B.1 OSF/1 implementation: default memory management architecture 43

B.2 OSF/1 implementation: location and status of source code 43

B.3 OSF/1 implementation: module layout 46

C References 47

List of Figures

1 System Control Block 5
2 OSF/1-based simulator architecture 14
3 Simulator task hierarchy 16
4 Code that runs within a process-context task 16
5 Simulator's address space layout 17
6 Two process contexts sharing a page 19
7 Page fault processing 21
8 Flush Operation 22
9 Virtual address format – default pmap 43
10 Virtual memory data structures – default pmap 44

1 Introduction and project goals

The *Operating System Development Environment* project provides libraries and run-time code that support the development of prototype operating systems. The primary goals of the project are:

1. to simulate a realistic machine architecture entirely in software, without being intrusive or causing unwanted side-effects in the operating system being developed.
2. to allow operating system code to be written entirely in a high-level language; initially "C" will be supported.
3. that the operating system and simulator should run native mode code generated by the host system's compilers. While running, the operating system should appear to the host system's kernel and development tools as a normal, user-mode application.

By building a machine simulator in this way, all existing development, debugging, and profiling tools supplied with the host system can be used to create, debug, and analyze the prototype operating system. No modifications are required to the host system's operating system, and performance is substantially improved as compared to an interpretive approach.

In order to simulate the actions of architecture-specific trap and interrupt handling, a machine-independent, C-callable interface is exported to the operating system writer. The simulator interface emulates the behavior of traps and interrupts in terms of the resulting flow of control between the different pieces of operating system code. For example, a trap function is provided that causes control to immediately transfer to the operating system's trap handler. Before making this control transfer, the simulator switches to a privileged mode and supervisor stack, emulating the way many hardware architectures respond to a trap instruction [6].

Other simulator functions are provided to manipulate privileged simulator registers, access simulated peripheral devices, and manage multiple process contexts. The simulator is structured so that the asynchronous, event-driven environment created by a typical multiple-device hardware system is emulated as closely as possible. Writing code to run in the simulator involves dealing with issues that must be faced when writing an operating system for such environments — such as handling multiple device priority levels, creating and switching between process contexts, handling hardware exceptions, and dealing directly with device I/O registers for both DMA and non-DMA type devices.

1.1 Structure of this document

In the following section, the general machine model presented by the simulator is discussed. Then, the low-level interface that exists between the simulated machine and the *Prototype Operating System* that logically runs within the simulated machine is presented. The design for a UNIX-based[1] implementation of the simulator is then outlined, followed by the design for an OSF/1-based[2] implementation. The OSF/1-based design avoids many of the limitations found in the UNIX-based design, and provides additional functionality mostly related to virtual memory support. In each

[1] UNIX is a registered trademark of UNIX Software Laboratories in the United States and other countries.
[2] OSF stands for the Open Software Foundation.

of the two design sections, the current status, limitations, and possible improvements for each implementation are also discussed. Finally, a summary of the project is provided, followed by appendices that contain the actual simulator programmable interface, and notes related to existing implementations.

1.2 Terminology

In order to avoid confusion, the operating system that is being developed to run in the simulator is termed the *prototype operating system*, or POS. The operating system that is running on the host machine is referred to as the *host operating system*. The host operating systems that have been used in the project are SunOs 4.1, VAX-ULTRIX V4.0, and OSF/1 (Mach 2.5 + OSF extensions).

2 Machine model

The machine that is emulated by the simulator combines features found in many existing hardware architectures [6]. Certain design decisions are mandated because the simulator runs native-mode code, and all simulator, POS, and user-mode code running in the POS must execute as a normal (non-privileged, user-mode) application on the target machine.

The simulated machine is based on a single-CPU model[3]. The CPU is logically attached to physical memory that is loaded with machine instructions and data accessed by the CPU. A fixed number of privileged and non-privileged registers are present in the machine. Privileged registers are employed to store addresses or constants that are used by the machine in order to perform system-level functions, and to control devices present in the simulated system. The non-privileged registers are the same general-purpose registers that exist on the host system and are accessed by both user-level and supervisor-level code.

Direct memory access (DMA) and non-DMA devices may be configured into the simulated system. The device interface is primitive, requiring direct manipulation of device registers to complete I/O operations. Devices may be set to interrupt when the I/O is complete, and the machine will then cause an interrupt service routine (ISR) to execute on the simulated machine's CPU.

2.1 Instruction set

Since the simulated machine will run native-mode code (i.e. code generated by the host system's compilers) as a user-mode application on the host system, the supported instruction set consists of the majority of non-privileged machine instructions available on the host machine. Code generated by the host system's compilers will run in the simulator without alteration[4]. The simulator is designed so that the POS and the user-mode code running within the POS can be written entirely in a high-level language, such as C.

Privileged and "special" machine instructions are emulated via a procedural interface. The simulator provides routines that perform the same logical functions as these instructions would perform on an actual machine. In this category are the following instructions:

- user-mode trap (controlled access to supervisor mode)
- return from trap, exception or interrupt (REI)
- process context creation, saving, loading, and modification

2.2 Execution modes and stacks

The simulated machine supports two execution modes — supervisor and user. The machine is in supervisor mode when POS code is running, and in user-mode when non-privileged code is running within the POS. Physical pages and their containing memory regions are assigned protections based

[3] In section 6.1, an possible extension to the simulator is discussed that would allow certain types of parallel machines to be simulated.

[4] relocation and address fix-up of the code may be required, depending on the POS being developed

on execution modes[5]. For example, POS code and data can be protected from read or write access by user-mode code, while at the same time allowing read/write access to supervisor-mode code (see section 2.6 for more information).

Each process context running within the POS has two program counters and two stack pointers — one of each for supervisor and user modes. There is currently no support for a system-wide interrupt stack. When a device interrupt occurs, the supervisor stack of the currently executing process context is used to deliver the interrupt handler.

Simulator privileged instructions (see below) can only be issued from supervisor mode, otherwise a *privileged-mode exception* occurs.

2.3 Machine registers

The simulated machine contains the following "privileged" registers:

PSR - the *processor status register*. Contains the current execution mode and interrupt priority level (IPL).

PIR - the *pending interrupt register*. Bits set in this register indicate an I/O has completed for a device at the corresponding IPL.

MMRs - a configurable number of *memory management registers* are available that can be used to support the particular memory management architecture being simulated. The use of these registers is described further in section 2.6.3.

Additionally, device registers are used to provide a low-level device interface as described in section 2.4.

2.4 Devices and interrupts

Simulated terminal, disk, and clock devices are supported. The number of each type of device is configurable. Associated with each device is an interrupt priority level, or IPL. IPLs range from 0 - 15, with 0 being reserved. The simulated machine stores the current IPL in the *processor status register* (PSR).

An interrupt hierarchy is supported using IPLs. When a device completes I/O, its IPL is compared with the current IPL of the simulated machine in the PSR. If the device IPL is higher, the code is interrupted, its state saved, and the interrupt handler for the device is run on the supervisor stack. If the current IPL is lower than the device IPL, the interrupt is deferred by setting a bit in the *pending interrupt register* (PIR).

When the IPL drops lower than the pending device's IPL, the interrupt handler is invoked through the "rei" mechanism (*return from exception or interrupt*). A special simulator instruction (`SIMrei`) must be issued whenever an interrupt, trap, or exception handler exits. During `SIMrei` processing in the simulator, the PIR is scanned to determine whether a pending interrupt can be delivered when the IPL is restored to its previous value. If so, the pending interrupt's IPL becomes the

[5] page protection is not supported in the UNIX implementation of the simulator.

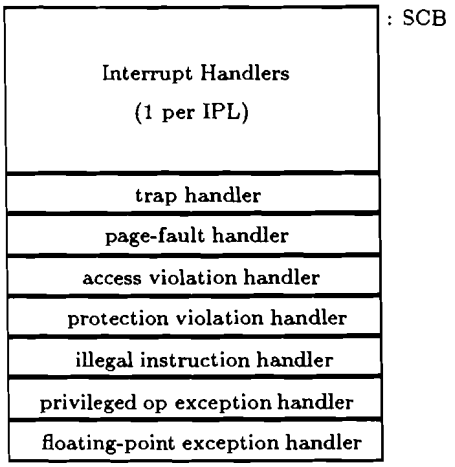

Figure 1: System Control Block

current machine IPL, and the interrupt is delivered immediately; otherwise the IPL is restored to the value that existed prior to the original interrupt. Control then resumes at the point the interrupt handler originally interrupted.

It is also possible for supervisor-mode code to directly change the current IPL by issuing a privileged machine instruction (`SIMsetipl`), so that critical code regions can be protected from interrupts. When the setipl instruction is used to lower IPL, the same checks for pending interrupts are made as with `SIMrei`.

2.5 The system control block (SCB)

A simple *system control block* is used by the simulator to locate the interrupt, trap, and exception handling routines supplied by the POS. The address of the SCB is contained in a simulator register, and may be accessed via privileged instructions. The format of the SCB is shown in figure 1.

A single interrupt handler exists for each IPL, even though multiple devices can use the same IPL. The interrupt handler can determine which specific device is interrupting by checking the first parameter supplied to the interrupt service routine. This argument is a longword uniquely identifying the interrupting device.

2.6 Memory Management

The memory management subsystem is designed with flexibility in mind. It is intended that various memory management architectures can be supported by replacing a single module of the simulator containing relatively few routines with simple interfaces. This module is termed the *pmap module* and is discussed in conjunction with the OSF/1-based implementation in section 5.2.1. Here, the general architecture of the memory management subsystem is presented.

2.6.1 Real versus Virtual mode

The simulated machine can run with memory management enabled or disabled[6]. When the simulator starts up, memory management is disabled — this state is known as *real mode*. While in real mode, there is no virtual memory translation. Every referenced address directly maps to the same address in simulated physical memory. Ideally, physical addresses start at zero and extend to the size of physical memory configured into the machine minus one. However, certain host machine limitations may require that the physical address space start elsewhere.

At some point during system startup, the POS may enable virtual memory, at which point the machine enters *virtual mode*. While in virtual mode, referenced addresses are mapped to physical addresses via the data structures and address translation mechanisms supported by the *pmap*. The writer of the POS is responsible for providing the page-fault handling code, and for maintaining the page-mapping tables required by the architecture. (As mentioned above, the simulator allows alternate memory management architectures and data structures to be supported by replacing the *pmap module*.)

2.6.2 Physical memory, pages, and page frames

Physical memory is divided into page-sized pieces known as *frames*. *Page frames* are numbered starting at zero, and extend to cover the amount of physical memory in the simulated machine. *Page frame numbers* are used during the virtual memory mapping process to support a non-contiguous memory allocation scheme [6]. Normally, page frame numbers will appear in the memory management data structures that are used to provide virtual mapping.

The simulated machine allows both user-mode and supervisor-mode page sharing. Multiple process contexts may map the same page frame to different virtual address ranges. It is up to the memory management architecture to manage protection and keep track of multiple references to the same page — the simulated machine merely provides a mechanism that allows multiple contexts to map the same region of physical memory.

The simulated machine does not have a fixed page size. Instead, various page sizes can be configured by changing a compile-time constant and rebuilding the simulator.

2.6.3 Memory management registers

A configurable number of longword registers are supplied for use with the memory management subsystem of the simulator; it is up to the pmap module (and thus the simulated machine's virtual memory architecture) to use and interpret their contents. For example, a register could be used as the system page table's system virtual address, and another could be used to contain the maximum system virtual page number. The simulator merely provides these registers at a known address and logically accesses them when required during the address translation process.

[6]The memory management functionality described in this section is only supported in the OSF/1-based implementation of the simulator

2.6.4 Page protection

While the simulated machine is in virtual mode, the following page *protection modes* are supported: read, write, and none. These protections are combined with the current execution mode to determine whether access is actually permitted. For example, a page may be protected user-mode read, supervisor-mode read/write; meaning user-mode code may read the page, but only supervisor-mode code may modify the page. The POS has complete control over page protection through the data structures employed to support the memory management architecture. If the paging architecture supports it, the granularity of protection may be a single page. The simulated machine does not support different protections for regions within the same page.

2.6.5 Additional paging support

The simulated machine provides four (logical) booleans associated with each page that may be supported in various degrees by the virtual memory architecture being simulated. The *mapped* flag indicates whether the page is mapped. A *valid* flag is used to indicate whether the page is currently present in memory (resident). A *modified* flag is provided that allows the POS to determine if a page has been written in to. Finally, a *referenced* flag allows the POS to determine if the page has been read or written. The specific paging architecture supported by an incarnation of the simulator's *pmap module* determines whether all or a subset of these flags are actually used. The *mapped* and *valid* flags, however, must be supported by all memory management architectures.

2.7 Machine support for contexts

The simulated machine supports the creation, saving, and loading of process contexts through special privileged simulator instructions. The inclusion of context manipulation support at this level serves the following purposes:

- It relieves the POS writer from dealing with many of the low-level operations that must typically be performed to support multiple contexts. This same rationale can be found in the design of certain CISC architectures, such as the VAX, which include similar high-level context manipulation instructions.
- It provides a mechanism for the simulator to be notified whenever a context is created, loaded, or saved. The simulator may need to know when this occurs because of the way multiple contexts are implemented on the host machine (for an example, see section 5.1.2).
- It provides a host-machine independent way for performing these functions. Most, if not all, of the machine-dependent operations required during context creation and switching are performed by the simulator's context-related procedures.

Process context manipulation is discussed further in the following section concerning low-level prototype operating system components.

3 Low-level components of the prototype operating system

This section describes components of a typical *prototype operating system* (POS) that interact with or are used directly by the simulated machine. The routines and data structures described here provide an illustrative example of the low-level machine support contained in a typical POS. There is no requirement that the code running in the simulator include all the routines discussed here, but a POS designed to support the features found in the majority of existing operating systems would include most, if not all, of them. Similarly, all of the data structures that are recognized by the simulated machine and described in section 3.3 would typically be used within a full-featured POS.

The POS would normally include initialization (system startup), trap-handling, exception-handling, and interrupt-handling routines. The POS code and static data are loaded into (simulated) physical memory, and portions may optionally be paged if *virtual* memory mode is supported in the implementation, and the paging data structures are set up correctly.

3.1 POS initialization routine

It is intended that POS code be written in a high-level language. After the POS is "bootstrapped", the POS initialization, or *main* routine gains control in supervisor mode. If virtual memory is to be supported, the POS main routine typically sets up page tables and enables virtual memory by issuing a special simulator instruction. The *main* routine also allocates and initializes supervisor data structures from the POS *free memory area* (described below), starts any I/O operations on devices that are used by the POS (such as a clock), and creates and loads the initial process context. When the initialization is complete, control may be transferred to the initial process context.

3.2 Interrupt service routines, trap, and exception handlers

After initialization, the POS is event-driven. POS routines are invoked when:

- user-mode code issues a trap instruction. When this occurs, the simulator transfers control to the POS trap handler in supervisor mode.
- user-mode code causes an exception, such as a page fault, floating point exception, access violation, etc. In this case, the simulator transfers control to the appropriate POS exception handler in supervisor mode.
- a (simulated) device completes I/O and the simulator causes an interrupt handler in the POS to be invoked.

Pointers to the trap, exception, and interrupt handlers are contained in a *system control block* (SCB) located in the POS *free memory area* (see next section). The POS *initialization routine* is responsible for setting up the SCB and storing its address in a simulator register.

3.3 Data structures

Data structures accessed by the simulated machine would normally be contained in the *free memory area*. The *free memory area* is the memory left over in (simulated) physical memory after the *prototype operating system* code and data are loaded by the bootstrap procedure. The POS is responsible for allocating the required data structures from the *free memory area*. Supervisor stacks, process control blocks, I/O buffers, and the *system control block* (section 2.5) are allocated from this region. If a page-mapped virtual-memory system is being implemented, then a physical page frame cache and any page-mapping data structures (such as page and/or segment tables) will also have to be allocated from this region.

The POS *free memory area* is fixed in size and can not be dynamically grown, since its size is directly determined by the amount of physical memory configured into the simulated system.

3.4 Process contexts

Multiple process contexts are managed through the use of a data structure known as the *context information block* (CIB). A new context is created (in supervisor mode) by using a simulator privileged instruction `SIMcreatectx()`. `SIMcreatectx()` writes the new CIB to an address supplied by the caller. The CIB is an opaque data type whose contents are used by the simulator to load the context when `SIMsaveloadctx()` is called.

`SIMcreatectx()` is callable by supervisor-mode code and requires the following as arguments:

- a supervisor and user stack pointer that are to be used by the new context.
- the address of a POS routine called the *context initialization routine*. When a process context is loaded for the first time, the *context initialization routine* gains control immediately in supervisor mode, using the new supervisor stack.
- a single longword argument that is supplied to the *context initialization routine*.
- a user-mode program counter that control is initially transferred to when the new context enters user mode. This address should be in the user-mode virtual address space of the simulated machine.
- an address that the newly-created CIB is to be copied to. This should be a location somewhere in the POS free memory area (typically, it would be part of a process control block).
- the address of a longword array that contains initial values for the *memory management registers* that will be set when the context is loaded[7].

The *context initialization routine* could be used to install page-mapping tables for the user-mode address space, and to complete the setup of the user-mode part of the context.

[7] this parameter is not used in the UNIX-based implementation of the simulator.

4 A UNIX-based implementation

An initial version of the simulator has been designed and implemented using BSD4.x ([3] and [4]) as the host operating system. This implementation supports most of the components of the model as presented in the previous section, with the exception of virtual memory.

4.1 Design overview

In the UNIX-based version of the simulator, all simulator routines are loaded into the same executable image as the prototype operating system. Thus the simulator and POS run as a single UNIX process. UNIX signals are used to simulate device interrupts and traps, and to detect exceptions. Host machine-specific code is required in the UNIX-based simulator to alter and restore signal contexts during interrupt, trap, and exception handler delivery and also to support context switching.

4.1.1 Device simulation and interrupts

Disk, terminal, and clock devices are simulated using standard UNIX system calls. Disks are implemented using read/write/seek calls within a single *container file* (a standard UNIX file that contains the data that is stored in the simulated disk). Clocks are implemented directly off the internal virtual timer described below. Terminals are implemented using either an X-window with polling [1] or the default tty the simulator was started from.

Device access is handled via a small number of simulator procedures that allow device registers to be read or written. The device is identified to these routines via a *handle*, which is an opaque data type corresponding to the device. In the UNIX implementation, the *handle* is simply the address of a data structure corresponding to the device.

An internal, periodic *virtual timer* is started during simulator boot. The timer interval is adjustable via a compile-time constant; the default is 10ms. The virtual timer handles device I/O completion, and polls the simulator terminal devices for any characters typed in the corresponding windows. When a device operation is initiated, an I/O delay is implemented by queuing the request to an I/O pending queue. There is a separate queue for each IPL. A delay timeout is associated with each request, and is configurable per device. In most cases, even though a delay is present, the actual I/O is completed immediately (synchronously) before the I/O request is inserted in the I/O pending queue. However, the appropriate status bits in the device registers are not set, and an interrupt is not delivered, until the I/O request completes its delay in the I/O pending queue.

Whenever the *virtual timer* expires, a `VTALRM` signal is delivered to the process, which in turn invokes the simulator's timer handler. The timer handler examines the I/O pending queues, starting with the highest IPL queue. If a request is found that is due to be delivered, the return signal context is modified so that the appropriate interrupt handler will be invoked. The previously interrupted context is stored on an internal "interrupt stack". Each entry on the interrupt stack contains the UNIX `sigcontext` structure for the interruption point, plus some additional information used by the simulator (such as the PSR at the time of the interrupt).

If an I/O request's delay has expired, but the current IPL of the simulated machine is too high

to deliver it, the timer handler sets a bit in the PIR register and exits. Whenever an `SIMrei` or `SIMsetipl` instruction is executed, the simulator checks the PIR register to see if any pending interrupts can now be delivered. Since the simulator's `SIMrei` and `SIMsetipl` routines are also initiated by sending UNIX signals, they also save and alter signal context when delivering previously-pending interrupts.

Since the default simulator terminal uses a simple X-window to provide terminal emulation, it is necessary to poll the window for any characters that may have been typed in it. In order to support polling, a special request is inserted in the I/O pending queues during terminal device configuration. This request is never removed from the queue; it is simply there to make sure the associated device is polled every virtual time-out interval. If a character is found during terminal device poll, and the appropriate bits are set in the control and status registers, the I/O is completed. This may or may not result in an interrupt, depending on whether interrupts are enabled for the device.

4.1.2 Traps and exceptions

When the POS issues a trap instruction, a simulator routine sends a UNIX signal (using the `kill` system service) to the process. The signal handler alters the return signal context so that the trap handler is invoked; using the same methods described in the previous section. In this sense, a trap is analogous to an IPL 0 interrupt.

During simulator startup, a special signal handler is defined to catch any exception-related signals (such as `SIGSEGV`, `SIGBUS`, and `SIGFPE`) generated by the running process. This signal handler again alters the return signal context so that the POS exception handler is invoked. When the exception handler returns, the instruction that caused the exception is executed again.

4.1.3 Context switching

Process context switching is supported via simulator procedures that create, save, and load contexts. The save and load operation is done within a single procedure, `SIMsaveloadctx()`.

The create context procedure `SIMcreatectx()` sets up the initial supervisor stack for the new context, and stores context state in a block of memory provided by the caller (usually the context block would be part of a process control block). Along with the context block's address, the caller supplies the initial supervisor-mode pc and sp for the new context, the initial user-mode pc and sp, and a single longword argument that is supplied to the initial supervisor-mode function. The supervisor-mode stack is used as the starting point to build the new context's supervisor stack.

Process context is normally switched while in an interrupt or trap handler, just before return to user mode. As described in the sections above, a signal initiated the interrupt or trap, and therefore the signal context to restore when the handler completes is stored on the simulator's interrupt stack. During context switch, this saved return context is modified according to the context state saved in the context block being loaded. The current context state is also saved in a different context block provided by the caller, so that it can be reloaded at a later time.

One final simulator routine exists for modifying an active context: `SIMsetusrctx()`. This routine allows modification of the user-mode portion of the context, specifically all the general-purpose registers. `SIMsetusrctx()` serves two purposes; it allows return status to be set for a trap in-

struction (by setting the register that normally contains return status; this is dependent on the host system's compiler), and setting the arguments to the initial user-mode routine that runs in a context. Since `SIMsetusrctx()` allows direct modification of machine registers, its use is highly host-machine dependent.

4.2 Current status and limitations

The UNIX implementation of the simulator was designed and implemented in approximately 3 months by a single grad student, with faculty assistance during the design phase. All functionality (except for virtual memory support) is present, and the resulting simulator library is currently being used in the operating systems course at Brown. The entire simulator library required only 20 C modules with a total of approximately 3300 lines of code. The most difficult part of implementing the library was providing the host machine-specific signal context modification routines.

Supported host machines are the SPARCstation (sun4) [10] running SunOs 4.1 and the VAX running VAX-ULTRIX V4.0. MIPS [2] support could be added quite easily, but was not completed due to time constraints.

5 An OSF/1-based implementation

The UNIX-based implementation described in the previous section allows many machine features to be simulated in a way that is relatively non-intrusive to the *prototype operating system*. However, the virtual memory capabilities presented in section 2.6 can not be easily simulated using the functionality available on standard UNIX systems. Notably absent from the UNIX system call interface are functions that allow direct manipulation of a process' address space — such as explicit mapping of address ranges, reading from or writing to another process' address space, and user-mode (application-level) detection of page-faults.

OSF/1, on the other hand, provides the system calls and memory management support required to implement the virtual memory architecture of the simulator. By using Mach memory management features present in OSF/1, the simulator can be built in a way that minimizes host machine-specific code, and avoids the necessity of leaving user-mode to perform virtual memory mapping and page-fault detection.

Other features of Mach and OSF/1[8] that are used to improve the implementation of the simulator are *threads* ([11]) and *dynamic loading*. Threads allow a a single simulator server task to logically divide its work so that a unique thread deals with each logical type of request. For example, separate threads are used within the simulator server task to simulate I/O on devices that are configured into the simulated machine. Mach system calls [8] allow the state of the single thread running the *prototype operating system* (POS) to be modified so that trap and interrupt handlers can be invoked. The *dynamic loading* capability available in OSF/1 allows code to be dynamically mapped into the simulator environment in a host machine-independent manner. The use of threads and dynamic loading is described further in the following sections.

The OSF/1-based implementation is envisioned to eventually contain the following components[9]:

- a simulator task that handles simulated machine startup, device I/O, context creation and switching, simulator instructions, and serves as a Mach external memory manager in order to provide simulator virtual memory support.
- a simulator library that is linked with POS code to produce one or more *process-context task(s)*. The simulator library is also linked against any user-mode images written to run within the POS.
- a console monitor routine (CMR) that is used during the startup sequence. One of the CMR's main functions is to start the boot sequence when the appropriate command is entered on the console terminal. The CMR also allows clean exit from the simulator via an "exit" command. The CMR could easily be extended to provide additional functions, such as commands to peek, poke, or dump physical or virtual memory locations in the simulated machine.

5.1 Task structure

The OSF/1-based version of the simulator is implemented using multiple tasks (Mach and OSF/1 terminology), otherwise known as processes (UNIX terminology). The first task to execute is the

[8] OSF/1 is a superset of Mach that contains extensions implemented by the Open Software Foundation. The dynamic loading feature was created by OSF and is not part of Mach.

[9] at the present time, prototypes of the first two items listed below have been created

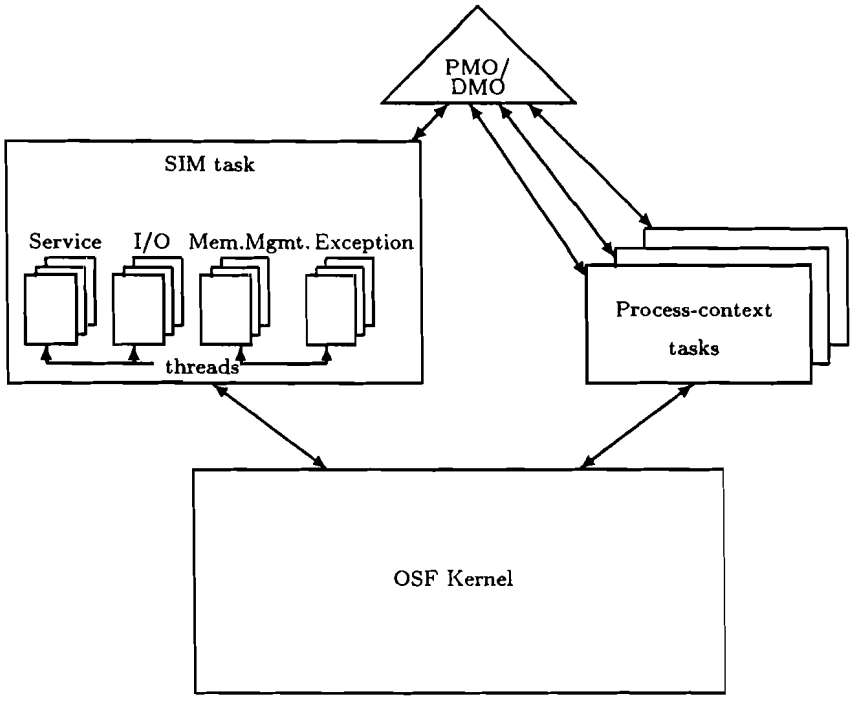

Figure 2: OSF/1-based simulator architecture

Sim task, which performs most simulated machine functions and manages all other tasks that are part of the simulator environment. The tasks controlled by the Sim task are called *process-context tasks*, because each of them corresponds to a single process context created by the POS. Only one process-context task is active at any given time. The Sim task activates (resumes) and deactivates (suspends) process-context tasks whenever the POS issues the simulator's context switching instruction.

The overall architecture of the simulator is shown in figure 2. The small boxes represent threads that execute within the Sim task. The lines between task boxes and triangles indicate the use of Mach memory objects to simulate physical memory, as described in section 5.2.

5.1.1 The Sim task

The Sim task is essentially a multi-threaded, user-mode server that handles all simulator device I/O, privileged and special instructions, process context creation and switching, and process context memory management functions. Mach message passing, memory, task, and thread management kernel calls are used extensively by the Sim task. Additionally, the dynamic load functionality provided in OSF/1 is used by the Sim task to load the CMR and POS image files into the initial *process context task* (see below). The OSF/1 load interface allows these images to be read into a target task's address space, and thus obviates the need for special-purpose, host machine-specific loader code to be part of the simulator.

The Sim task also serves as an external memory manager, or pager, for the *process-context task(s)*. In this way the Sim task can be notified whenever a *process context task* accesses a page that is

not logically resident in the simulated machine. This process is described in section 5.2.

The different types of routines that execute within the Sim task are shown in figure 2. The boxes containing *thread routines* indicate that the associated routines execute as a distinct thread or threads within the Sim task.

5.1.2 Process-context task(s)

In order to provide the same virtual address space to all process contexts running within the POS, a separate task is created on the host system for each distinct process context created by the POS. The Sim task will control the process-context tasks; activating the appropriate one when context switches occur. Only one process-context task will be active at any given time; the rest will be suspended. The process-context tasks run POS code, as well as code loaded by the POS. When the simulator is booted, the Sim task creates (forks) a single *initial process context task* running the *console monitor routine* (CMR) and later (after the "boot" command is issued to the CMR), the POS. Whenever a new process context is created by the POS, a new task will be forked (by the creating process-context task) in which the new context will run. If no new contexts are ever created, there will only be a single process context task. An example of the simulator task hierarchy with four active process-context tasks is shown in figure 3. The curved arcs in the figure represent the remote procedure call communication path between each process-context task and the Sim task, as described in section 5.1.3.

Each process-context task contiguously maps the same supervisor address range to the POS code and data areas. Thus supervisor-mode address space can be viewed as a shared memory segment existing between the process-context tasks and the Sim task.

Unlike the Sim task, each process-context task has only one thread. The state of this thread may be modified by the Sim task so that interrupt, trap, or exception handlers can be invoked. This procedure is described in section 5.4.4.

The code that runs within a process-context task is shown in figure 4. A process-context task runs *prototype operating system* code, as well as code dynamically loaded by the POS as part of the process context. The *simulator interface library* consists of remote procedure call interface routines that are used to communicate with the Sim task (see section 5.1.3).

5.1.3 Intertask communication

The Sim task and process-context task(s) communicate through shared memory (see section 5.2.2) and by issuing *remote procedure calls* (RPCs). Mach and OSF/1 support the RPC interface by supplying the *Mach Interface Generator* tool (MIG) [7]. MIG allows the interface between communicating tasks to be defined in a way that relieves the programmer from worrying about encoding or decoding messages.

RPCs are issued to ports (see [8]). Normally, the Sim task listens on several ports for messages explicitly or implicitly sent by a process-context task. Explicit messages are sent when a process-context task issues a simulator instruction, such as `SIMtrap()` or `SIMflushpmt()`. Implicit messages are sent to the Sim task by the OSF/1 kernel when a page fault or exception occurs in a process-context task.

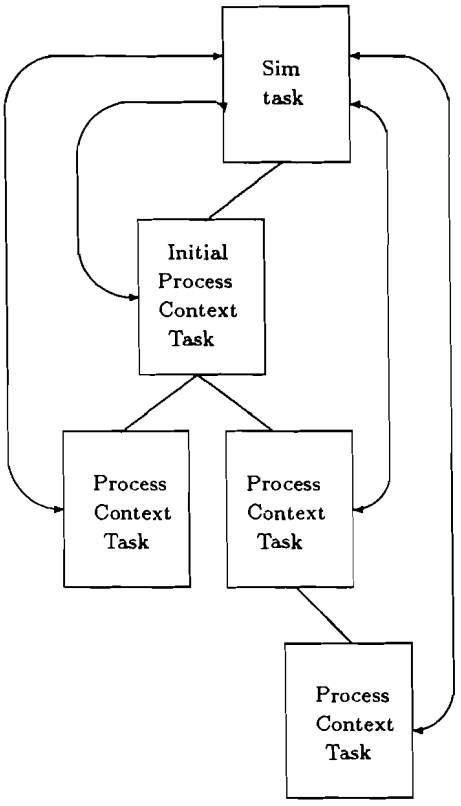

Figure 3: Simulator task hierarchy

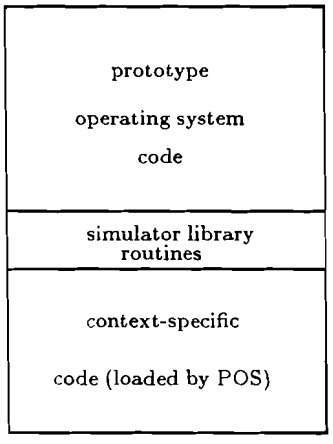

Figure 4: Code that runs within a process-context task

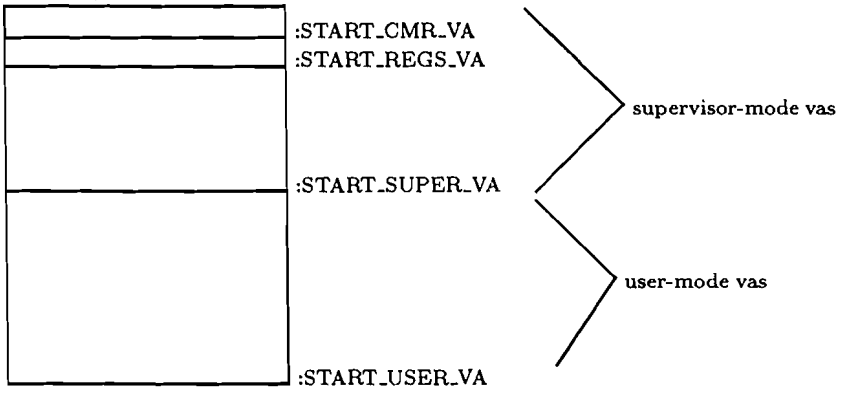

Figure 5: Simulator's address space layout

Most RPCs sent from the process-context task to the Sim task are bidirectional — i.e. the process-context task sends a request message, and then waits for a response from the Sim task. During the wait, the process-context task is in a synchronous wait state. However, even though the single thread within the process-context task is waiting for a response, the Sim task may suspend and then abort it. This results in the RPC call being cancelled, so that the state of the process context thread can be modified to invoke POS interrupt, trap, or exception handlers as described in section 5.4.4.

Practical examples of the use of MIG and remote procedure calls on Mach-based systems can be found in [9].

5.2 Virtual memory implementation

In this section, the complete design for supporting various virtual memory architectures in the OSF/1-based simulator is presented. The main design goal of the virtual memory component is to allow alternative memory management architectures to be implemented with minimal effort.

5.2.1 The pmap module

The Sim task contains a replaceable module that is dependent on the type of memory management architecture that is to be supported by the machine being simulated. This module is called the *pmap module* because it (roughly) serves a purpose similar to the Mach kernel module of the same name. For example, the a segmented, forward-based paging architecture could be replaced by a scheme that uses inverted page tables by replacing the Sim task's *pmap module* (and by changing the POS that interacts with it). The *pmap module* is discussed in the following sections on memory management where relevant.

5.2.2 Address space layout

A typical address space layout is shown in figure 5. In this configuration, user-mode code and data starts at `START_USER_VA`, supervisor-mode code and data at `START_SUPER_VA`. The *privileged register area* is contained within supervisor-mode space. The *console monitor routine* is also present

in supervisor-mode space – this area, and the *privileged register area* are presumed to be always resident and accessible while in supervisor mode. Any access is prevented while in user mode.

The values for START_USER_VA, START_SUPER_VA, START_REGS_VA, and START_CMR_VA are defined in the pmap module. These values are constrained by the host machine's available user-mode address space; that is, all simulator-defined address ranges must fit within the user-mode address range of the hosting machine.

The Sim task and all the process context tasks share the simulated machine's physical memory, privileged register, and CMR regions by mapping ranges of their address space to a single Mach memory object representing the simulated system's physical memory. This memory object is referred to in this document as the *physical memory object* (PMO). The process context tasks map the PMO (contiguously) using the supervisor-mode address range, and may map portions of it (non-contiguously) to pageable user-mode or supervisor address ranges. Thus, any given page of the PMO may be simultaneously mapped more than once by a process context task, depending on the mappings set up in the page mapping tables.

The Sim task will always be able to directly access locations mapped to the physical memory object, since it (the PMO) will be mapped to a contiguous region of the Sim task's virtual address space. If possible, the physical memory object will be mapped using the same supervisor-mode address range that maps the object in the process context tasks. Mapping to the same address range will simplify DMA operations, since the Sim task performs the "real" I/O initiated by a process context task. If a particular host architecture precludes using the supervisor-mode address region in the Sim task, conversion routines will be needed to convert a process context task supervisor-mode address to the corresponding address in the Sim task. The conversion should be straightforward; all that is required is the addition of an offset to the address in the process context tasks to get the Sim task's address for the same location in the physical memory object.

The following two conversion routines will be used: `PMAPpostosim()` and `PMAPsimtopos()`. In the case where no conversion is necessary, these routines will be defined as null macros.

While in both *real* and *virtual* modes, the physical memory object is mapped into the initial process context task starting at the beginning of supervisor-mode address range and extending to the size of configured physical memory. The CMR and privileged register areas are also mapped to the PMO in both real and virtual modes. However, references to addresses outside of these ranges (such as user-mode addresses) are always invalid while in real mode. Virtual mode allows page mapping tables to be defined so that these *non-physical* ranges can be mapped to page frames in simulator physical memory. Supervisor-mode ranges that are not mapped to actual physical memory, CMR, or privileged register areas can also be made pageable in the same way – if the *pmap* supports system page table mapping.

When virtual memory is enabled, different process context tasks may map the same physical page frame at different addresses. In this way, page sharing can be accomplished. Mapping a portion of the single physical memory object is done using the offset and size parameters of the Mach `vm_map()` system call, which allows part of a memory object to be mapped to a specified task address range. The size parameter will normally be set to the simulated machine's page size.

Figure 6 illustrates how two process contexts can share pages by mapping a range of their virtual address space to the same offset within the physical memory object.

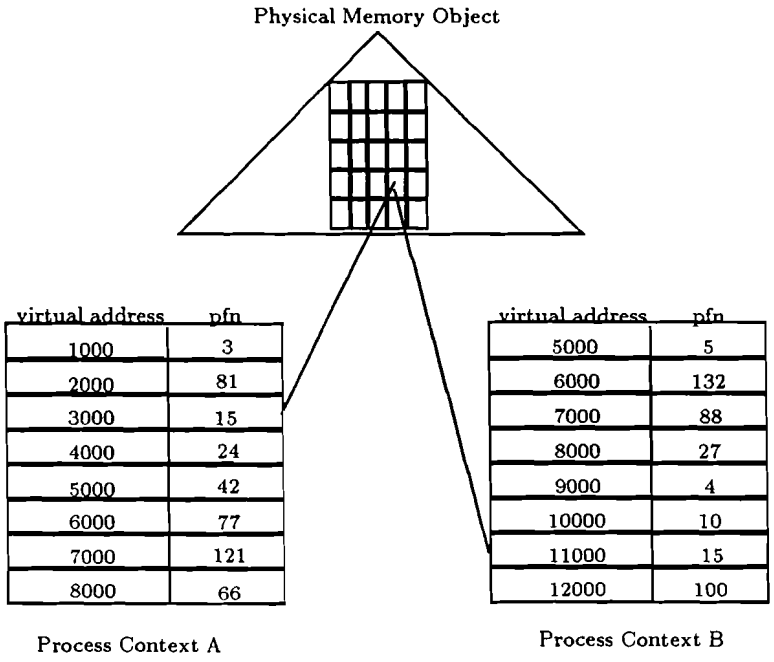

Figure 6: Two process contexts sharing a page

5.2.3 Enabling virtual memory

The system page mapping tables and any other data structures required by the memory management architecture must be created and installed by the POS before the simulator is switched from real to virtual memory management mode. Installation of page tables is usually accomplished by setting memory management registers with the addresses and lengths of the tables (supervisor and user — see next subsection). Switching to virtual memory management mode is done by calling `SIMenablevm()`.

Calling `SIMenablevm()` results in an remote procedure call (rpc) to the SIM task's service port. A thread in the Sim task receives the message and calls a routine (`PMAPenablevm()`) in the pmap module. `PMAPenablevm()` performs whatever remapping is required to set the correct memory management environment for virtual mode. Normally, this means scanning the page tables pointed to by the designated memory management registers, and performing mapping operations (Mach system calls) based on the entries found in the tables.

5.2.4 Use of the Mach external pager feature

OSF/1 and Mach provide an interface that allows a user-mode server to manage memory objects mapped by one or more processes [8]. By using this interface, the Sim task is able to detect access to non-resident pages (among other duties, the Sim task acts as a Mach external pager for the memory objects representing the simulator's physical memory). The Mach kernel essentially acts as a cache for pages mapped to memory objects. When the cache needs updating or flushing, the appropriate routine in the external pager is invoked.

The Mach external pager interface also allows pages cached by the Mach kernel to be locked from read or write access, and the external pager to be notified when either type of access is attempted. The server can then optionally unlock the referenced pages so that access will succeed. This feature is used by the Sim task to support the *referenced* and *modified* bits that are part of the generic memory management architecture supported by the simulator.

In the following sections, certain Mach memory management remote procedure calls (routines whose names start with the prefix `memory_object_`) are referenced. These routines are part of the external pager interface supported by Mach and described in [8].

5.2.5 Non-resident page access

Pages that are mapped by the active POS page tables, but are not resident, are mapped by the Sim task to a particular offset of a *dummy memory object* (DMO) that is also managed by the Sim task through the Mach external memory management interface. The protection of the the DMO is set to the protection value found in the POS page tables, but data is never actually provided for any page of the DMO. The reason for using a dummy memory object in this way is to cleanly detect attempted access of a non-resident page and, at the same time, handle protection violations.

When a non-resident page is accessed in a process-context task, and the protection value permits access, the Sim task is notified (via the external pager routine `memory_object_data_request`) and supplied with the offset within the DMO that produced the fault. The Sim task can then initiate page-fault processing in the faulting process-context task. When the page is made resident by the POS page-fault handler (see below), the Sim task unmaps the page from the DMO and maps it to an offset within the PMO that corresponds to the pfn used to hold the page's data.

If a non-resident page is accessed, and the protection value does not permit access, Mach will send a message to the task's exception port. A thread in the Sim task listens on this port, and changes the state of the process context task so that the exception handler is invoked (see section 5.4.3).

5.2.6 Responding to page faults

The detection of a page fault has been described in the previous section. Figure 7 illustrates the sequence of events that occur after the Sim task determines a page fault has occurred.

As shown in figure 7, normal page fault processing performed by the Simulator proceeds as follows:

1. The Sim task determines that a page fault has occurred because the Mach kernel issues a `memory_object_data_request()` RPC to the Sim task specifying a page mapped to the DMO.

2. The Sim task calls the pmap routine `PMAPget_page_data()` to obtain residency information about the accessed page from the page mapping tables.

3. If the page is not resident, the process context task's state is saved, and then modified so the POS page fault handler is invoked when the process context task is resumed. The page fault handler's address is obtained from the *system control block* (see section 2.5). Arguments to the page fault handler include the faulting address.

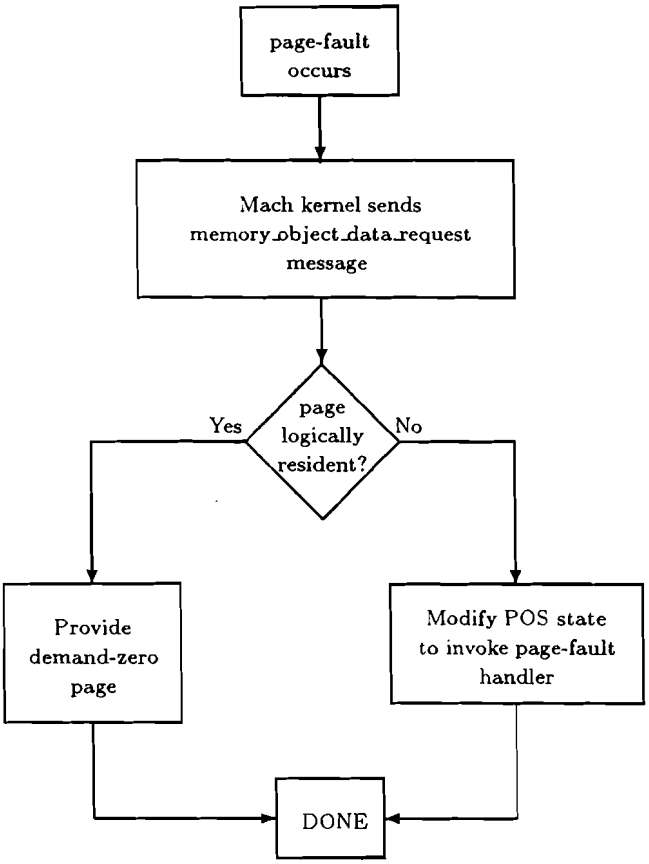

Figure 7: Page fault processing

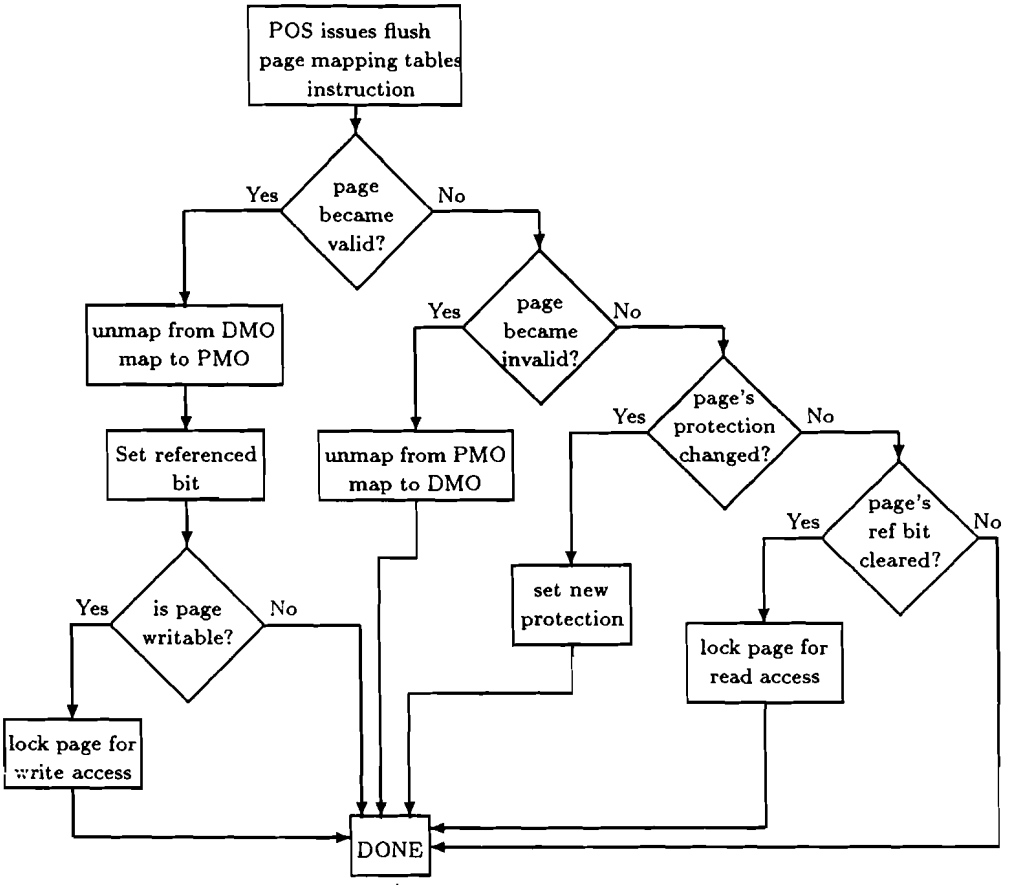

Figure 8: Flush Operation

4. The process context task is resumed. If the page-fault handler is invoked, it performs whatever actions are required to obtain the page data and update the page-mapping tables. When complete, the page fault handler must call `SIMflushpmt()`. `SIMflushpmt()` is described in the following section.

5.2.7 Altering and flushing page mapping tables

Any time the data structures used by the memory management architecture are modified, the Sim task must be notified so that the required changes in page protection and mapping can be performed. Therefore whenever the POS modifies the page tables, it should call `SIMflushpmt()`, which does the following (see figure 8):

1. It issues an rpc call to the Sim task's service port.
2. The Sim task receives the "flush" message, and calls `PMAPflushpmt()`, a routine in the pmap module. `PMAPflushpmt()` determines which pages have been changed in terms of validity, protection, or reference bit state. (Note that this may require the pmap to maintain a copy of the previous page-mapping tables for comparison with the newly-modified tables).

3. If a page has become invalid, the offset within the physical memory object to which the address range was mapped must be unmapped. The page is unmapped by calling the Mach routine `vm_deallocate()`.

4. If a page has become valid, it must be mapped. The physical memory object is mapped using the Mach call `vm_map()`. The pfn found in the page table entry is multiplied by the page size to determine the offset argument of the `vm_map()` call. Page access is set to the protection argument, and `PMAPreferenced()` is called for the page so that the reference bit is set, if supported by the pmap module. Finally, `memory_object_lock_request()` is called so that the Sim task is notified when write access is attempted to the page.

5. If a page's protection has been changed by the POS, and the page is valid, the Mach `vm_protect()` routine is called to set the new protection for the address range mapped to the page(s).

   If the protection is changed for an invalid page, no actions are required.

6. If the reference bit has been cleared by the POS, the page is locked from any access so that the mach kernel will call a memory manager routine (`memory_object_lock_request`) when access is attempted. The actions taken in this routine are described in the next section.

5.2.8 "Referenced" and "Modified" flags

As implied in the previous section, the *referenced* and *modified* flags will be implemented using the cache management routines provided by Mach. The basic idea is to lock a logically mapped, resident page so that the memory manager (i.e. the Sim task) is notified whenever a read or write access is attempted to the page. The locking only has to be applied until the first read or write attempt, after which the page can remain unlocked until the POS or pmap resets the bits in the page table entry indicating the access has occurred, or until the page is invalidated and later revalidated.

Upon notification of a lock request, the Sim task calls a routine in the pmap module that deals with the reference (typically by setting a bit in a page table entry). Afterwards, the page is unlocked so that when the process context task that caused the reference is resumed, the page will be accessed successfully.

The following describes, in detail, how the *referenced* and *modified* flags will be supported (also see figure 8):

1. When a page is made valid by the POS, the POS updates the page table entry within the page-mapping tables, and calls the `SIMflushpmt()` routine to flush the page tables. The Sim task receives the flush notification, and calls `PMAPflushpmt()` as described in the previous section. If the page has just switched from being invalid to being valid, go to step 2. If the *referenced bit* has just been cleared for a valid page, go to step 3.

2. If the page has just been made valid, the Sim task maps the page to the proper address range in the process context task, and sets the page protection. The Sim task then calls the `PMAPreferenced()` routine, which will cause the *referenced bit* in the page table entry to be set, if the memory management architecture supports such a bit.

If the page is protected so that write access is allowed, the Sim task then calls the Mach routine `memory_object_lock_request()` so that read access is permitted on the page, but any write access causes a lock request notification to be delivered to the memory manager (i.e. the Sim task). The routine `PMAPclearmodified()` is called so that the *modified bit* in the PTE is cleared.

3. If the *referenced bit* has just been cleared in a page table entry for a valid page, and read access is permitted to the page, then the `memory_object_lock_request()` call is issued so that read (and write, if the page is protected r/w) attempts cause a lock notification to be delivered to the memory manager.

The *modified* flag is supported in a similar manner:

1. When code running in a process context task attempts a write to the page, the Mach kernel calls the `memory_object_data_unlock()` routine in the memory manager (i.e. the Sim task) in order to deliver the write-lock request notification. This routine calls the `PMAPmodified()` routine, which will cause the *referenced* and *modified* bits in the page table entry to be set, if such bits exist in the architecture. Then, the Sim task releases all locks on the page, using `memory_object_lock_request()`.

To summarize; if a page allows write access, then a write lock will be placed on the page when its PTE is changed from invalid to valid, and the modified bit will be cleared in the PTE. When a write access is attempted, the lock will be released and the pmap notified so that the modified bit can be set in the PTE. Once set, the modified bit remains set until the page is invalidated and revalidated.

Also, when a PTE changes from invalid to valid, the *referenced bit* in the PTE is set. If the POS later clears the *referenced bit* for a valid page, a read lock will be placed on the page so that a subsequent read attempt will result in the *referenced bit* being reset.

Note that if a write attempt is made to a page that is protected read-only, an address exception will occur instead of a lock notification. Lock requests for write notification will never be issued to read-only pages.

5.2.9 User-mode code and data

The POS code is responsible for loading user-mode code, usually from image files stored in a simulated disk device. User-mode code can not call any of the simulator-supplied routines, with the exception of `SIMtrap()`, which is used to emulate a trap instruction. If a privileged simulator instruct is called from user mode, a *privileged instruction exception* is delivered to the process.

User-mode code is subject to paging, and therefore the POS must ensure that the required page mapping tables exist before any user-mode code is executed.

5.2.10 Switching between user and supervisor modes

The simulated machine supports two execution modes (also called *access modes*): supervisor and user. Page protection is based on the current execution mode; for example a page could be protected

user-mode no access, supervisor mode read access. In order to support different protections for each mode, the simulator has to be notified whenever the simulated machine is switching between the two modes, and has to issue Mach system calls to change the protection on pages that have different protections for each mode.

The simulated machine switches mode whenever a trap instruction is issued from user mode, or when the return from trap, exception, or interrupt instruction is returning to user mode. In both cases, the Sim task is notified via an rpc call from the active process context task (see sections 5.4.2 and 5.4.4). The Sim task calls `PMAPmodeswitch( protect_rtn )` whenever the mode is being changed. `PMAPmodeswitch()` must scan the page mapping tables and call `protect_rtn` for each page or range of pages that need their protections changed. `protect_rtn` calls the Mach routine `vm_protect()` to actually accomplish the protection switch.

5.3 Simulating device I/O

When a device register is updated in a way that semantically results in the commencement of an I/O operation, the Sim task detects the update, notifies a thread in the Sim task to process the I/O operation, and updates a device register when the I/O is completed. If an interrupt is to be delivered, the active process context task is suspended and one of the following is done:

1. If the current IPL of the machine is less than the IPL of the device, then the state of the POS thread is altered so that the interrupt handler for the just-completed I/O device is executed when the process context task is resumed. The Mach primitive `thread_set_state()` will be used to alter the state of the thread. The POS thread is then resumed and the interrupt handler executes immediately. See section 5.4.4 for details of the interrupt delivery mechanism.

2. If the simulator is running at an equal or higher IPL than the device that has just finished I/O, a bit in the simulator's Pending Interrupt Register (PIR) is set to indicate the pending interrupt. The `SIMrei()` or `SIMsetipl()` code will detect this bit and deliver the interrupt when IPL is being restored to a lower value.

`SIMrei()` and `SIMsetipl()` are described in detail in section 5.4.5. Also, the complete sequence of operations that need to be done when altering the state of the POS thread is described.

5.3.1 DMA support

The POS starts DMA operations by loading the *device address* and *memory address* registers, and then setting a bit in the device control register using `SIMdevctl()`. The memory address must be a supervisor-mode address. Since the Sim task maps the physical memory object using the same supervisor-mode address range as the process context tasks, the Sim task can use the memory address directly to perform the I/O. For example, to perform a disk read operation, an I/O thread running in the Sim task would use the value in the *memory address* register as the address of the target buffer.

5.4 Traps, exceptions, and interrupts

When a trap, exception, or device interrupt occurs, the Sim task takes special actions that may result in altering the state of the active POS thread. Mach kernel calls facilitate these activities and allow the machine-specific aspects to be reduced to very few lines of code.

5.4.1 Manipulating the POS thread's state

Before the details of trap, exception, and interrupt processing are presented, it is necessary to describe exactly how the state of the single POS thread will be altered when a trap, exception, or interrupt handler is invoked, and how the state will be restored when the handler exits.

The Sim task maintains a data structure known as the context stack. Whenever the flow of control in the POS thread is altered to invoke a handler, information pertaining to the thread and machine context that existed before the alteration is stored in a structure and pushed onto the context stack. This information minimally includes the program counter, the stack pointer, contents of the general-purpose registers, and the PSR.

The handler must always exit by issuing a `SIMrei()` instruction. When the simulator receives notification from `SIMrei()`, it pops the context stack and restores the context that was previously interrupted. At most, there can be 17 entries on the context stack, since there are 16 interrupt priority levels whose corresponding devices could interrupt an active trap or exception handler that interrupted user-mode code (note that using IPL 0 for a device interrupt level is not supported).

5.4.2 Trap handling

A trap is initiated by user-mode code executing in the process context task, but most of the work is done in the Sim task. The sequence of events is described in the following steps:

1. The user-mode code calls `SIMtrap()` supplying two parameters that are to be passed to the trap handler.
2. `SIMtrap()` packages the two parameters into an rpc to the Sim task service port, also passing the arguments supplied to `SIMtrap()`.
3. The Sim task suspends the POS thread, stores the context information on the context stack. and then aborts the POS thread (thus aborting the rpc). It then uses the `thread_set_state()` kernel call to modify the suspended state of the single POS thread so that the POS trap handler will be invoked when the thread resumes. The address of the trap handler is obtained from the *system control block* (SCB). The trap arguments are pushed on the target supervisor stack.
4. The `PMAPmodeswitch()` routine is called to set the protection of the memory regions in supervisor-mode address space.
5. The Sim task resumes the POS thread.

6. While the POS thread is handling the trap, it may call `SIMsetusrctx()` to set the return status for the trap. This routine allows modification of the context information stored on the context stack.

7. The POS thread then calls `SIMrei()` . `SIMrei()` will issue an rpc to the Sim task, which will result in the suspended POS thread being resumed (details below) in the previously interrupted thread context.

5.4.3 Exception handling

Exceptions occurring in the POS or the user-mode code running "within" the POS thread will be detected by Mach and a message will be sent to the thread's exception port. The receive rights for the exception port are passed to the Sim task during process context task creation (see section 5.7). Therefore, the Sim task receives exception notifications for all process context tasks, and initiates exception handler invocation in the same way described for traps above.

5.4.4 Delivering an Interrupt

The conditions under which an I/O thread causes an interrupt handler to be invoked in the currently active process context task was described in section 2.4. The I/O thread does the following in order to deliver the interrupt:

1. The Mach system call `msem_lock()` is used to obtain a the system-wide simulator semaphore (see section 5.6). This semaphore is used so that interrupts can not cause an interrupt handler to be delivered in the middle of the processing of a simulator instruction.

2. The POS thread is suspended and aborted. The context at the time of the suspend is saved on the simulator's internal interrupt stack.

3. The appropriate POS interrupt handler is obtained from the *system control block*, and the state of the POS thread is altered so that the interrupt handler will be invoked when the thread is resumed.

4. The Mach system callmsem_unlock() is used to release the simulator semaphore, and the POS thread is resumed.

5.4.5 `SIMrei()` and `SIMsetipl()`

The *return from exception or interrupt* instruction is handled mostly in the Sim task, and will be implemented as follows:

1. The code running in the POS thread (within a process context task) calls `SIMrei()` to exit from the exception, fault, trap, or interrupt handler.

2. `SIMrei()` calls `msem_lock` to obtain the system-wide simulator semaphore, and then issues an "rei instruction" rpc to the Sim task's service port.

3. The Sim task receives the rei message, and suspends, and then aborts, the POS thread that sent it. The simulator *Processor Status Register* (PSR) is reset to its previous value.

4. The simulator's *Pending Interrupt Register* (PIR) is then checked to see if an interrupt can now be delivered (if IPL is being lowered). If so, the state of the suspended POS thread is altered so that the new interrupt handler will be executed. The POS thread is then resumed, and `msem_unlock` is called to release the simulator semaphore.

5. If no interrupts are pending, the context stack is popped, and the context information stored in the popped entry is used to restore the POS thread's state. If control is returning to user mode, the Sim task calls `PMAPmodeswitch()` to protect the supervisor-mode regions of the simulator's virtual address space so that no user- mode access is possible. The POS thread is then resumed, and the simulator semaphore released.

The `SIMsetipl()` instruction behaves very similarly, except that there can be no return to user mode – in the case when there are no pending interrupts, the POS thread is simply resumed with no modification of thread context taking place.

5.5 Support for multiple process contexts

Context switching is performed using many of the mechanisms already discussed. The central idea is that when a new process context is created, a new host system task is forked in which the new context will run. In this way the new context can use the same virtual address space as its parent context. Supervisor-mode regions are shared between all contexts tasks, while user-mode regions are context task specific.

5.5.1 Context creation

When `SIMcreatectx()` is called from an active process-context task, the following processing occurs:

1. The active (parent) process-context task forks the new context task. After the fork, the parent context task suspends itself. The new context task issues an rpc call to the bootstrap port; the message is "new context task created". Included in the rpc call are the new task's *task_id*.

2. The Sim task receives the "new context task created" message, and returns send rights to the Sim task's service port to the new context task. The new context task then issues another rpc to the Sim task's service port (the message is "build context"), this time sending its unique name for the Sim task's service port. This is needed as part of the context information that is stored in the context block in the following steps. Also included as parameters in the rpc call are the arguments that were supplied to `SIMcreatectx()`. These parameters include state information that will be used to build the new context (stack pointers, memory management register contents, initial supervisor-mode program counter).

3. The Sim task receives the "build context" message, and suspends, then aborts the single thread in the new context task. The state of the new context thread is then modified using the parameters that were supplied to `SIMcreatectx()`.

4. The context information that is need to load the simulator's privileged registers is stored in a *context information block* whose address was supplied by the caller of `SIMcreatectx()`. This address must be in supervisor-mode address space.

5. The parent process-context task is resumed.

5.5.2 Context switching

Context switching is initiated when an existing process-context task calls `SIMsaveloadctx()`. The parameters to this call include a pointer to a block that will be used to store the current context, and a block that contains the context to be loaded. The following processing occurs when `SIMsaveloadctx()` is called:

1. The existing process-context task issues a "context switch" rpc call to the Sim task's service port.

2. The Sim task receives the message, and suspends the current process context task. The privileged simulator registers and other state information are saved in the *context information block* whose address was supplied by the caller of `SIMsaveloadctx()`.

3. The Sim task then retrieves the *context information block* for the context to be loaded, and uses it to load the simulator registers.

4. The Sim task then resumes the new process-context task that was previous created as described in the previous section.

5.6 Task and thread synchronization

The system-wide simulator semaphore has already been discussed in sections 5.4.4 and 5.4.5. In each rpc stub that is called by POS code to initiate a simulator instruction, the simulator semaphore is obtained by calling `msem_lock()`. When the rpc returns, or when the instruction is complete (some instructions never return, such as `SIMrei`) `msem_unlock()` is called to release the semaphore. The Sim task tries to obtain this semaphore whenever it is about to deliver an interrupt. Therefore, the semaphore prevents simulator instructions from being interrupted while being processed.

It is also necessary to coordinate access to device registers between I/O threads executing in the Sim task. In this case, the pthread mutex mechanism is used.

5.7 Startup and boot sequence

The Sim task is started manually by the user, with an optional configuration file as a command-line parameter. The Sim task performs the following initialization steps:

Configures the simulator – the configuration file, if present, is read, and the specified devices are "created" (data structures are allocated in the Sim task to represent the devices). If no configuration file is supplied, a default configuration is created.

Port allocation – ports are allocated for the *physical memory object*, the *dummy memory object* (see section 5.2.6), and the Sim task's general-purpose service port. A port is also allocated to serve as the *bootstrap port*, and the Mach system call to set the *bootstrap port* is called.

Initial process context task creation – the Sim task forks the initial process context task. After the fork, the Sim task starts a single thread listening on the ports allocated in the previous step (a port set is used to group the ports into a single entity). The process context task sends a *initial process context task created* message to the bootstrap port, which is received in the Sim task.

Virtual memory initialization – when the Sim task receives the *initial process context task created* message, it suspends and aborts[10] the single POS thread, then scans the address space of the process context task and deallocates all memory regions that overlap the address space of the simulated machine. The Sim task then maps the *physical memory object* (PMO) to the supervisor address range in the process context task.

Activation of the CMR – the Sim task loads the CMR image into its (i.e. the Sim task's) address space, and then uses Mach vm calls to copy the image into the process context task's address space[11]. The state of the single POS thread is modified so that the CMR main routine will be executed when the POS thread is resumed. The stack pointer is set to the highest physical address of simulator memory. The POS thread is then resumed.

CMR initialization – the CMR retrieves its bootstrap port, and issues a *POS started* rpc to the Sim task. The Sim task receives this message, deallocates the CMR from its own address space, and maps the supervisor mode address range to the PMO. At this point, the Sim task initializes device registers located in the register page of supervisor-mode address space. The Sim task then returns send rights to its service port as a return value to the *POS started* rpc. The CMR receives the port, and loads it into a reserved register located in the register page of supervisor-mode address space. (The service port name becomes part of the process context, since each rocess context task will have a unique name for this port). The CMR can now use the normal simulator mechanisms for I/O calls.

"Booting" the POS is initiated by a command typed on the console terminal. The CMR parses the command, and sends a *POS boot* message to the Sim task's service port. The Sim task immediately suspends and aborts the single POS thread. The Sim task then unmaps itself from the supervisor address space, loads the process context task image into its own address space, and uses mach vm calls to copy the POS image into the process context task's supervisor-mode address space.

(The POS will be linked with a special switch to set its starting address to the beginning of the supervisor-mode address range. Thus the OSF/1 loader will perform the correct address fix-up since this is the same address range in which it will be copied into the process context task. The reason for temporarily unmapping the supervisor area in the Sim task is to allow the loader to function properly – the load call will fail if the address range is already in use).

The POS thread's state is altered so that the main POS routine will be invoked when the thread is resumed. The pfn of the first free memory page is supplied as the first argument to the main

[10] the abort is required before the state of thread can be modified

[11] the OSF/1 load interface does not currently support directly loading images into another task's address space. This may be an option in future versions of OSF/1

routine, the size, pages of the entire physical memory is the second argument. The Sim task then re-maps the PMO to the supervisor-mode address range, and resumes the process context task.

5.8 Current status and limitations

The OSF/1-based implementation has been partially implemented. The basic virtual memory functionality has been prototyped; including page-fault detection, delivery, *referenced* and *modified* bits support, and page table flushing. A pmap module exists for the memory management architecture outlined in appendix B.1. I/O device emulation using threads in the Sim task has also been coded and tested, and interrupt delivery is working. Task and thread synchronization, after a great deal of debugging, is now working using a single system-wide simulator semaphore and multiple pthread mutexes to control access to device registers and internal device-related data structures.

User-mode support (the `PMAPmodeswitch()` routine and `SIMtrap()`) has not yet been implemented, nor has context creation and switching. The dynamic image file loading required to load the CMR and POS has not been coded, because the current version of OSF/1 (snapshot 5) does not support the loader interface. Dynamic image file loading and context switching support are the major pieces of work remaining to be done in order to complete the OSF/1-based simulator.

6 Conclusion

An operating system development environment has been created that allows operating system code to run in native mode on the host machine as a user-mode application. A generic machine model has been defined, and the operating system/machine interface is emulated using standard procedure calls. The simulated machine is configurable in terms of the virtual memory architecture supported. The asynchronous behavior of most multi-device computer systems is emulated fully using interrupt priority levels.

Two implementations have been prototyped – the first was built on standard BSD UNIX [3], the second on an OSF/1-based system. The OSF/1-based system allows emulation of a virtual address space, with full paging capability. This can be achieved using the sophisticated memory management kernel interface available with OSF/1.

6.1 Possibilities for further research

The following are ideas for possible further study related to the project:

Operating system analysis tools – it would be useful to measure operating system performance without including overhead introduced by the simulator.

An integrated programming environment – writing operating system code involves special programming paradigms. A tailored programming environment, optimized for operating system development, could be created that could further speed the development process. It should be relatively easy to build this environment on top of a general-purpose, extensible programming environment such as FIELD [12].

Parallel architecture support – the current OSF/1-based simulator could be extended to simulate a parallel machine by allowing multiple process-context tasks to execute simultaneously (given a parallel host system).

Transparent simulator operation – the current simulator suffers from the requirement that in order to modify some simulator registers, an explicit procedure call must be used so that the simulator can be notified. For example, modifying a device control register can not be accomplished simply by writing directly to the register. Instead, `SIMdevctl()` must be used – otherwise the simulator would not be notified when modifications to the register are made. If Mach offered a write-through cache option in conjunction with the external pager mechanism, the simulator could be notified transparently whenever such modifications are performed. The same technique could be used whenever page tables are modified, so that the explicit *flush page-mapping tables* instruction would not be required.

A Summary of the simulator interface

A complete user's guide for the UNIX-based implementation of the simulator can be found in [5]. This section summarizes the interface presented by both versions of the simulator.

A.1 Accessing privileged processor registers

A.1.1 Retrieving the contents of the Processor Status Register

An interface is provided to retrieve the current value of the PSR and to set the IPL value stored in it[12] :

`cur_psr = SIMgetpsr()` – returns the value of the current PSR. Note that the PSR is also the third parameter to interrupt service routines and exception handlers; therefore it should rarely be necessary to use this call to access it.

A.1.2 `SIMsetipl`

The ipl of the simulated machine can be altered by calling `SIMsetipl()`:

`old_ipl = SIMsetipl(new_ipl)` – sets the current interrupt priority level in the PSR to `new_ipl` and returns the previous IPL in `old_ipl`. This routine provides a way for lower-IPL code to raise IPL to block any potential interrupts from higher-priority devices. Lowering IPL below the value represented by an interrupt service routine is not recommended, as it may cause anomalies during REI processing.

A.1.3 `SIMsetscb`

The SCB register contains the address of the *system control block*. TVB, or *trap vector block*, is the name of the register in the UNIX-based implementation. The `SIMsetscb()` call sets this register in the OSF/1-based implementation; `SIMsettvb()` sets it in the UNIX-based implementation.

`SIMsetscb(tvb_adr)` – (OSF/1-based implementation) sets the SCB register to the address of the *system control block* contained in `scb_adr`.

`SIMsettvb(tvb_adr)` – (UNIX-based implementation) sets the TVB register to the address of the Trap/Vector Block contained in `tvb_adr`.

A.1.4 PIR – pending interrupt register

The `PIR` register contains bits which, when set, indicate an interrupt is pending at the IPL corresponding to the set bit's position. For example, if bit 5 is set, an interrupt is pending for some device at IPL 5.

[12] `SIMgetpsr` is only provided in the UNIX-based implementation of the simulator. The OSF/1-based simulator allows direct access to the register in the register area

There is currently no interface to obtain the contents of the PIR register; however, its contents at the time of an interrupt is supplied as the second argument to interrupt service routines — see section A.4.3 for more information.

A.1.5 MMRs – memory management registers

The memory management registers are used by the pmap to contain the addresses of memory management data structures such as page-mapping tables, or memory management parameters such as the size of a page table or a mapped segment. See appendix B.1 for a memory management architecture that uses two MMRs.

MMRs are only used on the OSF/1-based version of the simulator. There is no interface to update them; they exist in the *privileged register area* and may be accessed (read/write) directly by the POS. See the include file `sim.h` for the exact location of the MMRs within the *privileged register area*.

A.2 Virtual memory interface

The routines described in this section are only supported in the OSF/1 version of the simulator.

A.2.1 Enabling virtual memory

The simulator machine always begins operation in *real* mode, as described in section 2.6. In order to switch to *virtual* mode the following call is provided:

SIMenablevm() - enable virtual memory. It is assumed that any page-mapping tables or other data structures required by the pmap have been created and "installed" (normally by placing the addresses of structures in memory management registers) before this call is issued.

A.2.2 Flushing page tables

Whenever page tables are modified by the POS, the the simulator must be notified. The following routine performs this function:

SIMflushpmt() - flush page mapping tables. Normally, the address' of active page-mapping tables would be contained in a memory management register and thus no parameters are required for this call.

A.3 Device interface

The simulator library provides a set of functions that allow access to simulated peripheral device registers. These routines are the only way to control and monitor devices in the simulator environment. The interface has been designed so that accessing devices in the simulator closely resembles the way in which device registers are accessed in an actual operating system.

In the OSF/1-based implementation, all registers except the *device control registers* are accessed directly in the register area in supervisor-mode space . In both implementations, the *device control registers* are accessed through the system call `SIMdevctl()`. This is required because the simulator must be notified when device I/O is initiated. In the UNIX-based implementation, additional calls are provided so that all device registers can be accessed through procedures.

All the simulator device interface functions rely on the user supplying the correct device *handle* as the first parameter. If an incorrect handle is supplied to one of these calls, the results are unpredictable and are likely to result in a simulator crash. In the UNIX-based implementation, the handles are actually the addresses of data structures representing the associated device. In the OSF/1-based implementation, the handle is the supervisor-mode address of the register block corresponding to the device.a

The following simulator functions are provided to modify device registers[13]:

`SIMdevctl( handle, new_val )` — The value specified in `new_val` is written to the device control register of the device represented by `handle`. This function is used for both DMA and non-DMA devices.

`sts = SIMdevsts( handle )` — The value of the device status register of the device represented by `handle` is returned in `sts`. This function is used for both DMA and non-DMA devices.

`val = SIMdevrreg( handle)` — For non-DMA devices, returns the value in the read register of the device represented by `handle`.

`SIMdevwreg( handle, new_val )` — For the non-DMA device represented by `handle`, the value specified in `new_val` is written to the device's write register.

`SIMdevmadr( handle, new_val )` — For the DMA device represented by `handle`, the value specified in `new_val` is written to the device's memory address register.

`SIMdevdadr( handle, new_val )` — For the DMA device represented by `handle`, the value specified in `new_val` is written to the device's device address register.

The values supplied in the `new_val` parameters, which are device-specific, are described in the following sections.

A.3.1 Terminal device

Single-character I/O is supported in the simulator terminal device. If you are running on an Xwindows system [1], the terminal will appear as a new window during the boot process, unless the environment variable `SIM_NOX` is defined. If `SIM_NOX` is defined, or you are running on a non-Xwindows system, the current tty will be used. This means that any output from `printf` will be intermixed with actual terminal I/O from the simulator. By using X-windows, you can separate the debug `printf` statements from the "real" terminal I/O.

If Xwindows is used, multiple terminal devices can be created using the configuration interface supplied with the simulator.

[13] Only `SIMdevctl()` is supported in the OSF/1-based implementation. The other registers are accessed directly in the *privileged register* area of supervisor-mode address space.

The X terminal device is rather simple, and only responds to a few non-printable characters. When a write operation is performed, printable characters are printed at the current cursor location; when the line length of the terminal device is exceeded, wrap around will occur. The X terminal device processes the following non-printable keys:

key	value	action
backspace	0x08	move back one cell, don't delete char
delete	0x7F	move back one cell, delete char
return	0x0D	move to beginning of the line
linefeed	0x0A	move to next line, without carriage return

Non-printable characters, such as control characters, are returned in the device read register but are not echoed in any way during read or write operations.

The IPL of the default terminal is 10.

Terminal control register Four bits are defined in the terminal control register: two "GO" bits, `RGO` and `WGO` (one bit for read and one for write) and two "ENABLE" bits `RENABLE` and `WENABLE` (again, one for read and one for write). The bit positions are defined in the file `sim.h`.

The `RGO` bit is set to start a read from the terminal, the `WGO` bit is set to start a write to the terminal. On completion of the read, the character read from the terminal can be obtained from the read register using `SIMdevrreg`. The character to be output to the terminal during a write operation should be stored in the write register using `SIMdevwreg` before the `WGO` bit is set (using `SIMdevctl`).

The `RENABLE` and `WENABLE` bits are set if an interrupt is to be delivered when the read or write operation completes (see section A.4.3). The `GO` bits are set by the operating system to start I/O, and cleared by the simulator when the I/O is actually complete. However, there is no way to determine when I/O is complete by polling the GO bits; this must be done by polling the terminal status `READY` bits (see below).

Note that because of the separate read/write bits, read and write operations can be performed in parallel.

Terminal status register The terminal status register consists of two bits: `RREADY` and `WREADY`. The `RREADY` bit is set when a read has completed, `WREADY` is set when a write has completed. These bits are accessed by calling `SIMdevsts` as described above. The `RREADY` or `WREADY` bits are cleared when the user sets the corresponding `RGO` or `WGO` bits.

The operating system must make sure that the previous read/write operation is complete before beginning the next read/write operation.

Terminal read register The terminal read register contains the last character read from the terminal device. This register should not be accessed until the `RREADY` bit is set in the terminal status register.

Terminal write register The terminal write register contains the character that is to be output to the terminal device. This register should be set before the `WGO` bit is set initiating a terminal write operation.

A.3.2 Disk device

A direct memory access (DMA) disk device is supported in the simulator by means of a container file. In other words, the contents of the simulated disk appear in a file in the current directory from which the simulator is being run.

The default filename for the container file is `SIMdisk.dma`. If this file exists in the current directory when the simulator is started, it is used. Otherwise, a new empty container file is created. If a new container file is required (i.e. an initialized disk), the old one can simply be deleted.

The default disk is organized as 32 cylinders, 4 tracks per cylinder, 8 sectors per track, 1024 bytes per sector — and thus has a maximum capacity of 1 megabyte. Additional disks with different geometries and capacities can be defined using the simulator's configuration interface.

All disk operations are in units of sectors. Supported operations are *seek*, *read*, and *write*. It is strongly encouraged that seeks be done before each read and write to ensure that the disk head is at the correct location.

Since the disk device is DMA, up to one sector's worth of data is transferred directly to/from the disk (starting at the head's current location) from/to the buffer specified in the memory address register. When the I/O is complete, a bit is set in the disk status register and an interrupt is generated if interrupts have been enabled.

The IPL of the default disk is 7.

Disk control register A 2-bit field, `OP`, is used to specify the desired disk operation: currently `SEEK`, `READ`, or `WRITE` are supported (the values for the `OP` field are defined in `sim.h`). The `GO` bit is set to initiate the operation, and the `ENABLE` bit is set if an interrupt is desired when the operation completes.

Disk status register The status register has a single bit, `READY`, that is cleared when the `GO` bit is set in the control register, and set when the I/O operation has completed.

Disk device address register The device address register is used to set the cylinder, track, and sector position of the disk head. This register is accessed **only** during the seek operation, and is ignored during read and writes. The format of this register is defined in `sim.h`.

Disk memory address register The memory address register is used during read and write operations to determine where in physical memory the disk data should be copied to or from. The register is an unsigned longword that should be set to the desired address (`using SIMdevmadr`) before the read or write is started.

A.3.3 Clock device

The clock device provided by the simulator allows a single timer to be set. A bit is set in the clock status register when the time expires. An interrupt can be generated if interrupts have been enabled for the clock device.

The clock timer value is set by writing into the clock's write register the number of **microseconds** from the current time that the timer is to expire. The resolution of the clock is currently 10 milliseconds (10,000 microseconds) and runs off an internal simulator timer — therefore setting the clock's timeout value to less than 10 milliseconds will result in a time interval somewhere between 0 and 10 milliseconds and is equivalent to setting the timeout value to 0.

Additional clock devices can be created using the simulator configuration interface.

Clock control register The `START` bit causes the clock to begin operation. The `ENABLE` bit, if set, causes an interrupt to be generated when the clock timer expires. Setting the `HALT` bit stops the clock, and clears the `ALARM` and `STATE` bits in the status register (see below). The bit positions are defined in `sim.h`.

The IPL of the default clock device is 14.

Clock status register The clock status register has two bits: `ALARM` and `STATE`. The `ALARM` bit is cleared when the `HALT` bit is set in the clock control register, and is set by the simulator when the timer expires. An interrupt is delivered (or made pending, based on the current IPL) **only if** the `ALARM` bit is clear when the timer expires and the `ENABLE` bit has been set in the control register. The `STATE` bit is set after the `START` bit is set in the control register, and is also cleared when the clock is halted.

Clock device write register The clock write register is set by using `SIMdevwreg`. The value written to the register is a longword unsigned value corresponding to the number of **microseconds** from the current time until the time the clock timer is to expire.

A.4 Traps, exceptions, and interrupts

Traps, exceptions, and device interrupts are events that cause the current flow of execution to be interrupted, with possibly a change of mode and stack from user to supervisor. Traps always occur synchronously and from user-mode code — they are typically used as a way to enter supervisor mode to perform a system call. Exceptions also occur synchronously, and are detected by the simulator when an illegal operation or memory access is attempted. Device interrupts occur asynchronously when a device has completed I/O and the device has been enabled to generate interrupts (as described in section A.3).

A.4.1 Traps

Synchronous traps are generated using the following simulator function:

`val = SIMtrap( a1, a2 )` — causes a trap to occur, and the arguments `a1` and `a2` become the two arguments of the trap handler. Typically, `a1` would contain a trap code and `a2` would contain the address of an argument list, but there are no restrictions on what the arguments are used for. The return value in `val` is set, after the trap has been processed, to the contents of the user-mode register that is normally used to return status on the target architecture. On the SPARCstation, this is `i0`; on the VAX, it is `r0`. The contents of this register, and hence the value returned in `val`, can be set from supervisor mode during trap processing by calling `SIMsetusrctx` (see section A.4.8).

`SIMtrap` is the only simulator routine (besides `printf`) that may be called from user mode. If it is called from supervisor mode, an illegal instruction exception is generated.

When `SIMtrap` is called by the operating system code, the simulator freezes and saves the state of the user-mode code running on the user-mode stack, and starts the trap handler running on the supervisor stack. The address of the trap handler must have been previously stored in the system control (or trap vector) block. If this trap handler address is set to an invalid address in the SCB, or if the SCB register is set to an invalid address, unpredictable results will occur.

The trap handler `must` end with a `SIMrei` instruction in order for control to return to the frozen user-mode program counter and stack; see section A.4.4 for more information.

A.4.2 Exceptions

Exception handlers can be specified (by initializing pointers in the trap vector block) so that control is transferred to an exception handler, in supervisor mode, when one of the following events occurs:

- an attempt to issue privileged (simulator) instructions while in user mode or to issue `SIMtrap` while in supervisor mode
- access violation
- illegal instruction encountered
- floating-point error

When an exception occurs, the simulator switches mode and stack to supervisor (if not already running in supervisor mode). If an exception handler has been set in the TVB for the type of exception, the handler is called with the exception code (defined in `sim.h`) as the first argument, the program counter where the exception occurred as the second argument, and the PSR at the time of the exception as the third argument. Exceptions may occur in either supervisor or user mode, and at any IPL. If the exception handler does not take steps to remedy the faulting situation by altering the context, control will revert back to the offending instruction when the handler issues `SIMrei`, thus causing an endless cycle of exceptions.

A.4.3 Interrupts and interrupt handlers

All the devices described in chapter A.3 can be set up to generate interrupts when an operation completes. One interrupt service routine (ISR) exists for each interrupt priority level (IPL). The

addresses of the ISRs are contained in the trap/interrupt vector block, as described in section A.1.3. The ISR section of the trap vector block and TVB register itself must be initialized before starting any I/O operation that could result in an interrupt; otherwise the simulator may crash or control will be transferred to undesirable locations.

If a device has interrupts enabled, the interrupt service routine is called with three arguments when I/O completes on the device:

- the handle of the interrupting device
- the contents of the `PIR` register (section A.1.4)
- the interrupted (previous) PSR (section A.1.1)

Again, the ISR **must** end with a `SIMrei` instruction; see below.

A.4.4 The `SIMrei` instruction

The `SIMrei` instruction takes no arguments, and has the following syntax (note that `SIMrei` is defined as a macro and requires no parentheses when invoked):

`SIMrei` - return to the previously interrupted IPL and mode. If any pending interrupts exist for an IPL greater than the IPL being returned to, deliver the interrupt immediately.

If the mode being returned to is user, `SIMrei` will first check for pending interrupts, and if none are found, it will switch operation back to the user-mode stack and user-mode program counter. If the mode being returned to is supervisor, `SIMrei` will check for pending interrupts with IPL greater than the IPL being returned to. If any pending interrupts are found, they are delivered immediately by calling the appropriate ISR.

The operations described above will not be performed if the trap/exception handler or ISR returns normally (i.e. does not issue a `SIMrei` *instruction) — therefore it is critical that* `SIMrei` *be used as the single means of returning from all handlers and interrupt service routines.*

A.4.5 Context manipulation

Process context is defined here as a snapshot of the state of the general-purpose registers, the stack pointers (supervisor and user), the user and supervisor program counters, and other internal simulator information needed to properly save and restore a current running environment.

The simulator provides special functions that allow complete process contexts to be created, saved, and restored. Contexts are stored in and loaded from contiguous memory locations (context blocks) supplied and managed by the caller of these routines.

While running in supervisor mode, the operating system also can modify the frozen user-mode portion of its current context — this interface is described below in section A.4.8.

A.4.6 Context initialization

Two routines are provided that allow a new, complete process context to be created while running in supervisor mode in a different process context:

`SIMcreatectx( ctx_adr, sfunc, ssp, ufunc, usp, arg, mmrs )` - A new context is created and loaded into address pointed to by `ctx_adr`. `SIMsizectx` should be used to determine the size of the memory block required to store the context. The new context will initially start in supervisor mode in the *context initialization routine* `sfunc` on the stack starting at `ssp` (remember, stacks grow *downwards*). `arg` will be the single parameter supplied to the `sfunc` routine. The *mmrs* parameter is only used in the OSF/1-based implementation of the simulator. The *mmrs* argument is a pointer to an array of longwords that will be loaded into the simulator's memory management registers when the context is started.

The *context initialization routine* (`sfunc`) routine should terminate with a `SIMrei` instruction, after which control will revert to user mode and the `ufunc` routine, running on the supplied user-mode stack.

`SIMsizectx()` - returns the size, in bytes, that will be required to store the context.

`SIMcreatectx` simply creates a new context and sets up the stacks starting at the addresses provided by the caller. Control is not transferred to the new context until a `SIMsaveloadctx` call is issued (see below). If it is necessary to supply arguments to the initial user-mode function (`ufunc`) of a context, this should be done immediately after the context is loaded (normally in the *context initialization routine* `sfunc`) by calling the `SIMsetusrctx` routine described below in section A.4.8.

A.4.7 Process context switching

Process context switching refers to the action of saving the current process context, and loading into the simulator a different process context. The new process context gains control immediately. The following simulator function is provided to perform process context switching:

`SIMsaveloadctx(old_ctx_adr, new_ctx_adr)` - The active process context is saved in the memory location pointed to by `old_ctx_adr`, and the new process context is loaded from the memory location pointed to by `new_ctx_adr`. The new context could have been created via `SIMcreatectx` or could be the context returned in the location pointed to by the `old_ctx_adr` parameter during a previous `SIMsaveloadctx` call.

If the context being loaded has just been created using `SIMcreatectx`, control is immediately transferred to the *context initialization routine* (`sfunc`) that was specified in that call. Otherwise, the context, after being loaded, continues where it left off when it was saved in a previous `SIMsaveloadctx` call.

A.4.8 Altering the user-mode context

The simulator always ensures that whenever control is transferred from user mode to supervisor mode, all the general-purpose registers in use at the time are saved. Part of this information is stored internally in the simulator and not directly accessible to operating system code.

However, routines are provided that allow the contents of the saved user-mode general-purpose registers to be examined and modified by supervisor-mode code. By using these routines, it is possible to:

- set the return status of the `SIMtrap` routine, so that status can be returned from a system call. Note that using the normal C method of supplying a value to the `return` instruction of the trap handler will not work, because trap handlers *must* end with `SIMrei` and should not use `return`.
- pass arguments to the initial user-mode function (`ufunc`) of a new process context.
- set up an entire new user-mode stack, frame pointer, and program counter for the current process context. This may be useful if a user-mode exception occurs.

The following two procedures provide access to saved user-mode registers of the *current* process context[14]:

`SIMgetusrctx( mask, ret_adr )` - returns the values of the (user-mode) registers indicated by bits set in the mask argument (a single longword) to an array of longwords starting at `ret_adr`. If the 1st bit is set in `mask`, the value of `r0` at the time user-mode code was interrupted is returned; if the 8th bit is set, the value of `r7` is returned, etc. The values are returned consecutively in the return array from lowest to highest, according to the bits set in the `mask` argument.

`SIMsetusrctx( mask, src_adr )` - sets the registers indicated in `mask` to the values stored in the longword array beginning at `src_adr`. The `mask` argument has the same format as in `SIMgetusrctx`.

These routines will only modify the user-mode registers of the currently executing process context. There is no way to modify registers of a previously stored, inactive context.

[14] In this discussion, the rightmost bit is called bit 1, not bit 0.

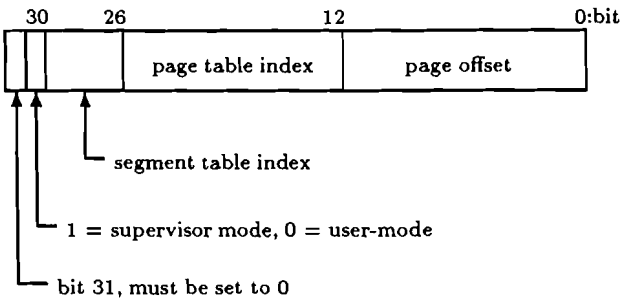

Figure 9: Virtual address format – default pmap

B OSF/1 implementation notes

B.1 OSF/1 implementation: default memory management architecture

The default pmap module developed for the OSF/1-based implementation using the MIPS hardware platform [2] supports a segment-based architecture. Each process context requires a single segment table with one or more page tables. Each segment table entry points to a single page table. Pages are 4096 bytes in length.

The format of a virtual address is shown in figure 9. Bit 31 must be 0, since all simulator addresses must be user-mode addresses on the host machine. Bit 30 is set to 1 if the address is a simulator supervisor-mode address, 0 if it is a simulator user-mode address. Bits 26 to 29 contain the index within the segment table (0-15), bits 12 to 25 contain the index within the associated page table (0-(2*14)-1), and bits 0-11 contain the offset within the page.

As shown in figure 10, a segment table entry consists of the address of the associated page table, the size of the segment (in bytes), and the protection associated with the segment (read-only, read-write, or none).

A page table entry contains the page frame number (pfn), the page protection, and a flags field. The pfn field contains the page frame number of a resident page, it may also be used to store the logical block number on disk of a non-resident page. The page protection in the pte allows more restrictive access to the page than that defined for the segment.

B.2 OSF/1 implementation: location and status of source code

The source code for the OSF/1-based version of the simulator is in `/pro/sim`. The information in this section is also contained in the `README.FIRST` file found in `/pro/sim`.

Due to bugs in OSF/1 and time contraints, approximate 40design presented in the design document has not yet been implemented. The source code in this area was used for testing certain key components of the design – page-fault processing and device I/O with interrupts. Most of the code should be re-usable, with some cleaning up. I've tried to comment things as thoroughly as possible.

The latex source for the design document is contained in the `.doc` subdirector and is called `bse.tex`.

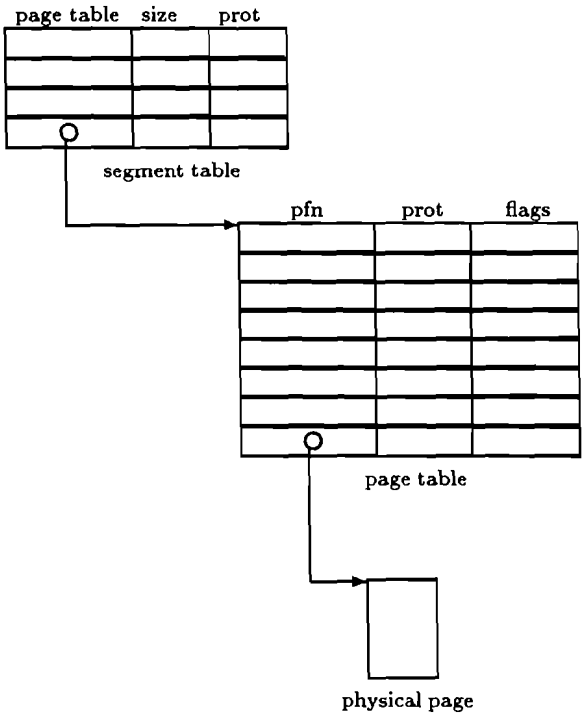

Figure 10: Virtual memory data structures – default pmap

The code should closely reflect the design presented there.

The following things still need to be done to complete this version of the simulator:

1. Because the OSF/1 loader interface wasn't complete as of 12/90, we couldn't implement dynamic loading of the console monitor routine and the prototype operating system image. Stubs performing the same functions are bound into a single image constructed by the Makefile. These need to be split out into seperate images and dynamically loaded during startup.

2. Context switching has not been implemented in any form. However, the design document contains a plan of how to do it. Many of the low-level routines needed to implement context creation and switching should already exist in the current code for other purposes, and could be used.

3. The address space used in the prototype is as follows:

   ```
   start_user_va: 0x20000000
   start_super_va:0x50000000
   registers_start_va: 0x64000000
   ```

   this will probably have to change so that user va starts lower (0x400000 on MIPS) and super_va at 0x40000000. The POS image should be loaded starting at 0x40000000 using the method described in the design doc.

4. Certain pmap modules are not implemented; such as `PMAPmodeswitch()`. The most important routines, such as `PMAPflushpmt()` are implemented, however.

5. The console monitor routine is really just a stub at this point. It needs to be extended to recognize certain commands (a parser should be created); at least the boot and exit commands should be supported.

6. `setipl` is not fully implemented.

7. Reading a device configuration file during startup and parsing it to actually create the machine configuration has not been implemented.

The following things have been coded and minimally tested/debugged:

1. Page fault detection, invocation of page-fault handler, and page-fault table flushing. The mach external pager interface is used as described in the design doc.

2. Device I/O initiatiation, processing, and interrupt on completion. Each device has at least one I/O thread. The terminal device should have two; one for read and one for write (currently only one thread is used, this needs to be changed to work properly).

3. REI instruction.

4. pmap module supporting paging, `PMAPenablevm()`, `PMAPmodified`, `PMAPreferenced()`.

5. The basic task synchronization method, using a system-wide semaphore (the address of the semaphore is a reserved word in the privileged register area). The macros `LOCK_SIM` and `UNLOCK_SIM` are defined in `sim_defs.h`, along with a lot of other useful macros.

6. The mig definition files for many of the required rpc calls. These files end in extension ".defs". The Makefile will compile these using mig to generate the corresponding ".c" files.

B.3 OSF/1 implementation: module layout

The following modules have been created or are supplied with OSF/1:
---

boot.c - initialization rpc routines
boot.defs - mig interface to routines in boot.c
clock.c - simulator clock device implementation
cmr.c - a placeholder for the console monitor routine
deliver.c - delivers interrupt,trap, or exception handlers. Alters thread state.
dev_common.c - handles operations common to all devices, like I/O completion.
device.c - routines that implement device interface; mig interface defined in device.defs.
device.defs - defines device interface offered by simulator
disk.c - simulator disk device implementation
exc.defs - defines exception handler interface (mach-provided file)
globals.c - actual file that containing global variables
init.c - called by initial process context task to initialize the world
kern_mem.c - a memory manager that could be used in test routines to manage supervisor-area free memory
Makefile - builds all modules and a couple of test routines
mem_mgmt.c - performs the Mach external pager functions for the simulator
memory_object.defs - (mach-provided) external pager interface
pmap.c - memory management architecture-specific routines
pos_main.c - a stub for a pos startup routine. Very hackish at this point.
queue.c - generic queue-handling routines
rei.c - handles "rei" simulator instruction
service.c - handles other special simulator instructionss or messages
service.defs - mig interface to routines in service.c
sim.h - include file exported to pos writers
sim_defs.h - internal simulator definitions and macros
sim_globals.h - simulator global variable definitions
sim_internal.h - master include file for use by simulator routines, includes all others.
sim_pmap.h - defines memory management architecture constants.
sim_structs.h - defines internal simulator data structures
sim_types.h - defines internal simulator data types.
startup.c - simulator main routine, generic port listen routine
term.c - simulator terminal device implementation
SIMenable_vm.c - POS interface to simulator to enable VM
SIMflush_pts.c - POS interface to simulator to flush page tables
SIMrei.c - POS interface to simulator to issue REI instruction
SIMdevctl.c - POS interface to simulator to modify device control register

C References

References

[1] R. W. Scheifler, J. Gettys, and R. Newman. *X Window System.* Digital Press, 1988.

[2] G. Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[3] S.J Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Inc. 1989.

[4] M. J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[5] R. C. Boyer *The Brown Simulator Library User's Guide.* Brown University, August 27, 1990.

[6] M. Milenković. *Operating Sytems Concepts and Design.* McGraw-Hill Book Company. 1987.

[7] R. P. Draves, M. B. Jones, M. R. Thompson. *MIG - The Mach Interface Generator.* Department of Computer Science, Carnegie-Mellon University. November 22, 1989.

[8] R. V. Baron, D. Black, W. Bolosky, J. Chew, R. P. Draves, D. B. Golub, R. F. Rashid, A. Tevanian, Jr, M. W. Young. *MACH Kernel Interface Manual.* Department of Computer Science, Carnegie-Mellon University, August 13, 1990.

[9] L. R. Walmer, M. R. Thompson. *A Programmer's Guide to the Mach System Calls.* Department of Computer Science, Carnegie-Mellon University, December 28, 1989.

[10] Sun Microsystems. *The SPARC^{TM} Architecture Manual.* Sum Microsystems, Inc. 1987.

[11] IEEE Computer Society. *Threads Extension for Portable Operating Systems — P1003.4a/D4.* Institute of Electrical and Electronics Engineers, Inc. August 10, 1990.

[12] S. P. Reiss. *Integration Mechanisms in the FIELD Environment.* Technical Report No. CS-88-18. Brown University Computer Science Department. 1988.