FMI/OS: A Comparative Study

by: Dimitri Hammond

Texas State University Press, San Marcos

(c)2006 Dimitri Hammond

Contents

Preface 7

Introduction 9

Part I: Kernel Land 13

Chapter 1: Bootup 15

1. BIOS 15
2. Linux 16
3. FMI/OS 17

Chapter 2: Processes/Threads 21

1. Process 21

2.Thread 24

3. FMI/OS 27

Chapter 3: Scheduling 29

1. Basics 29
2. Systems 32
3. Goals 32
4. Strategies, Techniques and Algorithms 34
5. Thread scheduling 36
6. O/S Comparisons 37

Chapter 4: Inter-Process Communication 43

1. Communication Mechanisms 43

2. OS Comparisons 44

Chapter 5: Memory Management 51

1. Virtual Memory Basics 51

2. Paging 53

3. Swapping and Page Faults 54

4. Page Tricks 56

5. Segmentation 58

6. Memory Implementation Standards 60

7. O/S Examples 60

Chapter 6: Multiple Processors 65

1. Multiprocessor O/S Configurations 65

2. Synchronizing CPUs and Lockouts 66

3. Scheduling 68

4. O/S Comparisons 70

Chapter 7: Deadlocks 73

1. What Causes Deadlocks 73

2. Deadlock Algorithms 74

3. O/S Comparisons 78

Part II: User Space 81

Chapter 8: I/O Layers 83

Layer 1: Device 83

Layer 2: Device Controller 83

Layer 3: Kernel Control 84

Layer 4: Device Driver 85

Layer 5: Device-Independent software 86

Layer 6: User-level software 86

Chapter 9: File Systems 87

1. Physical File Basics 87
2. Orthogonality 90
3. Monolithic OSes 91
4. Client-Server 92

Chapter 10: Environment Layer 95

1. Responsibilities 95
2. POSIX 96
3. FMI/OS 97

Chapter 11: Security 99

1. Basic Security Measures 99
2. FMI/OS 102

Part III: Open Source Development 103

Open Source Development 105

1. Communication 105
2. Coding/Submission 107

Part IV: Conclusion 109

Conclusion 111

1. Summary 111
2. Future Outlook 114

Bibliography 115

Index 117

Preface

This is a thesis that I, Dimitri Hammond, am writing as a comparative study between the open source project **Flexible Microkernel Infrastructure Operating System** (**FMI/OS**) and other popular operating systems. The goal of this thesis is to provide a topical view of the many different ways to piece together an operating system. The audience I am targeting is new (or seasoned) programmers who want to start down the difficult and scary path of coding an OS and those who want a good overview of how to do so.

My intention is that, upon completing this thesis, the reader will emerge with a good grasp of contemporary implementations of operating system concepts, especially those employed within the experimental OS, FMI/OS. Some details, such as specific algorithms and data structures, are left to future investigations of the reader. There are many other resources available that highlight the technical aspects of the topics to follow, and I will point the reader to some of them along the way.

Please keep in mind that this thesis is written on an online wiki, which is located at:

`https://trac.ocgnet.org/fmios/wiki/FMIOSAComparativeStudy`

Due to the open source nature of this project, many technical specifics of FMI/OS are subject to change since the printing of this work. For more information, read Part III: Open Source Development.

Introduction

(Why Another OS!?)

In this work each topic will start with some preliminary information and a quick refresher on a few terms. **Machine Code** (Assembly Language) is the language of the computer hardware; it is the natural language of computers. The **Operating System** (OS) is software that pulls together all of the hardware components, processors, disks, memory, etc., into a cohesive interface for all users and programs to use. The kernel is the name for the core of the operating system: the program that handles all the software-components controlling the hardware. Things that the kernel implements are considered to be in **kernel space** . Everything else, outside of the kernel, is considered **user space** , and is where software that uses the kernel's interface runs (managing files, graphical interfaces, games, word processing, etc).

Operating Systems come in all sizes. Windows, MacOS, UNIX, and SunOS are just a few of the popular platforms. One would argue that these few major forerunners are all that are really needed for the different niches to which they market. However, times are constantly changing and new and improved ways to do things are always in high demand. This is especially apparent when put into perspective of the fast-paced advances in the technology on which operating systems operate. Even today there is still a need for smaller, faster, more stable operating systems to control faster, more complex, and more compact devices.

One popular type of operating system design is the **monolithic kernel** design used in older UNIX and Windows. However, monolithic kernels tend to be very big, complex, and just plain messy in some areas. This is mainly due to their nature of having so much functionality implemented inside kernel space. The structure within the operating system that is needed to

handle all of this functionality must be complex and can be quite susceptible to unexpected modular interactions. The code required to successfully implement everything in a monolithic kernel system tends to be large and complex.

Since the early 1980s, a new way to model the kernel has developed: the **client-server microkernel** model. The design of FMI/OS falls into this category. This is a much simpler model where kernel space and user space are divided differently from the monolithic design. In this newer model, most of the traditional OS functionality occurs in programs that reside outside the kernel in user space. Essentially, the kernel handles only processes and threads, their scheduling, message passing between them, as well as their memory management. Everything else, from device drivers to file systems to networking, takes place outside the kernel. This was a pretty radical idea, but is now incorporated in operating systems such as Chrous, MachOS, QNX, Minix, and Plan9. Yet, none of these microkernel designs separate the kernel and user spaces in the way that FMI/OS is designed, especially while maintaining POSIX compliance (POSIX is a standard for compatibility between UNIX-like operating systems).

Because of the close relation FMIOS has to operating systems QNX and Plan9, I will discuss them a lot in this thesis. One of the uses of the microkernel design is as an embedded operating system. An **embedded OS** is hidden from users so they only speak to the OS via some nice interface. This is typically the case in cell phones or GameBoys or even fancy watches. These devices require a fairly small OS with limited functionality - only enough functionality needed for the limited application, necessitating the smallest, fastest, and cheapest way to build these devices. A microkernel is a good idea for this application because the kernel is small due to its simplicity, and the only things needed in user space are proprietary to the device.

Enter FMI/OS's future. This project is an endeavor tackled by a few people trying for a proof of concept. This project has grown from an academic project, called **VSTa** , to its practical counterpart now called FMI/OS. The history of its birth goes back a few years to

a small number of developers who have since abandoned the project once they proved the concept and learned what they wanted from it. Only one of the original researchers stayed with the project, seeing its potential and continuing to want to learn more. He has since rekindled the project to the quickly growing, open source project it is today. Although only a few people are working on it, they are very active and pushing the OS on its way to actual completion.

Over the next few pages I will reveal the ways that we, the developers of FMI/OS, have tackled different problems, implemented different innovative ideas, and even made mistakes. With this I intend to engage the reader in our learning process. At the end of this work I will provide a summary snapshot of the state of our open source project.

Part I: Kernel Land

Chapter 1: Bootup

(Kickin' the tires)

Computers spring to life in many different ways. The most commonly used platform is the i386 processor series (also the primary machine on which we, the FMI/OS development team, are writing our first version); so the boot-up procedure I will recap will be for this platform.

1. BIOS

The Basic Input Output System (**BIOS**) is the first stop for a typical i386 bootup. This memory holds basic I/O software such as procedures for reading from the keyboard, writing to the screen and writing to the disk. The BIOS first scans all the hardware. **Legacy** devices (typically plugged into the ISA slots inside the computer) have their own fixed interrupts and I/O addresses, and the BIOS records these. Then, the BIOS scans all the PCI Plug-n-Play devices and assigns them interrupts and I/O addresses. These BIOS-assigned interrupts and addresses are only tentative and the OS can change them as it pleases.

This is an ideal place to make a quick mention on what are **interrupts** . These are some of the keys when dealing with ways that the outside world communicates with the software. For instance, a program can be running, doing its own thing, with no contact with the environment outside of the computer. Events occurring outside the computer are typically unpredictable with respect to time and order of occurrence. So, when something does happen, such as a keyboard touch, a diskette-load, or an Internet connect, the interacting hardware generates a physical "interrupt" to the processor. At this point, software within the kernel must respond to this specific event. The details of how these are **handled** are covered in Chapter 3, Section 1.

After this initial hardware scan and gathering, the BIOS then checks which device contains the boot information. This could be a floppy disk, CD-ROM, or, typically, the primary hard drive. The first sector (**bootsector**) is read. The small program in this area reads a **partition table** that is kept at the end of the sector. This table has all of the hard drives, their partitions, and their purpose, listed. The one it looks for, specifically, is the one designated as the boot partition. Then, this little program turns over control to another program, called the **bootloader** , loading it from that boot partition and subsequently running it.

This bootloader is where things really start getting interesting, especially when it comes to the operating system. This is also where the first significant difference between monolithic kernels (such as UNIX and Linux) and microkernels arises.

2. Linux

For Linux, the bootloader must access a saved configuration file as well as the kernel. To do this it must have its own built-in hard disk and filesystem drivers. With these it locates the configuration file and the kernel, also on the boot partition. Once the bootloader locates these, it loads the kernel into memory and runs it according to specialized settings mentioned in the configuration file.

The kernel does a preliminary CPU and RAM scan to know what it has to work with and then sets up all of its own memory: for the kernel stack, debug messages, data structures, etc (Chapter 5). After this, Linux then begins its autoconfiguration where it sets up all of the I/O devices. It gets the data from the system, as far as which devices are installed and what their interrupts and address spaces are, and does its own scan to those devices, building its own table of what is installed. Linux then steps through this new table of installed devices and loads the appropriate **driver** to be able to speak to the device (Chapter 9) and poles each with a quick initialization routine.

After this painstaking hardware setup, Linux does a little further housework such as initializing the realtime clock, and mounting the root file system (Chapter 9). Then, Linux starts the creation of the first official process, called `init` . To run this, it does what it does with every other program it's about to run: sets up `init` 's stack, and then points the **instruction pointer** (the place marker for the processor which points to the next instruction to execute) to the beginning of `init` .

`Init` 's first job is running the initialization script to set up user level servers (such as networking services, mailserver daemons, etc.). `Init` also creates/runs the **shells** (Chapter 10) which the user (and other programs) use to talk to the computer. At this point the bootup process is nearly completed, unless a Graphical User Interface (GUI), such as X Windows, is run.

Keep in mind that the outline of these steps is intentionally vague. The actual details would require many more pages of explanation. However, the Linux bootup procedure does allow for a comparison to FMI/OS boot procedure.

3. FMI/OS

To begin talking about FMI/OS's bootup procedure we look back to the bootloader. Due to FMI/OS' microkernel design, there are many things it cannot do on its own. Because of this, it requires a little more from the bootloader other than just a simple run command.

One major part of the supposed micro-kernel handicap is its inability to talk to any of the devices, such as the hard disk. This is a slight problem because the kernel needs to somehow have access to the first few programs it will run, such as disk and filesystem drivers! To avoid this chicken and egg problem the kernel takes advantage of the bootloader's built-in disk-access feature.

Stored on the same small boot partition as the bootloader's configuration file and the kernel are these few small setup and initialization programs, which the kernel will run. The

bootloader, when loading the kernel into memory, also loads these small setup programs into memory and saves their begin and end addresses into a simple data structure called the **multiboot header** . When the bootloader surrenders its execution to the kernel, it passes along the address of the header as an argument.

The kernel reads the multiboot header, setting aside the addresses for the programs. It then takes the BIOS's hardware interrupt information and records them. Next, the kernel carves out its working memory, initializes the **virtual memory management** (Chapter 6), and allocates memory for the **process** and **thread** data structures (Chapter 2). The kernel then sets up process and thread information for the actual setup programs, flags them as runnable and turns over control to the external processes (Chapter 3).

One of these setup programs creates a lookup table for service names; another is the device driver for the hard drive; one is the filesystem driver; and the last one is the famous `init` . Although they can be loaded in any order, their execution still happens in a predetermined sequence because each will wait, or **block** (Chapter 3), until the one it needs is ready. For instance, `init` needs to be able to read off of the filesystem, so it'll block, until that program is running. However, the filesystem needs to be able to read the hard disk, but the disk needs a place to store the addresses of its service calls for others to find. So, when all of these are loaded, they start in order, like dominoes:

- `namer` -the service lookup table
- `wd` -the hard disk driver
- `fs` -the filesystem driver (DOS)
- `init` -main initialization

Whereas Linux initialized and set up all of the devices and their drivers, the root filesystem, etc., inside the kernel, FMI/OS leaves most of that up to modules outside of the kernel, in user space. This is just the first of many neat and nifty things that FMI/OS does differently that set it apart from other operating systems.

Now get ready for a description of FMI's processes and threads: the essential building blocks of activity on a computer.

Chapter 2: Processes/Threads

("If you want to destroy my sweater ...")

Processes (and threads) are one of the most significant resources of any computer, and managing them is one of the primary objectives when designing an operating system. This chapter is, therefore, one of the most important, and also one of the longest in the entire thesis.

1. Process

What *is* a process? **Processes** are essentially what one would consider a program that has started to execute: a process has instructions that are to be executed by the processor and has memory set aside to keep track of data and its state.

For instance, a word processor is considered a process. It sends instructions to the processor that result in formatting of text, spell checking, even reading-in from the keyboard. It also has some memory set aside to keep track of all of the user's text.

Now, let's say the word processor supports hyperlinks, where you click a link and it'll open a web browser for you. In this case, the word processor starts another process, the web browser. The web browser has its own instructions it needs executed. It also has its own memory to hold web pages, etc. The word processor and the web browser are two distinct processes. Their code is separate, their global data areas are separate, and their stack areas (where local variables are stored) must be separate. Each is a computing entity unto its own. Each is "independent" of the other and may be run in random order by the OS scheduler. In

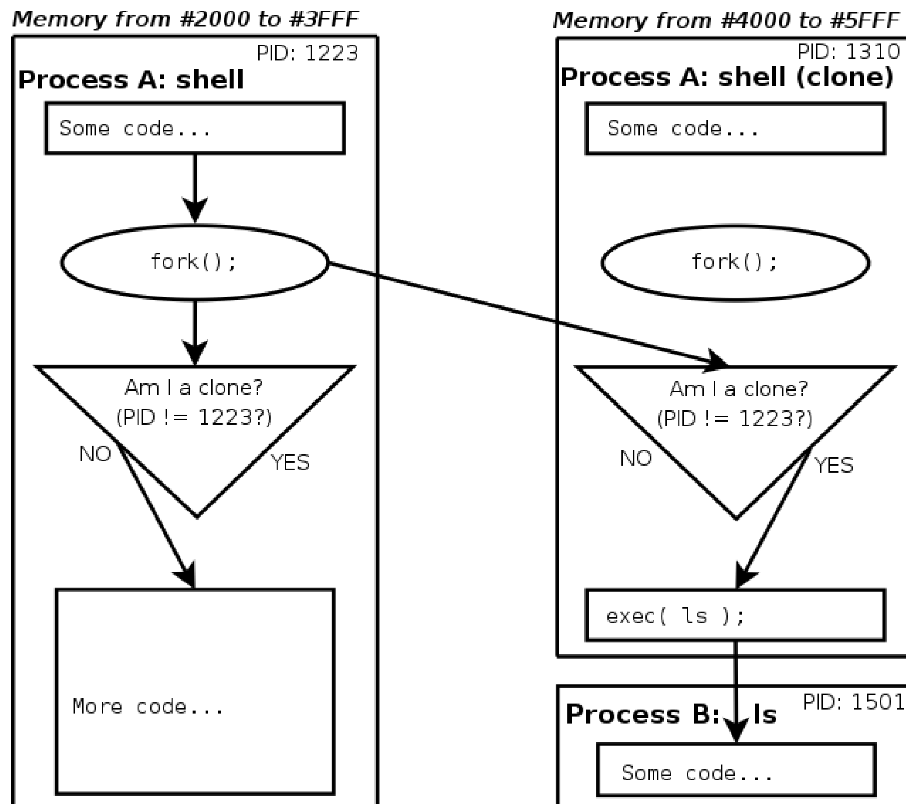
fact, the OS may run each in interwoven bursts of time, making it appear to the user that both are running simultaneously! This is an example of multiprogramming.

Whereas the example web browsers and word processors may alternate execution and continue to present simultaneously changing data to the user, some processes are intended as background processes. These are called **daemons** . These processes do not interact directly with humans but handle other external events, such as mail arrivals, network connections from external hosts, the "plugging in" of PCMCIA cards, etc.

It is the job of the OS to control processes and schedule each one's use of the processor. These stints of processor usage by each process may be short, measured in milliseconds (msec), yet add up in the end to complete whole tasks. Since they are so quick to the human eye, we may see the results of two (or more) running tasks at roughly the same time. The scheduling of these processes is handled by the OS component called the **scheduler** . Scheduling is a complicated endeavor that requires complex algorithms. Each OS will implement its scheduler differently and Chapter 3 covers a few examples of scheduling algorithms.

Keeping track of processes is important. A table is maintained within the kernel that has entries for each process. These entries contain everything that the process needs while running: instruction pointer, **registers** (temporary holding space for data the processor uses when executing a process), stack pointer, process state, priority, scheduler data, etc. When the kernel switches from one process to the next, it saves all of these data about the currently running process into their spot in the table, essentially freezing the process in place. The kernel then loads info about the next process into the processor and runs it. This change is transparent to the process itself. All of its registers are there; its stack is still the same; its files are still open when it is chosen to run again in the future. This procedure, called a **context switch** , is effective in multitasking between multiple processes, but it may be costly, in terms of processor time.

One important thing to look at is the creation of a process. Think of living cells: to create a new cell they basically split off a duplicate of themselves. This is really the easiest way because the best blueprint with which to make a new cell is their own! This is a lot like how computer processes are created in many operating systems. The two basic requests (syscalls) to the OS to create new processes doing new activities are `fork()` and `exec()` .



Let's say you were running a **command-line shell** . This is the typical program that the user directly interacts with to send commands to the operating system, such as "run this" or "list the contents of this directory." Suppose you were to type in `ls` (on **POSIX** systems, Chapter 10) to do the latter. The shell process would then make a **syscall** (commands you send to the kernel) called `fork()` . This `fork` command will then create a new process identical to the shell process: its new memory will be a copy of the shell's memory, its new table a copy of the shell's. The only thing it gets that's unique is a new process id (**PID**). Once this copy is made, and the new spawned process is run, it'll act just like another shell. However, the first thing this new shell does is make another syscall, `exec()` , which

“executes” the desired command (`ls` in this case). This execution replaces the memory of the running process with that of the new program and the new shell process is replaced by `ls` , And when `ls` stops, the process dies with it.

Now, there is another way of accomplishing multiple tasks at once without creating whole new processes: threads.

2.Thread

Lets say you are running your word processor, writing a 600-page paper, and you decide to save your document. However, saving a 600-page document takes some time and you could be stuck waiting for it. It would be much more efficient to enable the word processor to save the document while you were still allowed to type. Well, that’s actually what it does, and it does it using threads.

It would be silly for the word program to start a whole new process just to save the document. If it were to do this, it would have to copy over the entire memory of the document to the new process, still taking a good bit of time. However, lets say a new “sub-process” was started within the word program. This sub-process shares the same memory, but has its own task to execute, getting its own time to be scheduled on the processor. This sub-process is called a **thread** , for “thread of execution”. Machines that implement processing of multiple threads are called **multi-threaded** machines.

Threads also have their own table of information, but it is typically different than the process table. Implementation of threads is also sometimes separate from implementation of processes. Although processes are very much a kernel’s responsibility, threads don’t have to be. Let’s look at the operating system combination **MachOS/Hurd** , for example.

MachOS is a microkernel that implements only **IPC** (Inter-Process Communication, see Chapter 4), basic I/O interaction, basic memory handling, and basic process scheduling. Running on top of MachOS in a “**multi-server** “ (Cite: Le Mignot, 3) configuration is the

Hurd server system. This server system is basically the interim between the kernel and user environments. This is where services typically seen inside monolithic kernels reside, such as implementation and scheduling of threads, full memory handling, filesystems implementation, and network stacks. This system therefore allows many different servers to run on one single MachOS kernel, allowing a third "environment" layer (on top of each of the servers) with which the user actually interacts (such as a POSIX environment, MacOS, or even DOS).

MachOS and Hurd basically split apart what UNIX contains all within its monolithic kernel design. However, when it comes to threads, UNIX can still sometimes be found implementing them in user space in the form of a library. But the kernel is the most popular place for managing threads. When kernel threads are the case, the syscall `t_fork` is set aside for creating threads in much the same way `fork` creates processes.

In UNIX there are two main tables for handling processes (the threads get their own, separate tables).

- **Process table** (contains information needed at all times, even when process is not in memory):
  - Scheduler parameters (*Chapter 3*)
  - It's virtual memory image (*Chapter 5*)
  - Signals (*so the process can still receive them when it's not in memory*)
  - Miscellaneous Information (*process state, event it's waiting for, process id (PID) for itself, its parent, and its children, etc*)
- **User table** (*information needed only when process is in current memory and running*):
  - Machine registers (*for when trapping, see below*)
  - Syscall state (*parameters and results*)
  - File descriptor table (*list of files open*)
  - Kernel stack
  - Miscellaneous (*user & system CPU time, limits to CPU time, stack size, etc*)

The implementation of these tables is pretty straightforward. The process table is held in memory at all times because it is used by the scheduler, memory manager, and signal manager, even for processes that aren't active. Then, when the scheduler determines that it's ready for a process to come back to life, it performs the context switch, saving the current process' User Table and then loading the new process' User Stack.

One interesting note about UNIX and its process creation is something called **copy-on-write** . When a new process is forked it actually doesn't get a full copy of the memory of the **parent** process. In fact, the kernel actually cheats a little and only gives the new process a pointer to the parent's memory pages. This way the **child** process can read from the memory all it likes. It's not until the moment the child tries to *write* to the memory that the kernel copies it so the child can write and read to it at will. This, therefore, saves memory and processing power by not creating extraneous memory if a new process never needs to write to it.

Linux tackles the process/thread situation slightly different than its big brother, UNIX. Linux actually combines processes and threads into one general structure. Instead of `fork` and `t_fork` , it uses `clone` . Whether the new clone is a process or a thread is determined by the parameters sent to `clone` . These particular parameters are centered around **sharing flags** . These set of flags basically determine which address spaces and tables are shared between clones. Threads share the same address space, but processes have their own. This also implies that the contents of the Process Table and User Table in UNIX are combined under Linux.

To be compliant with the POSIX standard, Linux has Process and User table structures that just point to the appropriate parts of the combined tables. Linux also has `fork` and `t_fork` but they're basically just **wrapper** functions to `clone` that call `clone` with the particular share flags appropriate to process or thread.

3. FMI/OS

FMI/OS implements threads in the kernel, treating them not much differently than processes, much like Linux. Details on their implementation are in the next chapter, Scheduling.

As far as the Process and Thread tables, FMI/OS separates them. These structures contain:

- Process Structure
  - Permissions
  - Protections
  - Debugging information
  - Pointers to the parent process and child process(es)
  - List of its threads
  - Its memory pages, or **virtual address space** (**VAS**)
  - And the ports it has for use by the IPC (Chapter 4)
- Thread Structure
  - Parent process information
  - Pointer to its node in the schedule tree
  - Clock ticks ran for and ticks left to run (for use by scheduler)
  - State of process (whether running or blocking or sleeping)
  - Lists of what it's waiting for if blocking or sleeping
  - Stack info (for context switching)
  - Number of times it ran over its **quantum** (Chapter 3)
  - Miscellaneous flags

For a more detailed look at FMI/OS's thread and process structures check the source:

'`http://amatus.g-cipher.net/cgi-bin/archzoom.cgi/fmios-pqm@ocgnet.org--fmios/kernel--devel--1.0--patch-6/include/vsta/proc.h`'

'http://amatus.g-cipher.net/cgi-bin/archzoom.cgi/fmios-pqm@ocgnet.
org--fmios/kernel--devel--1.0--patch-6/include/vsta/thread.h'

Chapter 3: Scheduling

("Beware of bugs in the above code; I have only proved it correct, not tried it." - Donald Knuth)

Let's go back to the example of writing a 600-page document. Say you tell the word processor to save to disk. In the meantime, you switch to a web browser to do some research. The word processor and the browser are two different processes and allowing each to run in turn is the job of the scheduler. Scheduling is often applied to processes and threads in the same manner. Specifics on applying the scheduler to processes and threads differently will be mentioned when comparing the different case studies at the end of the chapter.

1. Basics

When looking at scheduling the first step is to look at *when* to schedule. The scheduler is typically not a daemon (background process). Therefore it has to be invoked during certain times through the course of a running system to make its scheduling decisions. These decisions on when to switch between processes (or threads) are usually made when one of these events occur:

- Creating or exiting processes
- Processes block
- Hardware interrupt (i.e. clock, I/O)
- Software interrupt (i.e. kill, message received)

When the word processor creates a thread to save your document, that thread must wait for the hard drive to write the data to the disk. While this is happening the thread may have

nothing else to do, so it is **blocked** . When it blocks, the OS writes a Blocked Flag into the word processor's table entry. On most systems it also mentions that it's waiting on the disk task to complete. Therefore, when the scheduler is making decisions on which process to run next it will look at this information in the table. If the thread is blocked, then it won't be considered for CPU time. However, in the future, when the awaited I/O interrupt from the disk is handled by the OS, then the blocked thread is flagged as Ready to Run, and it may be given special consideration when the scheduler is called again to make a decision.

When an **interrupt** occurs, the CPU stops what it's doing and switches attention to the recently arrived interrupt. What the kernel actually does is make a context switch to an **interrupt handler** , which is assigned specifically to that event. This handler does several things depending on the interrupt. In some systems, specifically FMI/OS, the handler informs all of the programs **subscribed** (Chapter 4) to it that the event happened. After the handler does its job, the scheduler is called to determine who runs next.

Other information in the process tables, which the scheduler considers, may be a process' priority level (Section 4). Also, whether the process is compute-bound or I/O-bound. **Compute-bound** processes include things such as graphics rendering or calculating a million digits of pi. These processes mostly use the processor so they should get more CPU time. **I/O-bound** processes include things like web browsing or searching through directories. These processes spend most of their time blocking, waiting for the disk drive (or other I/O devices). Therefore, these processes don't require as much CPU time.

The two major categories of process scheduling fall under the names preemptive or nonpreemptive scheduling. **Nonpreemptive** scheduling allows applications to run indefinitely until they create a new process, exit, block, or otherwise surrender their CPU time. **Preemptive** scheduling allows a scheduler to take the CPU away from a process even though it has not reached the point in its computation where it willingly gives up the CPU. Other than the typical I/O and software interrupts, preemptive scheduling makes heavy use of interrupts

from the computer's devices such as clock, disk, network devices, etc. In a preemptive-scheduled system, processes are given a small, fixed-length time slice called a **quantum** . If a time quantum expires, or a higher priority job arrives, the preemptive scheduler can context switch from the previous process to another.

Clarity of the differences between nonpreemptive and preemptive will come with some examples of different techniques and algorithms listed below.

Scheduling is also used for managing resources other than just CPU processing times. Many systems often employ a type of **three-level scheduling** which involves scheduling of three different resources:

- **Admission Scheduler**
  - Deals with admission of processes to the queue of active processes.
- **Memory Scheduler**
  - Applied to active processes.
  - **Swaps** out who's in memory and who's back-stored (written to disk).
  - Keeps a good mix of compute- and I/O- bound processes in memory so as to maximize throughput (i.e. more compute processes than I/O processes)
- **CPU Scheduler**
  - Applied to processes in memory.
  - Schedules processes on processor.

These "levels" work in simultaneous succession, where one level effects which processes are in the next level:

Systems may have different goals for the scheduler. The primary types of scheduled systems are: Batch, Interactive, and Realtime. All three share some common goals, but each has specialized goals, as well. Below are an overview of the different systems, the different goals of schedulers, and then, a short summary of some of the strategies used by schedulers.

2. Systems

- **Batch Systems** . Batch systems stem from the very first computers. They are used in today's mainframes whose primary purpose is high throughput of processes with little regard for waiting times of individual processes.
- **Interactive Systems** . Interactive systems are the most popular because they include most consumer computer systems. These systems are designed for regular user-interaction. Office workstations, gaming systems, web servers are all examples of interactive systems.
- **Realtime Systems** . Realtime systems are time-sensitive. Systems such as medical equipment or driving telematics are time-critical systems. For example, if a controlling process misses a deadline a patient could die or a driver could crash.

3. Goals

- **Fairness** . Fairness is important because every process should be treated with appropriate fairness. Allowing one process all of the CPU while five others of the same priority are waiting in line is not fair. However, allowing all processes to run with equal quanta even though one process is a much higher priority than the others (such as nuclear power plant safety program compared with the payroll program, or video playback compared with sending e-mail) is not fair, either. Fairness is a common goal among the different systems.
- **Policy enforcement** . Scheduling systems are not written without some policy to determine the appropriate fairness of processes. Such policies include priority. Policy enforcement is also a common goal.
- **Keeping everything busy** . To ensure the best efficiency when tackling tasks, the scheduler must keep the system as busy as possible. This is why compute-bound processes may run more than I/O-bound processes: while the I/O processes are

blocking, the system can be kept busy with the compute-bound processes. Keeping everything busy is another common goal.

- **Maximize throughput** . One of the ways to rate a system is by its **throughput** . Its throughput is determined by the amount of work accomplished in a certain length of time. To be deemed successful, some systems (such as mainframes processing millions of customer claims or employee payrolls) should have as high a throughput as possible. This goal is primarily seen in batch system schedulers.
- **Minimize turnaround** . Another way a system can be rated is by its **turnaround** . Turnaround is determined by the average time a task takes, measured from the time it is submitted to when it is completed. The quicker the turnaround time, the higher the rating of the system. This goal originated in batch system schedulers, but is currently seen in almost all other systems.
- **Minimize response time** . When interacting with a computer most users base their rating of the system on how quickly it responds. For example, if a user is typing into a word processor and doesn't promptly see the keystrokes appearing on the screen, it can be quite frustrating and unnerving. In this example a poorly designed scheduler may have not given the word processor sufficient privilege to run quickly enough to handle each keystroke in a timely manner. Therefore, a scheduler has to make informed decisions as to which processes to make higher priority, and when to make other processes lower priority. This goal is typically invoked in interactive system schedulers.
- **Meet deadlines** . Some processes are given real-time deadlines so they must be completed by a certain time. These are typically reserved for processes on realtime systems where these deadlines are important, even critical.

4. Strategies, Techniques and Algorithms

- **FIFO** . FIFO stands for First In First Out. This is perhaps the simplest of the schedulers. The first program submitted for processing is executed while the others line up after it. As the first completes, the second in the queue comes up for execution. The fairness of this technique comes from the first-come, first-served layout.
- **Shortest First** (*nonpreemptive*) & **Shortest Remaining Time Next** (*preemptive*). These techniques are fairly self-explanatory. If the scheduler is handed a batch of processes it chooses the shortest one to run first. Although in the end, regardless of execution order, the batch is completed in the same amount of time, the average turnaround time is much shorter. For example, let's say there were four processes with 1, 2, 4, and 8 estimated minutes for completion, respectively. If they were run in the order 8, 4, 2, 1, the average turnaround time would be $\frac{8+12+14+15}{4} = 12.25$minutes per task. However, if they were ran in order 1, 2, 4, 8, the average turnaround time would be $\frac{1+3+7+15}{4} = 6.5$minutes per task.

  The shortest remaining time next technique is a preemptive version of shortest first. If a new process arrives that has a shorter run time than the current process' remaining time, then it is run instead.

  Both of these algorithms have the added requirement of needing to know the estimated execution time of each process. Therefore, these algorithms are only possible on systems that provide that functionality. For those that do not, the scheduler must externally predict the estimated execution time. One such method is called **aging** . This method bases a process' estimated running time on its previous running times on the CPU. For instance, suppose a process runs for A seconds before blocking. Then, when it is run again, it has running time B. Its estimated next running time could then be calculated by adding $\frac{A}{2} + \frac{B}{2}$, with the aging factor being

$\frac{1}{2}$(which is easy to implement since it's only a bit-shift to the right). Now, if it is run once more and its running time is C then the estimated running time for the next run will be $\frac{A}{4} + \frac{B}{4} + \frac{C}{2}$. The predicted run time can be maintained by the scheduler with one variable (number), P, only. In fact, if the last prediction was P, and T is the last run time, then the new prediction can be found thusly: $P_{new} = \frac{P_{old}}{2} + \frac{T}{2}$.

- **Round Robin** (*purely preemptive*). This technique is one of the generic, and most widely used, scheduling algorithms. As the name implies, this circulates through the list of processes, giving each one its quantum on the CPU. In this manner it is very much preemptive. A big decision when writing a round-robin scheduling algorithm is striking a balance between short quanta (which cause many expensive context switches) and long quanta (which give perception of sluggishness to users).

  Since round-robin, in its pure form, does not account for priority, all processes get equal time on the CPU, regardless of priority. Therefore, this technique isn't ideal for implementing policy and fairness by itself. This is why round-robin algorithms are usually seen coupled with other algorithms that do incorporate priority.

- **Priority scheduling** . The easiest way to incorporate priority is by assigning different sized quanta. Higher priority requires longer quanta. Some tweaks to the algorithm can prevent one high priority algorithm from consuming the majority of the machine. This can be accomplished by diminishing the priority of a process after each complete quantum. Another tweak is to increase efficiency by increasing priority to processes that used only fractions of their last quantum. These quick processes will likely just block again and leave the ready queue.

  Another way of accomplishing priority scheduling is by using **Priority Classes** . Each class is set aside for each level of priority. Within that level

the scheduler will use round robin. Then the scheduler moves on to the the next level. To prevent starvation of lower-level processes the priority classes are frequently adjusted. One example of this is exponentially increasing quantum allocations depending on priority-class-tree depth. Therefore, when the scheduler does finally get to the lower processes, they will have a little more time to spend on the clock.

- **Lottery** . This method closely resembles a lottery. Each process is given a "lottery ticket" and tickets are picked at random after each quantum. To account for priorities, high-priority processes get more tickets for better chances of winning. Through this technique, controlling proportions is easy. If tickets are evenly distributed then each process has even time on the CPU. If there are 10 tickets and three processes are given 1, 3, 6 tickets, respectively, then, in time, the CPU times of the processes will average out to 10%, 30%, and 60% of the total time.
- **Fair-share** . This concept is based on of the idea that each user gets an even division of CPU time; regardless of the number of processes they are each running. If User A is running 8 processes and User B is running 2 processes then User B still gets $\frac{1}{2}$ of the CPU time rather than $\frac{2}{10}$.
- **Rate Monotonic Scheduling** . Time slices are computed depending on the rate at which periodic events occur. For example, if a video is playing back at 24 frames per second the process gets CPU time 24 times a second.
- **Earliest deadline** . This is similar to shortest-time first except it bases its selection on deadlines instead of shortest time. So, instead of each process announcing its estimated run time, they announce their deadline.

5. Thread scheduling

These algorithms and techniques can be applied to processes or threads. However, here are a couple different ways of implementing threads.

User Space threads. If threads are implemented in user space then each process has its own thread scheduling. This provides the process with the power of using the appropriate algorithm for scheduling the threads, depending on the special needs of the process. The implication here is that there are two schedulers: one managing the processes and another one managing the threads. This redundancy slows the system down and is one of the reasons why user space threads are often not implemented. Another disadvantage is that if one thread blocks then the *process* scheduler might block the entire process rendering the other threads in that process "blocked" and un-runnable, as well.

Kernel Space threads. In this method, the kernel handles process scheduling *and* thread scheduling. Therefore, it can manage both in the most efficient way possible. However, sometimes a better way is to not discern between threads and processes at all. This means that the scheduler manages *only* threads since all of the processes are made up of threads anyway. In this regard, the concept of "process" reverts back to being a static structure, purely a holder of resources. Since threads are what are actually executed, the process structure is just a collection of threads with a common commonly held resources, such as memory space or opened files, that they all may share. Although this sounds like a simple design, there are a few drawbacks. Since processes are transparent to the scheduler when it switches between threads, it is likely that the scheduler could also be switching between address spaces (i.e. different processes). This means that an expensive context switch, which happens when switching between different processes, could occur. To compensate for this, some scheduling algorithms take into consideration whether threads share an address space, thus minimizing address space switching and making the context switch less expensive.

6. O/S Comparisons

To help with some of the concepts discussed above a few case studies of different implementations are provided below, as well as FMI/OS's own implementation methods.

UNIX. UNIX has two levels of the three-level schedulers. There is a memory scheduler that operates in round-robin to ensure that all process get time in active memory.

The other scheduler is the CPU scheduler. This uses priority classes with priorities ranging from $n < 0$ (highest; kernel processes) to $n > 0$ (lowest; user processes). A round-robin approach is taken through each priority range with a quantum typically of 100 milliseconds. The highest priority range (most negative) is gone through first until the processes are done. Then the scheduler moves down to the next level, and so forth.

Priorities are recalculated every second based on a few factors: previous CPU usage (using a type of aging), "niceness" factor given to it by parent process, and a base priority. The **niceness** factor allows the process to assign a level of priority to its child (if it assigns a lower priority it is being nicer to other processes). The base value ranges from -4 for low-level processes, such as I/O-bound, to positive numbers for user processes. I/O-bound is highest so the scheduler can get those out quickly since they usually block, anyway, waiting for the I/O.

Linux. Linux uses priority classes, as well. They are: "Realtime" FIFO (highest priority), "Realtime" round-robin (high priority), and Timesharing (low priority). Although they are labeled Realtime they are not true Real-time. They basically mean that they meet realtime extensions to UNIX, standard IEEE1003.4 (POSIX substandard []). Linux also implements kernel threads and is strictly thread-scheduling.

The threads in the top level are scheduled in a FIFO configuration. These threads are not preemtable except from new FIFO threads. Once those threads are all completed the scheduler moves down to the next level, and so forth. Linux also reorganizes like UNIX does. It uses an encompassing algorithm that results in realtime threads getting most time, I/O-bound threads floating to top and gaining priority until done, and compute-bound threads getting fraction of CPU based on priority.

FMI/OS. The FMI/OS scheduling system has led it to be coined a Preemptive Cooperative Multitasking Environment, or **PCME** . The scheduling system relies mostly on nonpreemptive, **Cooperative Multitasking** techniques, which occur during message passing, and reverts only to preemption from the clock, or other interrupt handlers, when no messages are being passed.

There is no actual running scheduler. The scheduler is an **API** , or **Application Programming Interface** , whose commands are called whenever something is calling for a scheduling event. Semaphores-when they block, IPC message-passing commands, and interrupt handlers are usually what call scheduling events. These scheduling events also only deal with threads since FMI/OS implements kernel threads.

The heart of the scheduler is made up of two parts:

1. Runnable tree .

> This is a structure containing all of the threads that are able to be run. For these threads, everything is set up and ready for them to run, they're just waiting on CPU time. Most schedulers refer to this structure as the "ready queues."
>
> The runnable tree has a very important data structure to facilitate constant time scheduling (**O(1)** : see below). The main component of the structure is an array of linked lists. This array is as long as the number of priorities, typically 32 (a type of priority class scheduling). Each entry in the array is the head of a linked list of all of the threads that fall under that priority. The key to traversing the set in a timely manner is one of the other variables in the structure: `priority_mask` . This is a 32-bit variable (32 bits, one for each priority) where the bit number corresponding to the priority is set to 1 if there is a thread in that priority level, or 0 if there are none. For example, `priority_mask` could look like

`00000000001000000000000000001000` which indicates there are threads in the linked-lists of priority 21 and 3 (bits going from left to right: 31 to 0). So, all the scheduler needs to do, to determine who to run next, is call the command `ffs()` (**f** ind **f** irst **s** et bit) on the mask, which returns the position of the first `1` in the 32 bits (in this case, 21). It then uses that position as the index of the runnable tree array where it grabs the head of the linked list that resides there. The list that is there contains the threads that have that priority (priority level 21), so the first thread in it is run.

The beauty of this method is that it is **O(1)** efficient, implying scalability of its operation. Regardless of the number of threads waiting to be run, the scheduler will still only take a constant amount of time. If there are hundreds of threads, the scheduler still only needs 2 lines of code to determine the next process to run.

We chose this because the previous scheduler, from the original VSTA operating system, was bulky and inefficient. By invoking a scheduler that could not only easily implement POSIX-standard realtime capabilities, but still be able to handle a large number of processes without increasing search time, we are able to create a more versatile OS that could scale up to different sizes.

2. The `set_runnable()` and `run_next()` commands.

The two main commands of the scheduler API are `set_runnable()` and `run_next()` . The function `set_runnable()` moves a thread into the runnable tree, attaching the thread to the tail of the list for its priority, and sets the appropriate bit in `priority_mask` .

The function `run_next()` determines the next thread to be run. It takes the current thread (that was running) and reinserts it into the runnable tree depending on the scheduling algorithm the thread has specified in its own

table. If the thread flags itself as `sched_fifo` , it is put back in the head of the list; if it's flagged as `sched_rr` (for round-robin) or `sched-other` it is attached to the tail of the list. For more detail on this, check the IEEE 1003.1 standard (POSIX) ("System", SCHED_FIFO, etc). The next thread to run is now taken off of the runnable tree and is set to run.

So when does FMI/OS call these scheduler functions?

- After any message send or receive
- After a process exits (terminates)
- When a process blocks or yields
- When an interrupt is handled

For example, if a program sends a message to another, it blocks, waiting for a reply. The message sender sets the recipient of the message (if it is blocked, awaiting a message) as runnable (`set_runnable()`), then it calls `run_next()` to run the highest priority thread. This next thread could be the recipient of the message, or it could be some other thread with higher priority.

To handle clock preemption there is a clock timer that is set to the quantum when a thread is run (`run_next()` does this). On 3x86 systems, FMI/OS uses a hardware (CPU) timer that counts down with every CPU clock cycle or tick. When it gets to zero an interrupt happens. The handler for this first decrements the priority of the currently running process (see badness, below), then sets it as runnable and calls `run_next()` .

Fairness.

Fairness of the scheduler comes from the implementation of **badness** and **goodness** factors. When a thread is "good" its priority is increased, when it is "bad" its priority is decreased. Goodness points are gained whenever a thread blocks or yields. Badness points are taken away whenever a thread

has overrun its quantum. All threads start off as top priority, and eventually sort themselves out up and down the process tree in a sort of **accordion** effect, evening out the fairness of the scheduling.

One other important note is that of **priority inheritance** . Suppose the nuclear safety program forks off a thread for one of its safety subsystems. If that thread doesn't have the same priority as its parent then it could be treated as a lesser thread competing for the processor with, say, the payroll program. To compensate for this, the child thread inherits the priority of its parent. FMI/OS also incorporates priorities into message passing: a thread receiving a message inherits the priority of the thread sending it.

More of the message passing and interrupt calls will be looked at in Chapter 4: Inter-Process Communication.

Chapter 4: Inter-Process Communication

One of the most important tasks of the kernel, in a client-server model, is to provide a means for clients and servers to send and receive messages. Clients need to be able to talk to servers, servers need to be able to talk back to clients, and servers need to talk to servers.

1. Communication Mechanisms

There are four parts to this communication infrastructure:

1. Ports

> When a process that wants to be a server is created it requests a **port** number from the kernel. Ports are the endpoints where clients and servers send their messages, kind of like a mailbox address. Each port has its own queue of messages being sent to, or returned by, the server.

2. Translator

> When a client wants to talk to a server it needs a way to find what port number that server is bound to. This requires the need of a separate server that has access to the kernel's directory of port numbers. This server **translates** the server names to port numbers.

3. Message Passing Commands

> These are the actual syscalls for all user processes, clients and servers, to use to send and receive messages.

4. Interrupt Communication

There must also be a way for the interrupt handlers to communicate to other processes. These are separate from user space message passing commands because they are used within the kernel.

2. OS Comparisons

MachOS/Hurd.

MachOS/Hurd does things somewhat similar to the basic client-server system in that the microkernel (MachOS) handles the IPC. It uses ports and a translator, but its differences are significant enough to take a moment to look at it.

Remote Procedure Calls (RPCs).

MachOS utilizes RPCs, which are transport-transparent procedure calls. It is basically a way of making the functions exported by a server accessible by clients (local or remote). MachOS facilitates this by providing an interface for connecting client RPC calls to the server RPC functions. This is really nice for developers writing on the MachOS platform. However, internally, the kernel has to go through complex steps to make it all work. Because of this complexity I leave further details of an RPC system to the reader (see Le Mignot and The GNU Hurd in the Bibliography).

Asynchronous.

The MachOS/Hurd IPC is **asynchronous** in that a thread sending a message isn't blocked until the thread it sent it to receives it. This can easily cause scheduling headaches and other problems, as well as add a lot of overhead (such as complex buffer code).

Translators.

When a potential server is requesting a port it calls on the translator. The translator will actually **mount** the server to Hurd's **Virtual File System** (**VFS** and **mount** : Chapter 9: File Systems) so it is treated like any other directory.

So if a client wanted to talk to the `pfinet` (a TCP/IP internet driver) it would call the root translator (seen by all applications) for the **port right** to `/servers/socket/pfinet` . Port right is the right, or permission, to connect to a port. The root translator will then find the TCP/IP translator. The TCP/IP translator will then open the port by giving the port right to the client. At this point the client can make RPC calls to open TCP/IP sockets or send commands such as `ping` .

Any user who wants to run a server process can have it request a translator of its own. Upon bootup, though, most servers' translators are started by the root translator, such as `pfinet` .

This whole system has very powerful implications. Since all of the servers are accessible through the file system, file system commands, such as `cd` or `ls` can give the user access to different parts of the server if they were in the appropriate directory. This is controlled and restricted, however, by using UNIX file permissions (Chapter 11: Security).

FMI/OS and Plan9.

FMI/OS took many ideas for its IPC model from **Plan9** due to its functionality and simplicity. Some things we have kept, and some we have changed.

Namer.

Plan9's translator, called **namer** , is a user space server that facilitates the translation of a server name to its port number. When a server registers

itself with the namer, namer assigns it a port number that is provided by the kernel. The kernel keeps track of these associations in a lookup table.

When a process wants to send a message to a server, it must connect to that server and get the port number for it. This connection request, via `msg_connect()` , uses namer to get the port number. Once connected, the process can use that port number when sending messages. When the process does send a message, using `msg_send()` , the syscall uses the port/server lookup table in the kernel to translate the port number to the server it is associated with.

This method requires a lot of bouncing back and forth between user space and the kernel. Recently, we have discussed putting namer into the kernel. It will still issue, or **bind** , a port to a server upon request; however, ports are represented by strings instead of numbers. And instead of these strings being some arbitrary lookup number to the server, they actually contain the address (within the file system) of the server. This accomplishes two things:

1) The user space to kernel bounces are diminished.

2) It eliminates the server name/port number lookup table.

Message Passing Commands.

- `msg_connect( port )`

"Connecting" to a port first, before the thread can send messages to it, is important because the kernel has to **authenticate** the thread. For example, you don't want a user program to be able to connect straight to the `wd` disk server because they could cause some real damage. Instead, the user program has to connect to `fs` which can connect to `wd` because it knows

how to properly talk to `wd` without corrupting files or some other haphazard insanity.

- `msg_send( port, &msg )`

  This is where the thread can actually send a message (`msg`) to the server. Since the FMI/OS IPC is synchronous, every message send causes the sender to block. It can send as many as it wants, one at a time. When it wakes up, the message that `msg` points to has been changed to contain the server's reply.

- `msg_receive( &msg )`

  This allows server threads to receive a message (`msg`). Each receive blocks the thread, and it can do as many as it likes, one at a time.

- `msg_reply( &msg )`

  This is how the server replies to a message sent to it. It also enables the server to pass messages back to the sender by changing what's inside the original `msg` .

- `msg_close()`

  This closes the connection to the port and the process can no longer send or receive messages from it.

Interrupt Service Request (ISR).

Interrupts are implemented in the ISR. This is basically a set of message calls that allow interrupt handlers to directly inject messages into subscribed servers' queues without blocking (`kmesg_send()` for kernel-message send, etc.). Clients never talk directly to interrupts and they never have to. Besides, the interrupt handler has to have a port to send messages to,

and clients can't have ports. They always user servers as an in-between, thus further simplifying the ISR.

When still under the guise of VSTa, our kernel only allowed one server at at time to subscribe to an interrupt. And if that server were busy, the interrupt handler would keep track of all of the missed interrupts. This model limited the scalability of the kernel and required more overhead and complexity to handle the backed up, missed interrupts.

Our new implementation takes advantage of **shared interrupts** . This means that more than one server can subscribe to an interrupt. The handler also has two different lists of these subscribers: a *ready list* and a *pending list* . If a server were ready to receive a message from the handler it would be on the ready list. When the interrupt calls the handler it informs every server on the ready list and copies them to the pending list. When a server receives the message, off the queue, from the handler, that server is moved back over to the ready list.

This new model does away with missed interrupts, but allows for a much cleaner and more flexible way of dealing with numerous subscribers.

Typical IPC Flow.

Let's say a thread wants to use the keyboard. Keyboard is an I/O and uses interrupts. The in-between server is `cons` (whose program actually resides in the file system at location `//cons` for **console** . The procedure this thread would go through is as follows:

- `msg_connect( “//cons” )`
- `msg_send( “//cons”, &msg )` where `msg` contains a message that the `cons` server understands as "get keyboard input". The program blocks until `cons` replies with the actual keyboard input inside `msg` .

- `msg_close( “//cons” )` to close the connection after looping through send/receive/send/receive enough times to get all that it needs.

In the case of the 'cons' server, it would look something like this:

- `msg_receive( &msg )` which will block the server until it receives a message.
- `msg_send()` to the keyboard interrupt and block, putting returned data into `msg` when woken.
- `msg_reply( &msg )` which will wake up (i.e. `set_runnable()`) the blocked thread, with the new data in `msg`

The server will then loop back to `msg_receive()` and block, waiting for more messages.

Critical Region.

Due to the nature of multithreaded programming, special attention must be paid to the **critical regions** of code. These critical regions are parts of a program that access a shared resource. For instance, the message queue for a server is shared by everyone; in other words, any other client or server can send a message to it whenever they please (if they have the appropriate privileges). The part of the program that processes a message send request by putting a message on the server's queue, and then incrementing the index pointer on that queue, is a critical region. Keeping in mind that a preemptive scheduler (such as ours) can interrupt a process anywhere, imagine what may happen if that process were interrupted *after* adding a message to a queue, but *before* incrementing the index pointer! Now imagine if the new process to run was sending a message to the same queue. Since the index pointer was never incremented, the message from the previous process becomes overwritten!

This is an example of a **race condition** : two processes are racing for access to a shared resource the one that gets there first alters the state of the resource, adversely affecting the other's work on the resource.

To overcome this common occurrence, the kernel must ensure that all message commands are **atomic** , and thus are run as one command without interruption. The only case, then, where race conditions could still become a problem is in the context of multiprocessors. This is because two different message syscalls can be running at once (on two different processors). This will be covered more in Chapter 6, section 2.

Chapter 5: Memory Management

Allocating and managing memory is one of the primary tasks of an operating system. If there was just one process on the machine with plenty of memory to use, then the memory issue would be easy: the process talks directly to the physical memory using actual memory addresses. However, when you add multiple, simultaneous processes, multiple processors, and larger programs than the memory can hold at one time, you are faced with the need for a more complex memory management system.

1. Virtual Memory Basics

Let's start the discussion assuming there *is* plenty of memory. Managing this memory for multiprocessing machines is a full-time job, in and of itself.

Memory comes in three basic flavors:

- **Anonymous Memory**

  This is the typical form of memory people think of when they think of computer memory. It is memory **allocated** (set aside) for a process, or memory where the operating systems resides. This is also the memory that contains a processes' code, global data, stack and dynamic memory.

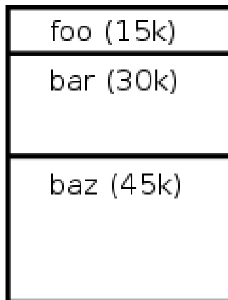
- **Files**

  Files often exist physically on a hard drive and are accessed as memory by using functions such as `mmap()` (Section 6).

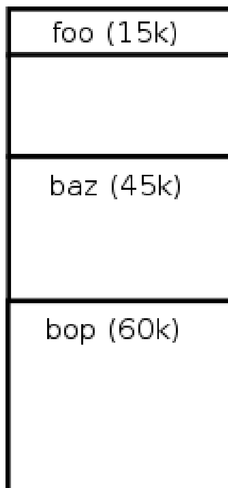
- **Permanently Allocated Memory**

This is memory that is allocated for special uses, is also mapped, but is not managed since it usually is dedicated for the duration of operating system runtime. An example of this is a video frame buffer.

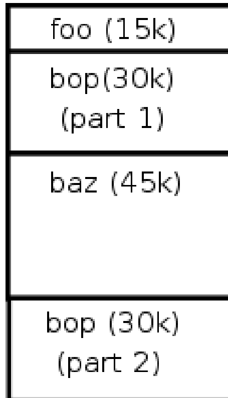
As processes are created and killed, anonymous memory and memory mappings pop up and go away regularly. If processes used physical memory alone without any structure or management, the many gaps that would form over time would result in a waste of memory space. Take, for example, the three processes, `foo` , `bar` , and `baz` . Suppose these processes fill the first 100 Kbytes of memory.



Now, assume `bar` dies and process `bop` starts.



As you can see, there is a gap between `foo` and `baz` while new processes continue to be allocated space at the end of inhabited memory. Since RAM is not infinite, some form of memory management is needed to make use of these gaps to recycle the memory.



As you can see in this diagram, `bop` filled in the gap between `foo` and `baz` . However, since `bop` required more memory than was available in the gap left by `bar` , it had to be split up into two chunks. This management results in memory being scattered in chunks all over. If memory has been chopped up enough, a single process could be storing information in ten or fifty different chunks scattered around memory. Another goal of memory management is for user processes to view memory as one contiguous entity, even though their memory is living within these scattered chunks in physical memory. Enter **virtual memory** (**VM**).

The best way to think of virtual memory is as a layer of abstraction between programs and physical memory space. Not only does this require a memory manager to connect the layers, it also protects physical memory from being accessed directly from user processes, thus minimizing corruption by user processes.

Two by-products of virtual memory are: providing an easy way to share memory between processes, and providing an easy way to **swap** rarely used memory to back-storage (on a hard drive or similar device; Section 5.3).

2. Paging

To keep memory chunks from getting broken up into smaller and smaller pieces, the size of a "chunk" is set to a standard. That way, they are created and erased in chunks of uniform size and contents may easily be moved from one chunk to another. These chunks are called

pages . Surprisingly, when a process is executing, all of its pages do not have to be in physical memory at once - just those memory locations that are needed to execute the next instruction! On a 32-bit system, addresses range from 0 to 232. Each page is typically 4K bytes (4096 bytes) in size. Each virtual page is mapped to physical page via a **page table** . Each process has its own mappings, and so sharing is made possible by mapping different virtual pages to a single physical page.

Whenever a user asks to access an address, that virtual address must be translated to the physical address. Typically, the CPU has to do this for *every* single memory access; thus can really slow things down. Machines that implement virtual memory have a **Memory Management Unit** , or **MMU** , which does this kind of translating at the hardware level (outside of the CPU). One technique to speed up translations is to employ a **Translation Lookaside Buffer** (**TLB**) by the MMU. This is a buffer that caches common virtual/physical address conversions. A **cache** is storage that the computer has quick and easy access to, and is usually closer than the system bus (thus it has a lower access time than ordinary memory).

3. Swapping and Page Faults

There are times when a process' contents may need to be taken out of physical memory and stored on disk. This is called **swapping** . This is another important task that the memory manager may handle. Some occasions that constitute swapping are:

- The program is too big to fit in memory
- The process has blocked for a significant period
- Space is needed for higher-priority jobs

With virtual memory, swapping of the entire process is often not needed. Sometimes it is enough just to remove one or two pages from physical memory, leaving the process still able to run. As long as the memory references actually required are resident in memory, the process can continue to execute.

What happens when the CPU requests a page that is not in physical memory? At this point the MMU throws an exception called a **page fault** . This will then cause the memory manager to load in the page from external memory. This acts like a type of interrupt handler. When a page fault happens, the guilty thread suspends and the pager daemon takes over, recalling the needed page from the disk. Sometimes this method is used when a process is first created and executed. The process must be loaded into memory, but since the program is stored onto disk, numerous page faults will occur until it is sufficiently loaded. This technique is called **demand paging** .

A major consideration for page fault handlers is which policy should be used to determine which pages to swap out to disk when space is needed.

- **Not Recently Used**

  In this policy, the pager swaps out pages that are resident but not accessed for the greatest time. This requires, however, that the pager keep tabs on when a page was used, as well as a fast method of searching through the vast page table to find times and compare them. An option for this might be a tree with most recently used pages towards the top (return a page to the top after use), pruning only the leaves (lowest nodes).

- **FIFO and Second Chance**

  Pages are swapped on a first-loaded-first-purged basis. This means that pages that are oldest are pruned first. In the case that a page slated for removal is actually commonly or recently used it can be skipped and given a **second chance** .

- **Clock**

  This algorithm is similar to the round robin scheduling algorithm (Section 3.4). Imagine all of the pages are arranged in a circle where the swapper

(synonymous for pager) goes from page to page like the hands on a clock. If this minute-hand of death comes to a page that is in use it skips over it and moves on.

- **Least Recently Used**

  This is similar to NRU (Not Recently Used). However, the pager swaps out pages that have been used the least since a certain point in time. This requires the pager to keep tabs of the times pages are accessed. This, and the search, both add to excessive overhead making this method very expensive.

- **Random**

  This is the easiest method to implement but it is seldom used. The pager picks a page at random to swap out. If the page is in use, or highly used, it can be skipped.

4. Page Tricks

Here are some useful data structures that are commonly used by a memory manager that works with pages.

- **Clean/Dirty Page Bits**

  When swapping out a page, the swapper must write the page that's in memory onto the disk. But what if the page wasn't altered while in memory? Why write it back to the disk if nothing is different? This is called a **clean page** . Only if the page is **dirty** would the pager bother writing it back to disk.

- **Pre-Paging**

Each process has its own set of pages that it uses in the course of a typical execution run. This set of pages is called that process' **working set** . Sometimes, the CPU spends a lot of time handling page faults to keep pulling in pages for this set. Pre-paging is a technique that involves loading the entire working set of a process before it is run, thus combining the entire loading-into-memory procedure into one operation, cutting down on the expensive page-fault calls during execution.

- **Page Protections**

  When pages are being shared or passed around, there should be a method to protect them. Also, having a means of informing other threads, and the pager, of some thread's intentions on a page would be useful. These intentions, also called **page protections** , can be labeled as read-only, read/write, none, locked, executable, etc. Read-only pages will not allow a process to write to it. Locked pages, also called **pinned** pages, are flagged to stay in memory.

- **Page Swap Allocations**

  There are two basic scopes for page allocations.

  1) **Local page allocation** : If a process needs a new page to be retrieved, one of its own pages is swapped out to disk make space for the new page.

  2) **Global page allocation** : If process `foo` needs a new page the pager swaps out any page (could be process `bar` 's page or process `baz` 's page, etc.) to make room.

  Local is useful because it allows for control of specific memory divisions among processes. For example, if processes `foo` , `bar` , and `baz` are each allotted 5 pages then they will always have 5 pages. If global is used then

the allotments for those processes will vary depending on those processes' individual working sets. So suppose process `foo` started requiring more and more memory. A global allocation system will give `foo` more and more pages, possibly taking them from `bar` and `baz` if memory is low. However, if local allocations were being used, `foo` would start faulting often because its page allotment is limited.

Mixing these two together can allow for dynamically controlling specific page allotments. If a process starts spitting out a lot of page faults, or its **Page Fault Frequency** (**PFF**) increases, then the memory manager could readjust the allotments globally while using local allocation to keep the processes within those new allotments.

- **Page Size**

  Page size is another consideration when designing a memory management system. If the page size is too small then the page tables get too big, adding to the management overhead, as well as storage of the tables. However, pages that are too large may result in wasted space because the pages may more often take up more space than the average process will need. The most widely used size is 4k (4096 bytes).

5. Segmentation

Segmentation is another useful tactic when approaching virtual memory. Instead of a process putting all of its information in one memory space, it splits that space into segments. These logical **segments** are individual virtual memory spaces, each with base address starting at 0. The way a programmer might use segments within their program is to put all of the variables in one segment during execution, all of the constant data into another, the stack into another, and the actual code into another, etc.

This definitive separation between the different memory spaces of a program is important: you would not put the *stack* in the *code* segment or put the read/write *variables* into the read-only *data* segment. An implication of this (and another one of the benefits of segmentation) is that each segment can have its own protections. For instance, the *code* segment would have an `execute` protection, meaning that segment's contents are read-only and executable. For this purpose you would not put the program stack in the same segment because the stack is *not* read-only and definitely *not* executable.

There are other many powerful things you can do with segments. One popular use is having each segment be a procedure, or a **library** of procedures. When you compile the program that uses these procedures you just link the segments. If you had a program that used these libraries and you wanted to change the code inside one of the libraries you could just edit the code in that segment, recompile it, and then link it back to the original program without having to recompile everything else.

These segmented libraries can also be shared (**shared libraries**) among programs. For instance, if you have a massive graphics library that takes a lot of memory and you have a couple programs that use it, you can just have them share the one library instead of them each having their own. This way you would only need one library loaded in memory instead of two.

In the beginning of the chapter, we talked about a process taking up one single chunk in memory and growing and shrinking according to how much memory it was using. Then we talked about paging as a way to keep these spaces uniform and easily manageable. Segments are not much different: just imagine them as *multiple* growing and shrinking chunks of memory within one process. We would just fill the physical memory with segments made up of uniform-sized pages. Each segment could have its own page table, and each process will have its own segment table.

6. Memory Implementation Standards

In order for memory to be usable by users, the operating system must provide certain functionality to them. Programming languages such as **C** need to have access to certain functionality so it can provide memory functions to programmers. The standard for this is called **ISO C** and calls for implementation of such functions as `malloc()` , `memcpy()` , `memcmp()` , `memmove()` , etc. These are functions for allocating memory, copying memory, comparing memory, and moving memory, just to name a few.

POSIX compliance not only calls for these C standards, but also its own proprietary functions such as `mmap()` and `memunmap()` . The function `mmap()` **maps** a physical object (such as a file or video buffer) to a virtual address so the program can access it like normal memory.

For more on both of these standards see (Definitions, "Memory Mapped Files").

7. O/S Examples

MINIX. **MINIX** is a minimized version of UNIX written by Andrew Tanenbaum. Because of its simplistic design it was intended for academic purposes to help programmers learn about writing operating systems. MINIX was also designed to be as **portable** as possible. This means that it can easily be implemented on different processors. This is important because different processors have different instructions built into their hardware, and thusly require different programming on any software that works directly with the CPU (such as the operating system or other machine-code programs).

Since MINIX is small and only meant to work with small processes on consumer machines, it keeps its memory management simple by using non-paging and non-swapping techniques.

First of all, memory is assigned by using **first-fit** . This means that when allocating memory for a process, the manager starts at the beginning of memory and scans along until it finds the first chunk of unused memory big enough to fit all that the process needs.

Another unique trait of MINIX is the fact that it implements the memory manager in user space! A user process determines where process memory is placed. This was done to allow for easy implementing of different allocation algorithms. To use something other than just first fit, you would only need to run a different memory manager rather than rewriting the kernel code. The mapping of process memory to physical memory, however, is handled in the kernel. The way that the user land memory manager talks to the kernel mapper is through typical **IPC** calls.

Memory is **allocated** when a process `fork()` s, copying the parent memory to the child's. Memory is "erased" (or released without necessarily destroying) and then reallocated when the child `exec()` s (`fork()` and `exec()` , see Chapter 2: Processes and Threads). Memory is then released when exiting or is killed by another program.

Windows. The basic structure of Windows' memory management (using the **Win32** standard) is fairly straightforward:

- Each process has its own **VAS** (**Virtual Address Space** , or virtual memory space), with 32bit addresses.
- The upper 2GB of memory is kernel info and is shared among all processes, but not accessible in user mode. Only when a syscall traps into kernel mode can the thread access that shared info. This may result in less data privacy, but means faster calls.
- The lower 2GB of memory is program code and data.
- Windows uses copy-on-write.
- Page protections are **free** , **committed** , or **reserved** (*pinned*).
- No pre-paging; strictly demand paging.
- Mixes local and global page allocations.

From here, though, the memory manager starts to get more complicated. Pages are shadowed, they can be shared, etc. There is also a 5-tiered physical memory manager. For more information, see (Windows).

FMI/OS. Memory management for FMI/OS is another area in which we chose to use an already-working and efficient model. The one we chose is the **UVM** used in the **NetBSD** operating system.

UVM Basics. The first thing worth mentioning about UVM is its separation between machine-dependent and machine-independent procedures. Getting full use of each MMU requires special considerations when programming. This specialized programming would not work on machines that implement theirs differently. The programs that have to make these special considerations are **machine-dependent** . However, much of the code managing virtual memory doesn't have to worry about the machine's specific hardware. Therefore, the VM manager is **machine-independent** .

UVM's machine-dependent code is called `pmap` . The `pmap` procedures are small and easy to **port** to other processors (i.e. re-program to make compatible with other computer architectures) to take advantage of different types of MMUs.

UVM also has other VM basics:

- Each process holds its own VAS information (page tables).
- Processes can map memory objects into their VAS.
- A pager (A daemon that runs in the background. Pointers to the pager procedures are given to each memory object. Therefore, if an object causes a page fault, all it has to do is call those procedures to handle it).
- Demand-paging is used.
- Prepaging is allowed
- Copy-on-write during process and thread forking.

But then, UVM has some extra features

(1) **Page Loanout** . This is a type of copy-on-write system for the IPC. Instead of copying messages from one process to another, the kernel "loans" the map over to

the other process, giving it read-only properties. If the other process chooses to write to it, only then does the kernel make a copy.

(2) **Scatter-Gather Buffers** . Instead of passing a single message from one process to the next, a list of messages (or "buffers") is passed. As each layer adds to the message, they would only need to add that message to the list. Therefore, the last process will receive a list of these buffers and will just write them all out at once. For instance, suppose a server is receiving a write-request message. This server will pass it on to an I/O server who will then add an **EOF** (End Of File) to the end of the list and write both buffers to disk, therefore not needing to copy the entire message to writable space just to add a byte of data. If the disk is using **DMA** (Chapter 8: I/O Layers), the DMA server will then "gather" all of the "scattered" buffers and copy them into one, contiguous buffer which it then sends to the DMA. If using a programmable I/O (which writes bytes to disk one at a time) then there would be no need for the last "gathering" copy. The function would just go through each buffer, byte-by-byte, and write each byte to disk.

(3) **Lazy Allocation** . This is a flag that a process can have set. On the rare occasion that the entire VM space is filled and a process is asking for yet more memory, then UVM has to make a decision on how to free some for it. If the offending process' lazy allocation flag is set, then the UVM takes the "lazy" route and kills that process. In the case that the offending process is not allowed to be killed (i.e. the flag's not set) then the pager daemon taps into its small buffer of reserve pages until it can kill another process.

There are many other specifics of UVM that are quite interesting. However, they fill a 270-page dissertation written by Chuck Cranor for his Ph.D. so I leave the research up to the curious reader. Cranor and Parulkar wrote a short abstract on UVM for the USENIX Tech Conference.

Applying UVM to FMI/OS. UVM has extensive functionality, but not all of it is needed in FMI/OS. UVM was written for NetBSD, which is a monolithic kernel, and FMI/OS is a client-server. For starters, the file-system implementation is in user space, thus swapping gets complicated. A temporary solution is that the swapping has been removed completely. Future solutions include putting a single pager daemon into user space. Another option is putting a pager daemon into the C library (that is included and compiled into every file that will run on the OS; see libc in Chapter 10: Environment Layer) so each program will have its own pager daemon thread that will just sleep until needed. Although threads do not need to know about this personal pager thread (since paging is handled automatically), it does allow threads that *do* know about it to employ specialized handling. Also, if the kernel were to be running low on memory, it could use its normal swapping algorithm to determine which pages to remove and just *signal* the process that owns the page to swap it out (see IRC discussion, 03-25-06).

UVM allows user processes to loan pages to the kernel itself, because that's where the drivers reside (i.e. disk I/O, printer, etc). FMI/OS's kernel does not implement any drivers inside of it so it does not need the loan ability. Therefore, to promote simplicity, it has not been implemented.

However, FMI/OS does implement writable loans. This means that the kernel's IPC will loan out a page with write permissions. For instance, if a program was blocking, waiting for a keyboard input, then the message it passed to the keyboard's interrupt handler is actually loaned so that the handler can write the key pressed to the message and return it.

For more information on porting UVM to FMI/OS see Amatus' UVM page on the wiki.

Chapter 6: Multiple Processors

(*"Foolproof systems don't take into account the ingenuity of fools." - Gene Brown.*)

Just when you thought it all sounded so easy, things start to get complex when you add multiple processors.

As computational tasks become larger, more complex, and more numerous, the demand for more computing power arises. With this, and the engineering limits of today, using more processors, or machines to do these tasks cooperatively is a common solution.

1. Multiprocessor O/S Configurations

On a **multiprocessor** machine, all of the processors share the same resources, such as memory and I/O devices. How these CPUs are managed varies according to the system's design. However, these methods fall under three general categories:

1. *Each CPU Has Its Own O/S.*

    This is perhaps the simplest of the three. Every CPU is running its own copy of the O/S with the memory divided among them. Therefore, each processor has its own execution table, process/thread table, memory manager, syscall handlers, etc. Although this successfully distributes tasks to multiple processors, there are some problems. First of all, if the process load for each CPU is unbalanced, then one CPU could be idle while another is working hard. This is a serious efficiency problem that could easily prevent the user from taking advantage of the multiprocessor system. Although memory could be dynamically assigned to different CPUs (in the case that

one CPU has to handle some memory-intensive processes, for example), it is hard to do this during run time because pages cannot be shared between processors.

2. *Master-Slave.*

   This model puts the O/S, and all its tables, onto one **master** CPU and distributes processes to the other **slave** CPUs. This allows for shared memory and better balance of process load on the processors. However, as the number of slave CPUs grows, the demand on the master grows and it will eventually become a bottleneck.

3. *Symmetric Multiprocessors (SMP).*

   With this model, there is also only one O/S and a common, shared memory. However, there is no bottleneck or process unbalance. All of the tables and other kernel info are shared. Also, each CPU may run in the kernel as well as in user space. As a CPU gets ready for a new process, it calls the common scheduler routines to assign it a new process from the common process tree. However, with all of this sharing of kernel structures, other problems arise. This leads into the next section.

2. Synchronizing CPUs and Lockouts

Let's assume a basic situation on an **SMP** machine where two CPUs want to call on the scheduler to assign them each a new thread to execute. Since executing a line of code is only a matter of changing the instruction pointer within the CPU and reading (only) from memory, executing shared memory can occur without any problems. However, when *writing* to shared memory, a type of synchronization is needed.

Mutexes. Any kernel must provide methods for processes to have **mutual exclusion** (abbreviated to **mutex**), which means only one process is allowed access to a shared memory or service at a time. Mutexes are also intended to prevent race conditions that can happen in processes' critical region (Section 4.2)

Semaphores.

> The primary tool for providing mutexes takes advantage of atomic actions. As mentioned in Chapter 4, an atomic action is a sequence of actions that cannot be interrupted. **Semaphores** are based off of the atomic actions `up()` and `down()` . These commands increment and decrement a given counter (called the semaphore). If the semaphore is 0 then the `down()` can't decrement anymore, so the thread calling it is blocked. It's not until an `up()` is called on the same semaphore (thus incrementing it from 0 to 1) that the blocked thread is allowed to proceed (**wakes up**). Therefore, a process would start off their critical region with a `down( foo )` and end it with an `up( foo )` .

Spinlocks.

> **Spinlocks** are also used to lock out other processes from a shared resource. The concept behind these is that there is a common variable that keeps track of whether it is safe to enter the critical region. When a process on one processor is accessing this shared resource, the common variable is set, stating that the resource is occupied, thereby *locking* it. Therefore, when another process, from another processor, tries to access that same resource, it notices that the resource is locked. As that process waits for the lock to be released it loops, or *spins* , until the lock is released. This isn't always the best solution because the processor, for the second process, is kept unnecessarily busy looping while waiting for the resource. However,

because of their simplicity, they are ideal for locking out short pieces of code that do not keep other processes waiting for long.

Commands in the scheduler are some of the functions that utilize a mutex. Although multiple processes (on multiple CPUs) are allowed to simultaneously *execute* the run-next function in the scheduler (since execution is a read-only action), they will eventually lock (or block) when they get to the part where they have to access the process table; this is the scheduler's *critical region* . Therefore, the first CPU to enter the critical region gets the lock while the other one must wait.

Some kernels handle mutual exclusion by locking the entire kernel and all its data. This isn't always the best option since some syscalls won't be using the same read/write data. For example, the IPC would not be accessing the process table. This is where **sectional spinlocks** come into place. These are locks that you place only on certain parts of the kernel data. For instance, there are locks for the page tables, IPC data structures, etc.

Locks, however, should still be used sparingly. In order to implement atomic instructions, some processors (such as the x86) lock the bus. This becomes a problem the more locks there are since every single mutex operation will lock the bus, hiccuping all the other CPUs (since the bus is what carries addresses and data, etc.).

One other possible problem that you must take into consideration is how to handle interrupts while locking within a critical region. These can cause major **deadlocks** , which are discussed in Chapter 7: Deadlocks.

3. Scheduling

There are three basic categories for multiprocessor scheduling.

1. *Time Sharing.*

   This isn't so much a modification as it is an inherent quality of the SMP model. With this, the run-tree is shared among all of the CPUs. Although

this has its timesharing and load-balancing advantages, it can slow down with many processors. Every time a process accesses the scheduler to request a new process, the process table is locked. This could easily cause back ups with the more processors that are added to the machine: if there are 128 processors and 50 of them are waiting in line for a new process then that's a waste of processing resources!

Something to consider when writing a timesharing scheduler is a CPU's **cache** . Every CPU has its own cache, which is very close, fast, private, and easily accessible memory (i.e. no bus) that the CPU uses to store temporary data (holding recently used pages, etc). With this in mind, if processor 1 was running process A, then if it runs process A again soon there's a good chance that A's data is still in that processor's cache, thus improving performance. Therefore, implementing this will improve the efficiency of Time-Sharing scheduling. One possible method of implementation is using a type of **two-level scheduling algorithm** . With this, the top level consists of a CPU, and the lower level consists of all of the processes attached to that CPU. So, whenever a process is created, it is assigned to a CPU.

2. *Space Sharing.*

Let's assume that a process has a group of threads and they're executed all at once. It is safe to say that these threads will probably have reasons to talk to one another, so having them run at the same time is a fairly intuitive jump. The **space sharing** takes these related threads and assigns one to each CPU. If a thread on one CPU blocks, then the CPU stops, or idles, until its thread is awakened. If its thread terminates, then the CPU is released to the pool of available processors that the scheduler can then assign to another group of related threads. The obvious drawback to this

is the abundance of CPU idle time. Also, although many threads can run at once, the number of CPUs limits the number of simultaneous processes.

3. *Gang Scheduling.*

   This method aims to combine Time and Space sharing. Groups of related threads are run at the same time, each on a different CPU. Every group of threads, typically grouped by process, is given a set quantum which all of those threads are required to adhere to. If a thread blocks, its CPU idles until the end of the quantum. By keeping the timeslices uniform, the threads that are likely to talk to one another will stay in-sync and be running simultaneously. At the end of every timeslice a scheduling decision is made and the next-up group of threads is run.

4. O/S Comparisons

Below are some descriptions of methods of managing mulitprocessors used by various operating systems.

Linux. Once again, this monolithic kernel takes radical performance hits trying to implement a particular functionality. Linux does use the SMP model. However, since the kernel has all of the drivers and services built into it, then there are a large number of locks that must be managed.

Linux also allows context switching when holding a lock. This is very deadlock-prone and requires special and complex programming to make up for it.

Because of all of the deadlock compensations as well as the shear amount of locks, Linux does not handle SMP well on machines with 16 or more processors.

FMI/OS. FMI/OS took a lot of its ideas from Plan9 on this topic, as well. It uses the SMP model with the timesharing scheduling algorithm. Because of its fixed-time, O(1)

scheduling, the possible scheduler bottleneck problem with numerous processors is solved easily.

Another interesting trait of FMI/OS's scheduling is its inherent affinity. When a thread blocks when sending a message to a server, the server is sent to the front of the list at its priority level, and so the odds of it being run next on that CPU are high. Also, if the server were to run, it gets the rest of the client's quantum on that CPU, a well as its own.

Other compensations FMI/OS makes for handling multiprocessor conditions are covered in the next chapter, Chapter 7: Deadlocks

Chapter 7: Deadlocks

(*Premature optimization is the root of all evil." -Knuth)*)

1. What Causes Deadlocks

Suppose process `foo` is trying to access the CD-ROM. Since only one process at a time can have the CD-ROM, there is a lock for it, and `foo` has that lock. Also assume that process `bar` is using (and has the lock for) the printer. Now, consider `foo` suddenly needs the printer. While still holding the CD-ROM lock, `foo` tries to get to the printer, but since `bar` has the printer locked, `foo` blocks, waiting for `bar` to release the printer. However, if `bar` has decided that it needs the CD-ROM for something, it ends up blocking to wait for the CD-ROM because `foo` has it locked. Therefore, both processes are locked indefinitely, waiting for a resource the other has locked. This situation is called a **deadlock** or, perhaps more descriptively, a **deadly embrace** .

> **Textbook definition:** *A set of processes is deadlocked if each process in the set is blocked, waiting for an event that only another process in the set can cause.* (Tanenbaum, 163)

When implementing SMP machines, since there are now multiple, simultaneous processes, and the requisite need for more locking, the possibility of deadlocks must be taken into consideration. An example is in sharing the run-tree. Let's say process A is killing process B. Process A puts a lock on process B and then has to go and put a lock on the entire tree so it can delete process B from it. However, before process A gets the tree lock another CPU running a command like `ps` (which lists all of the running processes) locks the tree in

order to traverse it. Therefore, process A blocks, waiting for `ps` to finish. However, when `ps` gets down to process B it blocks because process B is locked by process A. A deadlock has formed.

There are many more types of deadlocks, but they mostly fall under the general perspective of processes fighting for resources.

Resources. **Resources** are parts of a computer that a program uses, or is given access to. Resources come in all shapes and sizes: disk drives, printers, memory, mouse, etc. Whenever a process chooses to use a resource it must request it (lock it) either explicitly or implicitly, use it, and then release it (unlock).

Resources come in two basic flavors.

(1) **Preemptible resources** can actually be taken away from a process. An example of this is memory or disk access. Suppose a process has a lock on, say, a chunk of memory. Then, if the kernel desires (such as a to avoid a deadlock), it can just preempt that process' "lock" and page out its chunk of memory. The CPU itself is actually a preemptible resource!

(2) **Non-preemptible resources** are those that cannot be taken away from a process. An example of this is the printer. If process A prints half its job and process B wants the printer, the kernel can't just take the printer away from process A and give it to B because then the printout would be skewed. These non-preemptible resources are the biggest causes of deadlocks.

2. Deadlock Algorithms

There are two different ways to deal with deadlocks: prevent them from ever happening or catch them if they do happen. Actually, there is a third: ignore deadlocks altogether. This is purely a numbers game and gets riskier the more processes and resources there are. That aside, let's take a look at a few of the different active methods.

Detection and Recovery.

One method is to keep track of all of the resource requests and releases. If the kernel detects a circular trend that could lead to a possible deadlock, it could prevent the request altogether, or just kill the process. Another way of handling deadlocks is to just kill processes that have been blocking for a certain amount of time (like an hour).

These methods only work on machines where killing processes is actually acceptable (when done properly). Mainframes running batch jobs are such machines. If a process is killed then it can just be restarted later as long as its files, modified by the process before it was killed, were returned back to their original state.

Prevention.

There are a few ways to prevent deadlocks from happening. One possibility is to add a **spool** , or message queue, to certain resources that require mutual exclusion. An example of this is the printer. Jobs are queued to the printer's spool so no user process has a lock on the printer, itself. Instead, a daemon runs in the background, pulling jobs off of the spool and sending them to the printer. This daemon is the only thing that will ever need direct access to the printer so there are no fights for the lock.

Another option is to prevent processes from requesting new resources if they are already holding one or more. One way to accomplish this is to require process to release (temporarily or permanently) all of its resources before acquiring a new one. However, with processes that copy large amounts of data from one resource to another, this is not an option.

The other way to accomplish this is to have a process announce all of the resources it will use in the course of its execution, and then request

them all at once. This is difficult because it's hard (or impossible) for some processes to know all of the resources they will need, ahead of time. Also, locking out numerous parts of the machine, at once, will block a lot of other processes.

Banker's Algorithm.

Avoidance of deadlocks can also come from strategic scheduling of processes. A widely-used algorithm is called the **banker's algorithm** . This algorithm works similar to how a banker in a small town would handle lines of credit. The basic structures to allow for this algorithm is for each process to announce how many of what resource it's using, and the maximum amount it would use throughout that processes' full execution. For instance, if process A copies information from one tape drive to two others, then it has in its table *three* as the maximum number of tape drive resources it will need. Let's say, though, that process A is only using one, at the moment, so it also puts in its table that it is *using* 1 tape drive resource.

The scheduler will also need to know how many of each resource the machine has. For instance, let's assume that the machine running process A has 10 tape drives. Now, let's assume that the other processes' tables contain the following:

Process	**Using**	**Max**
A	1	3
B	2	4
C	2	5
D	3	8
========	=======	====
Remaining :	2	

So the "banker's" goal is to ensure that there are enough remaining tape drives to accommodate at *least* one process's max-needed to make sure the "bank" (i.e. system) doesn't go bankrupt on resources. In the case of the above table, there are enough tape drives left to satisfy process A or B if either of them cashed out and used its max. Although process C couldn't, it could wait until A or B finished their run and release their resources. Enough tape drives would be available, then, for process C to do its thing.

If process D was given one more tape drive then this table would claim the state of the machine unsafe (i.e. deadlock possible) because *none* of the processes would be able to be accommodated if any were to try and cash out. Therefore, the banker would not allow process D (or C) to claim any more resources.

To implement this simplified model to a system with many different types of non-preemptable resources, the scheduler would just keep two tables: one for resources being used, and one for max resources. Rows of these tables would consist of processes, and columns would consist of resources. The banker's job would be to ensure that at *least* one process can be granted all of its max resources.

There are two major drawbacks to this algorithm, though. First, it only effectively works on machines with multiple numbers of certain resources. On machines with one CD-ROM, for instance, if two or more processes exist on the run-tree that list, even just 1, as their max CD-ROMs needed, then neither of them will be allowed to access it. Also, it is nearly impossible, as stated earlier, for some processes to know their max resources needed (i.e. those processes that rely highly on user input).

FMI/OS:.

The number of possible deadlocks to compensate for is reduced dramatically on a client-server system. In FMI/OS, there are basically three:

Re-Entrant Interrupt Handler.

Let's assume that an interrupt handler has locked its subscriber list as it goes through and moves all of the processes from `ready` to `pending` . Then, while locked, the interrupt happens again. Thus, the handler is called again, starting over from the beginning. However, when it arrives at requesting the lock on its subscriber list it is blocked because the list is already locked. And who has the lock? The handler does, but it doesn't know it! Since the scheduler doesn't control the handlers, there was no context switch, so there's no way to return to the previous running instance of the handler to release the lock. A deadlock is born.

In an SMP system, two CPUs can simultaneously run an interrupt handler. One will just block while the other has the lock. A deadlock only happens if a single CPU is re-entering the interrupt it was just handling.

Fix: The easiest way that FMI/OS avoids this is to turn off interrupts on that CPU during a lock. We use something called **auto locks** that are locks which automatically turn off interrupts on that CPU within the duration of the lock. Auto locks also increment a counter keeping track of the number of locks being held on that CPU. With this, `run_next()` will throw an assertion when trying to context switch if the CPU calling it holds any locks (i.e. `locks_held` $>$ `0`).

Lock Releasing.

Another possible deadlock comes as a result of just bad programming. However, it's still worth mentioning because it is a common mistake. Some programmers make the mistake of allowing a program to grab a lock but not release it. Having these random locks floating around that will never be released could cause catastrophic deadlocks. Auto locks also prevent this from happening.

Fix: Although auto locks will prevent the deadlock, you should still always make sure to release all locks when writing code!

Lock Order.

The final deadlock FMI/OS must watch out for is the second example given in this chapter (the one with killing processes vs. `ps`). This deadlock is also a result of bad programming and can easily be prevented with proper ordering of grabbing the locks.

Fix: If a process wants to grab a lock on a thread in the run-tree (i.e. a leaf node of the tree) then it needs to grab a lock on the whole tree, *first* , before grabbing the leaf lock.

Part II: User Space

Chapter 8: I/O Layers

To take the discussion into User Space, an overview of the layering of system resources used in Input/Output will be made. Data flows through these layers as they make their way from the devices to the user, and vice versa. This will help set the stage for the chapters on file systems, environment layer, and security.

Layer 1: Device

This is the bottom-most layer consisting of the hardware device itself. Devices come in two basic flavors.

- **Block devices** do their reading and writing in blocks, such as on a disk.
- **Character devices** read and write as a stream of characters, such as on a network adapter or keyboard.

Layer 2: Device Controller

The device is electronically controlled by a **device controller** . The controller is typically built into the hardware of the device and has its own registers and data buffer used when dealing with the device. Controllers are programmable devices, their control language being specific to the device itself. The control commands may be standardized but often will differ according to the device manufacturer.

Layer 3: Kernel Control

The controllers are assigned I/O **ports** (for the IPC) or are often memory-mapped by the memory manager. E.g. data may be written to the device by "pretending" to write to ram at specified addresses (Section 5.6).

This layer is also where the kernel starts stepping in. One of the basic jobs of the kernel is to provide a layer of abstraction from the hardware to the software. It must also make sure that this layer is as device-independent as possible, for the sake of the users and of portability. This requires a method of uniform naming for these I/O ports, and uniform access. The kernel must also determine whether it implements *synchronous* access (where it blocks requesting threads while it's handling their request) or *asynchronous* access (where it allows the requesting thread to continue to run and notifying it, by way of interrupts, that the request is done).

This is also the layer in which the OS may implement any buffer caching.

There are different methods in which the kernel can deal with sending information to the I/O device:

(1) **Programmed I/O** . In this method the kernel sends data to the I/O device one buffer at a time. To do this it must busy-wait while polling the device to see if it's ready for the next datum, much like a spinlock would do. Also, like a spinlock, processor time is consumed in a waiting loop.

(2) **Interrupts** . If using this method, the kernel copies the data from the user to some temporary buffer. It then blocks the requesting thread and schedules as normal, leaving the device interrupt handler to copy the buffer, byte-by-byte.

(3) **DMA** . This method is the same as using interrupts except that the kernel hands the buffer address to a DMA controller and gets only ONE interrupt from the DMA saying it's done sending the buffer, in full, to the device. **DMA** stands for **Direct**

Memory Access and is a hardware layer between the device and the kernel that is sometimes part of the device controller or the CPU.

The kernel then provides a uniform standard for writing to the I/O, using such straightforward calls as `read()` and `write()` or redirections, such as < and > , that are commonly used in POSIX environments.

Layer 4: Device Driver

This layer consists of the **device driver** software. The driver is specific to the device (**device-dependent**) and knows how best to use the OS's access to the device, in accordance to the distinct device.

Monolithic kernels.

Since the drivers are in the kernel, the kernel must be recompiled to include any new device if one is added to the machine (some monolithic OSes have a modular device plugging system, like Linux). This, however, leaves the kernel vulnerable to faulty drivers. If a driver was to crash or misbehave, the entire kernel could shut down, leaving the computer incapacitated or worse, damage permanent data files.

Microkernel and Client-Server models.

With the drivers in user space, driver code is isolated from the kernel. Therefore, if drivers misbehave, will do so in their own space and not take the kernel with it. These user space drivers get their device access via syscalls and IPC.

Although these drivers are obviously device-dependent, their exported API must be device-independent and meet a certain functionality standard (i.e. Win32 or POSIX) for the next layer to use.

Layer 5: Device-Independent software

This layer bundles different I/O calls together into uniform/standard interfaces for programs to use. For example, writing to the printer is a similar interface as writing to disk and as writing to a network interface. This layer also takes care of uniform naming, protections, blocking (if the kernel doesn't automatically do it), buffering, allocation/releasing, error reporting, etc.

Layer 6: User-level software

This is the final (topmost) layer where all device access is now uniform and abstracted. Users can program using system calls, or useful libraries that bundle different calls together for improved functionality. Functions such as `read()` , `write()` , `open()` , etc., are part of this layer.

Now, with this generalized look at the I/O layers of a typical machine, we move on to File Systems.

Chapter 9: File Systems

(*"rm -rf /bin/laden"* - *Anonymous*)

The topic of file systems actually incorporates many subjects that haven't been discussed yet. And, being the first chapter of discussion for traditional User Space, this chapter must cover quite a bit of ground. Physical files, those stored on external media, will be looked at, first. Then, we will take a look at basic input/output (**I/O**) concepts (which are used with many things, not just disk I/O). Afterwards, we will start to discuss cases when the file system as a whole deals with "files" that may not be the physical variety (i.e. file representations of devices, running processes and their information as a file, etc).

1. Physical File Basics

Data stored in RAM and cache are only there when the computer is on. The moment the computer is turned off, this data vanishes. This memory is considered **volatile** . Hard drives, flash cards, CD-ROMs are all examples of **non-volatile** media. Any memory stored on these will stick around even when the computer is shut off. Data stored on these mediums may take the form of files.

Files come in different shapes and sizes:

- **Regular files** include text and binary files. Text files can be edited or otherwise opened to reveal readable text. Binary files are those that are not text. These include archives (compressed files such as JPG or ZIP or TGZ) and executables. Binary files have some sort of internal structure set by a particular standard. JPG files adhere to the Jpeg standard, MP3 files to the Mpeg2 Layer III standard, and

executable files to the ELF standard, see Trac Ticket #36 or "Executable and Linkable Format (ELF)" in the Bibliography.

- **Directories** of files are often files, themselves. The information they contain is a list of what files are in that directory.
- **Character-special files** are related to input/output and are used to model serial I/O devices such as terminals, printers, and networks.
- **Block-special files** are used to model disks such as floppies and CD-ROMs.

Accessing data inside files can be done in two different ways: sequential and random access. **Sequential access** is exemplified by tape drives where data can only be read and written in sequential order. If you wanted to read some data from the middle of the file you had to start from the beginning of the file and scan through to the middle. Hard drives and CD-ROMs are an example of **random-access** devices where you can read and write to any part of a file without having to start from the beginning.

To implement security, files are assigned attributes that the file system driver must strictly adhere to. These attributes include not just date, but also protections (similar to memory protection attributes: read-only, read/write, etc) and **permissions** (who can do what to the file, Chapter 11: Security).

The next issue is how file and directory structures are designed and allocated.

File Structure. There are a few different ways to implement file allocation. One way is to allocate data *contiguously* on the device, where the **blocks** (think: pages; see *Managing Disk Space* , below) containing the file are stored in contiguous memory. This only works if a file's max size is known ahead of time and it doesn't change much. Otherwise, serious **checkerboarding** (gaps between chunks of data) can occur as these files shrink, or are deleted, much like some of the early memory models discussed in Chapter 5.

Another allocation method is using **linked lists** . With this, file blocks are linked to one another (i.e. file block 1 has a pointer at the end of it that points to file block 2, etc).

This allows for files' blocks to be placed randomly on the disk. However, random access is difficult and time consuming with this method because the reader will have to start at block 1 and follow the links from one block to the other, turning it into a sequential-data model at run-time. However, keeping a **table of linked lists** for allocation would expedite random access. This table would contain all the links for the list. So, when attempting random access, instead of traversing through each block just to get to a single block in the center of the list, you can jump directly there using the link in the table. Since the table is held in memory by the file system driver, the seek can occur without making any disk references. However, keeping this table in memory takes a lot of time to load, and a lot of space to store.

The third, and most widely used, file allocation technique is using **i-nodes** (index nodes). Each file has an associated little table, called an i-node, that contains some of the attributes of the file and a tree of addresses to all the blocks that the file uses. If the file is large, then one of the entries in the i-node table is the address of a **single indirect block** , which contains addresses of more blocks of pointers. If the file is even larger, then **double indirect blocks** point to single indirect blocks, which point to single blocks, etc.

Directory Structure. The basic structure of most file directories has a **root** directory. This is the directory at the top of the hierarchy. All directories are contained within other directories, which are contained within this. Some file systems denote `/` as the root directory (POSIX systems) and some use \ (MS-DOS and Win32). Let's say you're on a POSIX system in your home directory, i.e. `/home/dimitri/` . Every file you attempt to access will be looked for within this directory. This is called your **working directory** (or **current directory**). A file in this directory, say `notes` , would actually have a full, **absolute path name** of `/home/dimitri/notes` . You can access files in any directory (not just the current one) using the absolute path name for that file.

Directory nodes are typically treated like files and are stored under the same name. They have the same types of attributes, and their data are actually lists of the files that are inside that directory.

Managing Disk Space. When thinking about disk space management, many of the same concepts that dealt with (volatile) memory management apply. Instead of pages, file systems use **blocks** . These blocks are kept at a uniform size, which simplifies moving and managing. As with pages, the choice of block size requires careful consideration, as well as how to store a list of the free blocks.

Disk drives do go bad, too. With older disk drives, a software method of keeping track of blocks that have gone bad on the drive is a necessity. However, modern disk drives have become more and more reliable with their own built-in bad block management systems. Therefore, this topic won't be discussed here.

Since disk references take so much longer than memory references, keeping a memory cache of recently used file data is useful and is implemented in different OS's. Many algorithms can be used for managing this; most of them are similar to those discussed in Chapter 5 such as LRU and FIFO.

2. Orthogonality

This is the name applied to the concept that everything is a file, not just physical files residing on a disk. This includes file representations of:

- devices (from UARTs to network interfaces)
- networks, protocols, services
- control and debugging of processes, system management, service discovery and naming (namer)
- graphics/video
- permanent storage structures (on disk, flash, mem, network)

This was actually a fairly wild concept when it was introduced, but now it is standard in many file systems (e.g. POSIX systems.) However, most, like UNIX and Linux, only implement this to a degree. For instance, their `dev` directory may show devices as files, but those "files" are just gateways to the actual block or character devices and can't be traversed with normal filesystem calls, such as `cd` , `ls` , or manipulations, such as those made with `cp` (copy) or `cat` (concatenate).

FMI/OS takes this orthogonality concept to the true definition. Below, in section 4, there is an example of one of its servers, `namer` , being accessed through the file system that illustrates the power of this concept.

3. Monolithic OSes

MS-DOS. On disk, DOS uses linked list table file allocation and avoids large tables by using large (32k) blocks. It also dynamically loads drivers into the kernel when executed.

UNIX.

File System Structure:

> Files in UNIX are stored using i-nodes. Directory files are just specially identified files that contain lists of all the i-node#/filename pairs of the files in that directory.

I/O:.

> Since drivers are compiled into the kernel, the entirety of the kernel must be recompiled every time a device is added. Linux provides loadable modules that can be loaded at runtime to prevent this, but they are still used by the kernel, directly; this does not help the vulnerability factor of the kernel to the drivers.

These resources are accessible through the file system via a visible path name. For example, access to the primary hard drive is through the file `/dev/hda` . This is the "name" given to the driver that controls the device. User-level software (*Level 6*) may speak to the hardware device through this file name. Although this concept simplifies communication with drivers, the communication among these drivers, and drivers to devices, take place in the kernel in the form of syscalls. UNIX implements 300 syscalls and counting, actually.

4. Client-Server

Plan9. In Plan9, device drivers are servers. All the kernel does is direct the messages from the clients to the appropriate servers (via the IPC). Since all the devices just look like servers to the programs then there is a true uniform interface to all resources.

Disks, graphics, security systems, etc., are all servers. A convenient implication of this is **network transparency** . This means that the introduction of a network is *transparent* to the system, i.e. remote computers or services appear local to users. This is because the IPC can send messages to processes across a network as easily as it can to local processes. So if a client wanted to access a disk drive that was actually on another computer, it would send a message to its server. The IPC then takes this message and sends it over the network to the appropriate server on the appropriate computer. This is possible because the kernel has its own **NIC** (Network Information Center: DNS server, etc) and **TCP** implementations.

Using different types of mappings and bindings each program assembles and computes its own **namespace** (environment in which it runs). Here are some common syscalls that Plan9 provides to facilitate this:

- `bind()` makes one file, or directory, appear as a different name. This is similar to a link usually created by the POSIX `ln` command, but this can do **unions** . For

instance, `bind(“/user/dimitri/lib”, “/lib”, MBEFORE)` will enable all searches into `/lib` to, not only look through the original `/lib` but `/user/dimitri/lib` as well. The `MBEFORE` flag tells the program to search through the latter first.

- `unmount()` reverses a `bind()` .
- `mount()` is like `bind()` but actually makes an entire file system appear under a different directory name.

FMI/OS. FMI/OS uses all of the Plan9 file system techniques mentioned above with only a few changes.

One of the key differences is the implementation of **Asynchronous Buffer Caching,** or ”ABC”. This useful tool makes synchronous I/O reads act like asynchronous reads: instead of blocking for sending a buffer at a time, the ABC blocks the thread while it reads the whole block. The ABC also reads a certain number of blocks ahead in anticipation of another read request. It stores these blocks in a linked list whose nodes are pointed to by the nodes of a hash table.

When looking for a block, if it exists in the table it moves that node to the top. If it doesn't exist it creates a new entry (i.e. read the block from the disk and store in the buffer) and places the node at the top of the table. This method provides the ABC an inherent implementation of the **LRU** (Least Recently Used) paging algorithm.

A prime example of our use of orthogonality can be seen using the `namer` service (Chapter 4). Suppose there was a server, `foo` , that wanted to be registered as a server, in order to open itself up for receiving messages. `Foo` would first have to ask the kernel for a port number, let's say it gets the number 1234. Then, as do most programs, `foo` would send a message to the namer requesting to be “registered” with the port number 1234. The namer will then mark `foo` down in the lookup table with 1234 as its port number.

Now, there is an alternate way to "register" `foo` with the namer. Since `namer` is a service, it has its own directory, `/namer` . So, if you were to create the file in that directory `/namer/foo` , and put 1234 in it, the you would have effectively "registered" `foo` with port number 1234.

Chapter 10: Environment Layer

(*"Smith & Wesson - the original point and click interface."*)

The **Environment Layer** of a computer is the final level of abstraction between the computer and the user. The environment is typically customized and geared towards the user's specific activity on the machine. Some are proprietary to the device (such as in embedded systems) and some are designed to meet a standard. Linux is an environment layer meeting the POSIX standard; Windows XP, Windows 2000 and Windows NT are all environment layers that meet the Win32 standard.

1. Responsibilities

The environment layer has many responsibilities, tying together all of the kernel and user level functionality into a convenient, machine and kernel independent standard interface. This allows a program to run on all of the operating systems that meet that particular standard, regardless of the type of processor or machine it's running on.

For instance, suppose you have one of your old favorite games, "The Incredible Machine," which you ran on Windows 3.1 on an old 3x86 machine many years ago. If you were to slide that disk into a newer computer, running on a Pentium or Athlon chip, with Windows XP, it would still run! When the game was written it used commands that conformed to the Win32 **API** , or **Application Programming Interface** , standard. Ancient Windows 3.1 (with the Win32 patch) and the modern Windows XP both conform to that standard. The Win32 environment layer will take the commands from the game, filter them and translate them appropriately on the current Windows kernel version and computer type.

Adhering to this standard isn't just a manner of accepting the appropriate commands from the programs; it's also about providing a user interface. DOS had a command-driven user interface much like UNIX. Windows is a **graphical user interface** (**GUI**) much like the MacOS and Linux' X Window System.

During development of FMI/OS, the environment layer of choice is that of a command line POSIX standard.

2. POSIX

POSIX systems require implementations and accommodations of many different C programming language calls and functionality. This is one of the reasons why they are used frequently by developers and programmers. To achieve this, POSIX systems have **libc** libraries. These are OS-independent libraries that provide such C commands as `malloc()` and `fork()` .

User programs very rarely make direct syscalls. Instead, they use higher-level commands (functions) that are part of a library such as the `libc` library. Programs, shells, and command-line commands can invoke libraries.

Many popular routines found in the libc library are:

- `termcap` routines (contain a database of the detailed compatibilities of hundreds of different **terminals** , which are monitor and keyboard interfaces) `http://www.delorie.com/gnu/docs/termcap/termcap_1.html`
- `regexp` routines (POSIX- standard pattern-matching library, provide commands for searching strings or text for certain patterns) `http://en.wikipedia.org/wiki/Regular_expression`
- `curses` routines (used for coloring, highlighting, moving, etc., text on the screen)
- `math` routines (common math routines, such as: `log()` , `sin()` , `sqrt()` , etc.)
- other miscellaneous helper routines
- routines that emulate POSIX system calls

The library bundles come in different flavors:

- ‘glibc‘:
  - GNU C library; most widely used and comes with all Linux distributions
- ‘uClibc‘, `newlibc` , `dietlibc` :
  - provides much more compact `libc` s for use in embedded devices.

3. FMI/OS

FMI/OS’s library structure is organized in such a way as to increase flexibility in the environment layer.

- `libfmi`

- sits right on the kernel
- Contains FMI/OS-specific routines for efficient use of kernel functions, such as message-passing and memory management, etc.
- `msg_connect()` , `msg_send()` , `t_fork()` , etc (see FMIOS Syscall API)

- `libposix`

- uses `libfmi`
- wrapper routines that create POSIX-like interface and syscalls which most `libc` s expect.
- some POSIX syscalls are wrappers of FMI/OS syscalls, but some (in regards to disk I/O, etc.) are actual wrappers for IPC calls.
- `read()` , `write()` , `fork()` , etc.

- `libc`

- uses `libposix`
- contains the core C libraries for use by all programs.
- `math.h` , `pthread.h` , `sched.h` , etc

There are two great advantages for this hierarchy. First, whereas FMI/OS contains **General Public Licensed** (**GPL**) code, `libfmi` is licensed under the **Lesser GPL** (**LGPL**) so that non-GPL applications can link to it. (For more GPL information see Chapter11OpenSourceDevelopment). It is essential to not limit the number of programs that can use FMI/OS.

The avoidance of licensing issues is also essential for the 2nd advantage of the library hierarchy: flexibility of environment layers. What this means is that many other environment standards can be implemented other than just POSIX. For instance, a `libwine` library can be created to create a Win32 environment, thus allowing Windows to run on an FMI/OS kernel.

Chapter 11: Security

(*"Those willing to give up a little security by using a little obscurity deserve neither security nor root privileges." - B. F.*)

Not everyone wants their private information accessed or altered by others. Were this allowed to happen, credit card numbers and other identity information could be stolen.

In much the same way, not every process wants their "private" information accessed or altered by any other random process. Were this allowed to happen, then system integrity would be compromised.

For these reasons, security measures are a must.

1. Basic Security Measures

Security within memory is covered via protections. This is discussed in Chapter 5: Memory Management under Page Protections. These protections basically allow processes to protect their memory by terms of non-readable, read-only, non-executable, etc. The kernel must honor these privileges to ensure system integrity.

Security among other objects, such as files and processes, is usually handled using Permissions, ACLs, or Capabilities.

Permissions.

> These are used in UNIX and Linux. Users each have their own user ID, or **uid** . There are also groups that users can be part of. There are usually groups such as `audio` or `cdrom` , which allow all members of each group

access to those resources. Therefore, each user has their own uid and a list of **gid** s (group ID) of which they are members.

The i-node of each file in UNIX has its own uid and gid, as well. The uid is of the file's owner, and the gid is whatever is assigned to the file. Also in the i-node is a 9 bit list of **permissions** . These permissions are readable, writable, and executable (`r` ,`w` ,`x`). The 9 bits are divided into three groups of three. The three bits in each group stand for `rwx` , in that order. The first group of three represent the permissions of the *owner* . The next three are for the users that belong to the file's *group* . The final set of three represent the permissions for everyone else, or *other* . This owner/group/other system is part of the POSIX standard.

Take the following files, for example:

```
rw-r--r-- 1 dimitri users 515 Aug 1 2005 notes
rwxrwx--- 1 dimitri users 26 Nov 22 2005 ocg
```

Both files are owned by the user `dimitri` and are part of the group `users` . The file `notes` allows the owner, `dimitri` , to read and write to the file. However, other members of the group `users` , as well as any one else for that matter, can only read the file. The file `ocg` is executable, but only by `dimitri` and anyone in the `users` group.

Access Control List (ACL).

An ACL is a list associated with each object. These are lists of different users and groups and their permissions to that object.

File	Access Control List
`notes`	`dimitri, users: RW; bob, coders: RW`
`ocg`	`dimitri, users: RX`

With this table of ACLs, the file `notes` provides the same permissions to `dimitri` and `users` as it did in the Permissions example, above. However, it also provides these permissions to the user `bob` and the group `coders` .

Capabilities.

In contrast to permissions and ACLs, **capabilities** are not contained within i-nodes, or otherwise referenced by the object being accessed. Instead, the situation is vice-versa. A process requesting the access must obtain a capability, first, and that capability is what allows references to the requested object. The requesting process does not have direct access to the requested object.

A capability can be thought of as a special admission ticket that a process uses to access what it needs. It can *use* this ticket (capability), *delete* it, or *give* a copy to another process of equal or lesser privilege.

These "admission tickets" can actually contain numerous capabilities at once, one for each object that process will ever need to access. These capabilities are stored in a list called the capabilities list, or **C-List** . Each capability in this list includes the type of object it's regarding, the rights to that object, and the pointer to the object, itself. Take the C-List below.

C-List1:

Type	**Rights**	**Object**
File	`RW-`	Pointer to `notes`
File	`RWX`	Pointer to `ocg`
Pointer	`R--`	Pointer to `C-List2`

A process could use this C-List, for example, to read or write to `notes` . The process could also use the third capability in this list to link to another C-List (`C-List2`), thereby adding on to the total available capabilities.

2. FMI/OS

FMI/OS currently uses a hierarchical ID model in conjunction with permissions. For instance, a user could have multiple uids and gids. Therefore, access to an object is the sum of access gained by each ID held (Valencia, "VSTa:").

This method was the original one used in VSTa (and was grandfathered into FMI/OS). It has many limitations, especially in regards to combinations of permissions on an object. Also, in the micro-kernel and orthogonal environment, a capabilities system would probably be more functional and secure. This is still in the hypothetical stages and no design or implementation decisions have yet been made for FMI/OS.

Part III: Open Source Development

Open Source Development

FMI/OS is being developed as **open source** , under the **GPL** (General Public License). What this means is that its source code can be read, and modified, by anyone, as long as the original authors are credited and all changes are documented. The idea is that by extending the availability of design and code documents, we would be broadening our access to more new and creative innovations without being restricted to developers that required financial reimbursement.

Not only does this provide for the potential of better, tighter, and more efficient programs, it also makes the art and science of computer programming more accessible to the common people. For those that do not want to, or can't, contribute to the actual writing of the program, they can still have access to, and learn from, the source code.

To learn more about the open-source and "Free is for Freedom" movement, see the GNU home page (see GNU in the Bibliography). They are one of the forerunners, as well as the creators, of the GPL.

1. Communication

When working on a project together, communication among the developers is key. This ensures that everyone is on the same wavelength and that code and concepts remain compatible and all work is towards a common goal. The FMI/OS development team is scattered throughout the world: USA, Germany, UK, Finland, and Sweden. They remain in contact via three means:

IRC Channel.

Perhaps the single most common and important form of communication is done on an **IRC** (Internet-Relay Chat; a chat-room-like protocol) channel. The two channels most commonly used are `#fmios` and `#ocgnet` on the irc.freenode.net IRC server. On these channels all developers can log on and talk to one another in a public forum where anyone can read all messages. Rarely are conversations about the project made outside of this common "room" because any and all discussions are logged and could benefit other current, and future, developers.

FMI/OS Homepage.

As a center of operations, the FMI/OS homepage offers a place to house news, archives, logs, and other developments. The homepage also provides a **wiki** interface. A wiki is a webpage that can be edited by anyone who can browse it. For example, since this thesis is on the wiki, I can edit it by logging in, going to the page that contains the chapter I want to work on, and clicking the 'edit' button. An editor window comes up which allows me to edit the text directly online. Anyone with an account can do this, as well. Although it is unlikely that anyone with an account has any malicious intent, were they to edit any of the thesis, there are still ways of tracking changes made to the page and revoking any unnecessary changes.

The use of the **Trac** wiki system provides a slew of useful tools. Any large-scale software development project requires a structured set of goals as well as issues that need be resolved. Trac provides a [trac:Roadmap Roadmap] to track issues (called **tickets**) and plot the goals of major steps (called **Milestones**) along the way to completion.

Source tree.

The final means of communication is through the **source tree** , itself. This is where the source code for the project is stored. This is discussed in the next section.

2. Coding/Submission

The source code is currently available on the website and anyone can find more information on downloading and installing it by reading the page on Getting The Source. Someone could also use an **SCM** (Source Code Manager) such as `bzr` to download the full source tree from other developers.

To prevent developers from stepping on each others' toes, each volunteer is assigned to certain tasks, which are called **tickets** on Trac. Each developer is responsible for working on, and submitting patches solely for, that specific ticket. Typically, working on these issues takes a lot of work, so developers have their own branches of the source code that they work on and test, independent of the primary (stable) branch.

As developers fix bugs or solve issues inside the source they can apply their fixes as **patches** to the main, working, **stable** branch of the source code. The concept of organizing, testing, and approving these patches is **patch-queue management** , or **pqm** . Once they pass the pqm process, patches are sent to the SCM. The job of the SCM is to merge a patch into the source code, line by line.

Here's an overview of the process:

(1) A developer works on his prescribed issue on his private development branch of the source code.
(2) Once the developer has accomplished his task, he documents all of the necessary changes needed to the source code.
  - These changes, also called **diffs** , are what make up the actual patch.
(3) The developer sends the patch to the system handling the pqm process.

- The patch is actually an attachment to a **pgp** -signed e-mail, addressed to the pqm system.

(4) Patches are sorted chronologically into a queue.

(5) Each patch is applied to a copy of the stable branch for testing

- A thorough automated test is conducted on the test source: bootup, message-passing, multiple users, I/O, successive & recursive filesystem scans and accesses, etc.

(6) If *any* part of the test fails the patch is rejected and the developer who submitted the patch is notified of the specifics of the failed test.

(7) Successfully tested patches are put to a democratic voting process for submission to the SCM.

- Voting can be made by all users and developers.

(8) Primary developers have over-riding vetoes on all voted decisions (in case a patch may compromise future development plans)

(9) Any patch that is approved will then be submitted to the SCM for assimilation into the final, stable source code.

- The SCM reads the diffs in the patch and changes the source code appropriately.

The stable branch is the primary branch for users and new developers to download from the website or from the SCM. In addition, when someone is working on an issue on their own branch, they can choose to solicit the location of their branch ("publish" it) to the SCM or to other developers to allow them to download and experiment with it.

Part IV: Conclusion

Conclusion

1. Summary

The open source project FMI/OS is a POSIX-compliant, microkernel-based, client-server OS modeled after QNX and Plan9. Below are some of the highlights of the implementations of key components.

Bootup Procedures.

> Booting a microkernel takes extra help from the bootloader, as opposed to monolithic kernels, which handle most things on their own (within the kernel). FMI/OS requires the bootloader to pre-load a few modules into memory and pass the addresses of those along to the kernel in the form of a multiboot header.
>
> During boot time, the kernel takes these modules, which are actually user space servers, and runs them to aid in the initialization of certain key services such as an IPC translator, disk driver, and filesystem server.

Scheduling.

> The FMI/OS scheduling API deals strictly with threads. It establishes a Preemptive Cooperative Multitasking Environment where scheduling events are only called in message passing routines, interrupt handlers, or other semaphores. They utilize POSIX Realtime-compliant round robin and FIFO techniques on a specialized priority class data structure. Accessing

the processes in this list occurs with a constant-time algorithm, regardless of the number of processes the list contains.

Badness and goodness factors are incorporated to allow an accordion effect of threads to prevent stagnation. Also incorporated is priority inheritance within child process (and thread) creation *and* message passing.

Internal Communication.

An IPC system is used for passing messages between clients and servers, and between servers, themselves. Servers are assigned port addresses which are handled with the translator `namer` . Message passing functions are atomic, within the kernel, to maintain the integrity of their critical regions.

Shared interrupts are used, with each having two subscriber lists: `ready` and `pending` . An interrupt handler would use ISR commands to send messages through the IPC to the subscribed servers residing in the handler's `ready` list. Servers that have already received a message from the handler, but have not yet responded, are in the handler's `pending` list.

Memory Management.

Memory is managed by a system similar to NetBSD's UVM. This employs a functional and portable API that's layered based upon machine-dependent and machine-independent routines. Each process has their own VAS, and copy-on-write is also often utilized.

Porting the UVM from a monolithic kernel to a microkernel has resulted in a culling of different functionality. As a result, the pager has been temporarily removed with plans to brainstorm more effective ways to implement it.

Symmetric Multiprocessing.

An SMP model has been used along with a timesharing scheduling algorithm. Due to the constant-time traversal of the run tree, heavy bottlenecking is avoided. Through the use of auto locks, which turn off a CPU's interrupts during a lock, re-entrance of interrupt handlers can be avoided. Other deadlocking is avoided by careful lock ordering.

I/O and Orthogonality.

One of the most important concepts that FMI/OS borrows from Plan9 is orthogonality. This means that everything, from physical files to devices to servers, are represented as files. With this, traditional manipulation of files and directories within the filesystem can serve as an interface to a device or server. Also, there is the added functionality of binding and mounting files and directories.

When communicating with an I/O device, FMI/OS guises synchronous communication as asynchronous using its ABC, or asynchronous buffer caching.

Orthogonality, used in conjunction with the IPC, also allows for the future possibility of easily achieving network transparency.

User Space Details.

Currently, we plan on porting glibc to FMI/OS, replacing its `libio` with a proprietary FMI/OS library to supply glibc with the "syscalls" it needs. This layered approach would allow for the possibility of future ports of different (non-POSIX) environments, such as Windows or MacOS.

Testing will occur with a test shell called `testsh` that will test some of the fundamental functionality of the kernel (such as the IPC and memory manager). Otherwise, the `bash` shell handles normal command line commands.

2. Future Outlook

As I finish writing this thesis, one-fifth of it will already be out of date. As it becomes published, one-third will be out of date. The sheer nature of an open-source project is that it is always changing. These changes come as better and more efficient ways for doing things come about.

FMI/OS is still in its academic stages as we all continue to learn and piece everything together. Despite this, our development, testing framework, and kernel are already getting recognition. Companies are approaching our development staff with different contracts, as a result. This is the beauty of Open-Source Development. You may not make money by selling licenses for people to use your software, but your achievements do not go unrecognized, and are often followed by a hefty requite.

However, rarely is it about the money. FMI/OS was started as an academic project for those involved to learn how to write a successful microkernel. And for that, it has succeeded immensely. In the course of writing this thesis, I, too, have learned much about, not just FMI/OS, but all operating systems and their methods. Hopefully those reading have shared in this insight, as well.

This thesis is a stepping-stone to much more work to do on FMI/OS, and much more to learn about computers and operating systems. I bid those who go on to further research the topics discussed: good luck and have many great adventures as you go on to learn and accomplish great things!

Bibliography

- "Capability-Based Security." November, 2006. Wikipedia. December, 2006. http://en.wikipedia.org/wiki/Capability-based_security
- "Definitions." IEEE Std 1003.1-2001. 2001. The Open Group Base Specifications Issue 6. March, 2006. http://man.chinaunix.net/unix/susv3/basedefs/xbd_chap03.html
- "Executable and Linkable Format (ELF)." Portable Formats Specification, Version 1.1. Tool Interface Standards. Mar, 2006 http://www.skyfree.org/linux/references/ELF_Format.pdf
- "The GNU Operating System." Dec., 2006. Free Software Foundation. Nov., 2006. http://www.gnu.org/home.html
- Le Mignot, Gael. "The GNU Hurd: Extended Abstract." http://kilobug.free.fr/hurd/pres-en/abstract/abstract.pdf
- "Libre Software Meeting 2005, Operating System Design and Implementation." http://medias.2005.libresoftwaremeeting.org/topics/os
- Marshall, Dave. "Remote Procedure Calls: RPC." Jan, 1999. Cardiff University. Mar, 2006 http://www.cs.cf.ac.uk/Dave/C/node33.html
- "Metasyntactic Variable." December, 2006. Wikipedia. November, 2006. http://en.wikipedia.org/wiki/Metasyntactic_variable
- Minnich, Ron. "Why Plan9 Is Not Dead Yet, and What We Can Learn From It." Los Alamos National Lab. March, 2006. http://www.cs.unm.edu/~fastos/05meeting/PLAN9NOTDEADYET.pdf

- "QNX Neutrino RTOS Optimizes Programming Systems." 2004. !XCell Journal. December, 2005. http://www.xilinx.com/publications/xcellonline/xcell_48/xc_pdf/xc_
- "Realtime (System Interfaces, Chapter 2)." IEEE Std 1003.1-2001. 2001. The Open Group Base Specifications Issue 6. March, 2006 http://man.chinaunix.net/unix/susv3/functions/xsh_chap02_08.html
- Tanenbaum, Andrew. Modern Operating Systems. New Jersey: Prentice-Hall, 2001.
- Tanenbaum, Andrew, and Woodhull, Albert. Operating Systems, Design and Implementation. Second Edition. New Jersey: Prentice-Hall, 1997.
- The GNU Hurd." Mar 2006. GNU Project. Dec, 2005. http://www.gnu.org/software/hurd/hurd.html
- Valencia, Andrew. "VSTa: Frequently Asked Questions." December, 2006. http://amatus.g-cipher.net/cgi-bin/archzoom.cgi/fmios-pqm@ocgnet.org--fmios/docs--devel--1.0--patch-3/FAQ.txt
- Valencia, Andrew. "An Overview of the VSTa Microkernel." December, 2006. Valencia Consulting. http://fmios.ocgnet.org/ftp.vsta.org/docs/vsta_intro.ps
- "Windows Memory Management." Windows Hardware Developer Central. 2006. Microsoft Corporation. December, 2006. http://www.microsoft.com/whdc/system/kernel/wmm.mspx

Index

- A -.

- absolute path name.....9.1
- accordion.....3.6
- access control list (ACL).....11.1
- affinity (scheduling) 6.3
- aging.....*see scheduling*
- allocate.....*see memory*
- anonymous memory.....*see memory*
- application programming interface (API).....3.6, 10.1
- asynchronous.....4.2, 8.2
- asynchronous buffer caching (ABC).....9.4
- assembly.....*see machine code*
- atomic.....4.2
- authenticate.....4.2
- auto locks.....*see spinlocks*

- B -.

- badness.....3.6
- banker's algorithm (deadlocks).....7.2
- batch systems.....3.2
- bind.....4.2
- `bind()` 9.4

- BIOS.....1.1
- blocks.....9.1
- (to) block.....1.3, 3.1
- block devices.....8.1
- block-special files.....9.1
- bootloader.....1.1
- bootsector.....1.1
- bzr.....III.2

- C -.

- C-List.....11.1
- cache.....5.2, 6.3
- capabilities.....11.1
- character devices.....8.1
- character-special files.....9.1
- checkerboarding.....9.1
- child.....2.2
- clean pages.....*see memory*
- client-server.....0.1
- clock (paging algorithm).....*see memory*
- command-line shell.....*see shell*
- compute-bound.....3.1
- console.....4.2
- context switch.....2.1
- contiguous file allocation..........8.1
- cooperative multitasking.....3.6
- copy-on-write.....2.2
- critical region.....4.2

- current directory.....*see working directory*

- D -.

- daemon.....2.1
- deadlock.....6.2, 7.1
- demand paging.....*see memory*
- device controller.....8.2
- device driver.....8.4
- diffs.....III.2
- directories.....9.1
- dirty pages.....*see memory*
- direct memory access (DMA).....8.3
- double indirect block.....*see indirect block*
- driver.....1.2

- E F -.

- earliest deadline.....*see scheduling*
- embedded OS.....0.1
- fair-share.....*see scheduling*
- FIFO
  - *see scheduling*
  - *see memory*
- files.....5.1
- first-fit.....*see memory*

- G H -.

- gang scheduling.....*see scheduling*
- global page allocation.....*see memory*

- goodness.....3.6
- GNU Public License (GPL).....10.3
- graphical user interface (GUI).....10.1
- group ID (gid).....11.1
- Hurd.....*see MachOS*

- I -.

- indirect block (single/double/triple).....9.1
- i-nodes.....9.1
- instruction pointer.....1.2
- interactive systems.....3.2
- internet relay chat (IRC).....III.1
- interprocess communication (IPC).....2.2
- interrupt.....1.1, 3.1, 4.2, 8.3
  - interrupt handler.....3.1, 4.2
  - Interrupt Service Routines (ISR).....4.2
  - shared interrupts.....4.2
  - subscribe to.....3.1
- I/O-bound.....3.1
- ISO C standard.....*see memory*

- J K L -.

- lazy allocation.....*see memory*
- least recently used (paging algorithm).....*see memory*
- Legacy.....1.1
- Lesser GPL.....10.3
- libc.....10.2
- library.....5.5

- linked list file allocation 8.1
- linked list table file allocation 8.1
- Linux.....1.2, 3.6, 6.4
- local page allocation.....*see memory*
- lock.....*see spinlocks*
- lottery.....*see scheduling*

- M -.

- machine code.....0.1
- machine-dependent/independent.....5.7
- MachOS.....2.2, 4.2
- mapping.....*see memory*
- memory.....5.1
  - allocate.....5.1
  - anonymous.....5.1
  - cache.....5.2
  - clean/dirty pages.....5.4
  - clock paging algorithm.....5.3
  - demand paging.....5.3
  - files.....5.1
  - first-fit.....5.7
  - global page allocation.....5.4
  - ISO C standard.....5.6
  - lazy allocation.....5.7
  - least recently used paging algorithm.....5.3
  - library.....5.5
  - local page allocation.....5.4
  - map.....5.6

  - Memory Management Unit (MMU).....5.2
  - non-volatile memory.....9.1
  - not recently used paging algorithm.....5.3
  - page, paging, page tables.....5.2
  - page fault.....5.3
  - page fault frequency.....5.4
  - page loanout.....5.7
  - page protections.....5.4
  - permanently allocated.....5.1
  - pinned (page).....5.4
  - pre-paging.....5.4
  - random paging algorithm.....5.3
  - scatter-gather buffers.....5.7
  - second chance.....5.3
  - segmentation, segments.....5.5
  - swap, swapping.....5.1,.....5.3
  - translation lookaside buffer (TLB).....5.2
  - UVM.....5.7
  - Virtual Address Space (VAS).....1.3, 5.7
  - virtual memory.....5.1
  - virtual memory management.....1.3
  - volatile memory.....9.1
  - working set.....5.4
- microkernel.....0.1
- milestones.....III.1
- MINIX 4.1,.....5.7
- mount.....4.2

- `mount()` 9.4
- `msg_close()` 4.2
- `msg_connect()` 4.2
- `msg_receive()` 4.2
- `msg_reply()` 4.2
- `msg_send()` 4.2
- MS/DOS.....9.3
- multiboot header.....1.3
- multiprocessor.....6.1
- multi-server.....2.2
- multi-threaded.....2.2
- mutual exclusion (mutex).....6.2

- N -.

- namer.....4.2
- namespace.....9.4
- NetBSD.....5.7
- network transparency.....9.4
- niceness.....3.6
- nonpreemptive.....3.1
  - *see resources*
  - *see scheduling*
- non-volatile memory.....*see memory*
- not recently used.....*see memory*

- O -.

- O(1).....3.6
- open source.....III.0

- orthogonal.....9.2

- P -.

- page, paging, page fault, page tables.....*see memory*
- parent.....2.2
- partition table.....1.1
- patch.....III.2
- patch-queue management (pqm).....III.2
- permanently allocated memory.....*see memory*
- permissions.....9.1, 11.1
- pinned (page).....*see memory*
- Plan9.....4.2,.....9.4
- port.....4.1,.....8.3
- (to) port.....5.7
- port right.....4.2
- portable.....5.7
- POSIX.....2.1, 10
- preemptive.....3.1
  - *see scheduling*
  - *see resources*
- Preemptive Cooperative Multitasking Environment (PCME).....3.6
- priority class.....*see scheduling*
- priority inheritance.....3.6
- priority scheduling.....*see scheduling*
- process.....2.1
- process ID (PID).....2.1
- process table.....2.2
- programmed I/O.....8.3

- Q R -.

- quantum.....1.3, 3.1
- race condition.....4.2
- RAM.....9.1
- random access.....9.1
- rate monotonic.....*see scheduling*
- realtime systems.....3.2
- register.....2.1
- regular files.....9.1
- remote procedure call (RPC).....4.2
- resources.....7.1
- root.....9.1
- round robin.....*see scheduling*
- runnable tree.....3.6
- `run_next()` 3.6

- S -.

- scatter-gather buffers.....*see memory*
- scheduler.....2.1,.....3.1
  - admission scheduler.....3.1
  - CPU scheduler.....3.1
  - memory scheduler.....3.1
- scheduling.....3.1
  - aging.....3.4
  - FIFO.....3.4
  - earliest deadline.....3.4
  - fair-share.....3.4

  - gang scheduling (multiprocessor).....6.3
  - goals.....3.3
  - lottery.....3.4
  - nonpreemptive.....3.1
  - preemptive.....3.1
  - priority classes.....3.4
  - priority scheduling.....3.4
  - rate monotonic.....3.4
  - round robin.....3.4
  - shortest first.....3.4
  - shortest remaining time next.....3.4
  - space sharing (multiprocessor).....6.3
  - thread scheduling.....3.5
  - three-level.....3.1
  - timesharing (multiprocessor).....6.3
  - two-level scheduling (multiprocessor).....6.3
- second chance (paging algorithm).....*see memory*
- sectional spinlock.....*see spinlocks*
- sequential access.....9.1
- semaphore.....6.2
- `set_runnable()` 3.6
- sharing flags.....2.2
- shell.....1.2
- shared library.....5.5
- shortest first.....*see scheduling*
- shortest remaining time next.....*see scheduling*
- single indirect block.....*see indirect block*

- source code manager (SCM).....III.2
- source tree.....III.1
- space sharing.....*see scheduling*
- spinlocks.....6.2
  - auto lock.....7.3
  - sectional spinlock.....6.2
- spool.....7.2
- stable branch.....III.2
- subscriber list.....*see interrupts*
- swap.....*see memory*
- Symmetric Multiprocessor (SMP).....6.1
- synchronous..........8.2
- syscall.....2.1

- T -.

- terminals.....10.2
- thread.....1.3, 2.2
  - kernel threads.....3.5
  - scheduling.....3.5
  - thread table.....2.2
  - user space threads.....3.5
- throughput.....3.3
- tickets.....III.1,.....III.2
- time sharing.....*see scheduling*
- Trac.....III.1
- translator.....4.1
- turnaround.....3.3
- two-leveling scheduling algorithm.....*see scheduling*

- U -.

- UNIX.....3.6, 9.3
- unions.....9.4
- `unmount()` 9.4
- user ID (uid).....11.1
- UVM.....*see memory*

- V -.

- Virtual Address Space (VAS).....*see memory*
- virtual file system (VFS).....4.2
- virtual memory.....*see memory*
- volatile memory.....*see memory*
- VSTa.....0.1

- W X Y Z -.

- wake up.....6.2
- wiki.....III.1
- Win32 API standard.....5.7
- Windows.....5.7
- working directory.....9.1
- working set.....*see memory*
- wrapper.....2.2