Notes on Operating Systems

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Contents

Part I

Background

We start with an introductory chapter, that deals with what operating systems are, and the context in which they operate. In particular, it emphasizes the issues of software layers and abstraction, and the interaction between the operating system and the hardware.

This is supported by an appendix reviewing some background information on computer architecture.

Chapter 1

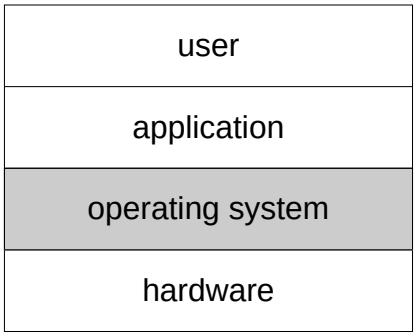
Introduction

In the simplest scenario, the operating system is the first piece of software to run on a computer when it is booted. Its job is to coordinate the execution of all other software, mainly user applications. It also provides various common services that are needed by users and applications.

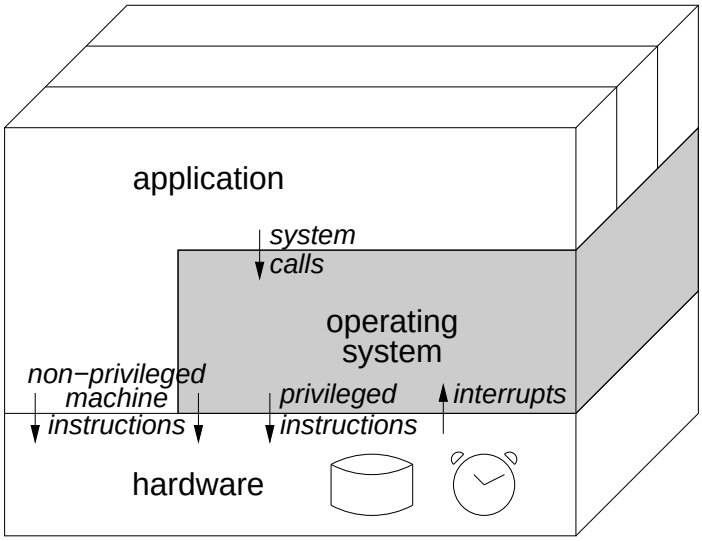
1.1 Operating System Functionality

The operating system controls the machine

It is common to draw the following picture to show the place of the operating system:



This is a misleading picture, because applications mostly execute machine instructions that do not go through the operating system. A better picture is:



where we have used a 3-D perspective to show that there is one hardware base, one operating system, but many applications. It also shows the important interfaces: applications can execute only non-privileged machine instructions, and they may also call upon the operating system to perform some service for them. The operating system may use privileged instructions that are not available to applications. And in addition, various hardware devices may generate interrupts that lead to the execution of operating system code.

A possible sequence of actions in such a system is the following:

1. The operating system executes, and *schedules* an application (makes it run).
2. The chosen application runs: the CPU executes its (non-privileged) instructions, and the operating system is not involved at all.
3. The system clock *interrupts* the CPU, causing it to switch to the clock's interrupt handler, which is an operating system function.
4. The clock interrupt handler updates the operating system's notion of time, and calls the scheduler to decide what to do next.
5. The operating system scheduler chooses another application to run in place of the previous one, thus performing a *context switch*.
6. The chosen application runs directly on the hardware; again, the operating system is not involved. After some time, the application performs a *system call* to read from a file.
7. The system call causes a *trap* into the operating system The operating system sets things up for the I/O operation (using some privileged instructions). It then puts the calling application to sleep, to await the I/O completion, and chooses another application to run in its place.
8. The third application runs.

The important thing to notice is that at any given time, only one program is running[1]. Sometimes this is the operating system, and at other times it is a user application. When a user application is running, the operating system loses its control over the machine. It regains control if the user application performs a system call, or if there is a hardware interrupt.

Exercise 1 *How can the operating system guarantee that there will be a system call or interrupt, so that it will regain control?*

The operating system is a reactive program

Another important thing to notice is that the operating system is a *reactive program*. It does not get an input, do some processing, and produce an output. Instead, it is constantly waiting for some *event* to happen. When the event happens, the operating system reacts. This usually involves some administration to handle whatever it is that happened. Then the operating system schedules another application, and waits for the next event.

Because it is a reactive system, the logical flow of control is also different. "Normal" programs, which accept an input and compute an output, have a `main` function that is the program's entry point. `main` typically calls other functions, and when it returns the program terminates. An operating system, in contradistinction, has many different entry points, one for each event type. And it is not supposed to terminate — when it finishes handling one event, it just waits for the next event.

Events can be classified into two types: interrupts and system calls. These are described in more detail below. The goal of the operating system is to run as little as possible, handle the events quickly, and let applications run most of the time.

Exercise 2 *Make a list of applications you use in everyday activities. Which of them are reactive? Are reactive programs common or rare?*

The operating system performs resource management

One of the main features of operating systems is support for multiprogramming. This means that multiple programs may execute "at the same time". But given that there is only one processor, this concurrent execution is actually a fiction. In reality, the operating system juggles the system's resources between the competing programs, trying to make it look as if each one has the computer for itself.

At the heart of multiprogramming lies resource management — deciding which running program will get what resources. Resource management is akin to the short blanket problem: everyone wants to be covered, but the blanket is too short to cover everyone at once.

[1]This is not strictly true on modern microprocessors with hyper-threading or multiple cores, but we'll assume a simple single-CPU system for now.

The resources in a computer system include the obvious pieces of hardware needed by programs:

- The CPU itself.
- Memory to store programs and their data.
- Disk space for files.

But there are also internal resources needed by the operating system:

- Disk space for paging memory.
- Entries in system tables, such as the process table and open files table.

All the applications want to run on the CPU, but only one can run at a time. Therefore the operating system lets each one run for a short while, and then *preempts* it and gives the CPU to another. This is called *time slicing*. The decision about which application to run is *scheduling* (discussed in Chapter 2).

As for memory, each application gets some memory frames to store its code and data. If the sum of the requirements of all the applications is more than the available physical memory, *paging* is used: memory pages that are not currently used are temporarily stored on disk (we'll get to this in Chapter 4).

With disk space (and possibly also with entries in system tables) there is usually a hard limit. The system makes allocations as long as they are possible. When the resource runs out, additional requests are failed. However, they can try again later, when some resources have hopefully been released by their users.

Exercise 3 *As system tables are part of the operating system, they can be made as big as we want. Why is this a bad idea? What sizes should be chosen?*

The operating system provides services

In addition, the operating system provides various services to the applications running on the system. These services typically have two aspects: abstraction and isolation.

Abstraction means that the services provide a more convenient working environment for applications, by hiding some of the details of the hardware, and allowing the applications to operate at a higher level of abstraction. For example, the operating system provides the abstraction of a file system, and applications don't need to handle raw disk interfaces directly.

Isolation means that many applications can co-exist at the same time, using the same hardware devices, without falling over each other's feet. These two issues are discussed next. For example, if several applications send and receive data over a network, the operating system keeps the data streams separated from each other.

1.2 Abstraction and Virtualization

The operating system presents an abstract machine

The dynamics of a multiprogrammed computer system are rather complex: each application runs for some time, then it is preempted, runs again, and so on. One of the roles of the operating system is to present the applications with an environment in which these complexities are hidden. Rather than seeing all the complexities of the real system, each application sees a simpler *abstract machine*, which seems to be dedicated to itself. It is blissfully unaware of the other applications and of operating system activity.

As part of the abstract machine, the operating system also supports some abstractions that do not exist at the hardware level. The chief one is files: persistent repositories of data with names. The hardware (in this case, the disks) only supports persistent storage of data blocks. The operating system builds the file system above this support, and creates named sequences of blocks (as explained in Chapter 5). Thus applications are spared the need to interact directly with disks.

Exercise 4 *What features exist in the hardware but are not available in the abstract machine presented to applications?*

Exercise 5 *Can the abstraction include new instructions too?*

The abstract machines are isolated

An important aspect of multiprogrammed systems is that there is not one abstract machine, but many abstract machines. Each running application gets its own abstract machine.

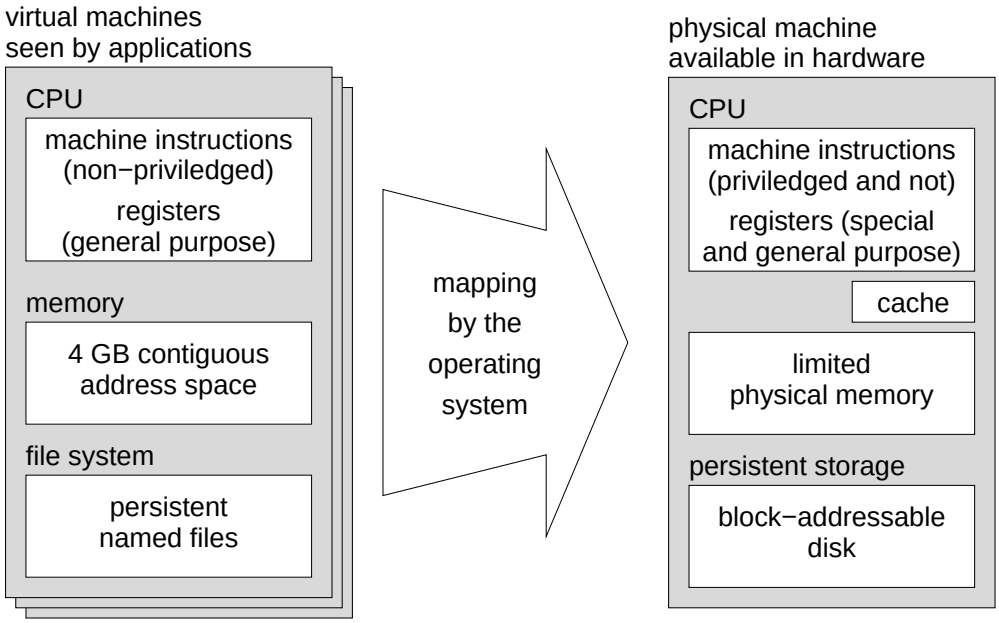
A very important feature of the abstract machines presented to applications is that they are isolated from each other. Part of the abstraction is to hide all those resources that are being used to support other running applications. Each running application therefore sees the system as if it were dedicated to itself. The operating system juggles resources among these competing abstract machines in order to support this illusion. One example of this is scheduling: allocating the CPU to each application in its turn.

Exercise 6 *Can an application nevertheless find out that it is actually sharing the machine with other applications?*

Virtualization allows for decoupling from physical restrictions

The abstract machine presented by the operating system is "better" than the hardware by virtue of supporting more convenient abstractions. Another important improvement is that it is also not limited by the physical resource limitations of the underlying hardware: it is a *virtual* machine. This means that the application does

not access the physical resources directly. Instead, there is a level of indirection, managed by the operating system.



The main reason for using virtualization is to make up for limited resources. If the physical hardware machine at our disposal has only 1GB of memory, and each abstract machine presents its application with a 4GB address space, then obviously a direct mapping from these address spaces to the available memory is impossible. The operating system solves this problem by coupling its resource management functionality with the support for the abstract machines. In effect, it juggles the available resources among the competing virtual machines, trying to hide the deficiency. The specific case of virtual memory is described in Chapter 4.

Virtualization does not necessarily imply abstraction

Virtualization does not necessarily involve abstraction. In recent years there is a growing trend of using virtualization to create multiple copies of the same hardware base. This allows one to run a different operating system on each one. As each operating system provides different abstractions, this decouples the issue of creating abstractions within a virtual machine from the provisioning of resources to the different virtual machines.

The idea of virtual machines is not new. It originated with MVS, the operating system for the IBM mainframes. In this system, time slicing and abstractions are completely decoupled. MVS actually only does the time slicing, and creates multiple *exact copies* of the original physical machine. Then, a single-user operating system called CMS is executed in each virtual machine. CMS provides the abstractions of the user environment, such as a file system.

As each virtual machine is an exact copy of the physical machine, it was also possible to run MVS itself on such a virtual machine. This was useful to debug new versions of the operating system on a running system. If the new version is buggy, only its virtual machine will crash, but the parent MVS will not. This practice continues today, and VMware has been used as a platform for allowing students to experiment with operating systems. We will discuss virtual machine support in Section 9.5.

To read more: History buffs can read more about MVS in the book by Johnson [7].

Things can get complicated

The structure of virtual machines running different operating systems may lead to a confusion in terminology. In particular, the allocation of resources to competing virtual machines may be done by a very thin layer of software that does not really qualify as a full-fledged operating system. Such software is usually called a *hypervisor*.

On the other hand, virtualization can also be done at the application level. A remarkable example is given by VMware. This is actually a user-level application, that runs on top of a conventional operating system such as Linux or Windows. It creates a set of virtual machines that mimic the underlying hardware. Each of these virtual machines can boot an independent operating system, and run different applications. Thus the issue of what exactly constitutes the operating system can be murky. In particular, several layers of virtualization and operating systems may be involved with the execution of a single application.

In these notes we'll ignore such complexities, at least initially. We'll take the (somewhat outdated) view that all the operating system is a monolithic piece of code, which is called the kernel. But in later chapters we'll consider some deviations from this viewpoint.

1.3 Hardware Support for the Operating System

The operating system doesn't need to do everything itself — it gets some help from the hardware. There are even quite a few hardware features that are included specifically for the operating system, and do not serve user applications directly.

The operating system enjoys a privileged execution mode

CPUs typically have (at least) two execution modes: *user* mode and *kernel* mode. User applications run in user mode. The heart of the operating system is called the *kernel*. This is the collection of functions that perform the basic services such as scheduling applications. The kernel runs in kernel mode. Kernel mode is also called *supervisor* mode or *privileged* mode.

The execution mode is indicated by a bit in a special register called the *processor status word* (PSW). Various CPU instructions are only available to software running

in kernel mode, i.e., when the bit is set. Hence these privileged instructions can only be executed by the operating system, and not by user applications. Examples include:

- Instructions to set the interrupt priority level (IPL). This can be used to block certain classes of interrupts from occurring, thus guaranteeing undisturbed execution.
- Instructions to set the hardware clock to generate an interrupt at a certain time in the future.
- Instructions to activate I/O devices. These are used to implement I/O operations on files.
- Instructions to load and store special CPU registers, such as those used to define the accessible memory addresses, and the mapping from each application's virtual addresses to the appropriate addresses in the physical memory.
- Instructions to load and store values from memory directly, without going through the usual mapping. This allows the operating system to access all the memory.

Exercise 7 *Which of the following instructions should be privileged?*

1. *Change the program counter*
2. *Halt the machine*
3. *Divide by zero*
4. *Change the execution mode*

Exercise 8 *You can write a program in assembler that includes privileged instructions. What will happen if you attempt to execute this program?*

Example: levels of protection on Intel processors

At the hardware level, Intel processors provide not two but four levels of protection.

Level 0 is the most protected and intended for use by the kernel.
Level 1 is intended for other, non-kernel parts of the operating system.
Level 2 is offered for device drivers: needy of protection from user applications, but not trusted as much as the operating system proper[2].
Level 3 is the least protected and intended for use by user applications.

Each data segment in memory is also tagged by a level. A program running in a certain level can only access data that is in the same level or (numerically) higher, that is, has the same or lesser protection. For example, this could be used to protect kernel data structures from being manipulated directly by untrusted device drivers; instead, drivers would be forced to use pre-defined interfaces to request the service they need from the kernel. Programs running in numerically higher levels are also restricted from issuing certain instructions, such as that for halting the machine.

Despite this support, most operating systems (including Unix, Linux, and Windows) only use two of the four levels, corresponding to kernel and user modes.

[2] Indeed, device drivers are typically buggier than the rest of the kernel [5].

Only predefined software can run in kernel mode

Obviously, software running in kernel mode can control the computer. If a user application was to run in kernel mode, it could prevent other applications from running, destroy their data, etc. It is therefore important to guarantee that user code will never run in kernel mode.

The trick is that when the CPU switches to kernel mode, it also changes the program counter[3] (PC) to point at operating system code. Thus user code will never get to run in kernel mode.

> **Note: kernel mode and superuser**
>
> Unix has a special privileged user called the "superuser". The superuser can override various protection mechanisms imposed by the operating system; for example, he can access other users' private files. However, this does not imply running in kernel mode. The difference is between restrictions imposed by the operating system software, as part of the operating system services, and restrictions imposed by the hardware.

There are two ways to enter kernel mode: interrupts and system calls.

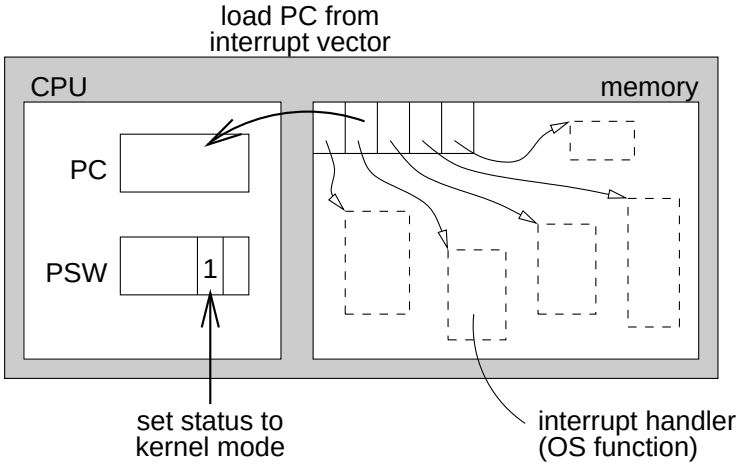
Interrupts cause a switch to kernel mode

Interrupts are special conditions that cause the CPU not to execute the next instruction. Instead, it enters kernel mode and executes an operating system interrupt handler.

But how does the CPU (hardware) know the address of the appropriate kernel function? This depends on what operating system is running, and the operating system might not have been written yet when the CPU was manufactured! The answer to this problem is to use an agreement between the hardware and the software. This agreement is asymmetric, as the hardware was there first. Thus, part of the hardware architecture is the definition of certain features and how the operating system is expected to use them. All operating systems written for this architecture must comply with these specifications.

Two particular details of the specification are the numbering of interrupts, and the designation of a certain physical memory address that will serve as an *interrupt vector*. When the system is booted, the operating system stores the addresses of the interrupt handling functions in the interrupt vector. When an interrupt occurs, the hardware stores the current PSW and PC, and loads the appropriate PSW and PC values for the interrupt handler. The PSW indicates execution in kernel mode. The PC is obtained by using the interrupt number as an index into the interrupt vector, and using the address found there.

[3]The PC is a special register that holds the address of the next instruction to be executed. This isn't a very good name. For an overview of this and other special registers see Appendix A.



Note that the hardware does this blindly, using the predefined address of the interrupt vector as a base. It is up to the operating system to actually store the correct addresses in the correct places. If it does not, this is a bug in the operating system.

Exercise 9 *And what happens if such a bug occurs?*

There are two main types of interrupts: asynchronous and internal. *Asynchronous (external) interrupts* are generated by external devices at unpredictable times. Examples include:

- Clock interrupt. This tells the operating system that a certain amount of time has passed. It's handler is the operating system function that keeps track of time. Sometimes, this function also calls the scheduler which might preempt the current application and run another in its place. Without clock interrupts, the application might run forever and monopolize the computer.

  Exercise 10 *A typical value for clock interrupt resolution is once every 10 milliseconds. How does this affect the resolution of timing various things?*

- I/O device interrupt. This tells the operating system that an I/O operation has completed. The operating system then wakes up the application that requested the I/O operation.

Internal (synchronous) interrupts occur as a result of an exception condition when executing the current instruction (as this is a result of what the software did, this is sometimes also called a "software interrupt"). This means that the processor cannot complete the current instruction for some reason, so it transfers responsibility to the operating system. There are two main types of exceptions:

- An error condition: this tells the operating system that the current application did something illegal (divide by zero, try to issue a privileged instruction, etc.). The handler is the operating system function that deals with misbehaved applications; usually, it kills them.

- A temporary problem: for example, the process tried to access a page of memory that is not allocated at the moment. This is an error condition that the operating system can handle, and it does so by bringing the required page into memory. We will discuss this in Chapter 4.

Exercise 11 *Can another interrupt occur when the system is still in the interrupt handler for a previous interrupt? What happens then?*

When the handler finishes its execution, the execution of the interrupted application continues where it left off — except if the operating system killed the application or decided to schedule another one.

To read more: Stallings [18, Sect. 1.4] provides a detailed discussion of interrupts, and how they are integrated with the instruction execution cycle.

System calls explicitly ask for the operating system

An application can also explicitly transfer control to the operating system by performing a *system call*. This is implemented by issuing the *trap* instruction. This instruction causes the CPU to enter kernel mode, and set the program counter to a special operating system entry point. The operating system then performs some service on behalf of the application. Technically, this is actually just another (internal) interrupt — but a desirable one that was generated by an explicit request.

As an operating system can have more than a hundred system calls, the hardware cannot be expected to know about all of them (as opposed to interrupts, which are a hardware thing to begin with). The sequence of events leading to the execution of a system call is therefore slightly more involved:

1. The application calls a library function that serves as a wrapper for the system call.
2. The library function (still running in user mode) stores the system call identifier and the provided arguments in a designated place in memory.
3. It then issues the trap instruction.
4. The hardware switches to privileged mode and loads the PC with the address of the operating system function that serves as an entry point for system calls.
5. The entry point function starts running (in kernel mode). It looks in the designated place to find which system call is requested.
6. The system call identifier is used in a big switch statement to find and call the appropriate operating system function to actually perform the desired service. This function starts by retrieving its arguments from where they were stored by the wrapper library function.

Exercise 12 *Should the library of system-call wrappers be part of the distribution of the compiler or of the operating system?*

Typical system calls include:

- Open, close, read, or write to a file.
- Create a new process (that is, start running another application).
- Get some information from the system, e.g. the time of day.
- Request to change the status of the application, e.g. to reduce its priority or to allow it to use more memory.

When the system call finishes, it simply returns to its caller like any other function. Of course, the CPU must return to normal execution mode.

The hardware has special features to help the operating system

In addition to kernel mode and the interrupt vector, computers have various features that are specifically designed to help the operating system.

The most common are features used to help with memory management. Examples include:

- Hardware to translate each virtual memory address to a physical address. This allows the operating system to allocate various scattered memory pages to an application, rather than having to allocate one long continuous stretch of memory.
- "Used" bits on memory pages, which are set automatically whenever any address in the page is accessed. This allows the operating system to see which pages were accessed (bit is 1) and which were not (bit is 0).

We'll review specific hardware features used by the operating system as we need them.

1.4 Roadmap

There are different views of operating systems

An operating system can be viewed in three ways:

- According to the services it provides to users, such as
  - Time slicing.
  - A file system.
- By its programming interface, i.e. its system calls.

- According to its internal structure, algorithms, and data structures.

An operating system is *defined* by its interface — different implementation of the same interface are equivalent as far as users and programs are concerned. However, these notes are organized according to services, and for each one we will detail the internal structures and algorithms used. Occasionally, we will also provide examples of interfaces, mainly from Unix.

To read more: To actually use the services provided by a system, you need to read a book that describes that system's system calls. Good books for Unix programming are Rochkind [15] and Stevens [19]. A good book for Windows programming is Richter [14]. Note that these books teach you about how the operating system looks "from the outside"; in contrast, we will focus on how it is built internally.

Operating system components can be studied in isolation

The main components that we will focus on are

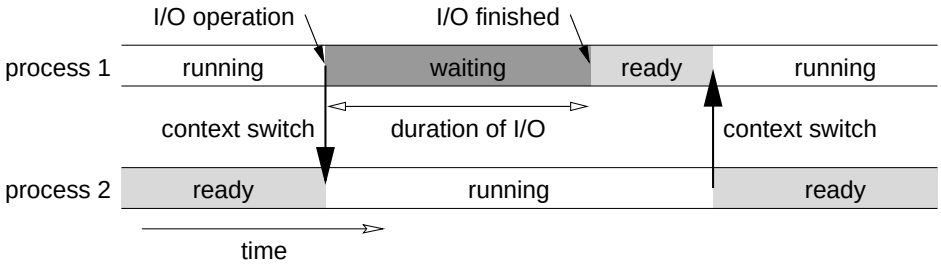
- Process handling. Processes are the agents of processing. The operating system creates them, schedules them, and coordinates their interactions. In particular, multiple processes may co-exist in the system (this is called *multiprogramming*).
- Memory management. Memory is allocated to processes as needed, but there typically is not enough for all, so paging is used.
- File system. Files are an abstraction providing named data repositories based on disks that store individual blocks. The operating system does the bookkeeping.

In addition there are a host of other issues, such as security, protection, accounting, error handling, etc. These will be discussed later or in the context of the larger issues.

But in a living system, the components interact

It is important to understand that in a real system the different components interact all the time. For example,

- When a process performs an I/O operation on a file, it is descheduled until the operation completes, and another process is scheduled in its place. This improves system utilization by overlapping the computation of one process with the I/O of another:



Thus both the CPU and the I/O subsystem are busy at the same time, instead of idling the CPU to wait for the I/O to complete.

- If a process does not have a memory page it requires, it suffers a page fault (this is a type of interrupt). Again, this results in an I/O operation, and another process is run in the meanwhile.
- Memory availability may determine if a new process is started or made to wait.

We will initially ignore such interactions to keep things simple. They will be mentioned later on.

Then there's the interaction among multiple systems

The above paragraphs relate to a single system with a single processor. The first part of these notes is restricted to such systems. The second part of the notes is about distributed systems, where multiple independent systems interact.

Distributed systems are based on networking and communication. We therefore discuss these issues, even though they belong in a separate course on computer communications. We'll then go on to discuss the services provided by the operating system in order to manage and use a distributed environment. Finally, we'll discuss the construction of heterogeneous systems using middleware. While this is not strictly part of the operating system curriculum, it makes sense to mention it here.

And we'll leave a few advanced topics to the end

Finally, there are a few advanced topics that are best discussed in isolation after we already have a solid background in the basics. These topics include

- The structuring of operating systems, the concept of microkernels, and the possibility of extensible systems
- Operating systems and mobile computing, such as disconnected operation of laptops
- Operating systems for parallel processing, and how things change when each user application is composed of multiple interacting processes or threads.

1.5 Scope and Limitations

The kernel is a small part of a distribution

All the things we mentioned so far relate to the operating system kernel. This will indeed be our focus. But it should be noted that in general, when one talks of a certain operating system, one is actually referring to a distribution. For example, a typical Unix distribution contains the following elements:

- The Unix kernel itself. Strictly speaking, this is "the operating system".
- The `libc` library. This provides the runtime environment for programs written in C. For example, is contains `printf`, the function to format printed output, and `strncpy`, the function to copy strings[4].
- Various tools, such as `gcc`, the GNU C compiler.
- Many utilities, which are useful programs you may need. Examples include a windowing system, desktop, and shell.

As noted above, we will focus exclusively on the kernel — what it is supposed to do, and how it does it.

You can (and should!) read more elsewhere

These notes should not be considered to be the full story. For example, most operating system textbooks contain historical information on the development of operating systems, which is an interesting story and is not included here. They also contain more details and examples for many of the topics that are covered here.

The main recommended textbooks are Stallings [18], Silberschatz et al. [17], and Tanenbaum [21]. These are general books covering the principles of both theoretical work and the practice in various systems. In general, Stallings is more detailed, and gives extensive examples and descriptions of real systems; Tanenbaum has a somewhat broader scope.

Of course it is also possible to use other operating system textbooks. For example, one approach is to use an educational system to provide students with hands-on experience of operating systems. The best known is Tanenbaum [22], who wrote the Minix system specifically for this purpose; the book contains extensive descriptions of Minix as well as full source code (This is the same Tanenbaum as above, but a different book). Nutt [13] uses Linux as his main example. Another approach is to emphasize principles rather than actual examples. Good (though somewhat dated) books in this category include Krakowiak [8] and Finkel [6]. Finally, some books concentrate on a certain class of systems rather than the full scope, such as Tanenbaum's book on distributed operating systems [20] (the same Tanenbaum again; indeed, one of the problems in the field is that a few prolific authors have each written a number of books on related issues; try not to get confused).

In addition, there are a number of books on specific (real) systems. The first and most detailed description of Unix system V is by Bach [1]. A similar description of 4.4BSD was written by McKusick and friends [12]. The most recent is a book on Solaris [10]. Vahalia is another very good book, with focus on advanced issues in different Unix versions [23]. Linux has been described in detail by Card and friends [4], by Beck and other friends [2], and by Bovet and Cesati [3]; of these, the first

[4] *Always* use `strncpy`, not `strcpy`!

gives a very detailed low-level description, including all the fields in all major data structures. Alternatively, source code with extensive commentary is available for Unix version 6 (old but a classic) [9] and for Linux [11]. It is hard to find anything with technical details about Windows. The best available is Russinovich and Solomon [16].

While these notes attempt to represent the lectures, and therefore have considerable overlap with textbooks (or, rather, are subsumed by the textbooks), they do have some unique parts that are not commonly found in textbooks. These include an emphasis on understanding system behavior and dynamics. Specifically, we focus on the complementary roles of hardware and software, and on the importance of knowing the expected workload in order to be able to make design decisions and perform reliable evaluations.

Bibliography

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

[2] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*. Addison-Wesley, 2nd ed., 1998.

[3] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[4] R. Card, E. Dumas, and F. Mével, *The Linux Kernel Book*. Wiley, 1998.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "*An empirical study of operating system errors*". In 18th *Symp. Operating Systems Principles*, pp. 73–88, Oct 2001.

[6] R. A. Finkel, *An Operating Systems Vade Mecum*. Prentice-Hall Inc., 2nd ed., 1988.

[7] R. H. Johnson, *MVS: Concepts and Facilities*. McGraw-Hill, 1989.

[8] S. Krakowiak, *Principles of Operating Systems*. MIT Press, 1988.

[9] J. Lions, *Lions' Commentary on UNIX 6th Edition, with Source Code*. Annabooks, 1996.

[10] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, 2001.

[11] S. Maxwell, *Linux Core Kernel Commentary*. Coriolis Open Press, 1999.

[12] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[13] G. J. Nutt, *Operating Systems: A Modern Perspective*. Addison-Wesley, 1997.

[14] J. Richter, *Programming Applications for Microsoft Windows*. Microsoft Press, 4th ed., 1999.

[15] M. J. Rochkind, *Advanced Unix Programming*. Prentice-Hall, 1985.

[16] M. E. Russinovic and D. A. Solomon, *Microsoft Windows Internals*. Microsoft Press, 4th ed., 2005.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 7th ed., 2005.

[18] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 5th ed., 2005.

[19] W. R. Stevens, *Advanced Programming in the Unix Environment*. Addison Wesley, 1993.

[20] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.

[21] A. S. Tanenbaum, *Modern Operating Systems*. Pearson Prentice Hall, 3rd ed., 2008.

[22] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd ed., 1997.

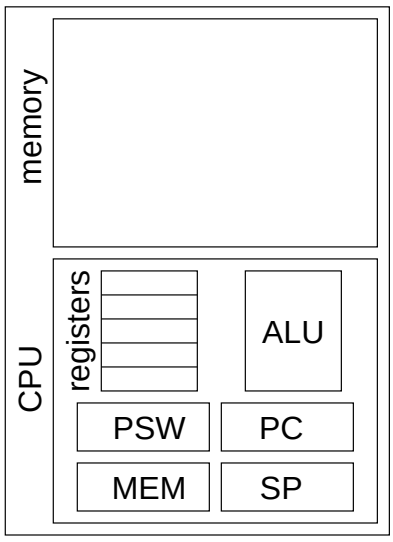
[23] U. Vahalia, *Unix Internals: The New Frontiers*. Prentice Hall, 1996.

Appendix A

Background on Computer Architecture

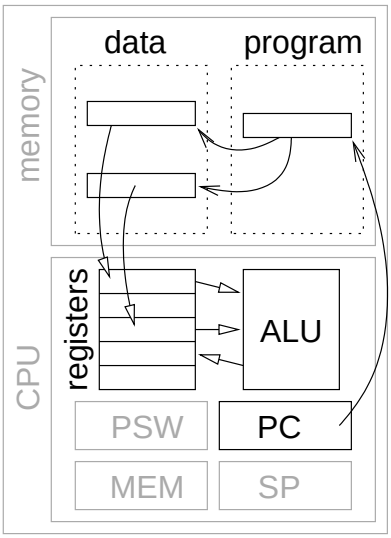
Operating systems are tightly coupled with the architecture of the computer on which they are running. Some background on how the hardware works is therefore required. This appendix summarizes the main points. Note, however, that this is only a high-level simplified description, and does not correspond directly to any specific real-life architecture.

At a very schematic level, we will consider the computer hardware as containing two main components: the memory and the CPU (central processing unit). The memory is where programs and data are stored. The CPU does the actual computation. It contains general-purpose registers, an ALU (arithmetic logic unit), and some special purpose registers. The general-purpose registers are simply very fast memory; the compiler typically uses them to store those variables that are the most heavily used in each subroutine. The special purpose registers have specific control functions, some of which will be described here.



The CPU operates according to a hardware clock. This defines the computer's "speed": when you buy a 3GHz machine, this means that the clock dictates 3,000,000,000 cycles each second. In our simplistic view, we'll assume that an instruction is executed in every such cycle. In modern CPUs each instruction takes more than a single cycle, as instruction execution is done in a pipelined manner. To compensate for this, real CPUs are superscalar, meaning they try to execute more than one instruction per cycle, and employ various other sophisticated optimizations.

One of the CPU's special registers is the *program counter* (PC). This register points to the next instruction that will be executed. At each cycle, the CPU loads this instruction and executes it. Executing it may include the copying of the instruction's operands from memory to the CPU's registers, using the ALU to perform some operation on these values, and storing the result in another register. The details depend on the architecture, i.e. what the hardware is capable of. Some architectures require operands to be in registers, while others allow operands in memory.



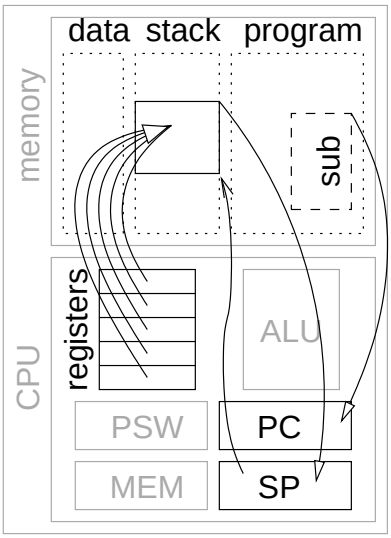
Exercise 13 *Is it possible to load a value into the PC?*

Exercise 14 *What happens if an arbitrary value is loaded into the PC?*

In addition to providing basic instructions such as add, subtract, and multiply, the hardware also provides specific support for running applications. One of the main examples is support for calling subroutines and returning from them, using the instructions `call` and `ret`. The reason for supporting this in hardware is that several things need to be done at once. As the called subroutine does not know the context from which it was called, it cannot know what is currently stored in the registers. Therefore we need to store these values in a safe place before the call, allow the called subroutine to operate in a "clean" environment, and then restore the register values when the subroutine terminates.

The `call` instruction does the first part:

1. It stores the register values on the stack, at the location pointed to by the stack pointer (another special register, abbreviated SP).
2. It also stores the return address (i.e. the address after the `call` instruction) on the stack.
3. It loads the PC with the address of the entry-point of the called subroutine.
4. It increments the stack pointer to point to the new top of the stack, in anticipation of additional subroutine calls.

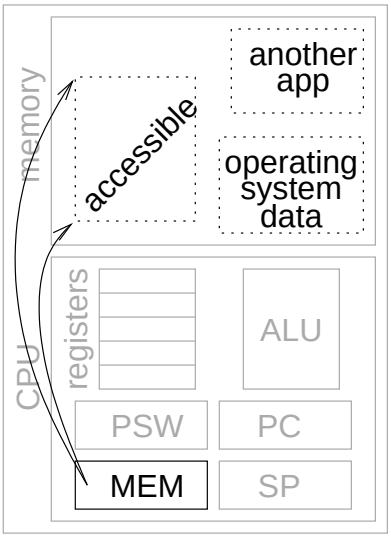


After the subroutine runs, the `ret` instruction restores the previous state:

1. It restores the register values from the stack.

2. It loads the PC with the return address that was also stored on the stack.

3. It decrements the stack pointer to point to the previous stack frame.



The hardware also provides special support for the operating system. One type of support is the mapping of memory. This means that at any given time, the CPU cannot access all of the physical memory. Instead, there is a part of memory that is accessible, and other parts that are not. This is useful to allow the operating system to prevent one application from modifying the memory of another, and also to protect the operating system itself. The simplest implementation of this idea is to have a pair of special registers that bound the accessible memory range. Real machines nowadays support more sophisticated mapping, as described in Chapter 4.

A special case of calling a subroutine is making a system call. In this case the caller is a user application, but the callee is the operating system. The problem is that the operating system should run in privileged mode, or kernel mode. Thus we cannot just use the `call` instruction. Instead, we need the `trap` instruction. This does all what `call` does, and in addition sets the mode bit in the processor status word (PSW) register. Importantly, when `trap` sets this bit, it loads the PC with the predefined address of the operating system entry point (as opposed to `call` which loads it with the address of a user function). Thus after issuing a `trap`, the CPU will start executing operating system code in kernel mode. Returning from the system call resets the mode bit in the PSW, so that user code will not run in kernel mode.

There are other ways to enter the operating system in addition to system calls, but technically they are all very similar. In all cases the effect is just like that of a trap: to pass control to an operating system subroutine, and at the same time change the CPU mode to kernel mode. The only difference is the trigger. For system calls, the trigger is a `trap` instruction called explicitly by an application. Another type of trigger is when the current instruction cannot be completed (e.g. division by zero), a condition known as an exception. A third is interrupts — a notification from an external device (such as a timer or disk) that some event has happened and needs handling by the operating system.

The reason for having a kernel mode is also an example of hardware support for the operating system. The point is that various control functions need to be reserved to the operating system, while user applications are prevented from performing them. For example, if any user application could set the memory mapping registers, they would be able to allow themselves access to the memory of other applications. Therefore the setting of these special control registers is only allowed in kernel mode. If a

user-mode application tries to set these registers, it will suffer an illegal instruction exception.

Part II

The Classics

Operating systems are complex programs, with many interactions between the different services they provide. The question is how to present these complex interactions in a linear manner. We do so by first looking at each subject in isolation, and then turning to cross-cutting issues.

In this part we describe each of the basic services of an operating system independently, in the context of the simplest possible system: a single autonomous computer with a single processor. Most operating system textbooks deal mainly with such systems. Thus this part of the notes covers the classic operating systems corriculum: processes, concurrency, memory management, and file systems. It also includes a summary of basic principles that underlie many of the concepts being discussed.

Part III then discusses the cross-cutting issues, with chapters about topics that are sometimes not covered. These include security, extending operating system functionality to multiprocessor systems, various technical issues such as booting the system, the structure of the operating system, and performance evaluation.

Part IV extends the discussion to distributed systems. It starts with the issue of communication among independent computers, and then presents the composition of autonomous systems into larger ensambles that it enables.

Chapter 2

Processes and Threads

A process is an instance of an application execution. It encapsulates the environment seen by the application being run — essentially providing it with a sort of virtual machine. Thus a process can be said to be an abstraction of the computer.

The application may be a program written by a user, or a system application. Users may run many instances of the same application at the same time, or run many different applications. Each such running application is a process. The process only exists for the duration of executing the application.

A thread is part of a process. In particular, it represents the actual flow of the computation being done. Thus each process must have at least one thread. But multithreading is also possible, where several threads execute within the context of the same process, by running different instructions from the same application.

To read more: All operating system textbooks contain extensive discussions of processes, e.g. Stallings chapters 3 and 9 [15] and Silberschatz and Galvin chapters 4 and 5 [14]. In general, Stallings is more detailed. We will point out specific references for each topic.

2.1 What Are Processes and Threads?

2.1.1 Processes Provide Context

A process, being an abstraction of the computer, is largely defined by:

- Its CPU state (register values).
- Its address space (memory contents).
- Its environment (as reflected in operating system tables).

Each additional level gives a wider context for the computation.

The CPU registers contain the current state

The current state of the CPU is given by the contents of its registers. These can be grouped as follows:

- Processor Status Word (PSW): includes bits specifying things like the mode (privileged or normal), the outcome of the last arithmetic operation (zero, negative, overflow, or carry), and the interrupt level (which interrupts are allowed and which are blocked).
- Instruction Register (IR) with the current instruction being executed.
- Program Counter (PC): the address of the next instruction to be executed.
- Stack Pointer (SP): the address of the current stack frame, including the function's local variables and return information.
- General purpose registers used to store addresses and data values as directed by the compiler. Using them effectively is an important topic in compilers, but does not involve the operating system.

The memory contains the results so far

Only a small part of an applications data can be stored in registers. The rest is in memory. This is typically divided into a few parts, sometimes called segments:

Text — the application's code. This is typically read-only, and might be shared by a number of processes (e.g. multiple invocations of a popular application such as a text editor).

Data — the application's predefined data structures.

Heap — an area from which space can be allocated dynamically at runtime, using functions like `new` or `malloc`.

Stack — where register values are saved, local variables allocated, and return information kept, in order to support function calls.

All the addressable memory together is called the process's *address space*. In modern systems this need not correspond directly to actual physical memory. We'll discuss this later.

Exercise 15 *The different memory segments are not independent — rather, they point to each other (i.e. one segment can contain addresses that refer to another). Can you think of examples?*

The environment contains the relationships with other entities

A process does not exist in a vacuum. It typically has connections with other entities, such as

- A terminal where the user is sitting.
- Open files that are used for input and output.
- Communication channels to other processes, possibly on other machines.

These are listed in various operating system tables.

Exercise 16 *How does the process affect changes in its register contents, its various memory segments, and its environment?*

All the data about a process is kept in the PCB

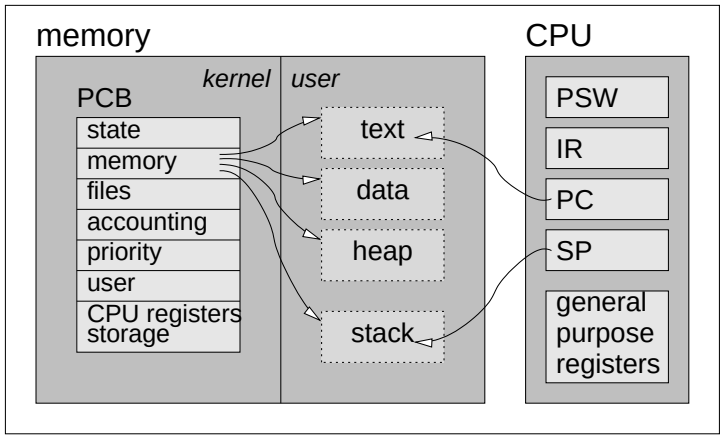
The operating system keeps all the data it needs about a process in the *process control block* (PCB) (thus another definition of a process is that it is "the entity described by a PCB"). This includes many of the data items described above, or at least pointers to where they can be found (e.g. for the address space). In addition, data needed by the operating system is included, for example

- Information for calculating the process's priority relative to other processes. This may include accounting information about resource use so far, such as how long the process has run.
- Information about the user running the process, used to decide the process's access rights (e.g. a process can only access a file if the file's permissions allow this for the user running the process). In fact, the process may be said to represent the user to the system.

The PCB may also contain space to save CPU register contents when the process is not running (some implementations specifically restrict the term "PCB" to this storage space).

Exercise 17 *We said that the stack is used to save register contents, and that the PCB also has space to save register contents. When is each used?*

Schematically, all the above may be summarized by the following picture, which shows the relationship between the different pieces of data that constitute a process:



2.1.2 Process States

One of the important items in the PCB is the process *state*. Processes change state during their execution, sometimes by themselves (e.g. by making a system call), and sometimes due to an external event (e.g. when the CPU gets a timer interrupt).

A process is represented by its PCB

The PCB is more than just a data structure that contains information about the process. It actually represents the process. Thus PCBs can be linked together to represent processes that have something in common — typically processes that are in the same state.

For example, when multiple processes are ready to run, this may be represented as a linked list of their PCBs. When the scheduler needs to decide which process to run next, it traverses this list, and checks the priority of the different processes.

Processes that are waiting for different types of events can also be linked in this way. For example, if several processes have issued I/O requests, and are now waiting for these I/O operations to complete, their PCBs can be linked in a list. When the disk completes an I/O operation and raises an interrupt, the operating system will look at this list to find the relevant process and make it ready for execution again.

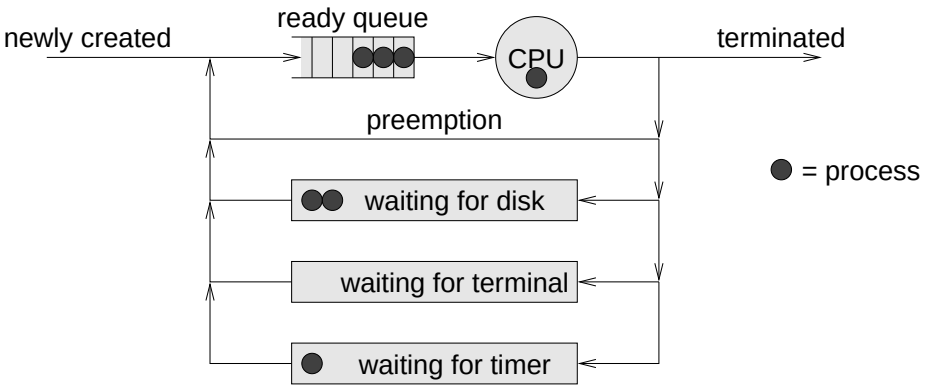
Exercise 18 *What additional things may cause a process to block?*

Processes changes their state over time

An important point is that a process may change its state. It can be ready to run at one instant, and blocked the next. This may be implemented by moving the PCB from one linked list to another.

Graphically, the lists (or states) that a process may be in can be represented as different locations, and the processes may be represented by tokens that move from

one state to another according to the possible transitions. For example, the basic states and transitions may look like this:

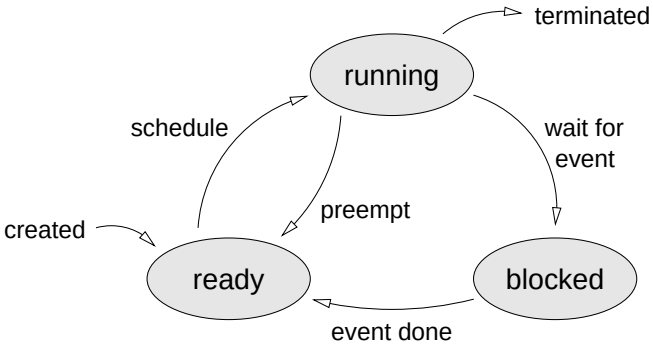


At each moment, at most one process is in the running state, and occupying the CPU. Several processes may be ready to run (but can't because we only have one processor). Several others may be blocked waiting for different types of events, such as a disk interrupt or a timer going off.

Exercise 19 *What sort of applications may wait for a timer?*

States are abstracted in the process states graph

From a process's point of view, the above can be abstracted using three main states. The following graph shows these states and the transitions between them:



Processes are created in the *ready* state. A ready process may be scheduled to *run* by the operating system. When running, it may be preempted and returned to the ready state. A process may also *block* waiting for an event, such as an I/O operation. When the event occurs, the process becomes ready again. Such transitions continue until the process terminates.

Exercise 20 *Why should a process ever be preempted?*

Exercise 21 *Why is there no arrow directly from blocked to running?*

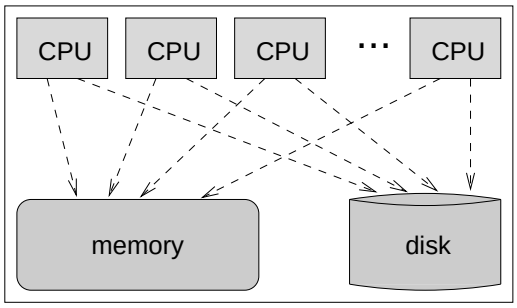
Exercise 22 *Assume the system provides processes with the capability to suspend and resume other processes. How will the state transition graph change?*

2.1.3 Threads

Multithreaded processes contain multiple threads of execution

A process may be *multithreaded*, in which case many executions of the code co-exist together. Each such thread has its own CPU state and stack, but they share the rest of the address space and the environment.

In terms of abstractions, a thread embodies the abstraction of the flow of the computation, or in other words, what the CPU does. A multithreaded process is therefore an abstraction of a computer with multiple CPUs, that may operate in parallel. All of these CPUs share access to the computer's memory contents and its peripherals (e.g. disk and network).



The main exception in this picture is the stacks. A stack is actually a record of the flow of the computation: it contains a frame for each function call, including saved register values, return address, and local storage for this function. Therefore each thread must have its own stack.

Exercise 23 *In a multithreaded program, is it safe for the compiler to use registers to temporarily store global variables? And how about using registers to store local variables defined within a function?*

Exercise 24 *Can one thread access local variables of another? Is doing so a good idea?*

Threads are useful for programming

Multithreading is sometimes useful as a tool for structuring the program. For example, a server process may create a separate thread to handle each request it receives. Thus each thread does not have to worry about additional requests that arrive while it is working — such requests will be handled by other threads.

Another use of multithreading is the implementation of asynchronous I/O operations, thereby overlapping I/O with computation. The idea is that one thread performs the I/O operations, while another computes. Only the I/O thread blocks to wait for the I/O operation to complete. In the meanwhile, the other thread can continue to run.

For example, this can be used in a word processor when the user requests to print the document. With multithreading, the word processor may create a separate thread that prepares the print job in the background, while at the same time supporting continued interactive work.

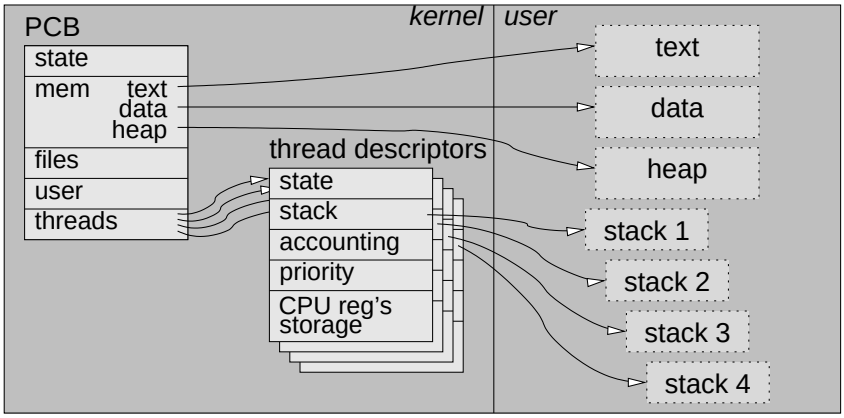
Exercise 25 *Asynchronous I/O is obviously useful for writing data, which can be done in the background. But can it also be used for reading?*

The drawback of using threads is that they may be hard to control. In particular, threads programming is susceptible to race conditions, where the results depend on the order in which threads perform certain operations on shared data. As operating systems also have this problem, we will discuss it below in Chapter 3.

Threads may be an operating system abstraction

Threads are often implemented at the operating system level, by having multiple thread entities associated with each process (these are sometimes called kernel threads, or light-weight processes (LWP)). To do so, the PCB is split, with the parts that describe the computation moving to the thread descriptors. Each thread then has its own stack and descriptor, which includes space to store register contents when the thread is not running. However they share all the rest of the environment, including the address space and open files.

Schematically, the kernel data structures and memory layout needed to implement kernel threads may look something like this:



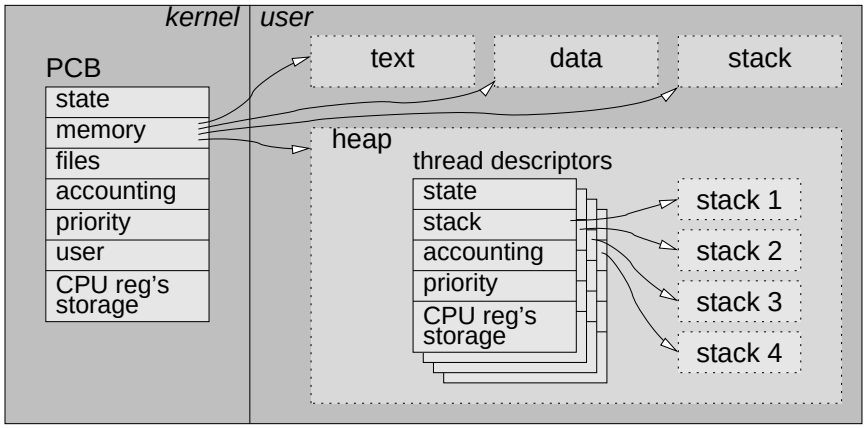
Exercise 26 *If one thread allocates a data structure from the heap, can other threads access it?*

At the beginning of this chapter, we said that a process is a program in execution. But when multiple operating-system-level threads exist within a process, it is actually the threads that are the active entities that represent program execution. Thus it is threads that change from one state (running, ready, blocked) to another. In particular, it is threads that block waiting for an event, and threads that are scheduled to run by the operating system scheduler.

Alternatively, threads can be implemented at user level

An alternative implementation is user-level threads. In this approach, the operating system does not know about the existence of threads. As far as the operating system is concerned, the process has a single thread of execution. But the program being run by this thread is actually a thread package, which provides support for multiple threads. This by necessity replicates many services provided by the operating system, e.g. the scheduling of threads and the bookkeeping involved in handling them. But it reduces the overhead considerably because everything is done at user level without a trap into the operating system.

Schematically, the kernel data structures and memory layout needed to implement user threads may look something like this:



Note the replication of data structures and work. At the operating system level, data about the process as a whole is maintained in the PCB and used for scheduling. But when it runs, the thread package creates independent threads, each with its own stack, and maintains data about them to perform its own internal scheduling.

Exercise 27 *Are there any drawbacks for using user-level threads?*

The problem with user-level threads is that the operating system does not know about them. At the operating system level, a single process represents all the threads. Thus if one thread performs an I/O operation, the whole process is blocked waiting for the I/O to complete, implying that *all* threads are blocked.

Exercise 28 *Can a user-level threads package avoid this problem of being blocked when any thread performs an I/O operation? Hint: think about a hybrid design that also uses kernel threads.*

> **Details: Implementing user-level threads with setjmp and longjmp**
> The hardest problem in implementing threads is the need to switch among them. How is this done at user level?

If you think about it, all you really need is the ability to store and restore the CPU's general-purpose registers, to set the stack pointer (SP) to point into the correct stack, and to set the program counter (PC) to point at the correct instruction. This can actually be done with the appropriate assembler code (you can't do it in a high-level language, because such languages typically don't have a way to say you want to access the SP or PC). You don't need to modify the special registers like the PSW and those used for memory mapping, because they reflect shared state that is common to all the threads; thus you don't need to run in kernel mode to perform the thread context switch.

In Unix, jumping from one part of the program to another can be done using the `setjmp` and `longjmp` functions that encapsulate the required operations. `setjmp` essentially stores the CPU state into a buffer. `longjmp` restores the state from a buffer created with `setjmp`. The names derive from the following reasoning: `setjmp` sets things up to enable you to jump back to exactly this place in the program. `longjmp` performs a long jump to another location, and specifically, to one that was previously stored using `setjmp`.

To implement threads, assume each thread has its own buffer (in our discussion of threads above, this is the part of the thread descriptor set aside to store registers). Given many threads, there is an array of such buffers called `buf`. In addition, let `current` by the index of the currently running thread. Thus we want to store the state of the current thread in `buf[current]`. The code that implements a context switch is then simply

```
switch() {
   if (setjmp(buf[current]) == 0) {
      schedule();
   }
}
```

The `setjmp` function stores the state of the current thread in `buf[current]`, and returns 0. Therefore we enter the `if`, and the function `schedule` is called. Note that this is the general context switch function, due to our use of `current`. Whenever a context switch is performed, the thread state is stored in the correct thread's buffer, as indexed by `current`.

The schedule function, which is called from the context switch function, does the following:

```
schedule() {
   new = select-thread-to-run
   current = new;
   longjmp(buf[new], 1);
}
```

`new` is the index of the thread we want to switch to. `longjmp` performs a switch to that thread by restoring the state that was previously stored in `buf[new]`. Note that this buffer indeed contains the state of that thread, that was stored in it by a previous call to `setjmp`. The result is that we are *again* inside the call to `setjmp` that originally stored the state in `buf[new]`. But this time, that instance of `setjmp` will return a value of 1, not 0 (this is specified by the second argument to `longjmp`). Thus, when the function returns, the `if` surrounding it will fail, and `schedule` will *not* be called again immediately.

Instead, `switch` will return and execution will continue where it left off before calling the switching function.

User-level thread packages, such as pthreads, are based on this type of code. But they provide a more convenient interface for programmers, enabling them to ignore the complexities of implementing the context switching and scheduling.

Exercise 29 *How are `setjmp` and `longjmp` implemented? do they need to run in kernel mode?*

Exploiting multiprocessors requires operating system threads

A special case where threads are useful is when running on a multiprocessor (a computer with several physical processors). In this case, the different threads may execute simultaneously on different processors. This leads to a possible speedup of the computation due to the use of parallelism. Naturally, such parallelism will only arise if operating system threads are used. User-level threads that are multiplexed on a single operating system process cannot use more than one processor at a time.

The following table summarizes the properties of kernel threads and user threads, and contrasts them with processes:

<table>
<tr><th>processes</th><th>kernel threads</th><th>user threads</th></tr>
<tr><td>protected from each other, require operating system to communicate</td><td colspan="2">share address space, simple communication, useful for application structuring</td></tr>
<tr><td>high overhead: all operations require a kernel trap, significant work</td><td>medium overhead: operations require a kernel trap, but little work</td><td>low overhead: everything is done at user level</td></tr>
<tr><td colspan="2">independent: if one blocks, this does not affect the others</td><td>if a thread blocks the whole process is blocked</td></tr>
<tr><td colspan="2">can run on different processors in a multiprocessor</td><td>all share the same processor</td></tr>
<tr><td colspan="2">system specific API, programs are not portable</td><td>the same thread library may be available on several systems</td></tr>
<tr><td colspan="2">one size fits all</td><td>application-specific thread management is possible</td></tr>
</table>

In the following, our discussion of processes is generally applicable to threads as well. In particular, the scheduling of threads can use the same policies described below for processes.

2.1.4 Operations on Processes and Threads

As noted above, a process is an abstraction of the computer, and a thread is an abstraction of the CPU. What operations are typically available on these abstractions?

Create a new one

The main operation on processes and threads is to create a new one. In different systems this may be called a `fork` or a `spawn`, of just simply `create`. A new process is typically created with one thread. That thread can then create additional threads within that same process.

Note that operating systems that support threads, such as Mach and Windows NT, have distinct system calls for processes and threads. For example, the "process_create" call can be used to create a new process, and then "thread_create" can be used to add threads to this process. This is an important distinction, as creating a new process is much heavier: you need to create a complete context, including its memory space. Creating a thread is much easier, as it simply hooks into an existing context.

Unix originally did not support threads (it was designed in the late 1960's). Therefore many Unix variants implement threads as "light-weight" processes, reusing a relatively large part of the process abstraction.

Terminate an existing one

The dual of creating a process is terminating it. A process or thread can terminate itself by returning from its main function, or by calling the `exit` system call.

Exercise 30 *If a multithreaded process terminates, what happens to its threads?*

Allowing one process to terminate another is problematic — what if the other process belongs to another user who does not want his process to be terminated? The more common interface is to allow one process to send a signal to another, as described below.

Threads within the same process are less restricted, as it is assumed that if one terminates another this is part of what the application as a whole is supposed to do.

Suspend execution

A thread embodies the flow of a computation. So a desirable operation on it may be to stop this computation.

A thread may suspend itself by going to sleep. This means that it tells the system that it has nothing to do now, and therefore should not run. A sleep is associated with a time: when this future time arrives, the system will wake the thread up.

Exercise 31 *Can you think of an example where this is useful?*

Threads (in the same process) can also suspend each other. Suspend is essentially another state in the thread state transition graph, which is similar to the blocked state. The counterpart of suspend is to resume another thread. A resumed thread is moved from the suspend state to the ready state.

Control over execution is sometimes also useful among processes. For example, a debugger process may control the execution of a process executing the application being debugged.

Send a signal or message

A common operation among processes is the sending of signals. A signal is often described as a software interrupt: the receiving process receives the signal rather than continuing with what it was doing before. In many cases, the signal terminates the process unless the process takes some action to prevent this.

2.2 Multiprogramming: Having Multiple Processes in the System

Multiprogramming means that multiple processes are handled by the system at the same time, typically by time slicing. It is motivated by considerations of responsiveness to the users and utilization of the hardware.

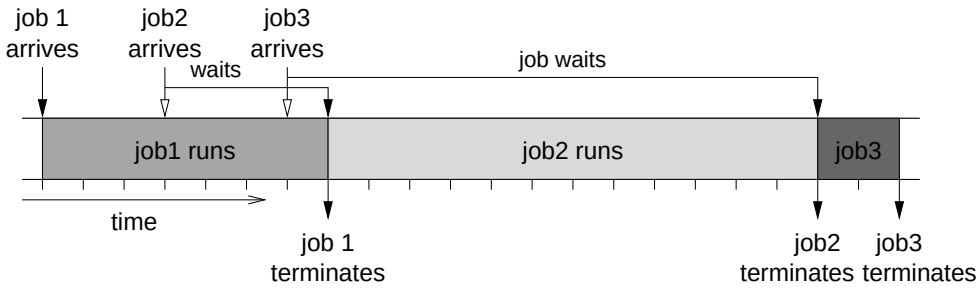
> **Note: terminology may be confusing**
>
> "Job" and "process" are essentially synonyms.
>
> The following terms actually have slightly different meanings:
>
> ***Multitasking*** — having multiple processes time slice on the same processor.
>
> ***Multiprogramming*** — having multiple jobs in the system (either on the same processor, or on different processors)
>
> ***Multiprocessing*** — using multiple processors for the same job or system (i.e. parallel computing).
>
> When there is only one CPU, multitasking and multiprogramming are the same thing. In a parallel system or cluster, you can have multiprogramming without multitasking, by running jobs on different CPUs.

2.2.1 Multiprogramming and Responsiveness

One reason for multiprogramming is to improve responsiveness, which means that users will have to wait less (on average) for their jobs to complete.

with FCFS, short jobs may be stuck behind long ones

Consider the following system, in which jobs are serviced in the order they arrive (First Come First Serve, or FCFS):



Job 2 is ahead of job 3 in the queue, so when job 1 terminates, job 2 runs. However, job 2 is very long, so job 3 must wait a long time in the queue, even though it itself is short.

If the CPU was shared, this wouldn't happen

Now consider an ideal system that supports *processor sharing*: when there are k jobs in the system, they all run simultaneously, but at a rate of $1/k$.

Now job 3 does not have to wait for job 2. The time it takes is proportional to its own length, increased according to the current load.

Regrettably, it is impossible to implement this ideal. But we'll see below that it can be approximated by using time slicing.

Responsiveness is important to keep users happy

Users of early computer systems didn't expect good responsiveness: they submitted a job to the operator, and came back to get the printout the next day. But when interactive systems were introduced, users got angry when they had to wait just a few minutes. Actually good responsiveness for interactive work (e.g. text editing) is measured in fractions of a second.

Supporting interactive work is important because it improves productivity. A user can submit a job and get a response while it is "still in his head". It is then possible to make modifications and repeat the cycle.

To read more: The effect of responsiveness on users' anxiety was studied by Guynes, who showed that bad responsiveness is very annoying even for people who are normally very relaxed [5].

Actually, it depends on workload statistics

The examples shown above had a short job stuck behind a long job. Is this really a common case?

Consider a counter example, in which all jobs have the same length. In this case, a job that arrives first and starts running will also terminate before a job that arrives later. Therefore preempting the running job in order to run the new job delays it and *degrades* responsiveness.

Exercise 32 *Consider applications you run daily. Do they all have similar runtimes, or are some short and some long?*

The way to go depends on the coefficient of variation (CV) of the distribution of job runtimes. The coefficient of variation is the standard deviation divided by the mean. This is a sort of normalized version of the standard deviation, and measures how wide the distribution is. "Narrow" distributions have a small CV, while very wide (or fat tailed) distributions have a large CV. The exponential distribution has CV = 1.

Returning to jobs in computer systems, if the CV is smaller than 1 than we can expect new jobs to be similar to the running job. In this case it is best to leave the running job alone and schedule additional jobs FCFS. If the CV is larger than 1, on the other hand, then we can expect new jobs to be shorter than the current job. Therefore it is best to preempt the current job and run the new job instead.

Measurements from several different systems show that the distribution of job runtimes is heavy tailed. There are many very short jobs, some "middle" jobs, and few long jobs, but some of the long jobs are *very* long. The CV is always larger than 1 (values from about 3 to about 70 have been reported). Therefore responsiveness is improved by using preemption and time slicing, and the above examples are correct.

To read more: The benefit of using preemption when the CV of service times is greater than 1 was established by Regis [13].

> **Details: the distribution of job runtimes**
> There is surprisingly little published data about real measurements of job runtimes and their distributions. Given the observation that the CV should be greater than 1, a common procedure is to choose a simple distribution that matches the first two moments, and thus has the correct mean and CV. The chosen distribution is usually a two-stage hyper-exponential, i.e. the probabilistic combination of two exponentials. However, this procedure fails to actually create a distribution with the right shape, and might lead to erroneous performance evaluations, as demonstrated by Lazowska [9].
>
> An interesting model for interactive systems was given by Leland and Ott [10], and later verified by Harchol-Balter and Downey [7]. This model holds for processes that are longer

than a couple of seconds, on Unix systems. For such processes, the observed distribution is

$$\Pr(r > t) = 1/t$$

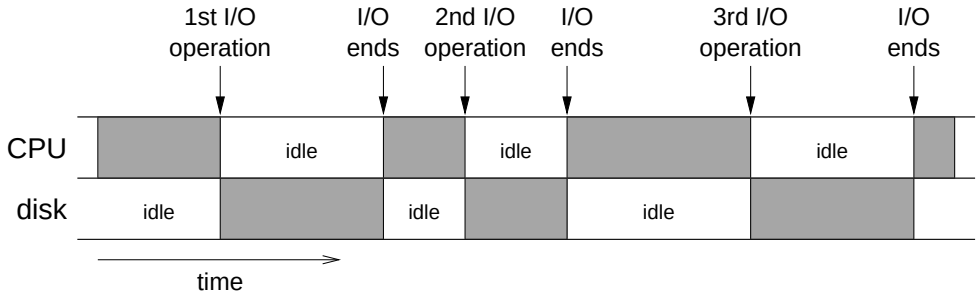
(where r denotes the process runtime). In other words, the tail of the distribution of runtimes has a Pareto distribution.

2.2.2 Multiprogramming and Utilization

The second reason for multiprogramming is to improve hardware utilization.

Applications use one system component at a time

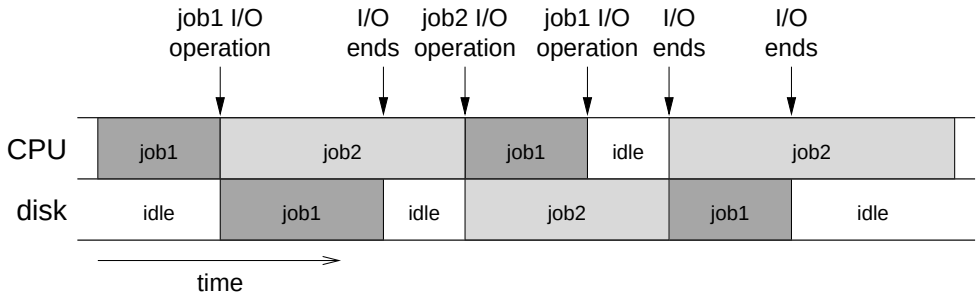
Consider a simple example of a system with a CPU and disk. When an application reads data from the disk, the CPU is idle because it has to wait for the data to arrive. Thus the application uses either the CPU or the disk at any given moment, but not both, as shown in this Gantt chart:



Note: The time to perform I/O
An important issue concerning the use of different devices is the time scale involved. It is important to note that the time scales of the CPU and I/O devices are typically very different. The cycle time of a modern microprocessor is on the order of part of a nanosecond. The time to perform a disk operation is on the order of several milliseconds. Thus an I/O operation takes the same time as millions of CPU instructions.

Multiprogramming allows for simultaneous use of several components

If more than one job is being serviced, then instead of waiting for an I/O operation to complete, the CPU can switch to another job. This does not guarantee that the CPU never has to wait, nor that the disk will always be kept busy. However it is in general possible to keep several systems components busy serving different jobs.



Improved utilization can also lead to improved responsiveness

Improved utilization is good in itself, because it means that the expensive hardware you paid for is being used, rather than sitting idle. You are getting more for your money.

In addition, allowing one job to use resources left idle by another helps the first job to make progress. With luck, it will also terminate sooner. This is similar to the processor sharing idea described above, except that here we are sharing all the system resources, not just the CPU.

Finally, by removing the constraint that jobs have to wait for each other, and allowing resources to be utilized instead of being left idle, more jobs can be serviced. This is expressed as a potential increase in throughput. Realization of this potential depends on the arrival of more jobs.

Exercise 33 *In the $M/M/1$ analysis from Chapter 10, we saw that the average response time grows monotonically with utilization. Does this contradict the claims made here?*

All this depends on an appropriate job mix

The degree to which multiprogramming improves system utilization depends on the requirements of the different jobs.

If all the jobs are *compute-bound*, meaning they need a lot of CPU cycles and do not perform much I/O, the CPU will be the bottleneck. If all the jobs are *I/O-bound*, meaning that they only compute for a little while and then perform I/O operations, the disk will become a bottleneck. In either case, multiprogramming will not help much.

In order to use all the system components effectively, a suitable *job mix* is required. For example, there could be one compute-bound application, and a few I/O-bound ones. Some of the applications may require a lot of memory space, while others require only little memory.

Exercise 34 *Under what conditions is it reasonable to have only one compute-bound job, but multiple I/O-bound jobs? What about the other way around?*

The operating system can create a suitable job mix by judicious *long-term scheduling*. Jobs that complement each other will be loaded into memory and executed. Jobs that contend for the same resources as other jobs will be swapped out and have to wait.

The question remains of how to classify the jobs: is a new job going to be compute-bound or I/O bound? An estimate can be derived from the job's history. If it has already performed multiple I/O operations, it will probably continue to do so. If it has not performed any I/O, it probably will not do much in the future, but rather continue to just use the CPU.

2.2.3 Multitasking for Concurrency

When multiple applications are active concurrently they can interact

A third reason for supporting multiple processes at once is that this allows for concurrent programming, in which the multiple processes interact to work on the same problem. A typical example from Unix systems is connecting a set of processes with pipes. The first process generates some data (or reads it from a file), does some processing, and passes it on to the next process. Partitioning the computational task into a sequence of processes is done for the benefit of application structure and reduced need for buffering.

Exercise 35 *Pipes only provide sequential access to the data being piped. Why does this make sense?*

The use of multitasking is now common even on personal systems. Examples include:

- Multitasking allows several related applications to be active simultaneously. For example, this is especially common with desktop publishing systems: a word processor may embed a figure generated by a graphic editor and a graph generated by a spread-sheet. It is convenient to be able to switch among these applications dynamically rather than having to close one and open another each time.
- Multitasking allows the system to perform certain tasks in the background. For example, a fax handling application can be started when the computer is booted, but left in the background to wait for arriving faxes. This enables it to receive a fax that arrived while the user is busy with something else.

2.2.4 The Cost

Multitasking also has drawbacks, which fall into three categories:

Overhead: in order to perform a context switch (that is, stop running one process and start another), register values have to be stored in memory and re-loaded from memory. This takes instruction cycles that would otherwise be dedicated to user applications.

Degraded performance: even when the CPU is running application code, its performance may be reduced. For example, we can see

- Contention for resources such as memory: in order to run, multiple applications need their address spaces to be loaded into memory. If the total requirements exceed the physically available memory, this can lead to swapping or even thrashing (Section 4.4).
- Cache interference: switching among applications causes a corruption of cache state, leading to degraded performance due to more cache misses.

Another example is possible interference with real-time tasks, such as viewing a movie or burning a CD.

Complexity: a multitasking operating system has to deal with issues of synchronization and resource allocation (Chapter 3). If the different processes belong to different users, the system also needs to take care of security (this has been the standard in Unix since the 1970s, but supporting multiple users at once still doesn't exist on Windows desktop systems).

However, on the bottom line, the benefits of multiprogramming generally far outweigh the costs, and it is used on practically all systems.

2.3 Scheduling Processes and Threads

The previous section argued that time slicing should be used. But which process[1] should be executed at each moment? This question, which relates to the processes that are ready to run and are loaded into memory, is called *short-term scheduling*. The action of loading the process state into the CPU is known as *dispatching*.

2.3.1 Performance Metrics

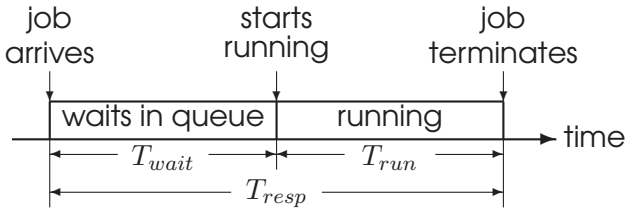
Performance depends on how you measure it

The decision about which process to schedule might depend on what you are trying to achieve. There are several possible metrics that the system could try to optimize, including

[1] Whenever we say process here, we typically also mean thread, if the system is thread based.

Response time or turnaround time — the average time from submitting a job until it terminates. This is the sum of the time spent waiting in the queue, and the time actually running:

$$T_{resp} = T_{wait} + T_{run}$$



If jobs terminate faster, users will be happier. In interactive systems, this may be more of a sharp threshold: if resopnse time is less than about 0.2 seconds, it is OK. If it is above 2 or 3 seconds, it is bad.

> **Note: Interactivity, response, and termination**
> Many interactive applications are reactive, just like the operating system; for example, a text editor spends most of its time waiting for user input, and when it gets some, it quickly handles it and returns to wait for more. Thus the notion that a job is submitted, waits, runs, and terminates seems to be irrelevant for such applications.
>
> However, we can ignore the fact that we have one continuous application running, and regard the handling of each input as a distinct job. Thus when the user hits "enter" or some other key, he is submitting a job, which waits, runs, produces a response, and terminates (or at least, becomes dormant). The time to termination is then actually the time to produce a response to the user input, which is the right metric. In essence, the model is that the application does not compute continuously, but rather performs a sequence of "CPU bursts", interspersed by I/O activity (and specifically, terminal I/O waiting for user input).
>
> In the past, many textbooks made a distinction between turnaround time which was the time till the whole application terminated, and response time which was the time until it produced its first response. This is based on a model of an application that computes continuously, and generates many outputs along the way, which does not seem to correspond to the way that any real applications work.

Variants on this idea are

Wait time — reducing the time a job waits until it runs also reduces its response time. As the system has direct control over the waiting time, but little control over the actual run time, it should focus on the wait time (T_{wait}).

Response ratio or slowdown — the ratio of the response time to the actual run time:

$$slowdown = \frac{T_{resp}}{T_{run}}$$

This normalizes all jobs to the same scale: long jobs can wait more, and don't count more than short ones.

In real systems the definitions are a bit more complicated. For example, when time slicing is used, the runtime is the sum of all the times that the job runs, and the waiting time is all the times it waits in the ready queue — but probably not the times it is waiting for an I/O operation to complete.

Throughput — the number of jobs completed in a unit of time. If there are more jobs completed, there should be more happy users.

Utilization — the average percentage of the hardware (or the CPU) that is actually used. If the utilization is high, you are getting more value for the money invested in buying the computer.

Exercise 36 *The response time can be any positive number. What are the numerical ranges for the other metrics? When are high values better, and when are low values better?*

Other desirable properties are predictability and fairness. While it is harder to quantify these properties, they seem to correlate with low variability in the service to different jobs.

The chosen metric should be one that reflects the scheduler's performance

Not all metrics are equally applicable. For example, in many real-world situations, users submit jobs according to their needs. The ratio of the requirements of all the jobs to the available resources of the system then determines the utilization, and does not directly reflect the scheduler's performance. There is only an indirect effect: a bad scheduler will discourage users from submitting more jobs, so the utilization will drop.

In a nutshell, utilization and throughput are more directly linked to the workload imposed by users than to the scheduler. They become important when the system is overloaded. Metrics related to response time are generally better for the evaluation of schedulers, especially in an interactive setting.

2.3.2 Handling a Given Set of Jobs

While the operating system in general is reactive, the scheduling algorithm itself is not. A scheduling algorithm has inputs, performs a computation, and terminates producing an output. The input is information about the jobs that need to be scheduled. the output is the schedule: a decision when to run each job, or at least a decision which job to run now.

We start by considering off-line algorithms, that only handle a given set of jobs that are all available at the outset.

Off-line means everything is known in advance

One important distinction is between on-line and off-line algorithms. Off-line algorithms receive complete information in advance. Thus they can benefit from two assumptions:

1. All jobs are available at the outset and none arrive later[2].
2. The job runtimes are also known in advance.

The assumption that all jobs are known in advance implies that this is the set of jobs that the scheduling algorithm needs to handle. This is a reasonable assumption in the context of an algorithm that is invoked repeatedly by the operating system whenever it is needed (e.g. when the situation changes because additional jobs arrive).

The assumption that runtimes are known in advance is somewhat problematic. While there are some situations in which runtimes are known in advance, in most cases this is not the case.

Off-line algorithms run jobs to completion

Given that off-line algorithms get all their required information at the outset, and that all relevant jobs are available for scheduling, they can expect no surprises during execution. In fact, they complete the schedule before execution even begins.

Under these circumstances, there is no reason to ever preempt a job. Response time and throughput metrics depend on job completion times, so once a job is started it is best to complete it as soon as possible; if running another job would improve the metric, that other job should have been scheduled in the first place. Utilization is maximized as long as any job is running, so it is not a very relevant metric for off-line algorithms.

The result is that off-line algorithms use "run to completion" (RTC): each job is executed until it terminates, and the algorithm is only concerned with the order in which jobs are started.

Exercise 37 *The above argument is correct when we are only scheduling on one resource, e.g. the CPU. Does this change if there are multiple CPUs, but each job only needs one of them? What about jobs that may need more than one CPU (that is, parallel jobs)?*

FCFS is the base case

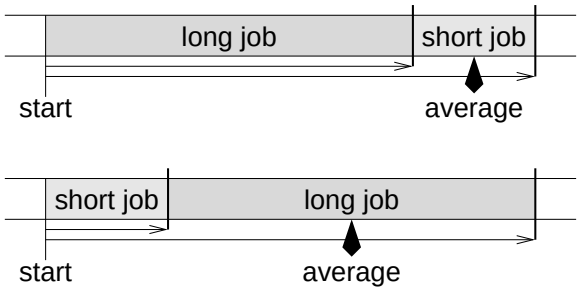
The base case for RTC scheduling algorithms is First-Come First-Serve (FCFS). This algorithm simply schedules the jobs in the order in which they arrive, and runs each

[2]Another variant allows jobs to arrive at arbitrary times, but assumes such future arrivals are also known in advance. This scenario is mainly of theoretical interest, as it provides a bound of what may be achieved with full knowledge. We will not discuss it further here.

one to completion. For the off-line case, the jobs are run in the order that they appear in the input.

Running short jobs first improves the average response time

Reordering the jobs so as to run the shortest jobs first (SJF) improves the average response time. Consider two adjacent jobs, one longer than the other. Because the total time for both jobs is constant, the second job will terminate at the same time regardless of their order. But if the shorter one is executed first, its termination time will be shorter than if the long one is executed first. As a result, the average is also reduced when the shorter job is executed first:



By repeating this argument, we see that for every two adjacent jobs, we should run the shorter one first in order to reduce the average response time. Switching pairs of jobs like this is akin to bubble-sorting the jobs in order of increasing runtime. The minimal average response time is achieved when the jobs are sorted, and the shortest ones are first.

Exercise 38 *Is SJF also optimal for other metrics, such as minimizing the average slowdown?*

But real systems are on-line

In real system you typically don't know much in advance. In particular, new jobs may arrive unexpectedly at arbitrary times. Over the lifetime of a system, the scheduler will be invoked a very large number of times, and each time there will only be a small number of new jobs. Thus it seems ill-advised to emphasize the behavior of the scheduler in a single invocation. Instead, one should consider how it handles all the arrivals that occur over a long stretch of time.

On-line algorithms get information about one job at a time, and need to decide immediately what to do with it: either schedule it or put it in a queue to be scheduled later. The decision may of course be influenced by the past (e.g. by previously arrived jobs that are already in the queue), but not by the future.

The on-line model is also applicable to interactive or I/O-bound jobs, which have bursts of CPU activity separated by I/O operations (for interactive jobs, this is terminal I/O). Whenever an I/O operation completes, the job has to be scheduled again for its next CPU burst.

An important tool for handling a changing situation is preemption.

2.3.3 Using Preemption

Preemption is the action of stopping a running job and scheduling another in its place. Switching from one job to another is called a *context switch*. Technically, this is done by storing the CPU register values of the running process in its PCB, selecting an alternative process that is ready to run, and loading the CPU register values of the selected process. As this includes the PC, the CPU will now continue to run the selected process from where is left off when it was preempted.

On-line algorithms use preemption to handle changing conditions

On-line algorithms do not know about their input in advance: they get it piecemeal as time advances. Therefore they might make a scheduling decision based on current data, and then regret it when an additional job arrives. The solution is to use preemption in order to undo the previous decision.

We start by reversing the first of the two assumptions made earlier. Thus we now assume that

1. Jobs may arrive unpredictably and the scheduler must deal with those that have already arrived without knowing about future arrivals
2. Nevertheless, when a job arrives its runtime is known in advance.

The version of SJF used in this context is called "shortest remaining time first" (SRT)[3]. As each new job arrives, its runtime is compared with the *remaining* runtime of the currently running job. If the new job is shorter, the current job is preempted and the new job is run in its place. Otherwise the current job is allowed to continue its execution, and the new job is placed in the appropriate place in the sorted queue.

Exercise 39 *Is SRT also optimal for other metrics, such as minimizing the average slowdown?*

A problem with actually using this algorithm is the assumption that the run times of jobs are known. This may be allowed in the theoretical setting of off-line algorithms, but is usually not the case in real systems.

An interesting counter example is provided by web servers that only serve static pages. In this case, the service time of a page is essentially proportional to the page

[3]Sometimes also called "shortest remaining *processing* time first", or SRPT.

size. Thus when a request for a certain page arrives, we can get a pretty accurate assessment of how long it will take to serve, based on the requested page's size. Scheduling web servers in this way turns out to improve performance significantly for small pages, without too much effect on large ones [6].

Preemption can also compensate for lack of knowledge

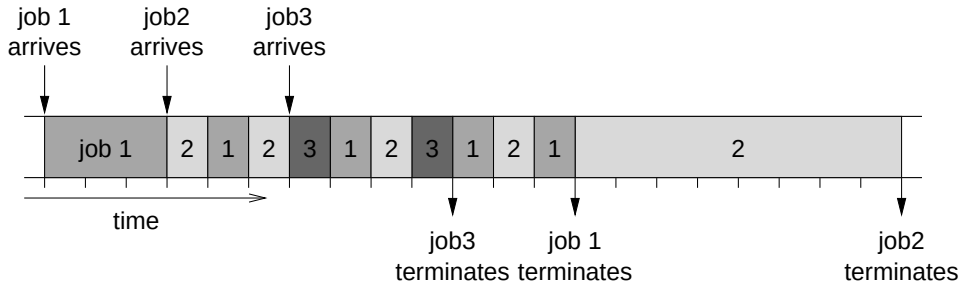
SRT only preempts the current job when a shorter one arrives, which relies on the assumption that runtimes are known in advance. But a more realistic set of assumptions is that

1. Jobs may arrive unpredictably, and
2. Job runtimes are *not* known in advance.

Using more preemptions can compensate for this lack of knowledge. The idea is to schedule each job for a short *time quantum*, and then preempt it and schedule another job in its place. The jobs are scheduled in *round robin* order: a cycle is formed, and each gets one time quantum. Thus the delay until a new job gets to run is limited to one cycle, which is the product of the number of jobs times the length of the time quantum. If a job is short, it will terminate within its first quantum, and have a relatively short response time. If it is long, it will have to wait for additional quanta. The system does not have to know in advance which jobs are short and which are long.

Exercise 40 *Round robin scheduling is often implemented by keeping a circular linked list of the PCBs, and a pointer to the current one. Is it better to insert new jobs just before the current pointer or just after it?*

Note that when each process just runs for a short time, we are actually time slicing the CPU. This results in a viable approximation to processor sharing, which was shown on page 36 to prevent situations in which a short job gets stuck behind a long one. In fact, the time it takes to run each job is more-or-less proportional to its own length, multiplied by the current load. This is beneficial due to the high variability of process runtimes: most are very short, but some are very long.



Exercise 41 *Should the time slices be long or short? Why?*

Using preemption regularly ensures that the system stays in control

But how are time quanta enforced? This is another example where the operating system needs some help from the hardware. The trick is to have a hardware clock that causes periodic interrupts (e.g. 100 times a second, or every 10 ms). This interrupt, like all other interrupts, is handled by a special operating system function. Specifically, this handler may call the scheduler which may decide to preempt the currently running process.

An important benefit of having such periodic clock interrupts is that they ensure that the system regains control over the CPU. Without them, a rogue process may never relinquish the CPU, and prevent all other processes from using the system.

In reality, however, the question of scheduling quanta is almost moot. In the vast majority of cases, the *effective quantum* is much shorter than the allocated quantum. This is so because in most cases the process performs a system call or an external interrupt happens [4]. The clock interrupt therefore serves mainly as a backup.

And it also improves fairness

Finally, we note that even if processes do not engage in infinite loops, some are compute-bound while others are interactive (or I/O-bound). Without preemption, CPU-bound processes may lock out the interactive processes for excessive periods, which is undesirable. Preempting them regularly allows interactive processes to get a chance to run, at least once in a while. But if there are many compute-bound processes, this may not be enough. The solution is then to give the interactive processes higher priorities.

2.3.4 Priority Scheduling

Interactive jobs can get priority based on past behavior

Round robin scheduling is oblivious — it does not take into account any information about previous behavior of the processes. All processes receive equal treatment.

However, the system can easily accumulate information about the processes, and *prioritize* them. Interactive jobs, such as a text editor, typically interleave short bursts of CPU activity with I/O operations to the terminal. In order to improve responsiveness, the system should give high priority to such jobs. This can be done by regarding the CPU burst as the unit of computing, and scheduling processes with short bursts first (this is a variant of SJF).

The question remains of how to estimate the duration of the next burst. One option is to assume it will be like the last one. Another is to use an average of all previous bursts. Typically a weighted average is used, so recent activity has a greater influence on the estimate. For example, we can define the $n+1$st estimate as

$$E_{n+1} = \frac{1}{2}T_n + \frac{1}{4}T_{n-1} + \frac{1}{8}T_{n-2} + \frac{1}{16}T_{n-3} + \ldots$$

where E is the estimate, T is the duration of the burst, and as bursts become more distant they are given half the weight (a simple generalization is to use another factor $0 \leq \alpha \leq 1$ to set the relative weights).

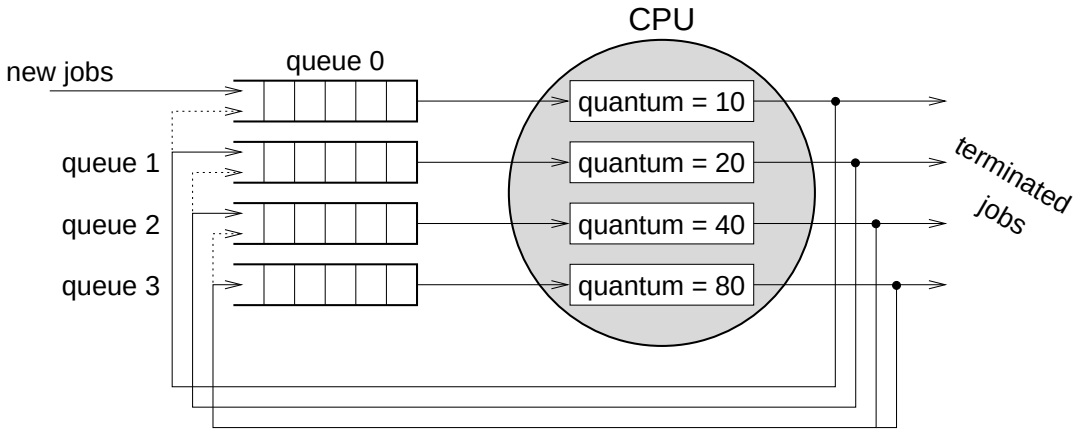
Exercise 42 *Can this be computed without storing information about all the previous bursts?*

Multi-level feedback queues learn from past behavior

Multi-level feedback queues are a mechanism to differentiate among processes based on their past behavior. It is similar to the SJF-like policy described above, but simpler and does not require the system to maintain a lot of historical information. In effect, the execution history of the process is encoded in the queue in which it resides, and it passes from queue to queue as it accumulates CPU time or blocks on I/O.

New processes and processes that have completed I/O operations are placed in the first queue, where they have a high priority and receive a short quantum. If they do not block or terminate within this quantum, they move to the next queue, where they have a lower priority (so they wait longer), but then they get a longer quantum. On the other hand, if they do not complete their allocated quantum, they either stay in the same queue or even move back up one step.

In this way, a series of queues is created: each additional queue holds jobs that have already run for a longer time, so they are expected to continue to run for a long time. Their priority is reduced so that they will not interfere with other jobs that are assumed to be shorter, but when they do run they are given a longer quantum to reduce the overhead of context switching. The scheduler always serves the lowest-numbered non-empty queue.



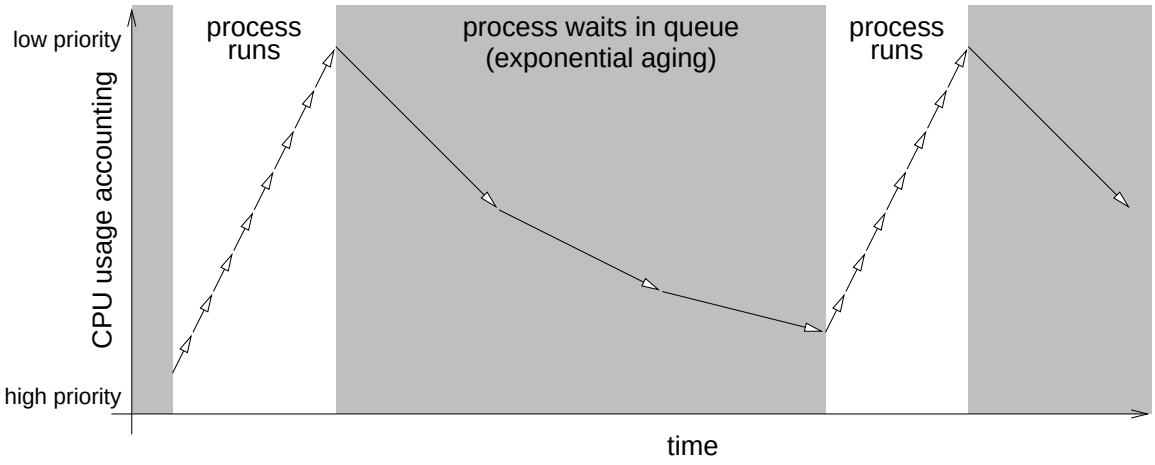
In Unix, priority is set by CPU accounting

The Unix scheduler also prioritizes the ready processes based on CPU usage, and schedules the one with the highest priority (which is the *lowest* numerical value).

The equation used to calculate user-level priorities is the following (when running in kernel mode, the priority is fixed):

$$pri = cpu\_use + base + nice$$

cpu_use is recent CPU usage. This value is incremented for the running process on every clock interrupt (typically 100 times a second). Thus the priority of a process goes down as it runs. In order to adjust to changing conditions, and not to over-penalize long running jobs, the cpu_use value is divided in two for all processes once a second (this is called exponential aging). Thus the priority of a process goes up as it waits in the ready queue.



$base$ is the base priority for user processes, and distinguishes them from kernel priorities. A process that goes to sleep in a system call, e.g. when waiting for a disk operation to complete, will have a higher priority. Thus when it wakes up it will have preference over all user-level processes. This will allow it to complete the system call and release kernel resources. When it returns to user mode, its priority will be reduced again.

$nice$ is an optional additional term that users may use to reduce the priority of their processes, in order to be nice to their colleagues.

Exercise 43 *Does the Unix scheduler give preference to interactive jobs over CPU-bound jobs? If so, how?*

Exercise 44 *Can a Unix process suffer from starvation?*

The implementation of Unix scheduling is essentially similar to multi-level feedback queues, except that time quanta are not varied. As CPU accounting is only done at a rather coarse granularity, there are actually a relatively small number of possible priority levels: in most systems this is 32, 64, or 128 (if the range of possible accounting values is larger, it is scaled to fit). The system maintains an array of pointers, one for each priority level. When a process is preempted or blocks, it is linked to the pointer that matches its current priority. The scheduler always chooses the process at the head of the topmost non-empty list for execution.

Exercise 45 *Assuming that CPU usage is accounted at a rate of 100Hz (that is, the usage of the running process is incremented by 100 every 10ms), and that all accounts are halved once a second, what is the maximal priority value that a process may achieve?*

To read more: Unix scheduling is described in Bach [1, Sect. 8.1] (system V) and in McKusick [11, Sect. 4.4] (4.4BSD). The BSD formula is slightly more complicated.

2.3.5 Starvation, Stability, and Allocations

Starvation may be a problem

The problem with priority scheduling algorithms like multi-level feedback is that they run the risk of *starving* long jobs. As short jobs continue to arrive, they continuously populate the first queue, and the long jobs in lower queues never get to run.

But selective starvation may be a feature

However, it is debatable whether this is a real problem, as the continued arrival of short jobs implies that the system is overloaded, and actually cannot handle the load imposed on it. Under such conditions, jobs will necessarily be delayed by arbitrary amounts. Using multi-level queues just makes a distinction, and only delays the long jobs, while allowing the short ones to complete without undue delay.

Exercise 46 *Which of the previously described algorithms also suffers from starvation?*

The effect of the runtime distribution on starvation is also important. When the distribution is fat-tailed, a very small fraction of the jobs is responsible for a rather large fraction of the load (where load is defined here as CPU seconds of actual computation). By starving only these jobs, the system can tolerate overload while still providing adequate service to nearly all jobs [2].

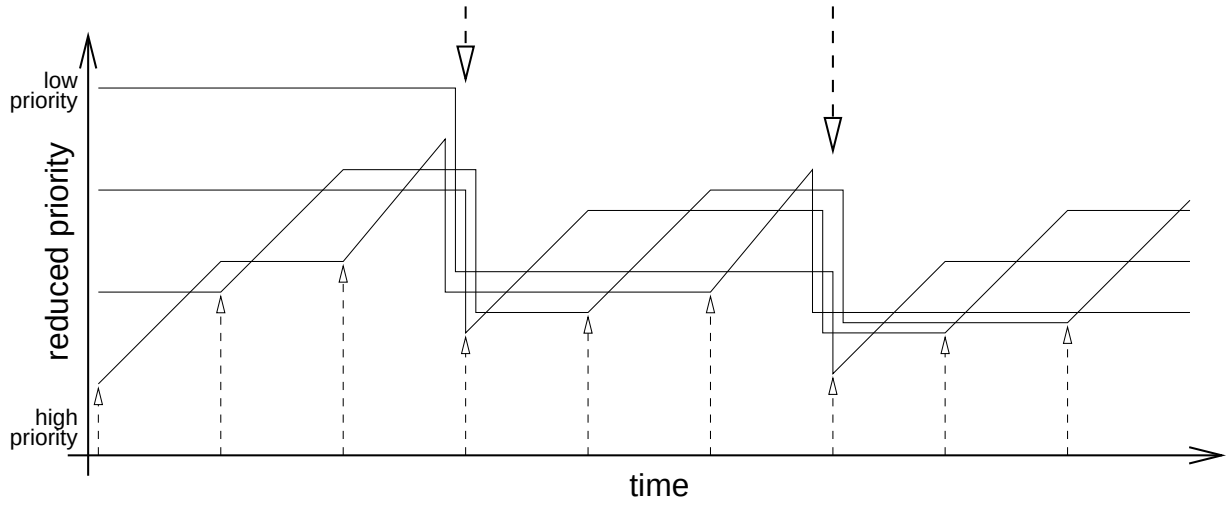
In fact, this is an example of a more general principle. When a system is overloaded, it is extremely important to prioritize its work. Without prioritization, all processes are equal and all suffer an infinite delay! (at least for true processor sharing; short ones will eventually terminate when using finite quanta). Prioritization allows the system to selectively reject some jobs, at the same time giving reasonable service to others. As rejecting some jobs is unavoidable in overloaded conditions, it is better to do so based on system considerations and not at random.

Unix avoids starvation by having negative feedback

Starvation depends on prioritization being a one-way street. In multi-level feedback queues, as described above, a process's priority can only drop. As a result, it may starve as new processes with higher priorities continue to arrive.

But in the Unix scheduler priority is a two-way street: a process's priority drops as it runs and accumulates CPU seconds, but it grows when it waits in the queue. This leads to a stabilizing negative feedback effect: the very act of running reduces your ability to run more.

An example of how the priorities of 4 processes change over time is shown in the following figure. Periodic scheduling decisions (that choose the process with the lowest priority value) are indicated by dashed arrows from below, and periodic divisions of all priorities by 2 are indicated by arrows from above. Even though the processes start with quite different priorities, they all get to run, and their priorities tend to converge to a relatively narrow range.



The result of this behavior is that CPU allocations tend to be equalized — if there are many competing processes, they will eventually all get about the same share of the CPU. The only way that a process may get less than its peers is if it needs less, e.g. if it often blocks on I/O. As a result it will have a higher priority, and on those occasions when it wants to run, it will get the processor first.

Linux avoids starvation by using allocations

An alternative approach for avoiding starvation is to impose pre-defined allocations. Each process is given an allocation of CPU seconds, and then they are all allowed to use up their allocations before anyone's allocation is renewed.

An allocation-based scheme is used in the Linux system. Linux divides its time into epochs. At the beginning of each epoch, every process receives an allocation. The allocation also doubles as a priority, so processes with a larger allocation also get a higher priority. The scheduler always selects the process with the highest priority (that is, the process with the largest remaining allocation) to run. But allocations may run out. When this happens, the process is effectively prevented from running until the epoch ends.

The duration of epochs is somewhat flexible. An epoch ends, and all allocations are renewed, when the scheduler finds that it has no process that can be scheduled to run. This may happen because all processes have exhausted their allocations, or because all processes with remaining allocations are currently blocked waiting for various events.

Technically all this is done by keeping processes on either of two arrays: the active array and the expired array. Initially all processes are on the active array and eligible for scheduling. Each processes that exhausts its allocation is moved to the expired array. When there are no more runnable processes on the active array, the epoch ends and the arrays are switched.

Note that Linux sets process allocations based on scheduling considerations. But users or system administrators may also want to be able to control the relative allocations of different processes. This is enabled by fair-share scheduling, dicussed below.

But allocations may affect interactivity

The Linux approach has a subtle effect on interactive processes. The crucial point is that allocations and priorities are correlated (in fact, they are the same number). Therefore if we have several high-priority interactive processes, each may run for a relatively long time before giving up the processor. As a result such interactive may have to wait a long time to get their turn to run.

The allocations made by the simple basic multi-level feedback queues scheme avoid this problem [**?**]. In this scheme, the effective quanta are inversely proportional to the priority: the higher the priority, the shorter each allocation. This ensures rapid cycling among the high priority processes, and thus short latencies until a process gets scheduled.

2.3.6 Fair Share Scheduling

Most schedulers attempt to serve all jobs as best they can, subject to the fact that jobs have different characteristics. Fair share scheduling is concerned with achieving predefined goals. This is related to quality of service guarantees.

Administrative considerations may define “fairness”

Fair share scheduling tries to give each job what it deserves to get. However, “fair” does not necessarily imply “equal”. Deciding how to share the system is an administrative policy issue, to be handled by the system administrators. The scheduler should only provide to tools to implement the chosen policy. For example, a fair share of the machine resources might be defined to be “proportional to how much you paid when the machine was acquired”. Another possible criterion is your importance in the

organization. Your fair share can also change with time, reflecting the current importance of the project you are working on. When a customer demo is being prepared, those working on it get more resources than those working on the next generation of the system.

Shares can be implemented by manipulating the scheduling

A simple method for fair share scheduling is to add a term that reflects resource usage to the priority equation. For example, if we want to control the share of the resources acquired by a group of users, we can add a term that reflects cumulative resource usage by the group members to the priority equation, so as to reduce their priority as they use more resources. This can be "aged" with time to allow them access again after a period of low usage. The problem with this approach is that it is hard to translate the priority differences into actual shares [8, 3].

An alternative approach is to manipulate the time quanta: processes that should receive more runtime are just given longer quanta. The problem with this approach is that the actual runtime they receive is the product of the quantum length and how often they are scheduled to run. It may therefore be necessary to monitor the actual shares and adjust the quanta dynamically to achieve the desired division of resources.

Exercise 47 *This presentation provides fair shares at the expense of performance-driven priorities as used in the multi-level feedback scheme described above. Can the two be combined?*

To read more: Various other schemes for fair-share scheduling have been proposed in the literature. Two very nice ones are lottery scheduling, which is based on a probabilistic allocation of resources [16], and virtual time round robin (VTRR), which uses a clever manipulation of the queue [12].

2.4 Summary

Abstractions

A process is essentially an abstraction of a computer. This is one of the major abstractions provided by multiprogrammed operating systems. It provides the operating environment for applications, which is based on the hardware, but with some changes such as lack of access to privileged instructions. But the important point is the isolation from other applications, running as other processes. Each one only sees its own resources, and is oblivious to the activities of the others.

Another important abstraction is that of a thread. This actually breaks the process abstraction into two parts: the treads, which can be viewed as an abstraction of the CPU and the flow of the computation, and the process, which abstracts the environment including the address space, open files, etc. For single-threaded processes

the distinction is of course moot, and we can say that the process abstracts the whole computer.

Implementation

Most modern operating systems support processes and threads directly. They have one system call to create a process, and another to create an additional thread within a process. Then there are many other system calls to manipulate processes and threads. For example, this is the situation with Windows.

Unix is somewhat unique, in providing the `fork` system call. Instead of creating a new process from scratch, this creates a clone of the calling process. This new "child" process can then use the `exec` system call to load another executable program instead of the one running in the parent process.

Linux is also unique. It does not have a distinction between processes and threads. In effect, it only has threads (but they are called "tasks"). New ones are created by the `clone` system call, which is similar to `fork`, but provides detailed control over exactly what is shared between the parent and child.

Resource management

Threads abstract the CPU, and this is the main resource that needs to be managed. Scheduling — which is what "resource management" is in this context — is a hard problem. It requires detailed knowledge, e.g. how long a job will run, which is typically not available. And then it turns out to be NP-complete.

However, this doesn't mean that operating systems can't do anything. The main idea is to use preemption. This allows the operating system to learn about the behavior of different jobs, and to reconsider its decisions periodically. This is not as pompous as it sounds, and is usually embodied by simple priority rules and simple data structures like multi-level feedback queues.

Workload issues

Periodical preemptions are not guaranteed to improve average response times, but they do. The reason is that the distribution of process runtimes is heavy-tailed. This is one of the best examples of a widespread operating system policy that is based on an empirical observation about workloads.

Hardware support

There are two types of hardware support that are related to processes. One is having a clock interrupt, and using this to regain control, to perform preemptive scheduling, and to implement timers.

The other is support for isolation — preventing each process from seeing stuff that belongs to another process. This is a main feature of memory management mechanisms, so we will review it in Chapter 4.

Bibliography

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

[2] N. Bansal and M. Harchol-Balter, "*Analysis of SRPT scheduling: investigating unfairness*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 279–290, Jun 2001.

[3] D. H. J. Epema, "*Decay-usage scheduling in multiprocessors*". *ACM Trans. Comput. Syst.* **16(4)**, pp. 367–415, Nov 1998.

[4] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[5] J. L. Guynes, "*Impact of system response time on state anxiety*". *Comm. ACM* **31(3)**, pp. 342–347, Mar 1988.

[6] M. Harchol-Balter, N. Bansal, B. Schroeder, and M. Agrawal, "*SRPT scheduling for web servers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 11–20, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[7] M. Harchol-Balter and A. B. Downey, "*Exploiting process lifetime distributions for dynamic load balancing*". *ACM Trans. Comput. Syst.* **15(3)**, pp. 253–285, Aug 1997.

[8] G. J. Henry, "*The fair share scheduler*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1845–1857, Oct 1984.

[9] E. D. Lazowska, "*The use of percentiles in modeling CPU service time distributions*". In *Computer Performance*, K. M. Chandy and M. Reiser (eds.), pp. 53–66, North-Holland, 1977.

[10] W. E. Leland and T. J. Ott, "*Load-balancing heuristics and process behavior*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 54–69, 1986.

[11] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[12] J. Nieh, C. Vaill, and H. Zhong, "*Virtual-Time Round Robin: an O(1) proportional share scheduler*". In *USENIX Ann. Technical Conf.*, pp. 245–259, Jun 2001.

[13] R. C. Regis, "*Multiserver queueing models of multiprocessing systems*". *IEEE Trans. Comput.* **C-22(8)**, pp. 736–745, Aug 1973.

[14] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[15] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[16] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.

Appendix B

UNIX Processes

Unix started out as a very simple operating system, designed and implemented by two programmers at AT&T. They subsequently received the Turing award (the Nobel prize of computer science) for this work.

It took some time, but eventually AT&T turned Unix into a commercial product. Some current versions of Unix, such as Solaris, have their roots in that version. At the same time another variant of Unix was designed and implemented at Berkeley, and was distributed under the name BSD (for Berkeley Software Distribution). IBM wrote their own version, called AIX. Linux is also and independent version largely unrelated to the others. The main unifying aspect of all these systems is that they support the same basic system calls, although each has its own extensions.

One unique feature in Unix is how processes are created. This is somewhat anachronistic today, and would probably be done differently if designed from scratch. However, it is interesting enough for an appendix.

To read more: Unix was widely adopted in academia, and as a result there is a lot of written material about it. Perhaps the best known classic is Bach's book on Unix version V [1]. Another well-known book is McKusick et al. who describe the BSD version [2].

Unix processes are generally similar to the description given in Chapter 2. However, there are some interesting details.

The PCB is divided in two

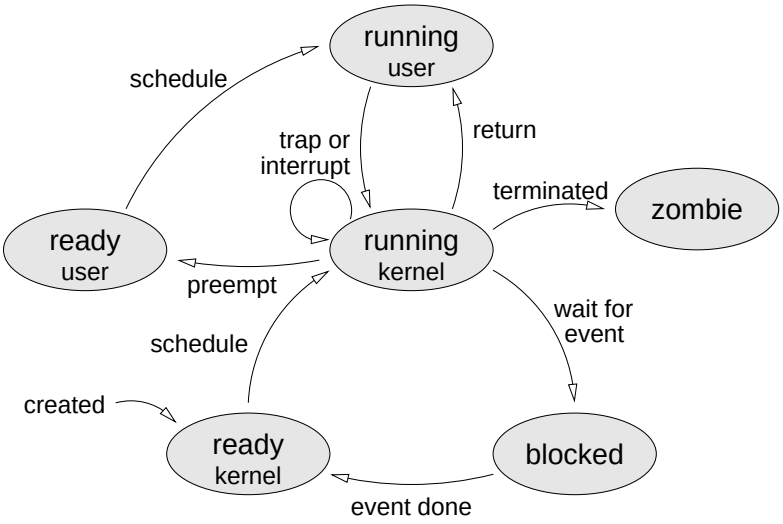
The Unix equivalent of a PCB is the combination of two data structures. The data items that the kernel may need at any time are contained in the process's entry in the process table (including priority information to decide when to schedule the process). The data items that are only needed when the process is currently running are contained in the process's u-area (including the tables of file descriptors and signal handlers). The kernel is designed so that at any given moment the current process's

u-area is mapped to the same memory addresses, and therefore the data there can be accessed uniformly without process-related indirection.

Exercise 48 *Should information about the user be in the process table or the u-area? Hint: it's in the process table. Why is this surprising? Can you imagine why is it there anyway?*

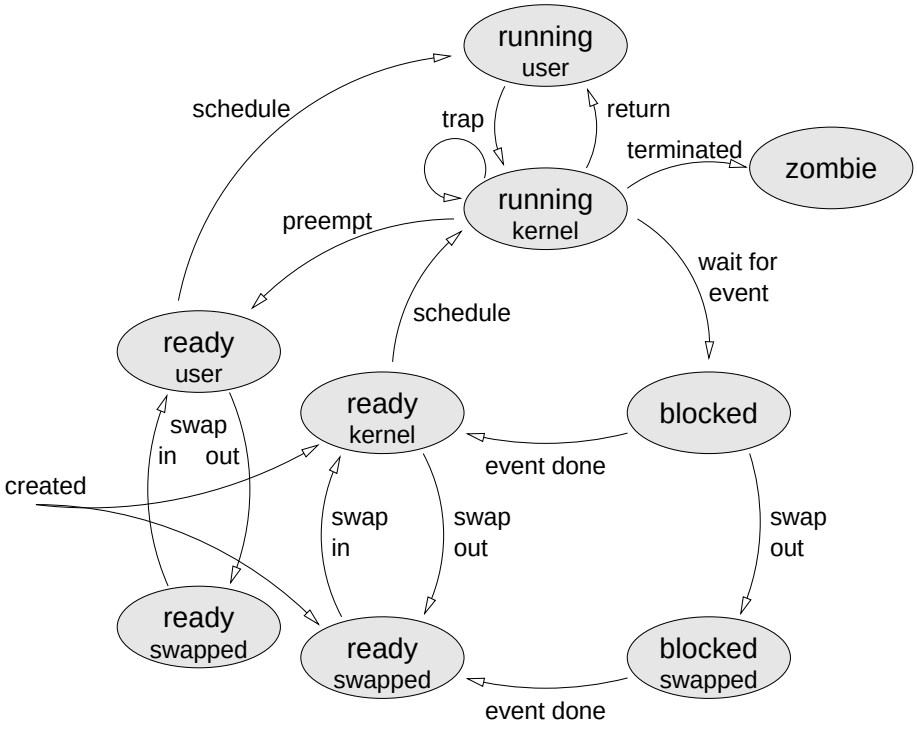
There are many states

The basic process state graph in Unix is slightly more complicated than the one introduced above, and looks like this:



Note that the *running* state has been divided into two: running in user mode and in kernel mode. This is because Unix kernel routines typically run within the context of the current user process, rather than having a separate environment for the kernel. The *ready* state is also illustrated as two states: one is for preempted processes that will continue to run in user mode when scheduled, and the other is for processes that blocked in a system call and need to complete the system call in kernel mode (the implementation actually has only one joint ready queue, but processes in kernel mode have higher priority and will run first). The zombie state is for processes that terminate, but are still kept in the system. This is done in case another process will later issue the `wait` system call and check for their termination.

When swapping is considered, even more states are added to the graph, as shown here:



As previously, the distinction between processes that are ready to run in kernel mode and those that are ready to return to user mode is artificial — this is actually the same state. The distinction is made for illustrative reasons, to stress that only processes that are ready to return to user mode may be preempted. When new processes are created, they may be placed in the swapped out state if enough memory is not available. The default, however, is to make them ready to run in memory.

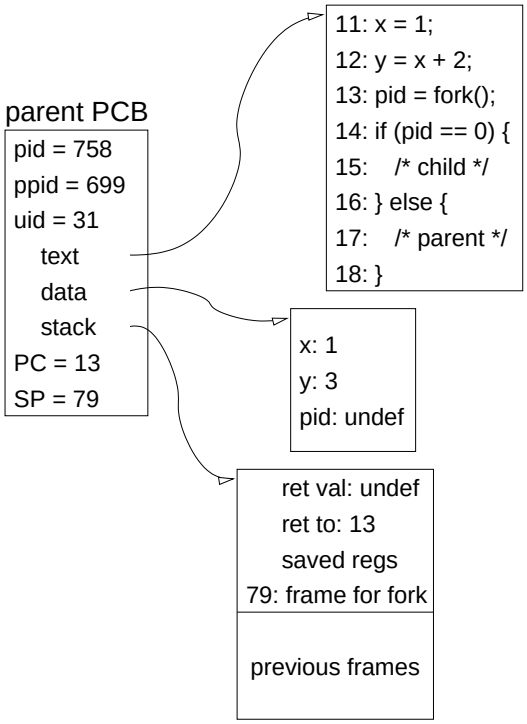
Exercise 49 *Why isn't the* blocked *state divided into blocked in user mode and blocked in kernel mode?*

Exercise 50 *The arrow from* ready user *to* running user *shown in this graph does not really exist in practice. Why?*

The `fork` system call duplicates a process

In Unix, new processes are not created from scratch. Rather, any process can create a new process by *duplicating itself*. This is done by calling the `fork` system call. The new process will be identical to its parent process: it has the same data, executes the same program, and in fact is at exactly the same place in the execution. The only differences are their process IDs and the return value from the `fork`.

Consider a process structured schematically as in the following figure:

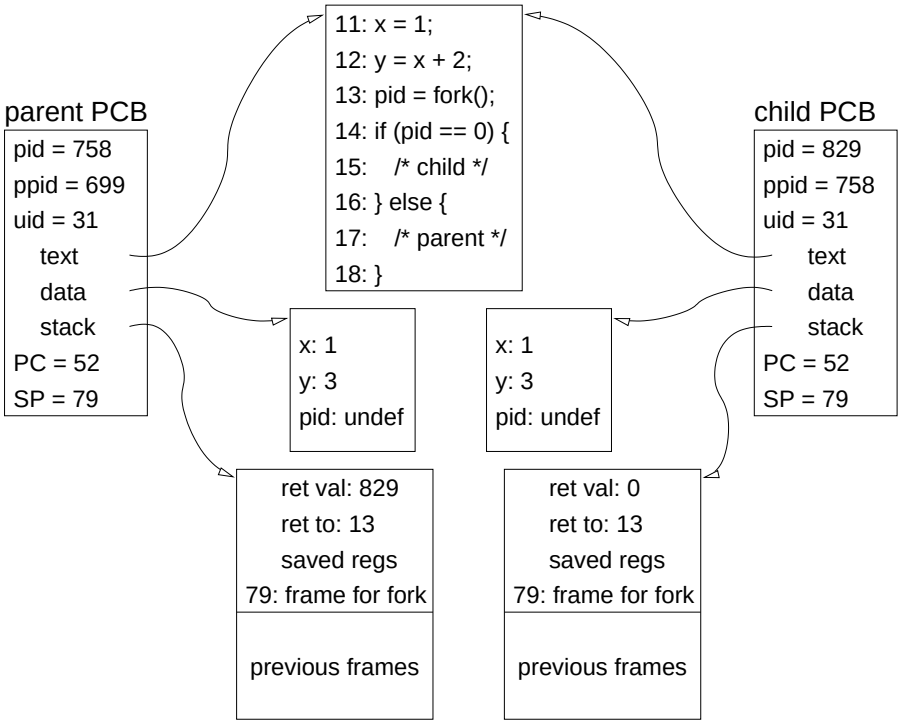


It has a process ID (pid) of 758, a user ID (uid) of 31, text, data, and stack segments, and so on (the "pid" in the data segment is the name of the variable that is assigned in instruction 13; the process ID is stored in the PCB). Its program counter (PC) is on instruction 13, the call to `fork`.

Calling `fork` causes a trap to the operating system. From this point, the process is not running any more. The operating system is running, in the `fork` function. This function examines the process and duplicates it.

First, `fork` allocates all the resources needed by the new process. This includes a new PCB and memory for a copy of the address space (including the stack). Then the contents of the parent PCB are copied into the new PCB, and the contents of the parent address space are copied into the new address space. The text segment, with the program code, need not be copied. It is shared by both processes.

The result is two processes as shown here:



The new process has a process ID of 829, and a parent process ID (ppid) of 758 — as one might expect. The user ID and all other attributes are identical to the parent. The address space is also an exact copy, except for the stack, where different return values are indicated: in the parent, `fork` will return the process ID of the new child process, whereas in the child, it will return 0. When the processes are scheduled to run, they will continue from the same place — the end of the `fork`, indicated by a PC value of 52. When the `fork` function actually returns, the different return values will be assigned to the variable pid, allowing the two processes to diverge and perform different computations.

Exercise 51 *What is the state of the newly created process?*

Note that as the system completes the `fork`, it is left with two ready processes: the parent and the child. These will be scheduled at the discretion of scheduler. In principle, either may run before the other.

To summarize, `fork` is a very special system call: it is "a system call that returns twice" — in two separate processes. These processes typically branch on the return value from the `fork`, and do different things.

The exec system call replaces the program being executed

In many cases, the child process calls the `exec` system call which replaces the program that is being executed. This means

1. Replace the text segment with that of the new program.

2. Replace the data segment with that of the new program, initialized as directed by the compiler.
3. Re-initialize the heap.
4. Re-initialize the stack.
5. Point the program counter to the program's entry point.

When the process is subsequently scheduled to run, it will start executing the new program. Thus `exec` is also a very special system call: it is "a system call that never returns", because (if it succeeds) the context in which it was called does not exist anymore.

Exercise 52 *One of the few things that the new program should inherit from the old one is the environment (the set of* $\langle name, value \rangle$ *pairs of environment variables and their values). How can this be done if the whole address space is re-initialized?*

The environment can be modified between the `fork` and the `exec`

While `exec` replaces the program being run, it does not re-initialize the whole environment. In particular, the new program inherits open files from its predecessor. This is used when setting up pipes, and is the reason for keeping `fork` and `exec` separate. It is described in more detail in Section 12.2.4.

Originally, Unix did not support threads

Supporting threads in the operating system should be included in the operating system design from the outset. But the original Unix systems from the 1970s did not have threads (or in other words, each process had only one thread of execution). This caused various complications when threads were added to modern implementations. For example, in Unix processes are created with the `fork` system call, which duplicates the process which called it as described above. But with threads, the semantics of this system call become unclear: should all the threads in the forked process be duplicated in the new one? Or maybe only the calling thread? Another example is the practice of storing the error code of a failed system call in the global variable `errno`. With threads, different threads may call different system calls at the same time, and the error values will overwrite each other if a global variable is used.

Bibliography

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

[2] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

Chapter 3

Concurrency

Given that the operating system supports multiple processes, there may be various interactions among them. We will study three rather different types of interactions:

- Access to shared operating system data structures.

  This issue is concerned with internal operating system integrity. The problem is that several processes may request the operating system to take related actions, which require updates to internal operating system data structures. Note that there is only one operating system, but many processes. Therefore the operating system data structures are shared in some way by all the processes. Updates to such shared data structures must be made with care, so that data is not corrupted.

- Deadlock due to resource contention.

  This issue is concerned with the resource management functionality of the operating system. Consider a scenario in which one application acquires lots of memory, and another acquires access to a tape drive. Then the first application requests the tape, and the second requests more memory. Neither request can be satisfied, and both applications are stuck, because each wants what the other has. This is called deadlock, and should be avoided.

- Mechanisms for user-level inter-process communication.

  This issue is concerned with the abstraction and services functionality of the operating system. The point is that multiple processes may benefit from interacting with each other, e.g. as part of a parallel or distributed application. The operating system has to provide the mechanisms for processes to identify each other and to move data from one to the other.

We'll discuss the first two here, and the third in Chapter 12.

3.1 Mutual Exclusion for Shared Data Structures

An operating system is an instance of concurrent programming. This means that multiple activities may be ongoing at the same time. For example, a process may make a system call, and while the system call is running, an interrupt may occur. Thus two different executions of operating system code — the system call and the interrupt handler — are active at the same time.

3.1.1 Concurrency and the Synchronization Problem

Concurrency can happen on a single processor

Concurrency does not necessarily imply parallelism.

In a *parallel* program, different activities actually occur simultaneously on different processors. That is, they occur at the same time in different locations.

In a *concurrent* program, different activities are interleaved with each other. This may happen because they really occur in parallel, but they may also be interleaved on the same processor. That is, one activity is started, but before it completes it is put aside and another is also started. Then this second activity is preempted and a third is started. In this way many different activities are underway at the same time, although only one of them is actually running on the CPU at any given instant.

The operating system is a concurrent program

An operating system is such a concurrent program. It has multiple entry points, and several may be active at the same time. For example, this can happen if one process makes a system call, blocks within the system call (e.g. waiting for a disk operation to complete), and while it is waiting another process makes a system call. Another example is that an interrupt may occur while a system call is being serviced. In this case the system call handler is preempted in favor of the interrupt handler, which is another operating system activity.

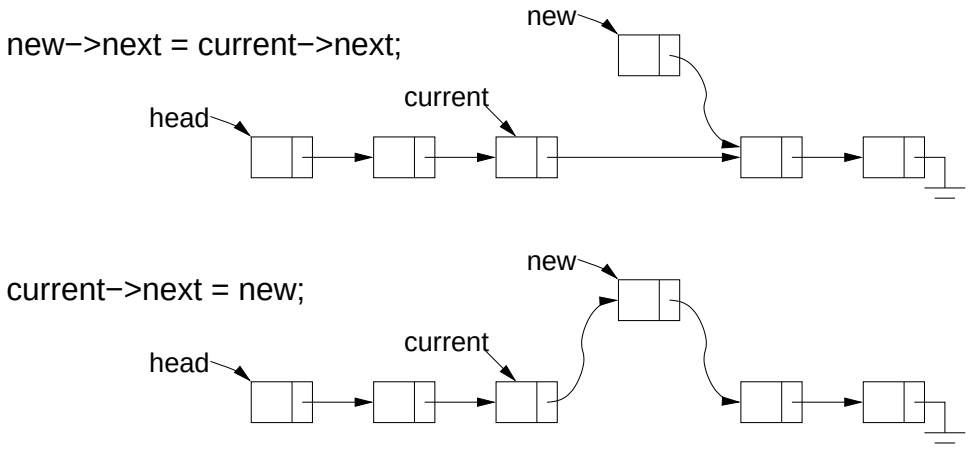
As operating system activities are often executed on behalf of user processes, in the sequel we will usually talk of processes that are active concurrently. But we typically mean operating system activities on behalf of these processes. On the other hand, user-level processes (or threads) may also cooperatively form a concurrent program, and exactly the same problems and solutions apply in that case as well.

Concurrent updates may corrupt data structures

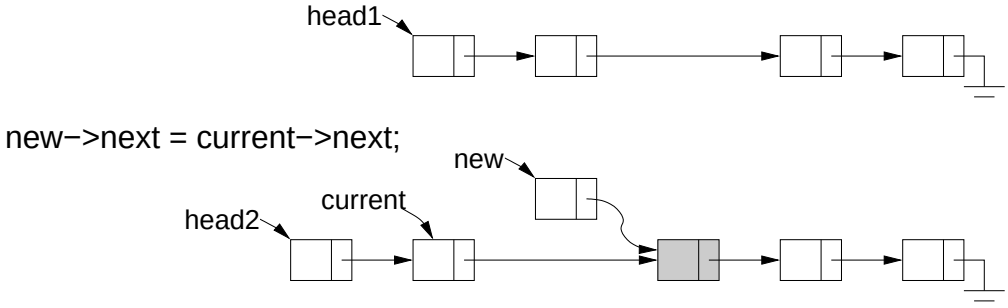
The problem with concurrency is that various operations require multiple steps in order to complete. If an activity is interrupted in the middle, the data structures on which it operated may be in an inconsistent state. Such a situation in which the

outcome depends on the relative speed and the order of interleaving is called a *race condition*.

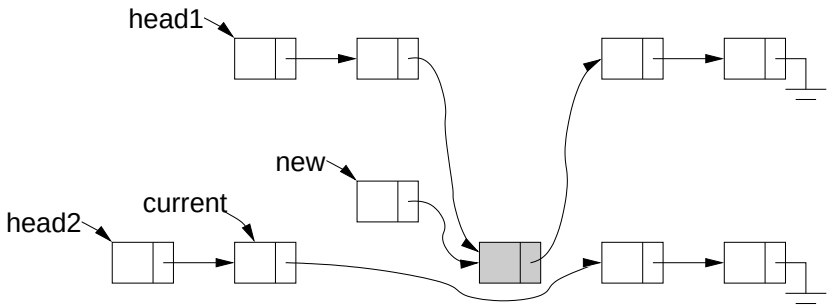
Consider adding a new element to a linked list. The code to insert the element pointed to by `new` after the element pointed to by `current` is trivial and consists of two statements:



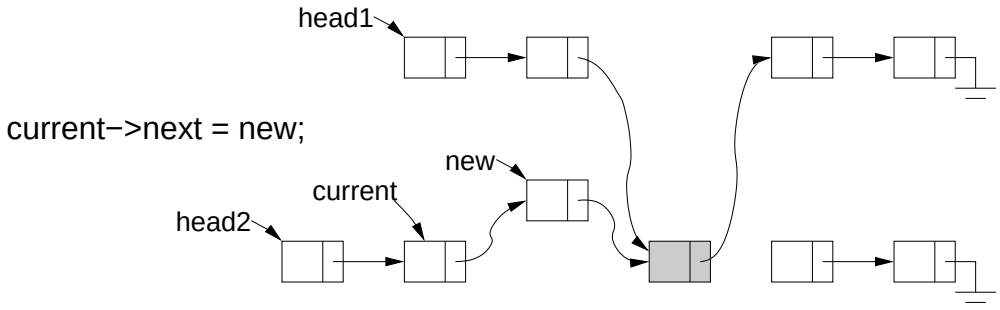
But what if the activity is interrupted between these two statements? `new->next` has already been set to point to another element, but this may no longer be valid when the activity resumes! For example, the intervening activity may *delete* the pointed element from the list, and insert it into another list! Let's look at what happens in detail. Initially, there are two lists, and we are inserting the new element into one of them:



Now we are interrupted. The interrupting activity moves the gray item we are pointing at into the other list, and updates `current->next` correctly. It doesn't know about our new item, because we have not made it part of the list yet:



However, we don't know about this intervention. So when we resume execution we therefore overwrite `current->next` and make it point to our new item:



As a result, the two lists become merged from the gray element till their end, and two items are completely lost: they no longer appear in any list! It should be clear that this is very very bad.

Exercise 53 *What would happen if the interrupting activity also inserted a new item after* `current`*, instead of moving the gray item? And what would happen if the interrupting activity inserted a new item after the gray item?*

Exercise 54 *Can you think of a scenario in which only the new item is lost (meaning that it is not linked to any list)?*

Does this happen in real life? You bet it does. Probably the most infamous example is from the software controlling the Therac-25, a radiation machine used to treat cancer patients. A few *died* due to massive overdose induced by using the wrong data. The problem was traced to lack of synchronization among competing threads [13].

3.1.2 Mutual Exclusion Algorithms

The solution is to define mutually exclusive critical sections

The solution is that all the steps required to complete an multi-step action must be done *atomically*, that is, as a single indivisible unit. The activity performing them must not be interrupted. It will then leave the data structures in a consistent state.

This idea is translated into program code by identifying *critical sections* that will be executed atomically. Thus the code to insert an item into a linked list will be

```
begin_critical_section;
new->next = current->next;
current->next = new;
end_critical_section;
```

`begin_critical_section` is a special piece of code with a strange behavior: if one activity passes it, another activity that also tries to pass it will get stuck. As a result only one activity is in the critical section at any time. This property is called *mutual exclusion*. Each activity, by virtue of passing the `begin_critical_section` code and being in the critical section, excludes all other activities from also being in the critical section. When it finishes the critical section, and executes the `end_critical_section` code, this frees another activity that was stuck in the `begin_critical_section`. Now that other activity can enter the critical section.

Exercise 55 *Does the atomicity of critical sections imply that a process that is in a critical section may not be preempted?*

Mutual exclusion can be achieved by sophisticated algorithms

But how do you implement `begin_critical_section` and `end_critical_section` to achieve the desired effect? In the 1960s the issue of how and whether concurrent processes can coordinate their activities was a very challenging question. The answer was that they can, using any of several subtle algorithms. While these algorithms are not used in real systems, they have become part of the core of operating system courses, and we therefore review them here. More practical solutions (that are indeed used) are introduced below.

The basic idea in all the algorithms is the same. They provide code that implements `begin_critical_section` and `end_critical_section` by using an auxiliary shared data structure. For example, we might use shared variables in which each process indicates that it is going into the critical section, and then checks that the others are not. With two processes, the code would be

process 1:	**process 2:**
`going_in_1 = TRUE;`	`going_in_2 = TRUE;`
`while (going_in_2) /*empty*/;`	`while (going_in_1) /*empty*/;`
`critical section`	*`critical section`*
`going_in_1 = FALSE;`	`going_in_2 = FALSE;`

where `going_in_1` and `going_in_2` are shared memory locations that can be accessed by either process. Regrettably, this code is not very good, as it may create a deadlock: if process 1 sets `going_in_1` to `TRUE`, is interrupted, and then process 2 sets `going_in_2` to `TRUE`, the two processes will wait for each other indefinitely.

Exercise 56 *Except for the problem with deadlock, is this code at least correct in the sense that if one process is in the critical section then it is guaranteed that the other process will not enter the critical section?*

A better algorithm is the following (called Peterson's algorithm [14]):

process 1:

```
going_in_1 = TRUE;
turn = 2;
while (going_in_2 && turn==2)
    /*empty*/;
critical section
going_in_1 = FALSE;
```

process 2:

```
going_in_2 = TRUE;
turn = 1;
while (going_in_1 && turn==1)
    /*empty*/;
critical section
going_in_2 = FALSE;
```

This is based on using another shared variable that indicates whose turn it is to get into the critical section. The interesting idea is that each process tries to let the *other* process get in first. This solves the deadlock problem. Assume both processes set their respective `going_in` variables to `TRUE`, as before. They then both set `turn` to conflicting values. One of these assignments prevails, and the process that made it then waits. The other process, the one whose assignment to `turn` was overwritten, can then enter the critical section. When it exits, it sets its `going_in` variable to `FALSE`, and then the waiting process can get in.

Exercise 57 *What happens if each process sets `turn` to itself?*

The bakery algorithm is rather straightforward

While Peterson's algorithm can be generalized to more than two processes, it is not very transparent. A much simpler solution is provided by Lamport's bakery algorithm [9], which is based on the idea that processes take numbered tickets, and the one with the lowest number gets in. However, there remains the problem of assigning the numbers. Because we have more processes, we need more shared variables. The algorithm uses two arrays: `i_am_choosing[`N`]`, initialized to `FALSE`, and `my_ticket[`N`]` initialized to `0`. The code for process i (out of a total of N) is

```
i_am_choosing[i] = TRUE;
for (j=0 ; j<N ; j++) {
    if (my_ticket[i] <= my_ticket[j])
        my_ticket[i] = my_ticket[j] + 1;
}
i_am_choosing[i] = FALSE;
for (j=0 ; j<N ; j++) {
    while (i_am_choosing[j]) /*empty*/;
    while ((my_ticket[j] > 0) &&
           ((my_ticket[j] < my_ticket[i]) ||
           ((my_ticket[j] == my_ticket[i]) && (j < i)))) /*empty*/;
}
critical section
my_ticket[i] = 0;
```

The first block (protected by `i_am_choosing[`i`]`) assigns the ticket number by looking at all current numbers and choosing a number that is larger by 1 than any number seen. Note that there are no loops here, so no process will be delayed, so tickets indeed represent the arrival order.

However, note that several processes may be doing this at the same time, so more than one process may end up with the same number. This is solved when we compare our number with all the other processes, in the second `for` loop. First, if we encounter a process that is in the middle of getting its ticket, we wait for it to actually get the ticket. Then, if the ticket is valid and smaller than ours, we wait for it to go through the critical section (when it gets out, it sets its ticket to 0, which represents invalid). Ties are simply solved by comparing the IDs of the two processes.

Exercise 58 *A simple optimization of this algorithm is to replace the loop of choosing the next ticket with a global variable, as in*

```
my_ticket[i] = current_ticket;
current_ticket = current_ticket + 1;
```

Note, however, that incrementing the global variable is typically not atomic: each process reads the value into a register, increments it, and writes it back. Can this cause problems? Hint: if it would have worked, we wouldn't use the loop.

There are four criteria for success

In summary, this algorithm has the following desirable properties:

1. *Correctness:* only one process is in the critical section at a time.
2. *Progress:* there is no deadlock, and if one or more processes are trying to get into the critical section, some process will eventually get in.
3. *Fairness:* there is no starvation, and no process will wait indefinitely while other processes continuously sneak into the critical section. There are also other stronger versions of fairness, culminating with the requirement that processes enter the critical section strictly in the order of arrival.
4. *Generality:* it works for N processes.

Exercise 59 *How strong is the fairness provided by the Bakery algorithm?*

Exercise 60 *Can you show that all these properties indeed hold for the bakery algorithm?*

Details: A Formal Proof

After showing that the algorithms are convincing using rapid hand waving, we now turn to a formal proof. The algorithm used is the n-process generalization of Peterson's algorithm, and the proof is due to Hofri [8].

The algorithm uses two global arrays, `q[`n`]` and `turn[`$n-1$`]`, both initialized to all 0's. Each process p_i also has three local variables, `j`, `k`, and its index `i`. The code is

```
1  for (j=1; j<n; j++) {
2      q[i] = j;
3      turn[j] = i;
4      while ((∃k≠i s.t.  q[k]≥j) && (turn[j] = i))
5          /*empty*/;
6  }
7  critical section
8  q[i] = 0;
```

The generalization from the two-process case is that entering the critical section becomes a multi-stage process. Thus the Boolean `going_in` variable is replaced by the integer variable `q[i]`, originally 0 to denote no interest in the critical section, passing through the values 1 to $n-1$ to reflect the stages of the entry procedure, and finally hitting n in the critical section itself. But how does this work to guarantee mutual exclusion? Insight may be gained by the following lemmas.

Lemma 3.1 *A process that is ahead of all others can advance by one stage.*

Proof: The formal definition of process i being ahead is that `∀k≠i, q[k] < q[i]`. Thus the condition in line 4 is not satisfied, and `j` is incremented and stored in `q[i]`, thus advancing process i to the next stage. ∎

Lemma 3.2 *When a process advances from stage j to stage $j+1$, it is either ahead of all other processes, or there are other processes at stage j.*

Proof: The first alternative is a rephrase of the previous lemma: if a process that is ahead of all others can advance, than an advancing process may be ahead of all others. The other alternative is that the process is advancing because `turn[j]`$\neq$`i`. This can only happen if at least one other process reached stage j after process i, and modified `turn[j]`. Of these processes, the last one to modify `turn[j]` must still be at stage j, because the condition in line 4 evaluates to true for that process. ∎

Lemma 3.3 *If there are at least two processes at stage j, there is at least one process at each stage $k \in \{1, \ldots, j-1\}$.*

Proof: The base step is for $j = 2$. Given a single process at stage 2, another process can join it only by leaving behind a third process in stage 1 (by Lemma 3.2). This third process stays stuck there as long as it is alone, again by Lemma 3.2. For the induction step, assume the Lemma holds for stage $j - 1$. Given that there are two processes at stage j, consider the instance at which the second one arrived at this stage. By Lemma 3.2, at that time there was at least one other process at stage $j - 1$; and by the induction assumption, this means that all the lower stages were also occupied. Moreover, none of these stages could be vacated since, due to Lemma 3.2. ∎

Lemma 3.4 *The maximal number of processes at stage j is $n - j + 1$.*

Proof: By Lemma 3.3, if there are more than one process at stage j, all the previous $j - 1$ stages are occupied. Therefore at most $n - (j - 1)$ processes are left for stage j. ∎

Using these, we can envision how the algorithm works. The stages of the entry protocol are like the rungs of a ladder that has to be scaled in order to enter the critical section. The algorithm works by allowing only the top process to continue scaling the ladder. Others are restricted by the requirement of having a continuous link of processes starting at the bottom rung. As the number of rungs equals the number of processors minus 1, they are prevented from entering the critical section. Formally, we can state the following:

Theorem 3.1 *Peterson's generalized algorithm satisfies the conditions of correctness, progress, and fairness (assuming certain liveness properties).*

Proof: According to Lemma 3.4, stage $n - 1$ can contain at most two processes. Consider first the case where there is only one process at this stage. If there is another process currently in the critical section itself (stage n), the process at stage $n - 1$ must stay there according to Lemma 3.2. Thus the integrity of the critical section is maintained. If there are two processes in stage $n - 1$, then according to Lemma 3.3 all previous stages are occupied, and the critical section is vacant. One of the two processes at stage $n - 1$ can therefore enter the critical section. The other will then stay at stage $n - 1$ because the condition in line 4 of the algorithm holds. Thus the integrity of the critical section is maintained again, and correction is proved.

Progress follows immediately from the fact that some process must be either ahead of all others, or at the same stage as other processes and not the last to have arrived. For this process the condition in line 4 does not hold, and it advances to the next stage.

Fairness follows from the fact that the last process to reach a stage is the one that cannot proceed, because the stage's `turn` cell is set to its ID. Consider process p, which is the last to arrive at the first stage. In the worst case, all other processes can be ahead of it, and they will enter the critical section first. But if they subsequently try to enter it again, they will be behind process p. If it tries to advance, it will therefore manage to do so, and one of the other processes will be left behind. Assuming all non-last processes are allowed to advance to the next stage, process p will have to wait for no more than $n - j$ other processes at stage j, and other processes will overtake it no more than n times each. ∎

To read more: A full description of lots of wrong mutual exclusion algorithms is given by Stallings [16, Sect. 5.2]. The issue of sophisticated algorithms for mutual exclusion has been beaten to death by Lamport [10, 11, 12].

An alternative is to augment the instruction set with atomic instructions

As noted, the problem with concurrent programming is the lack of atomicity. The above algorithms provide the required atomicity, but at the cost of considerable headaches. Alternatively, the hardware may provide a limited measure of atomicity, that can then be amplified.

The simplest example is the test_and_set instruction. This instruction operates on a single bit, and does two things atomically: it reads the bit's value (0 or 1), and then it sets the bit to 1. This can be used to create critical sections as follows, using a single bit called `guard` initialized to 0 (= FALSE):

```
while ( test_and_set(guard) ) /*empty*/;
critical section
guard = 0;
```

Because the hardware guarantees that the test_and_set is atomic, it guarantees that only one process will see the value 0. When that process sees 0, it atomically sets the value to 1, and all other processes will see 1. They will then stay in the while loop until the first process exits the critical section, and sets the bit to 0. Again, only one other process will see the 0 and get into the critical section; the rest will continue to wait.

Exercise 61 *The compare_and_swap instruction is defined as follows:*

```
compare_and_swap( x, old, new )
   if (*x == old)
      *x = new;
      return SUCCESS;
   else
      return FAIL
```

where `x` is a pointer to a variable, `old` and `new` are values, and the whole thing is done atomically by the hardware. How can you implement a critical section using this instruction?

3.1.3 Semaphores and Monitors

Programming is simplified by the abstraction of semaphores

Algorithms for mutual exclusion are tricky, and it is difficult to verify their exact properties. Using hardware primitives depends on their availability in the architecture.

A better solution from the perspective of operating system design is to use some more abstract mechanism.

The mechanism that captures the abstraction of inter-process synchronization is the *semaphore*, introduced by Dijkstra [1]. Semaphores are a new data type, that provides only two operations:

- the `P` operation checks whether the semaphore is free. If it is, it occupies the semaphore. But if the semaphore is already occupied, it waits till the semaphore will become free.
- The `V` operation releases an occupied semaphore, and frees one blocked process (if there are any blocked processes waiting for this semaphore).

The above specification describes the semantics of a semaphore. Using semaphores, it is completely trivial to protect a critical section. If we call the semaphore `mutex` (a common name for mutual exclusion mechanisms), the code is

```
P(mutex);
critical section
V(mutex);
```

Thus semaphores precisely capture the desired semantics of `begin_critical_section` and `end_critical_section`. In fact, they more generally capture an important *abstraction* very succinctly — they are a means by which the process can tell the system that it is waiting for something.

The most common way to implement this specification is by using an integer variable initialized to 1. The value of 1 indicates that the semaphore is free. A value of 0 or less indicates that it is occupied. The `P` operation decrements the value by one, and blocks the process if it becomes negative. The `V` operation increments the value by one, and frees a waiting process. This implementation is described by the following pseudo-code:

```
class semaphore
   int value = 1;
   P() {
      if (--value < 0)
         block_this_proc();
   }
   V() {
      if (value++ < 0)
         resume_blocked_proc();
   }
```

The reasons for calling the operations `P` and `V` is that they are abbreviations of the Dutch words “proberen” (to try) and “verhogen” (to elevate). Speakers of Hebrew have the advantage of regarding them as abbreviations for פחות and עלה. In English, the words *wait* and *signal* are sometimes used instead of `P` and `V`, respectively.

Exercise 62 *"Semáforo" means "traffic light" in Spanish. What is similar and what is different between semaphores and traffic lights? How about the analogy between semaphores and locks?*

Semaphores can be implemented efficiently by the operating system

The power of semaphores comes from the way in which they capture the essence of synchronization, which is this: the process needs to wait until a certain condition allows it to proceed. The important thing it that the abstraction does not specify *how* to implement the waiting.

All the solutions we saw so far implemented the waiting by burning cycles. Any process that had to wait simply sat in a loop, and continuously checked whether the awaited condition was satisfied — which is called *busy waiting*. In a uniprocessor system with multiple processes this is very very inefficient. When one process is busy waiting for another, it is occupying the CPU all the time. Therefore the awaited process cannot run, and cannot make progress towards satisfying the desired condition.

The P operation on a semaphore, on the other hand, conveys the information that the process cannot proceed if the semaphore is negative. If this is the case, the operating system can then preempt this process and run other processes in its place. Moreover, the process can be blocked and placed in a queue of processes that are waiting for this condition. Whenever a V operation is performed on the semaphore, one process is removed from this queue and placed on the ready queue.

Exercise 63 *Some implementations don't just wake up one waiting process — they wake up all waiting processes! Can you think of a reason for such wasteful behavior? (And why do we say it is wasteful?)*

And they have additional uses beyond mutual exclusion

Semaphores have been very successful, and since their introduction in the context of operating system design they have been recognized as a generally useful construct for concurrent programming. This is due to the combination of two things: that they capture the abstraction of needing to wait for a condition or event, and that they can be implemented efficiently as shown above.

An important advantage of the semaphore abstraction is that now it is easy to handle multiple critical sections: we can simply declare a separate semaphore for each one. More importantly, we can use the same semaphore to link several critical sections of code that manipulate the same shared data structures. In fact, a semaphore can be considered as a mechanism for *locking* data structures as described below.

Exercise 64 *Consider a situation in which we have several linked lists, and occasionally need to move an item from one list to another. Is the definition of a unique semaphore for each pair of lists a good solution?*

Moreover, the notion of having to wait for a condition is not unique to critical sections (where the condition is "no other process is executing this code"). Thus semaphores can be used for a host of other things.

One simple example comes from resource allocation. Obviously a semaphore can represent the allocation of a resource. But it also has the interesting property that we can initialize the semaphore to a number different from 1, say 3, thus allowing up to 3 processes "into" the semaphore at any time. Such *counting semaphores* are useful for allocating resources of which there are several equivalent instances.

Exercise 65 *A common practical problem is the producer/consumer problem, also known as the bounded buffer problem. This assumes two processes, one of which produces some data, while the other consumes this data. The data is kept in a finite set of buffers in between. The problem is to keep the producer from overflowing the buffer space, and to prevent the consumer from using uninitialized data. Give a solution using semaphores.*

Exercise 66 *Is your solution to the previous exercise good for only one producer and one consumer, or also for arbitrary numbers of producers and consumers? Give a solution for the general case.*

Monitors provide an even higher level of abstraction

Another abstraction that was introduced for operating system design is that of monitors [7]. A monitor encapsulates some state, and provides methods that operate on that state. In addition, it guarantees that these methods are executed in a mutually exclusive manner. Thus if a process tries to invoke a method from a specific monitor, and some method of that monitor is already being executed by another process, the first process is blocked until the monitor becomes available.

A special case occurs when a process cannot complete the execution of a method and has to wait for some event to occur. It is then possible for the process to enqueue itself *within* the monitor, and allow other processes to use it. Later, when the time is right, some other process using the monitor will resume the enqueued process.

To read more: Additional issues in concurrent programming are covered in Sections 5.4 through 5.7 of Stallings [17]. Similar material is covered in Sections 6.4 through 6.9 of Silberschatz and Galvin [15].

3.1.4 Locks and Disabling Interrupts

So where are we?

To summarize, what we have seen so far is the following.

First, the mutual exclusion problem can be solved using a bare bones approach. It is possible to devise algorithms that only read and write shared variables to determine whether they can enter the critical section, and use busy waiting to delay themselves if they cannot.

Second, much simpler algorithms are possible if hardware support is available in the form of certain atomic instructions, such as test_and_set. However, this still uses busy waiting.

Third, the abstraction of semaphores can be used to do away with busy waiting. The P operation on a semaphore can be used to tell the system that a process needs to wait for a condition to hold, and the system can then block the process.

However, it is actually not clear how to implement semaphores within the operating system. The pseudo-code given on page 74 includes several related operations, e.g. checking the value of the semaphore and updating it. But what happens if there is an interrupt between the two? We need to do this in a critical section... Reverting to using busy waiting to implement this "internal" critical section may void the advantage of blocking in the semaphore!

The simple solution is to disable interrupts

In order to avoid busy waiting, we need a mechanism that allows selected operating system code to run without interruption. As asynchronous interruptions are caused by external interrupts, this goal can be achieved by simply disabling all interrupts. Luckily, such an option is available on all hardware platforms. Technically, this is done by setting the interrupt level in the PSW. The processor then ignores all interrupts lower than the level that was chosen.

Exercise 67 *Should the disabling and enabling of interrupts be privileged instructions?*

Example: classical Unix used a non-preemptive kernel

A rather extreme solution to the issue of mutual exclusion is to consider the whole kernel as a critical section. This was done in early versions of the Unix system, and was one of the reasons for that system's simplicity. This was accomplished by disabling all interrupts when in the kernel. Thus kernel functions could complete without any interruption, and leave all data structures in a consistent state. When they returned, they enabled the interrupts again.

Exercise 68 *The Unix process state graph on page 59 shows an arrow labeled "preempt" leaving the "kernel running" state. How does this fit in with the above?*

The problem with a non-preemptive kernel is that it compromises the responsiveness of the system, and prevents it from reacting in real time to various external conditions. Therefore modern versions of Unix do allow preemption of kernel functions, and resort to other means for synchronization — mainly locks.

Exercise 69 *Does the blocking of interrupts provide the required atomicity on multiprocessors?*

Locks can be used to express the desired granularity

Disabling interrupts can be viewed as locking the CPU: the current process has it, and no other process can gain access. Thus the practice of blocking all interrupts whenever a process runs in kernel mode is akin to defining a single lock, that protects the whole kernel. But this may be stronger than what is needed: for example, if one process wants to open a file, this should not interfere with another that wants to allocate new memory.

The solution is therefore to define multiple locks, that each protect a part of the kernel. These can be defined at various granularities: a lock for the whole file system (and by implication, for all the data structures involved in the file system implementation), a lock for a single large data structure such as a system table, or a lock for a a single entry within such a table. By holding only the necessary locks, the restrictions on other processes are reduced.

Locks apply to data structures

Note the shift in focus from the earlier parts of this chapter: we are no longer talking about critical sections of *code* that should be executed in a mutually exclusive manner. Instead, we are talking of *data structures* which need to be used in a mutually exclusive manner. This shifts the focus from the artifact (the code handling the data structure) to the substance (the data structure's function and why it is being accessed).

This shift in focus is extremely important. To drive the point home, consider the following example. As we know, the operating system is a reactive program, with many different functions that may be activated under diverse conditions. In particular, we may have

- The scheduler, which is called by the clock interrupt handler. When called, it scans the list of ready processes to find the one that has the highest priority and schedule it to run.
- The disk interrupt handler, which wakes up the process that initiated the I/O activity and places it on the ready queue.
- The function that implements the creation of new processes, which allocates a PCB for the new process and links it to the ready queue so that it will run when the scheduler selects it.

All these examples and more are code segments that manipulate the ready queue and maybe some other data structures as well. To ensure that these data structures

remain consistent, they need to be locked. In particular, all these functions need to lock the ready queue. Mutually exclusive execution of the code segments won't work, because they are distinct code fragments to begin with.

And locks can have special semantics

Once locks are in place, it is possible to further reduce the restrictions on system activities by endowing them with special semantics. For example, if two processes only need to read a certain data structure, they can do so concurrently with no problem. The data structure needs to be locked only if one of them modifies it, leading to possible inconsistent intermediate states that should not be seen by other processes.

This observation has led to the design of so called *readers-writers locks*. Such locks can be locked in two modes: locked for reading and locked for writing. Locks for reading do not conflict with each other, and many processes can obtain a read-lock at the same time. But locks for writing conflict with all other accesses, and only one can be held at a time.

Exercise 70 *Write pseudo-code to implement the three functions of a readers-writers lock: lock_read, lock_write, and release_lock. Hint: think about fairness and starvation.*

But locks may also lead to problems

Locks provide a straightforward an intuitive mechanism for protecting data structures during concurrent access. However, using them may lead to two types of problems. Both relate to the fact that one process may need to wait for another.

The first problem is called priority inversion. This happens when a low priority process holds a lock, and a higher priority process tries to acquire the same lock. Given the semantics of locks, this leads to a situation where a high-priority process waits for a low-priority one, and worse, also for processes with intermediate priorities that may preempt the low-priority process and thus prevent it from releasing the lock. One possible solution is to temporarily elevate the priority of the process holding the lock to that of the highest process that also want to obtain it.

The second problem is one of deadlocks, where a set of processes all wait for each other and none can make progress. This will be discussed at length in Section 3.2.

3.1.5 Multiprocessor Synchronization

But what about multiprocessor systems? The operating system code running on each processor can only block interrupts from occurring *on that processor*. Related code running on another processor will not be affected, and might access the same data structures.

The only means for synchronizing multiple processors is atomic instructions that are implemented by the hardware in some shared manner — for example, the atomicity may be achieved by locking the shared bus by which all the processors access memory.

The problem with using these atomic instructions is that we are back to using busy waiting. This is not so harmful for performance because it is only used to protect very small critical sections, that are held for a very short time. A common scenario is to use busy waiting with test_and_set on a variable that protects the implementation of a semaphore. The protected code is very short: just check the semaphore variable, and either lock it or give up because it has already been locked by someone else. Blocking to wait for the semaphore to become free is a local operation, and need not be protected by the test_and_set variable.

3.2 Resource Contention and Deadlock

One of the main functions of an operating system is resource allocation. But when multiple processes request and acquire multiple resources, deadlock may ensue.

3.2.1 Deadlock and Livelock

Contention may lead to deadlock

We already saw examples of deadlock in the discussion of critical sections, where two processes busy wait forever in their respective loops, each waiting for the other to proceed. But this can also happen with mechanisms like locks.

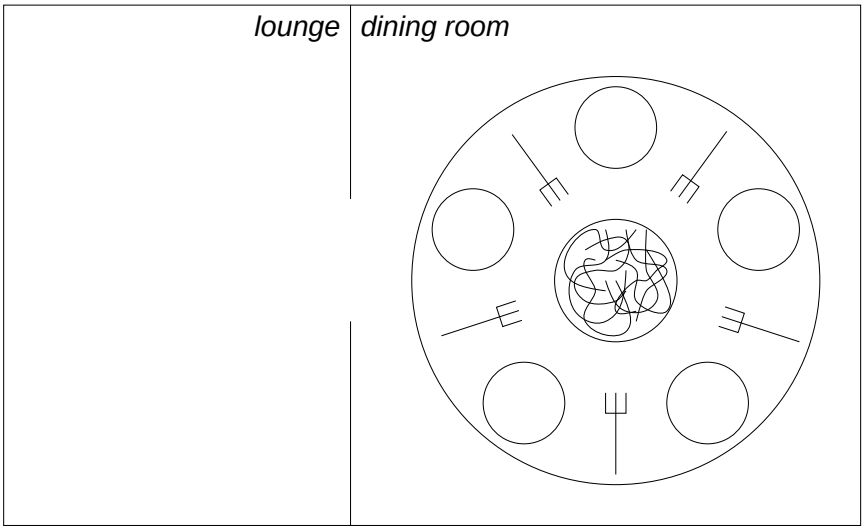
For example, consider a function that moves an item from one list to another. To do so, it must lock both lists. Now assume that one process wants to move an item from list A to list B, and at the same time another process tries to move an item from list B to list A. Both processes will lock the source list first, and then attempt to lock the destination list. The problem is that the first activity may be interrupted after it locks list A. The second activity then runs and locks list B. It cannot also lock list A, because list A is already locked, so it has to wait. But the first activity, which is holding the lock on list A, is also stuck, because it cannot obtain the required lock on list B. Thus the two activities are deadlocked waiting for each other.

The dining philosophers problem provides an abstract example

One of the classical problems in operating system lore is the following. Assume five philosophers live together, spending their time thinking. Once in a while they become hungry, and go to the dining room, where a table for five is laid out. At the center of the table is a large bowl of spaghetti. Each philosopher sits in his place, and needs

two forks in order to shovel some spaghetti from the central bowl onto his personal plate. However, there is only one fork between every two philosophers.

The problem is that the following scenario is possible. All the philosophers become hungry at the same time, and troop into the dining room. They all sit down in their places, and pick up the forks to their left. Then they all try to pick up the forks to their right, only to find that those forks have already been picked up (because they are also another philosopher's left fork). The philosophers then continue to sit there indefinitely, each holding onto one fork, and glaring at his neighbor. They are deadlocked.



For the record, in operating system terms this problem represents a set of five processes (the philosophers) that contend for the use of five resources (the forks). It is highly structured in the sense that each resource is potentially shared by only two specific processes, and together they form a cycle. The spaghetti doesn't represent anything.

Exercise 71 *Before reading on, can you think of how to solve this?*

A tempting solution is to have each philosopher relinquish his left fork if he cannot obtain the right fork, and try again later. However, this runs the risk that all the philosophers will put down their forks in unison, and then try again at the same time, resulting in an endless sequence of picking up forks and putting them down again. As they are active all the time, despite not making any progress, this is called *livelock* rather than deadlock.

Luckily, several solutions that do indeed work have been proposed. One is to break the cycle by programming one of the philosophers differently. For example, while all the philosophers pick up their left fork first, one might pick up his right fork first. Another is to add a footman, who stands at the dining room door and only allows four philosophers in at a time. A third solution is to introduce randomization: each

philosopher that finds his right fork unavailable will relinquish his left fork, and try again only after a random interval of time.

Exercise 72 *Why do these solutions work?*

Exercise 73 *Write a program that expresses the essentials of this problem and one of its solutions using semaphores.*

However, these solutions are based on the specific structure of this problem. What we would like is a more general strategy for solving such problems.

3.2.2 A Formal Setting

The system state is maintained in the resource allocation graph

It is convenient to represent the system state — with respect to resource management and potential deadlocks — by the *resource allocation graph*. This is a directed graph with two types of nodes and two types of edges.

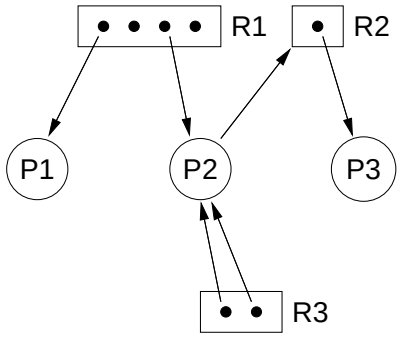
Processes are represented by round nodes.

Resource types are represented by square nodes. Within them, each instance of the resource type is represented by a dot.

Requests are represented by edges from a process to a resource type.

Allocations are represented by edges from a resource instance to a process.

For example, in this graph process P2 has one instance of resource R1 and two of R3, and wants to acquire one of type R2. It can't get it, because the only instance of R2 is currently held by process P3.



Exercise 74 *What are examples of resources (in a computer system) of which there is a single instance? A few instances? Very many instances?*

Exercise 75 *A computer system has three printers attached to it. Should they be modeled as three instances of a generic "printer" resource type, or as separate resource types?*

Cycles in the graph may imply deadlock

Assuming requests represent resources that are really needed by the process, an unsatisfied request implies that the process is stuck. It is waiting for the requested resource to become available. If the resource is held by another process that is also stuck, then that process will not release the resource. As a result the original process will stay stuck. When a group of processes are stuck waiting for each other in this way, they will stay in this situation forever. We say they are *deadlocked*.

More formally, four conditions are necessary for deadlock:

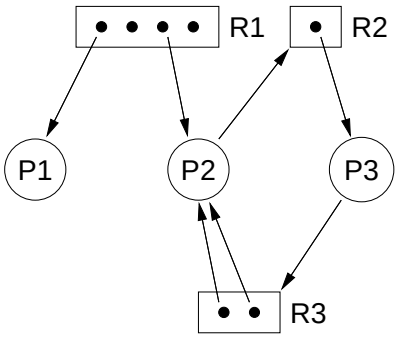
1. Resources are allocated exclusively to one process at a time, and are not shared.
2. The system does not preempt resources that are held by a processes. Rather, processes are expected to release resources when they are finished with them.
3. Processes may simultaneously hold one resource and wait for another.
4. There is a cycle in the resource allocation graph, with each process waiting for a resource held by another.

The first two are part of the semantics of what resource allocation means. For example, consider a situation in which you have been hired to run errands, and given a bicycle to do the job. It would be unacceptable if the same bicycle was given to someone else at the same time, or if it were taken away from under you in mid trip.

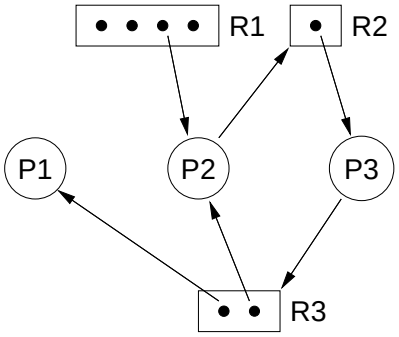
The third condition is a rule of how the system operates. As such it is subject to modifications, and indeed some of the solutions to the deadlock problem are based on such modifications.

The fourth condition may or may not happen, depending on the timing of various requests. If it does happen, then it is unresolvable because of the first three.

For example, ths graph represents a deadlocked system: Process P3 is waiting for resource R3, but both instances are held by P2 who is waiting for R2, which is held by P3, thus forming a cycle.



Note, however, that having a cycle in the graph is not a sufficient condition for deadlock, unless there is only one instance of each resource. If there are multiple instances of some resources, the cycle may be resolved by some other process releasing its resources. For example, in this graph process P1 may release the instance of R3 it is holding, thereby allowing the request of P3 to be satisfied; when P3 subsequently completes its work it will release the instance of R2 it is holding, which will be given to P2.



Exercise 76 *Under what conditions is having a cycle a sufficient condition? Think of conditions relating to both processes and resources.*

3.2.3 Deadlock Prevention

There are several ways to handle the deadlock problem. Prevention involves designs in which deadlock simply cannot happen.

Deadlock is prevented if one of the four conditions does not hold

Deadlock is bad because the involved processes never complete, leading to frustrated users and degraded service from the system. One solution is to design the system in a way that *prevents* deadlock from ever happening. We have identified four conditions that are necessary for deadlock to happen. Therefore, designing the system so that it violates any of these conditions will prevent deadlock.

One option is to annul the "hold and wait" condition. This can be done in several ways. Instead of acquiring resources one by one, processes may be required to give the system a list of all the resources they will need at the outset. The system can then either provide *all* the resources immediately, or block the process until a later time when all the requested resources will be available. However, this means that processes will hold on to their resources for more time than they actually need them, which is wasteful.

An alternative is to require processes to release all the resources they are currently holding before making new requests. However, this implies that there is a risk that the resources will be allocated to another process in the meanwhile; for example, if the resources are a system data structure, it must be brought to a consistent state before being released.

In some cases we can nullify the second condition, and allow resources to be preempted. For example, in Section 4.4 we will introduce swapping as a means to deal with overcommitment of memory. The idea is that one process is selected as a victim

and swapped out. This means that its memory contents are backed up on disk, freeing the physical memory it had used for use by other processes.

Exercise 77 *Can locks be preempted? Can processes holding locks be swapped out?*

Prevention can be achieved by acquiring resources in a predefined order

A more flexible solution is to prevent cycles in the allocation graph by requiring processes to acquire resources in a predefined order. All resources are numbered in one sequence, and these numbers dictate the order. A process holding some resources can then only request additional resources that have strictly higher numbers. A process holding a high-number resource cannot request a low-numbered one. For example, in the figure shown above, process P2 holds two instances of resource R3. It will therefore not be allowed to request an instance of R2, and the cycle is broken. If P2 needs both R2 and R3, it should request R2 first, before it acquires R3.

Exercise 78 *With n resources there are $n!$ possible orders. Which should be chosen as the required order?*

Exercise 79 *Why do we require "strictly higher" numbers?*

Exercise 80 *Consider an intersection of two roads, with cars that may come from all 4 directions. Is there a simple uniform rule (such as "always give right-of-way to someone that comes from your right") that will prevent deadlock? Does this change if a traffic circle is built?*

3.2.4 Deadlock Avoidance

Avoidance is for systems where deadlock may in fact happen. But the system manages to stay away from deadlock situations by being careful.

Deadlock is avoided by allocations that result in a safe state

Another approach is deadlock avoidance. In this approach the system may in principle enter a deadlock state, but the operating system makes allocation decisions so as to avoid it. If a process makes a request that the operating system deems dangerous, that process is blocked until a better time. Using the cars at the intersection analogy, the operating system will not allow all the cars to enter the intersection at the same time.

An example of this approach is the banker's algorithm (also due to Dijkstra) [1]. This algorithm is based on the notion of *safe states*. A safe state is one in which all the processes in the system can be executed in a certain order, one after the other, such that each will obtain all the resources it needs to complete its execution. The assumption is that then the process will release all the resources it held, which will

then become available for the next process in line. Such an ordering provides the operating system with a way to run the processes without getting into a deadlock situation; by ensuring that such an option always exists, the operating system avoids getting stuck.

Safe states are identified by the banker's algorithm

In order to identify safe states, each process must declare in advance what its *maximal* resource requirements may be. Thus the algorithm works with the following data structures:

- The maximal requirements of each process $\vec{M}_p$,
- The current allocation to each process $\vec{C}_p$, and
- The currently available resources $\vec{A}$.

These are all vectors representing all the resource types. For example, if there are three resource types, and three units are available from the first, none from the second, and one from the third, then $\vec{A} = (3, 0, 1)$.

Assume the system is in a safe state, and process p makes a request $\vec{R}$ for more resources (this is also a vector). We tentatively update the system state *as if* we performed the requested allocation, by doing

$$\vec{C}_p = \vec{C}_p + \vec{R}$$
$$\vec{A} = \vec{A} - \vec{R}$$

Where operations on vectors are done on respective elements (e.g. $\vec{X} + \vec{Y}$ is the vector $(x_1 + y_1, x_2 + y_2, ...x_k + y_k)$). We now check whether this new state is also safe. If it is, we will really perform the allocation; if it is not, we will block the requesting process and make it wait until the allocation can be made safely.

To check whether the new state is safe, we need to find an ordering of the processes such that the system has enough resources to satisfy the maximal requirements of each one in its turn. As noted above, it is assumed that the process will then complete and release all its resources. The system will then have a larger pool to satisfy the maximal requirements of the next process, and so on. This idea is embodied by the following pseudo-code, where P is initialized to the set of all processes:

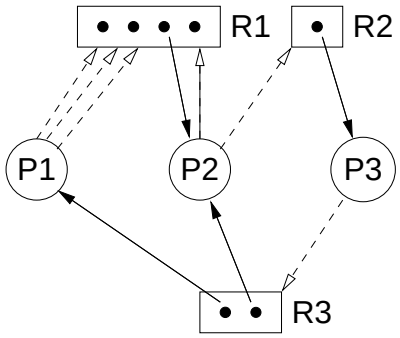
```
while (P ≠ ∅) {
    found = FALSE;
    foreach p ∈ P {
        if (M_p − C_p ≤ A) {
            /* p can obtain all it needs, terminate, */
            /* and releases what it already has.     */
            A = A + C_p;
```

```
                P = P − {p};
                found = TRUE;
            }
        }
        if (!  found) return FAIL;
    }
    return OK;
```

where comparison among vectors is also elementwise. Note that the complexity of the algorithm is $O(n^2)$ (where $n = |P|$), even though the number of possible orders is $n!$. This is because the resources available to the system increase monotonically as processes terminate, so if it is possible to execute any of a set of processes, the order that is chosen is not important. There is never any need to backtrack and try another order.

Exercise 81 *Show that the complexity is indeed* $O(n^2)$.

For example, consider the following resource allocation graph, where dashed edges represent potential future requests as allowed by the declared maximal requirements:



This state is described by the following vectors:

$$\begin{array}{llll} \vec{C}_1 = (0,0,1) & \vec{C}_2 = (1,0,1) & \vec{C}_3 = (0,1,0) & \vec{A} = (3,0,0) \\ \vec{M}_1 = (3,0,1) & \vec{M}_2 = (2,1,1) & \vec{M}_3 = (0,1,1) & \end{array}$$

Assume process P1 now actually requests the allocation of an instance of resource R1 (that is, $R = (1,0,0)$). This request can be granted, because it leads to a safe state. The sequence of allocations showing this is as follows. First, granting the request leads to the state

$$\begin{array}{llll} \vec{C}_1 = (1,0,1) & \vec{C}_2 = (1,0,1) & \vec{C}_3 = (0,1,0) & \vec{A} = (2,0,0) \\ \vec{M}_1 = (3,0,1) & \vec{M}_2 = (2,1,1) & \vec{M}_3 = (0,1,1) & \end{array}$$

In this state there are enough available instances of R1 so that all of P1's potential additional requests can be satisfied. So P1 can in principle run to completion and will then terminate and release all its resources, including its instance of R3, leading to the state

$$\begin{array}{lll} \vec{C}_2 = (1,0,1) & \vec{C}_3 = (0,1,0) & \vec{A} = (3,0,1) \\ \vec{M}_2 = (2,1,1) & \vec{M}_3 = (0,1,1) & \end{array}$$

This is not good enough to fill all the requests of P2, but we can fulfill all of P3's potential requests (there is only one: a request for an instance of R3). So P3 will be able to run to completion, and will then release its instance of R2. So in the second iteration of the external `while` loop we will find that P2 too can acquire all the resources it needs and terminate.

As another example, consider the initial situation again, but this time consider what will happen if P2 requests an instance of resource R1. This request cannot be granted, as it will lead to an unsafe state, and therefore might lead to deadlock. More formally, granting such a request by P2 will lead to the state

$$\begin{array}{llll} \vec{C}_1 = (0,0,1) & \vec{C}_2 = (2,0,1) & \vec{C}_3 = (0,1,0) & \vec{A} = (2,0,0) \\ \vec{M}_1 = (3,0,1) & \vec{M}_2 = (2,1,1) & \vec{M}_3 = (0,1,1) & \end{array}$$

In this state the system does not have sufficient resources to grant the maximal requirements of any of the processes:

P1 may request 3 instances of R1, but there are only 2;
P2 may request an instance of R2, but there are none; and
P3 may request an instance of R3, but again there are none.

Therefore the first iteration of the algorithm will fail to find a process that can in principle obtain all its required resources first. Thus the request of process P2 will *not* be granted, and the process will be blocked. This will allow P1 and P3 to make progress, and eventually, when either of them terminate, it will be safe to make the allocation to P2.

Exercise 82 *Show that in this last example it is indeed safe to grant the request of P2 when either P1 or P3 terminate.*

Exercise 83 *What can you say about the following modifications to the above graph:*

1. *P1 may request another instance of R3 (that is, add a dashed arrow from P1 to R3)*
2. *P3 may request an instance of R1 (that is, add a dashed arrow from P3 to R1)*

Exercise 84 *Consider a situation in which the resources we are interested in are locks on shared data structures, and several processes execute functions that all handle the same set of data structures. What is the result of using the banker's algorithm in this scenario?*

Exercise 85 *Avoidance techniques check each resource request with something like the Banker's algorithm. Do prevention techniques need to check anything?*

3.2.5 Deadlock Detection

Detection allows deadlock and handles it after the fact.

Deadlock detection and recovery is the last resort

The banker's algorithm is relatively flexible in allocating resources, but requires rather complicated checks before each allocation decision. A more extreme approach is to allow all allocations, without any checks. This has less overhead, but may lead to deadlock. If and when this happens, the system detects the deadlock (using any algorithm for finding cycles in a graph) and recovers by killing one of the processes in the cycle.

Exercise 86 *What happens if you kill a process holding a lock?*

3.2.6 Real Life

While deadlock situations and how to handle them are interesting, most operating systems actually don't really employ any sophisticated mechanisms. This is what Tanenbaum picturesquely calls the ostrich algorithm: pretend the problem doesn't exist [18]. However, in reality the situation may actually not be quite so bad. Judicious application of simple mechanisms can typically prevent the danger of deadlocks.

Note that a process may hold two different types of resources:

Computational resources: these are the resources needed in order to run the program. They include the resources we usually talk about, e.g. the CPU, memory, and disk space. They also include kernel data structures such as page tables and entries in the open files table.

Synchronization resources: these are resources that are not needed in themselves, but are required in order to coordinate the conflicting actions of different processes. The prime example is locks on kernel data structures, which must be held for the duration of modifying them.

Deadlocks due to the first type of resources are typically prevented by breaking the "hold and wait" condition. Specifically, processes that attempt to acquire a resource

and cannot do so simply do not wait. Instead, the request fails. It is then up to the process to try and do something intelligent about it.

For example, in Unix an important resource a process may hold onto are entries in system tables, such as the open files table. If too many files are open already, and the table is full, requests to open additional files will fail. The program can then either abort, or try again in a loop. If the program continues to request that the file be opened in a loop, it might stay stuck forever, because other programs might be doing the same thing. However it is more likely that some other program will close its files and terminate, releasing its entries in the open files table, and making them available for others. Being stuck indefinitely can also be avoided by only trying a finite number of times, and then giving up.

Exercise 87 *If several programs request something in a loop until they succeed, and get stuck, is this deadlock? Can the operating system detect this? Should it do something about it? Might it take care of itself? Hint: think quotas.*

Deadlock due to locks can be prevented by acquiring the locks in a specific order. This need not imply that all the locks in the system are ordered in one grand scheme. Consider a kernel function that handles several data structures, and needs to lock them all. If this is the only place where these data structures are locked, then multiple instances of running this function will always lock them in the same order. This effectively prevents deadlock without requiring any cognizant global ordering of all the locks in the system. If a small number of functions lock these data structures, it is also easy to verify that the locking is done in the same order in each case. Finally, system designers that are aware of a potential deadlock problem can opt to use a non-blocking lock, which returns a fail status rather than blocking the process if the lock cannot be acquired.

Exercise 88 *The Linux kernel defines a function called* `double_rq_lock` *to lock two runqueues safely. Part of the code is the following:*

```
double_rq_lock(struct rq *rq1, struct rq *rq2)
    ...
    if (rq1 < rq2) {
        spin_lock(&rq1->lock);
        spin_lock(&rq2->lock);
    } else {
        spin_lock(&rq2->lock);
        spin_lock(&rq1->lock);
    }
```

What is this for?

3.3 Lock-Free Programming

As demonstrated in the beginning of this chapter, the existence of multiple processes or threads that access shared data can cause problems — in particular, the corruption of data structures such as linked lists. The common solution to such problems is to use locking, and thereby provide each process with mutually exclusive access in its turn. But causing processes to wait for each other reduces the concurrency in the system and may reduce performance, especially when extra care is taken to also avoid deadlock. An alternative is to use lock-free programming.

Lock-free programming is based on atomic hardware operations that can be leveraged to obtain the desired synchronization effects. The most commonly used operation is the *compare_and_swap* (CAS), which is defined by the following code:

```
compare_and_swap( x, old, new )
   if (*x == old)
      *x = new;
      return SUCCESS;
   else
      return FAIL;
```

where `x` is a pointer to a variable, and `old` and `new` are values; what this does is to verify that `*x` (the variable pointed to by `x`) has the expected old value, and if it does, to swap this with a new value. Of course, the whole thing is done atomically by the hardware, so the value cannot change between the check and the swap.

To see why this is useful, consider the same example from the beginning of the chapter: adding an element to the middle of a linked list. The regular code to add an element `new` after the element `current` is

```
new->next = current->next;
current->next = new;
```

But as we saw, the list could become corrupted if some other process changes it between these two instructions. The alternative, using compare_and_swap, is as follows:

```
new->next = current->next;
compare_and_swap(&current->next, new->next, new);
```

This first creates a link from the new item to the item after where it is supposed to be inserted. But then the insertion itself is done atomically by the compare_and_swap; and moreover, this is conditioned on the fact that the previous pointer, `current->next`, did not change in the interim.

Of course, using compare_and_swap does not guarantee that another process will not barge in and alter the list. So the compare_and_swap may fail. It is therefore imperative to check the return value from the compare_and_swap, in order to know whether the desired operation had been performed or not. In particular, the whole

thing can be placed in a loop that retries the compare_and_swap operation until it succeeds:

```
do {
   current = ⟨element after which to insert new⟩;
   new->next = current->next;
} until (compare_and_swap(&current->next, new->next, new));
```

Exercise 89 *Why did we also put the assignment to current and to new.next in the loop?*

The code shown here is lock-free: it achieves the correct results without using locks and without causing one process to explicitly wait for another. However, a process may have to retry its operation again and again an unbounded number of times. It is also possible to devise algorithms and data structures that are *wait-free*, which means that all participating processes will complete their operations successfully within a bounded number of steps.

Exercise 90 *Given that a process may need to retry the compare_and_swap many times, is there any benefit relative to locking with busy waiting?*

To read more: Non-blocking algorithms and data structures were introduced by Herlihy [3, 2, 6]. Since then a significant body of research on this topic has developed, and it has reached a level of maturity allowing it to be considered a general approach to concurrent programming [5]. This is also the basis for interest in transactional memory [4].

3.4 Summary

Abstractions

The main abstraction introduced in this chapter is semaphores — a means of expressing the condition of waiting for an event that will be performed by another process. This abstraction was invented to aid in the construction of operating systems, which are notoriously difficult to get right because of their concurrency. It was so successful that it has since moved to the user interface, and is now offered as a service to user applications on many systems.

A closely related abstraction is that of a lock. The difference is that semaphores are more "low level", and can be used as a building block for various synchronization needs. Locks focus on the functionality of regulating concurrent access to resources, and ensuring mutual exclusion when needed. This includes the definition of special cases such as readers-writers locks.

Another abstraction is the basic notion of a "resource" as something that a process may need for its exclusive use. This can be anything from access to a device to an entry in a system table.

Resource management

Deadlock prevention or avoidance have a direct effect on resource management: they limit the way in which resources can be allocated, in the interest of not entering a deadlock state.

Mechanisms for coordination have an indirect effect: providing efficient mechanisms (such as blocking) avoids wasting resources (as is done by busy waiting).

Workload issues

Workload issues are not very prominent with regard to the topics discussed here, but they do exist. For example, knowing the distribution of lock waiting times can influence the choice of a locking mechanism. Knowing that deadlocks are rare allows for the problem to be ignored.

In general, the load on the system is important in the context of concurrency. Higher loads imply more concurrency, giving the system more flexibility in choosing what process or thread to run. But higher loads also tend to expose more problems where concurrency was not handled properly. If the load is low, e.g. when there is only one active process in the system, concurrency is almost never a problem.

Hardware support

Hardware support is crucial for the implementation of coordination mechanisms — without it, we would have to use busy waiting, which is extremely wasteful. The most common form of hardware support used is the simple idea of blocking interrupts. More sophisticated forms of support include atomic instructions such as test_and_set and compare_and_swap. These can then be amplified to create arbitrarily complex synchronization mechanisms.

Bibliography

[1] E. W. Dijkstra, "*Co-operating sequential processes*". In *Programming Languages*, F. Genuys (ed.), pp. 43–112, Academic Press, 1968.

[2] M. Herlihy, "*A methodology for implementing highly concurrent data structures*". In 2nd *Symp. Principles & Practice of Parallel Programming*, pp. 197–206, Mar 1990.

[3] M. Herlihy, "*Wait-free synchronization*". *ACM Trans. Prog. Lang. & Syst.* **13(1)**, pp. 124–149, Jan 1991.

[4] M. Herlihy and J. E. B. Moss, "*Transactional memory: Architectural support for lock-free data structures*". In 20th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 289–300, May 1993.

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[6] M. P. Herlihy and J. E. B. Moss, "*Lock-free garbage collection for multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **3(3)**, pp. 304–311, May 1992.

[7] C. A. R. Hoare, "*Monitors: An operating system structuring concept*". *Comm. ACM* **17(10)**, pp. 549–557, Oct 1974.

[8] M. Hofri, "*Proof of a mutual exclusion algorithm – a 'class'ic example*". *Operating Syst. Rev.* **24(1)**, pp. 18–22, Jan 1990.

[9] L. Lamport, "*A new solution of Dijkstra's concurrent programming problem*". *Comm. ACM* **17(8)**, pp. 453–455, Aug 1974.

[10] L. Lamport, "*The mutual exclusion problem: Part I — a theory of interprocess communication*". *J. ACM* **33(2)**, pp. 313–326, Apr 1986.

[11] L. Lamport, "*The mutual exclusion problem: Part II — statement and solutions*". *J. ACM* **33(2)**, pp. 327–348, Apr 1986.

[12] L. Lamport, "*A fast mutual exclusion algorithm*". *ACM Trans. Comput. Syst.* **5(1)**, pp. 1–11, Feb 1987.

[13] N. G. Leveson and C. S. Turner, "*An investigation of the Therac-25 accidents*". *Computer* **26(7)**, pp. 18–41, Jul 1993.

[14] G. L. Peterson, "*Myths about the mutual exclusion problem*". *Inf. Process. Lett.* **12(3)**, pp. 115–116, Jun 1981.

[15] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[16] W. Stallings, *Operating Systems*. Prentice-Hall, 2nd ed., 1995.

[17] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[18] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd ed., 1997.

Chapter 4

Memory Management

Primary memory is a prime resource in computer systems. Its management is an important function of operating systems. However, in this area the operating system depends heavily on hardware support, and the policies and data structures that are used are dictated by the support that is available.

Actually, three players are involved in handling memory. The first is the compiler, which structures the address space of an application. The second is the operating system, which maps the compiler's structures onto the hardware. The third is the hardware itself, which performs the actual accesses to memory locations.

4.1 Mapping Memory Addresses

Let us first review what the memory is used for: it contains the context of processes. This is typically divided into several segments or regions.

Different parts of the address space have different uses

The address space is typically composed of regions with the following functionalities:

Text segment — this is the code, or machine instructions, of the application being executed, as created by the compiler. It is commonly flagged as read-only, so that programs cannot modify their code during execution. This allows such memory to be shared among a number of processes that all execute the same program (e.g. multiple instances of a text editor).

Data segment — This usually contains predefined data structures, possibly initialized before the program starts execution. Again, this is created by the compiler.

Heap — this is an area in memory used as a pool for dynamic memory allocation. This is useful in applications that create data structures dynamically, as is common in object-oriented programming. In C, such memory is acquired using the

`malloc` library routine. The implementation of `malloc`, in turn, makes requests to the operating system in order to enlarge the heap as needed. In Unix, this is done with the `brk` system call.

Stack — this is the area in memory used to store the execution frames of functions called by the program. Such frames contain stored register values and space for local variables. Storing and loading registers is done by the hardware as part of the instructions to call a function and return from it (see Appendix A). Again, this region is enlarged at runtime as needed.

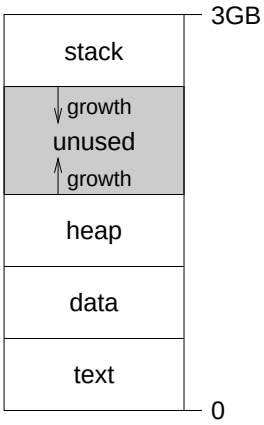
In some systems, there may be more than one instance of each of these regions. For example, when a process includes multiple threads, each will have a separate stack. Another common examle is the use of dynamically linked libraries; the code for each such library resides in a separate segment, that — like the text segment — can be shared with other processes. In addition, the data and heap may each be composed of several independent segments that are acquired at different times along the execution of the application. For example, in Unix it is possible to create a new segment using the `shmget` system call, and then multiple processes may attach this segment to their address spaces using the `shmat` system call.

These different parts have to be mapped to addresses

The compiler creates the text and data segments as relocatable segments, that is with addresses from 0 to their respective sizes. Text segments of libraries that are used by the program also need to be included. The heap and stack are only created as part of creating a process to execute the program. All these need to be mapped to the process address space.

The sum of the sizes of the memory regions used by a process is typically much smaller than the total number of addresses it can generate. For example, in a 32-bit architecture, a process can generate addresses ranging from 0 to $2^{32}-1 = 4,294,967,295$, which is 4 GB, of which say 3GB are available to the user program (the remainder is left for the system, as described in Section 11.7.1). The used regions must be mapped into this large virtual address space. This mapping assigns a virtual address to every instruction and data structure. Instruction addresses are used as targets for branch instructions, and data addresses are used in pointers and indexed access (e.g. the fifth element of the array is at the address obtained by adding four element sizes to the the array base address).

Mapping static regions, such as the text, imposes no problems. The problem is with regions that may grow at runtime, such as the heap and the stack. These regions should be mapped so as to leave them ample room to grow. One possible solution is to map the heap and stack so that they grow towards each other. This allows either of them to grow much more than the other, without having to know which in advance. No such solution exists if there are multiple stacks.



Exercise 91 *Is it possible to change the mapping at runtime to increase a certain segment's allocation? What conditions must be met? Think of what happens if the code includes statements such as "x = &y;".*

Exercise 92 *Write a program which calls a recursive function that declares a local variable, allocates some memory using malloc, and prints their addresses each time. Are the results what you expected?*

And the virtual addresses need to be mapped to physical ones

The addresses generated by the compiler are *relative* and *virtual*. They are relative because they assume the address space starts from address 0. The are virtual because they are based on the assumption that the application has all the address space from 0 to 3GB at its disposal, with a total disregard for what is physically available. In practice, it is necessary to map these compiler-generated addresses to physical addresses in primary memory, subject to how much memory you bought and contention among different processes.

In bygone days, dynamic memory allocation was not supported. The size of the used address space was then fixed. The only support that was given was to map a process's address space into a contiguous range of physical addresses. This was done by the *loader*, which simply set the *base address register* for the process. Relative addresses were then interpreted by adding the base address to each of them.

Today, paging is used, often combined with segmentation. The virtual address space is broken into small, fixed-size pages. These pages are mapped independently to frames of physical memory. The mapping from virtual to physical addresses is done by special hardware at runtime, as described below in Section 4.3.

4.2 Segmentation and Contiguous Allocation

In general, processes tend to view their address space (or at least each segment of it) as contiguous. For example, in C it is explicitly assumed that array elements will

reside one after the other, and can therefore be accessed using pointer arithmetic. The simplest way to support this is to indeed map contiguous segments of the address space to sequences of physical memory locations.

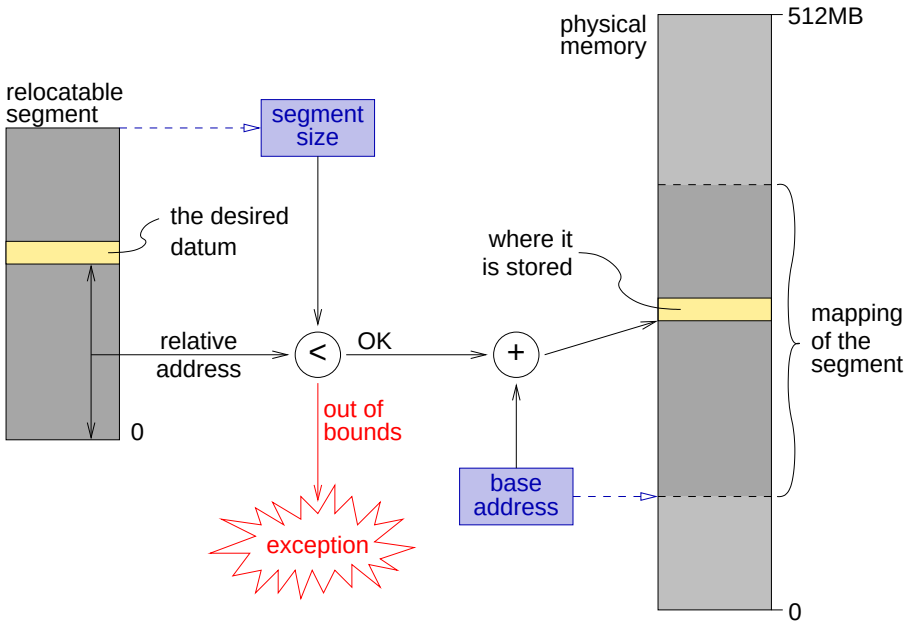
4.2.1 Support for Segmentation

Segments are created by the programmer or compiler

Segments are arbitrarily-sized contiguous ranges of the address space. They are a tool for structuring the application. As such, the programmer or compiler may decide to arrange the address space using multiple segments. For example, this may be used to create some data segments that are shared with other processes, while other segments are not.

Segments can be mapped at any address

As explained above, compilers typically produce *relocatable code*, with addresses relative to the segment's base address. In many systems, this is also supported by hardware: the memory access hardware includes a special architectural register[1] that is loaded with the segment's base address, and this is automatically added to the relative address for each access. In addition, another register is loaded with the size of the segment, and this value is compared with the relative address. If the relative address is out of bounds, a memory exception is generated.



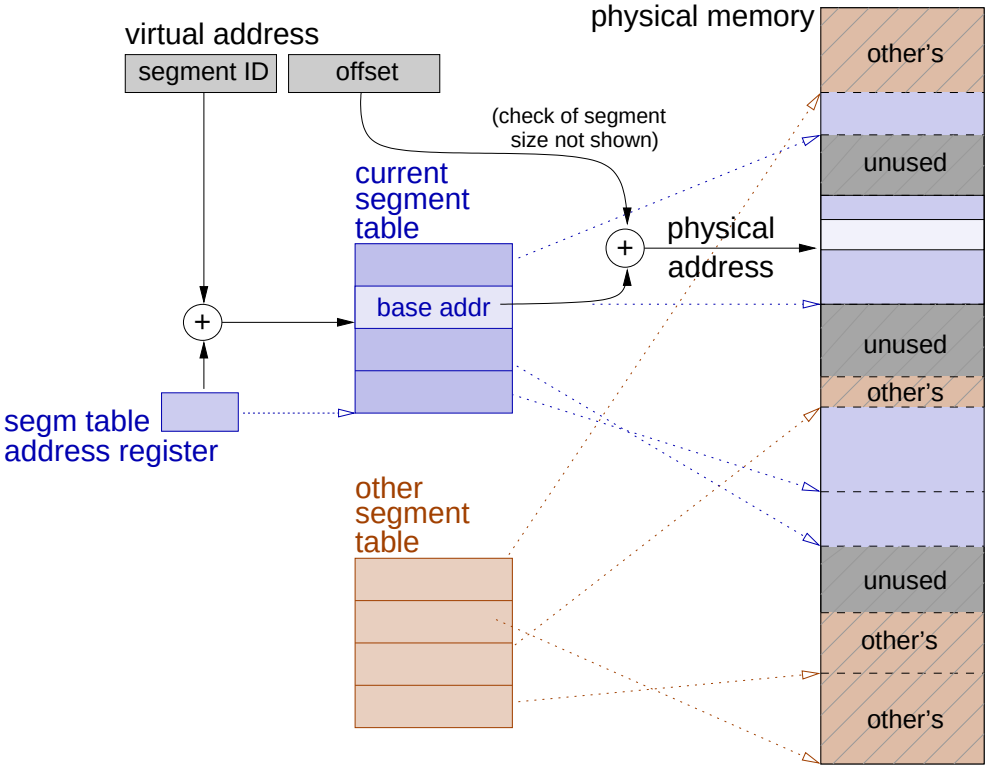
[1]This means that this register is part of the definition of the architecture of the machine, i.e. how it works and what services it provides to software running on it. It is not a general purpose register used for holding values that are part of the computation.

Exercise 93 *What are the consequences of not doing the bounds check?*

Multiple segments can be supported by using a table

Using a special base registers and a special length register is good for supporting a single contiguous segment. But as noted above, we typically want to use multiple segments for structuring the address space. To enable this, the segment base and size values are extracted from a segment table rather than from special registers. The problem is then which table entry to use, or in other words, to identify the segment being accessed. To do so an address has to have two parts: a segment identifier, and an offset into the segment. The segment identifier is used to index the segment table and retrieve the segment base address and size, and these can then be used to find the physical address as described above.

Exercise 94 *Can the operating system decide that it wants to use multiple segments, or does it need hardware support?*



Once we have multiple segments accessed using a table, it is an easy step to having a distinct table for each process. Whenever a process is scheduled to run, its table is identified and used for memory accesses. This is done by keeping a pointer to the table in the process's PCB, and loading this pointer into a special register when the process is scheduled to run. This register is part of the architecture, and serves as the basis of

the hardware address translation mechanism: all translations start with this register, which points to the segment table to use, which contains the data regarding the actual segments. As part of each context switch the operating system loads this register with the address of the new process's segment table, thus effectively switching the address space that will be used.

Note that using such a register to point to the segment table also leads to the very important property of memory protection: when a certain process runs, only its segment table is available, so only its memory can be accessed. The mappings of segments belonging to other processes are stored in other segment tables, so they are not accessible and thus protected from any interference by the running process.

The contents of the tables, i.e. the mapping of the segments into physical memory, is done by the operating system using algorithms described next.

4.2.2 Algorithms for Contiguous Allocation

Assume that segments are mapped to contiguous ranges of memory. As jobs are loaded for execution and then terminate, such ranges are allocated and de-allocated. After some time, the memory will be divided into many allocated ranges, separated by unallocated ranges. The problem of allocation is just one of finding a contiguous range of free memory that is large enough for the new segment being mapped.

First-fit, best-fit, and next-fit algorithms just search

The simplest data structure for maintaining information about available physical memory is a linked list. Each item in the list contains the start address of a free range of memory, and its length.

Exercise 95 *The two operations required for managing contiguous segments are to allocate and de-allocate them. Write pseudo-code for these operations using a linked list of free ranges.*

When searching for a free area of memory that is large enough for a new segment, several algorithms can be used:

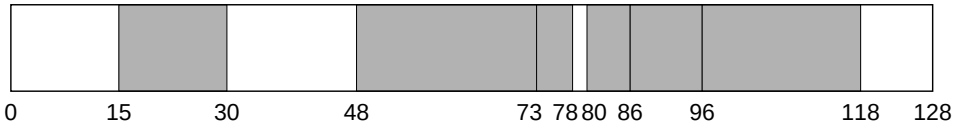
First-fit scans the list from its beginning and chooses the first free area that is big enough.

Best-fit scans the *whole* list, and chooses the smallest free area that is big enough. The intuition is that this will be more efficient and only waste a little bit each time. However, this is wrong: first-fit is usually better, because best-fit tends to create multiple small and unusable fragments [11, p. 437].

Worst-fit is the opposite of best-fit: it selects the biggest free area, with the hope that the part that is left over will still be useful.

Next-fit is a variant of first-fit. The problem is that first-fit tends to create more fragmentation near the beginning of the list, which causes it to take more time to reach reasonably large free areas. Next-fit solves this by starting the next search from the point where it made the previous allocation. Only when the end of the list is reached does it start again from the beginning.

Exercise 96 *Given the following memory map (where gray areas are allocated), what was the last segment that was allocated assuming the first-fit algorithm was used? and what if best-fit was used?*

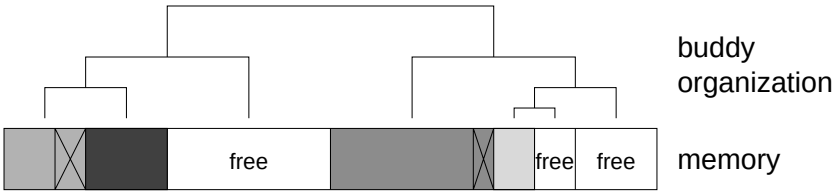


Exercise 97 *Given a certain situation (that is, memory areas that are free and allocated), and a sequence of requests that can be satisfied by first-fit, is it always true that these requests can also be satisfied by best-fit? How about the other way around? Prove these claims or show counter examples.*

To read more: There is extensive literature regarding the detailed analysis of these and other packing algorithms in an off-line setting. See Coffman et al. for a good survey [3].

Buddy systems use predefined partitions

The complexity of first-fit and best-fit depends on the length of the list of free areas, which becomes longer with time. An alternative with constant complexity is to use a buddy system.



With a buddy system, memory is partitioned according to powers of two. Each request is rounded up to the nearest power of two (that is, the size of the address space that is allocated is increased — represented by an area marked with an X in the figure). If a block of this size is available, it is allocated. If not, a larger block is split repeatedly into a pair of buddies until a block of the desired size is created. When a block is released, it is re-united with its buddy if the buddy is free.

> **Overhead Vs. Utilization**
> The complexities of the various algorithms differ: best fit is linear in the number of free ranges, first fit is also linear but with a constant smaller than one, and buddy systems

are logarithmic. On the other hand, the different algorithms achieve different levels of memory utilization. The question is then which effect is the dominant one: is it worth while to use a more sophisticated algorithm to push up the utilization at the price of more overhead, or is it better to forgo some utilization and also reduce overhead?

In order to answer such questions it is necessary to translate these two metrics into the same currency. A promising candidate is the effect on the performance of user applications. With this currency, it is immediately evident that algorithms with a higher complexity are detrimental, because time spent running the algorithm is not spent running user programs. Likewise, an inefficient algorithm that suffers from fragmentation and reduced memory utilization may cause user processes to have to wait for memory to become available. We can use detailed simulations to assess whether the more sophisticated algorithm manages to reduce the waiting time enough to justify its cost, or whether it is the other way round. The results will depend on the details of the workload, such as the distribution of job memory requirements.

Fragmentation is a problem

The main problem with contiguous allocation is fragmentation: memory becomes split into many small pieces that cannot be used effectively. Two types of fragmentation are distinguished:

- *Internal fragmentation* occurs when the system allocates more than the process requested and can use. For example, this happens when the size of the address space is increased to the next power of two in a buddy system.
- *External fragmentation* occurs when unallocated pieces of memory are left between allocated areas, and these unallocated pieces are all too small to satisfy any additional requests (even if their sum may be large enough).

Exercise 98 *Does next-fit suffer from internal fragmentation, external fragmentation, or both? And how about a buddy system?*

One solution to the problem of fragmentation is *compaction*: relocate the various segments so as to accumulate all the free space in one place. This will hopefully be large enough for additional allocations. However this suffers from considerable overhead. A better solution is to use paging rather than contiguous allocation.

Exercise 99 *Does compaction solve internal fragmentation, external fragmentation, or both?*

Allocations from the heap may use caching

While operating systems typically use paging rather than contiguous allocation, the above algorithms are by no means obsolete. For example, allocation of memory from

the heap is done by similar algorithms. It is just that the usage has shifted from the operating system to the runtime library.

Modern algorithms for memory allocation from the heap take usage into account. In particular, they assume that if a memory block of a certain size was requested and subsequently released, there is a good chance that the same size will be requested again in the future (e.g. if the memory is used for a new instance of a certain object). Thus freed blocks are kept in lists in anticipation of future requests, rather than being merged together. Such reuse reduces the creation of additional fragmentation, and also reduces overhead.

4.3 Paging and Virtual Memory

The instructions and data of a program have to be in main memory for the CPU to use them. However, most applications typically use only a fraction of their instructions and data at any given time. Therefore it is possible to keep only the needed parts in memory, and put the rest on disk. This decouples the memory as seen and used by the application from its physical implementation, leading to the concept of "virtual memory".

Technically, the allocation and mapping of virtual memory is done in fixed-size units called *pages*. The activity of shuttling pages to the disk and back into main memory is called *paging*. It is used on practically all contemporary general purpose systems.

To read more: Jacob and Mudge provide a detailed survey of modern paging schemes and their implementation [9]. This doesn't exist in most textbooks on Computer Architecture.

4.3.1 The Concept of Paging

Paging provides the ultimate support for virtual memory. It not only decouples the addresses used by the application from the physical memory addresses of the hardware, but also decouples the amount of memory used by the application from the amount of physical memory that is available.

Paging works by shuttling pages of memory between the disk and physical memory, as needed

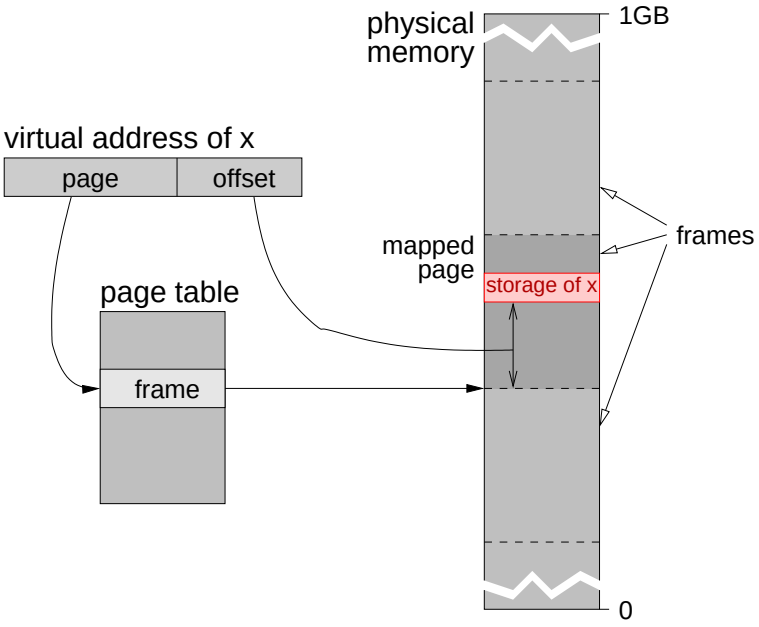
The basic idea in paging is that memory is mapped and allocated in small, fixed size units (the pages). A typical page size is 4 KB. Addressing a byte of memory can then be interpreted as being composed of two parts: selecting a page, and specifying an offset into the page. For example, with 32-bit addresses and 4KB pages, 20 bits indicate the page, and the remaining 12 identify the desired byte within the page. In mathematical notation, we have

$$\text{page} = \left\lfloor \frac{\text{address}}{4096} \right\rfloor$$

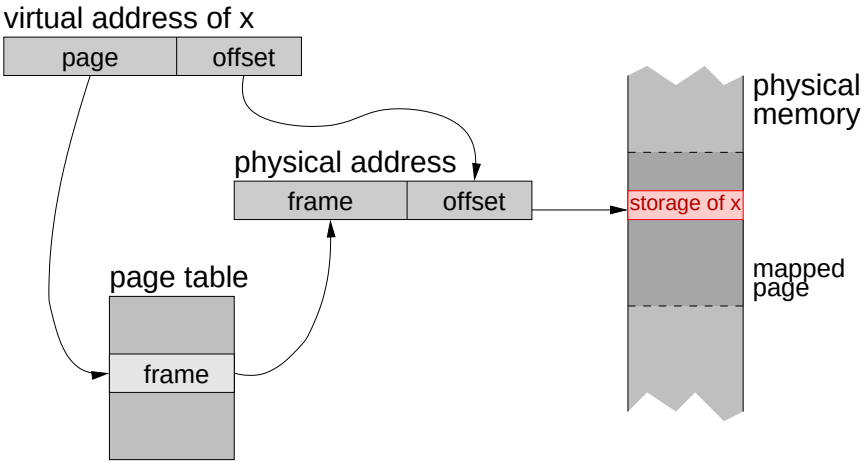
$$\text{offset} = \text{address} \bmod 4096$$

Using bit positions like this leads to page sizes that are powers of two, and allows for simple address translation in hardware.

The operating system maps pages of virtual addresses to *frames* of physical memory (of the same size). The mapping is kept in the *page table*. When the CPU wants to access a certain virtual address (be it the next instruction or a data item), the hardware uses the page table to translate the virtual page number to a physical memory frame. It then accesses the specified byte in that frame.



Note that the hardware does not need to perform mathematical operations to figure out which frame is needed and the offset into this frame. By virtue of using bits to denote pages and offsets, it just needs to select a subset of the bits from the virtual address and append them to the bits of the frame number. This generates the required physical address that points directly at the desired memory location.



If a page is not currently mapped to a frame, the access fails and generates a *page fault*. This is a special type of interrupt. The operating system handler for this interrupt initiates a disk operation to read the desired page, and maps it to a free frame. While this is being done, the process is blocked. When the page becomes available, the process is unblocked, and placed on the ready queue. When it is scheduled, it will resume its execution with the instruction that caused the page fault — but this time it will work. This manner of getting pages as they are needed is called *demand paging*.

Exercise 100 *Does the page size have to match the disk block size? What are the considerations?*

A possible cloud on this picture is that there may be no free frames to accommodate the page. Indeed, the main issue in paging systems is choosing pages to page out in order to make room for other pages.

> **Virtual and Physical Addresses**
> Understanding the meaning of virtual and physical addresses is crucial to understand paging, so we'll give an analogy to drive the definition home. In a nutshell, a virtual or logical address is the name you use to refer to something. The physical address is the location where this something can be found.
>
> This distinction doesn't always make sense in real life. For example, when referring to the building at 123 Main Street, the string "123 Main Street" serves both as its name and as an indication of its location.
>
> A better example for our purpose is provided by articles in the scientific literature. For example, we may have the citation "*Computer* Magazine, volume 23, issue 5, page 65". This is a logical address, or name, of where the article was published. The first part, "*Computer* Magazine", is like the page number. Your librarian (in the capacity of an operating system) will be able to translate it for you into a physical address: this is the journal in the third rack on the left, second shelf from the top. The rest, "volume 23, issue 5, page 65", is an offset into the journal. Using the physical address you got, you go to the shelf and find the desired article in the journal.

4.3.2 Benefits and Costs

Paging provides improved flexibility

When memory is partitioned into pages, these pages can be mapped into unrelated physical memory frames. Pages that contain contiguous virtual addresses *need not* be contiguous in physical memory. Moreover, all pages and all frames are of the same size, so any page can be mapped to any frame. As a result the allocation of memory is much more flexible.

Paging relieves the constraints of physical availability

An important observation is that not all the used parts of the address space need to be mapped to physical memory simultaneously. Only pages that are accessed in the current phase of the computation are mapped to physical memory frames, as needed. Thus programs that use a very large address space may still be executed on a machine with a much smaller physical memory.

Exercise 101 *What is the maximal number of pages that may be required to be memory resident (that is, mapped to physical frames) in order to execute a single instruction?*

Paging provides reduced fragmentation

Because page sizes match frame sizes, external fragmentation is eliminated. Internal fragmentation is also small, and occurs only because memory is allocated in page units. Thus when memory is allocated for a segment of B bytes, this will require a total of $\lceil B/P \rceil$ pages (where P is the page size). In the last page, only $B \bmod P$ bytes will be used, and the rest wasted. On average, this will lead to a waste of half a page per segment.

An obvious way to reduce the internal fragmentation is to reduce the page size. However, it is best not to make pages too small, because this will require larger page tables, and will also suffer from more page faults.

Exercise 102 *Can the operating system set the page size at will?*

It depends on locality

The success of paging systems depends on the fact that applications display *locality of reference*. This means that they tend to stay in the same part of the address space for some time, before moving to other remote addresses. With locality, each page is used many times, which amortizes the cost of reading it off the disk. Without locality, the system will *thrash*, and suffer from multiple page faults.

Luckily, this is a good assumption. The vast majority of applications do indeed display a high degree of locality. In fact, they display locality at several levels. For

example, locality is also required for the efficient use of caches. Indeed, paging may be viewed as a coarse-grain version of caching, where physical memory frames are used to cache pages of the virtual address space. This analogy is shown by the following table:

	fast storage	*slow storage*	*transfer unit*
processor cache	cache	primary memory	cache line
paging	primary memory	disk	page

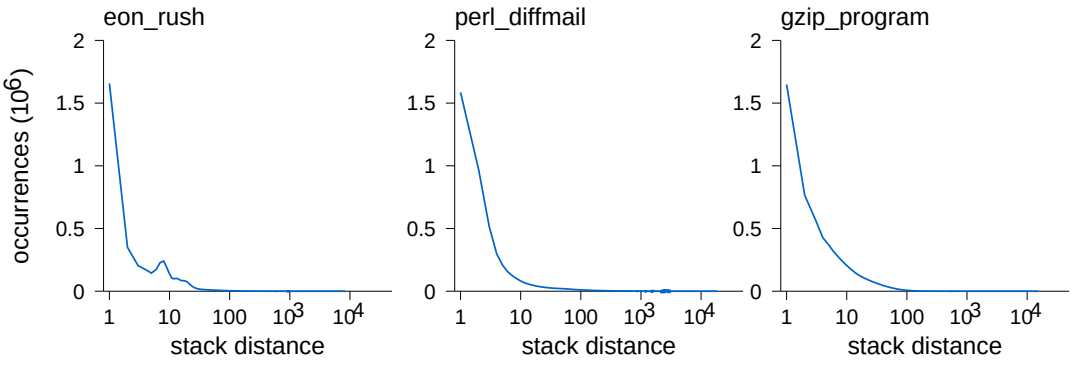
However, one difference is that in the case of memory the location of pages is fully associative: any page can be in any frame, as explained below.

Exercise 103 *The cache is maintained by hardware. Why is paging delegated to the operating system?*

Detail: measuring locality

Locality can be measure using the "average stack depth" where data would be found, if all data items were to be kept in a stack. The idea is as follows. Assume everything is kept in one large stack. Whenever you access a new datum, one that you have not accessed before, you also deposit it on the top of the stack. But when you re-access a datum, you need to find it in the stack. Once it is found, you note its depth into the stack, and then you move it from there to the top of the stack. Thus the order of items in the stack reflects the order in which they were last accessed.

But how does this measure locality? If memory accesses are random, you would expect to find items "in the middle" of the stack. Therefore the average stack depth will be high, roughly of the same order of magnitude as the size of all used memory. But if there is significant locality, you expect to find items near the top of the stack. For example, if there is significant temporal locality, it means that we tend to repeatedly access the same items. If such repetitions come one after the other, the item will be found at the top of the stack. Likewise, if we repeatedly access a set of items in a loop, this set will remain at the top of the stack and only the order of its members will change as they are accessed.



experimental results from several SPEC 2000 benchmarks, shown in these graphs, confirm that they have very strong locality. Low stack depths, typically not much more than

10, appear hundreds of thousands of times, and account for the vast majority of references. Only in rare cases an item is found deep into the stack.

Another measure of locality is the size fo the *working set*, loosely defined as the set of addresses that are needed at a particular time [7]. The smaller the set (relative to the full set of all possible addresses), the stronger the locality. This may change for each phase of the computation.

It also depends on the memory/disk cost ratio

Another underlying assumption in paging systems is that disk storage costs less than primary memory. That is why it is worth while to store most of the data on (slower) disks, and only keep the working set in (fast CPU-accessible) memory. For the time being, this is a safe assumption. While the price of memory has been dropping continuously, so has that of disks. But it is possible that in the future memory will become cheap enough to call the use of paging into question [8, sect. 2.4].

Exercise 104 *So when memory becomes cheaper than disk, this is the end of paging? Or are there reasons to continue to use paging anyway?*

The cost is usually very acceptable

So far we have only listed the benefits of paging. But paging may cause applications to *run slower*, due to the overheads involved in interrupt processing. This includes both the direct overhead of actually running the interrupt handler, and the indirect overhead of reduced cache performance due to more context switching, as processes are forced to yield the CPU to wait for a page to be loaded into memory. The total effect may be a degradation in performance of 10–20% [10].

To improve performance, it is crucial to reduce the page fault rate. This may be possible with good prefetching: for example, given a page fault we may bring additional pages into memory, instead of only bringing the one that caused the fault. If the program subsequently references these additional pages, it will find them in memory and avoid the page fault. Of course there is also the danger that the prefetching was wrong, and the program does not need these pages. In that case it may be detrimental to bring them into memory, because they may have replaced other pages that are actually more useful to the program.

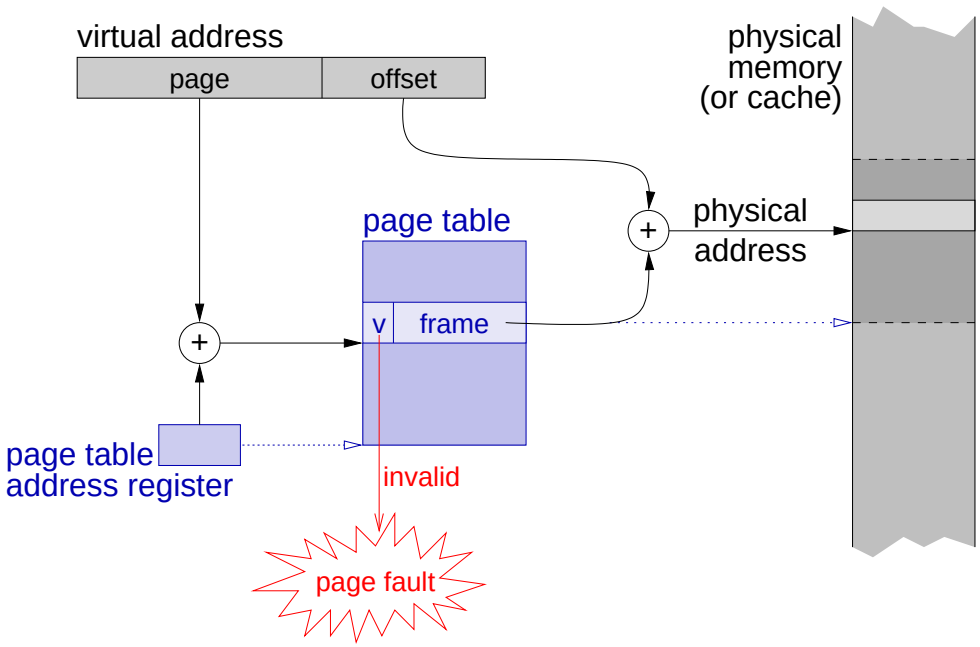
4.3.3 Address Translation

As mentioned above, paging hinges on hardware support for memory mapping that allows pages to be mapped to any frame of memory.

Address translation is done by the hardware, not the operating system

Essentially every instruction executed by the CPU requires at least one memory access — to get the instruction. Oftentimes it also requires 1 to 3 additional accesses to fetch operands and store the result. This means that memory access must be fast, and obviously cannot involve the operating system. Therefore translating virtual addresses to physical addresses must be done by the hardware.

Schematically, the translation is done as follows.



The page part of the virtual address is used as an index into the page table, which contains the mapping of pages to frames (this is implemented by adding the page number to a register containing the base address of the page table). The frame number is extracted and used as part of the physical address. The offset is simply appended. Given the physical address, an access to memory is performed. This goes through the normal mechanism of checking whether the requested address is already in the cache. If it is not, the relevant cache line is loaded from memory.

To read more: Caching is covered in all textbooks on computer architecture. It is largely orthogonal to the discussion here.

The page table also contains some additional bits of information about each page, including

- A *valid bit* (also called *present bit*), indicating whether the page is assigned to a frame or not. If it is not, and some word in the page is accessed, the hardware will generate a page fault. This bit is set by the operating system when the page is mapped.

- A *modified* or *dirty bit*, indicating whether the page has been modified. If so, it has to be written to disk when it is evicted. This bit is cleared by the operating system when the page is mapped. It is set automatically by the hardware each time any word in the page is written.
- A *used bit*, indicating whether the page has been accessed recently. It is set automatically by the hardware each time any word in the page is accessed. "Recently" means since the last time that the bit was cleared by the operating system.
- *Access permissions*, indicating whether the page is read-only or read-write. These are set by the operating system when the page is mapped, and may cause the hardware to trigger an exception if an access does not match the permissions.

We discuss the use of these bits below.

Useful mappings are cached in the TLB

One problem with this scheme is that the page table can be quite big. In our running example, there are 20 bits that specify the page, for a total of $2^{20} = 1048576$ pages in the address space. This is also the number of entries in the page table. Assuming each one requires 4 bytes, the page table itself uses 1024 pages of memory, or 4 MB. Obviously this cannot be stored in hardware registers. The solution is to have a special cache that keeps information about recently used mappings. This is again based on the assumption of locality: if we have used the mapping of page X recently, there is a high probability that we will use this mapping many more times (e.g. for other addresses that fall in the same page). This cache is called the *translation lookaside buffer* (TLB).

With the TLB, access to pages with a cached mapping can be done immediately. Access to pages that do not have a cached mapping requires two memory accesses: one to get the entry from the page table, and another to the desired address.

Note that the TLB is separate from the general data cache and instruction cache. Thus the translation mechanism only needs to search this special cache, and there are no conflicts between it and the other caches. One reason for this separation is that the TLB includes some special hardware features, notably the used and modified bits described above. These bits are only relevant for pages that are currently being used, and must be updated by the hardware upon each access.

Inverted page tables are smaller

The TLB may make access faster, but it can't reduce the size of the page table. As the number of processes increases, the percentage of memory devoted to page tables also increases. This problem can be solved by using an inverted page table. Such a table only has entries for pages that are allocated to frames. Inserting pages and searching

for them is done using a hash function. If a page is not found, it is inferred that it is not mapped to any frame, and a page fault is generated.

The drawback of inverted page tables is that another structure is needed to maintain information about where pages are stored on disk, if they are not mapped to a frame. With conventional page tables, the same space can be used to hold the mapping to a frame *or* the location on disk.

Exercise 105 *Another way to reduce the size of the page table is to enlarge the size of each page: for example, we can decide that 10 bits identify the page and 22 the offset, thus reducing the page table to 1024 entries. Is this a good idea?*

The operating system only handles problems

So far we have only discussed hardware support. Where does the operating system come in? Only when things don't go smoothly.

Pages representing parts of a process's virtual address space can be in one of 3 states:

- Mapped to a physical memory frame. Such pages can be accessed directly by the CPU, and do not require operating system intervention. This is the desired state.
- Backed on disk. When such a page is accessed, a page fault occurs. This is a type of interrupt. The operating system handler function then initiates a disk operation to read the page and map it to a frame of physical memory. When this is completed, the process continues from the instruction that caused the page fault.
- Not used. Such pages are not part of any used memory segment. Trying to access them causes a memory error, which causes an interrupt. In this case, the operating system handler kills the job.

This involves maintaining the mappings

When a page fault occurs, the required page must be mapped to a free frame. However, free frames are typically in short supply. Therefore a painful decision has to be made, about what page to evict to make space for the new page. The main part of the operating system's memory management component is involved with making such decisions.

In a nutshell, the division of labor is as follows: the operating system maps pages of virtual memory to frames of physical memory. The hardware uses the mapping, as represented in the page table, to perform actual accesses to memory locations.

In fact, the operating system maintains multiple mappings. If there are multiple processes, each has its own page table, that maps its address space to the frames that have been allocated to it. If a process has multiple segments, they may each have

an independent page table (as shown below). Therefore the operating system has to notify the hardware which mapping is in effect at any given moment. This is done by loading a special architectural register with the address of the table that should be used. To switch to another mapping (e.g. as part of a context switch), only this register has to be reloaded.

It also allows various tricks

Controlling the mapping of pages allows the operating system to play various tricks. A few famous examples are

- Leaving unmapped regions to catch stray memory accesses.

  As noted above, access to unmapped pages causes a memory exception, which is forwarded to the operating system for handling. This can be used to advantage by leaving unmapped regions in strategic places. For example, access to local variables in functions is done by using a calculated offset from the stack pointer (SP). By leaving a few unmapped pages beyond the end of the stack, it is possible to catch erroneous accesses that extend farther than the pre-allocated space.

  Exercise 106 *Will all erroneous accesses be caught? What happens with those that are not caught?*

- Implementing the copy-on-write optimization.

  In Unix, forking a new process involves copying all its address space. However, in many cases this is a wasted effort, because the new process performs an exec system call and runs another application instead. A possible optimization is then to use copy-on-write. Initially, the new process just uses its parent's address space, and all copying is avoided. A copy is made only when either process tries to modify some data, and even then, only the affected page is copied.

  This idea is implemented by copying the parent's page table to the child, and marking the whole address space as read-only. As long as both processes just read data, everything is OK. When either process tries to modify data, this will cause a memory error exception (because the memory has been tagged read-only). The operating system handler for this exception makes separate copies of the offending page, and changes their mapping to read-write.

- Swapping the Unix u-area.

  In Unix, the u-area is a kernel data structure that contains important information about the currently running process. Naturally, the information has to be maintained somewhere also when the process is not running, but it doesn't have to be *accessible*. Thus, in order to reduce the complexity of the kernel code, it is convenient if only one u-area is visible at any given time.

This idea is implemented by compiling the kernel so that the u-area is page aligned (that is, the data structure starts on an address at the beginning of a page). For each process, a frame of physical memory is allocated to hold its u-area. However, these frames are not mapped into the kernel's address space. Only the frame belonging to the current process is mapped (to the page where the "global" u-area is located). As part of a context switch from one process to another, the mapping of this page is changed from the frame with the u-area of the previous process to the frame with the u-area of the new process.

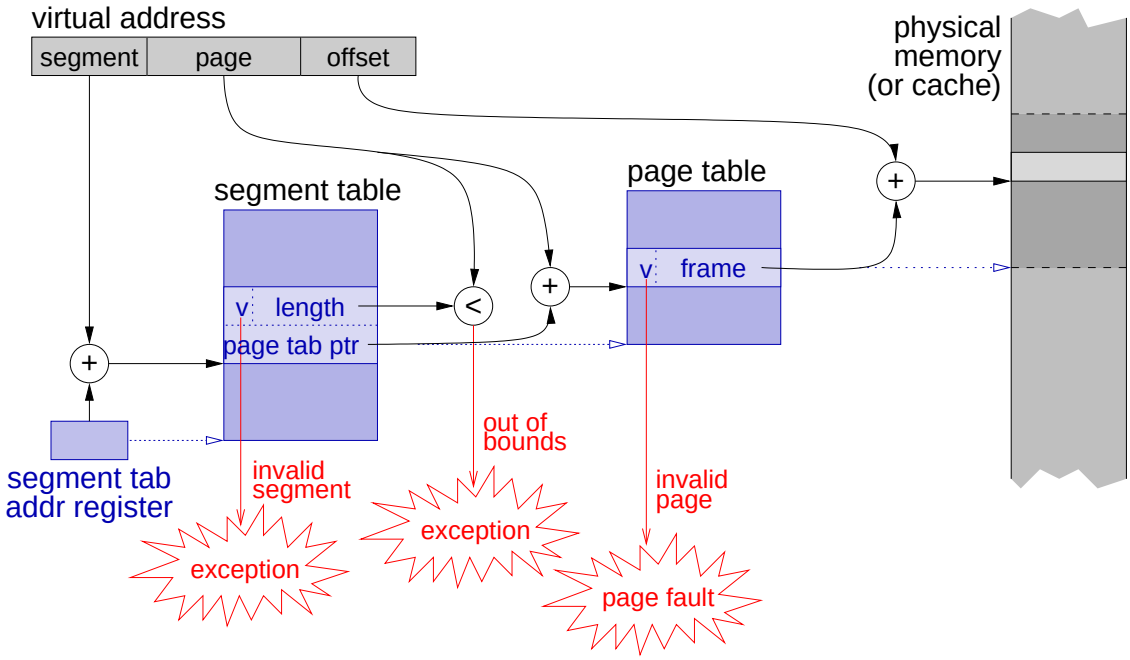
If you understand this, you understand virtual memory with paging.

Paging can be combined with segmentation

It is worth noting that paging is often combined with segmentation. All the above can be done on a per-segment basis, rather than for the whole address space as a single unit. The benefit is that now segments do not need to be mapped to contiguous ranges of physical memory, and in fact segments may be larger than the available memory.

Exercise 107 *Does this mean that there will be no fragmentation?*

Address translation for paged segments is slightly more complicated, though, because each segment has its own page table. The virtual address is then parsed into three parts:



1. The segment number, which is used as an index into the segment table and finds the correct page table. Again, this is implemented by adding it to the contents of a special register that points to the base address of the segment table.

2. The page number, which is used as an index into the page table to find the frame to which this page is mapped. This is mapped by adding it to the base address of the segment's page table, which is extracted from the entry in the segment table.
3. An offset into the page.

As far as the compiler or programmer is concerned, addresses are expressed as a segment and offset into the segment. The system interprets this offset as a page and offset into the page. As before, the segment offset is compared with the segment length to check for illegal accesses (the drawing assumes this is done in page units). In addition, a page may be absent from memory, thereby generating a page fault.

Exercise 108 *Why is the case of an invalid segment shown as an exception, while an invalid page is a page fault?*

Exercise 109 *When segments are identified by a set of bits in the virtual address, does this impose any restrictions?*

To read more: Stallings [13, Sect. 7.3] gives detailed examples of the actual structures used in a few real systems (this was deleted in the newer edition). More detailed examples are given by Jacob and Mudge [9].

Protection is built into the address translation mechanism

An important feature of address translation for paging is that each process can only access frames that appear in its page table. There is no way that it can generate a physical address that lies in a frame that is allocated to another process. Thus there is no danger that one process will inadvertently access the memory of another.

Moreover, this protection is cheap to maintain. All the address translation starts from a single register, which holds the base address of the page table (or the segment table, in case paged segmentation is used). When the operating system decides to do a context switch, it simply loads this register with the address of the page table of the new process. Once this is done, the CPU no longer "sees" any of the frames belonging to the old process, and can only access pages belonging to the new one.

A slightly different mechanism is used by architectures that use a single page table, rather than a separate one for each process. In such architectures the operating system loads a unique address space identifier (ASID) into a special register, and this value is appended to each address before the mapping is done. Again, a process is prevented from generating addresses in the address spaces of other processes, and switching is done by changing the value of a single register.

4.3.4 Algorithms for Page Replacement

As noted above, the main operating system activity with regard to paging is deciding what pages to evict in order to make space for new pages that are read from disk.

FIFO is bad and leads to anomalous behavior

The simplest algorithm is FIFO: throw out the pages in the order in which they were brought in. You can guess that this is a bad idea because it is oblivious of the program's behavior, and doesn't care which pages are accessed a lot and which are not. Nevertheless, it is used in Windows 2000.

In fact, it is even worse. It turns out that with FIFO replacement there are cases when adding frames to a process causes *more* page faults rather than less page faults — an effect known as Belady's anomaly. While this only happens in pathological cases, it is unsettling. It is the result of the fact that the set of pages maintained by the algorithm when equipped with n frames is not necessarily a superset of the set of pages maintained when only $n-1$ frames are available.

In algorithms that rely on usage, such as those described below, the set of pages kept in n frames *is* a superset of the set that would be kept with $n-1$ frames, and therefore Belady's anomaly does not happen.

The ideal is to know the future

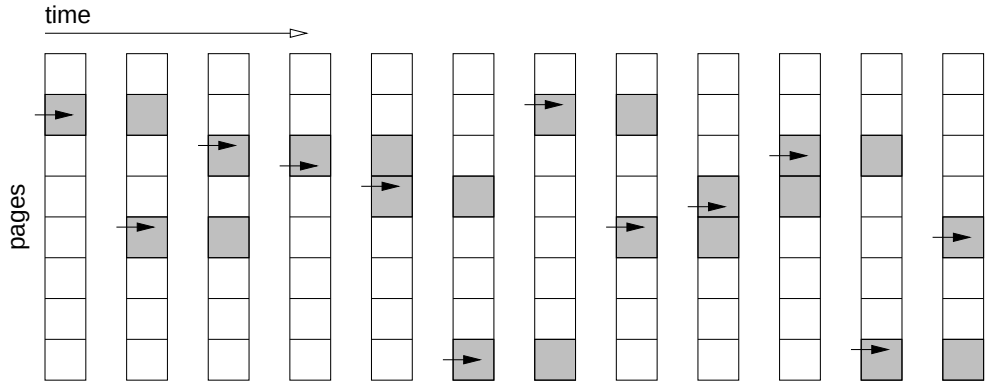
At the other extreme, the best possible algorithm is one that knows the future. This enables it to know which pages will not be used any more, and select them for replacement. Alternatively, if no such pages exist, the know-all algorithm will be able to select the page that will not be needed for the longest time. This delays the unavoidable page fault as much as possible, and thus reduces the total number of page faults.

Regrettably, such an optimal algorithm cannot be implemented. But if we do not know the future, we can at least leverage our knowledge of the past, using the principle of locality.

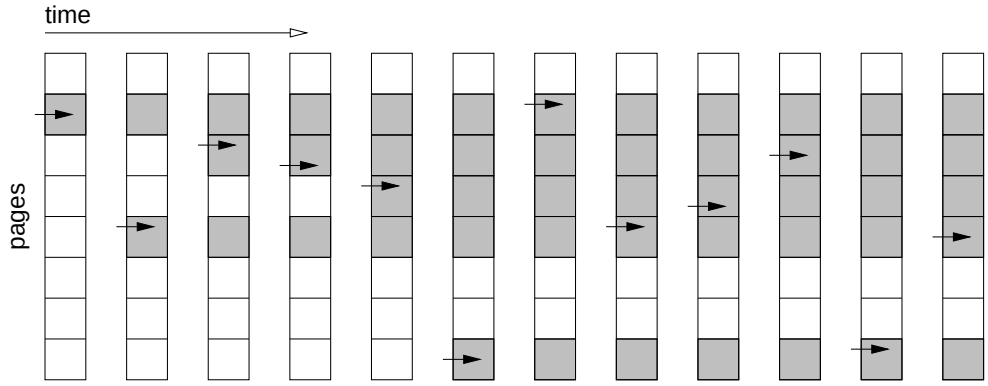
The alternative is to have each process's working set in memory

The most influential concept with regard to paging is the *working set*. The working set of a process is a dynamically changing set of pages. At any given moment, it includes the pages accessed in the last Δ instructions; Δ is called the *window size* and is a parameter of the definition.

The importance of the working set concept is that it captures the relationship between paging and the principle of locality. If the window size is too small, then the working set changes all the time. This is illustrated in this figure, where each access is denoted by an arrow, $\Delta = 2$, and the pages in the working set are shaded:



But with the right window size, the set becomes static: every additional instruction accesses a page that is already in the set. Thus the set comes to represent that part of memory in which the accesses are localized. For the sequence shown above, this requires $\Delta = 6$:



Knowing the working set for each process leads to two useful items of information:

- How many pages each process needs (the *resident set size*), and
- Which pages are not in the working set and can therefore be evicted.

However, keeping track of the working set is not realistically possible. And to be effective it depends on setting the window size correctly.

Evicting the least-recently used page approximates the working set

The reason for having the window parameter in the definition of the working set is that memory usage patterns change over time. We want the working set to reflect the current usage, not the pages that were used a long time ago. This insight leads to the LRU page replacement algorithm: when a page has to be evicted, pick the one that was least recently used (or in other words, was used farthest back in the past).

Note that LRU automatically also defines the resident set size for each process. This is so because *all* the pages in the system are considered as potential victims for eviction, not only pages belonging to the faulting process. Processes that use few

pages will be left with only those pages, and lose pages that were used in the more distant past. Processes that use more pages will have larger resident sets, because their pages will never be the least recently used.

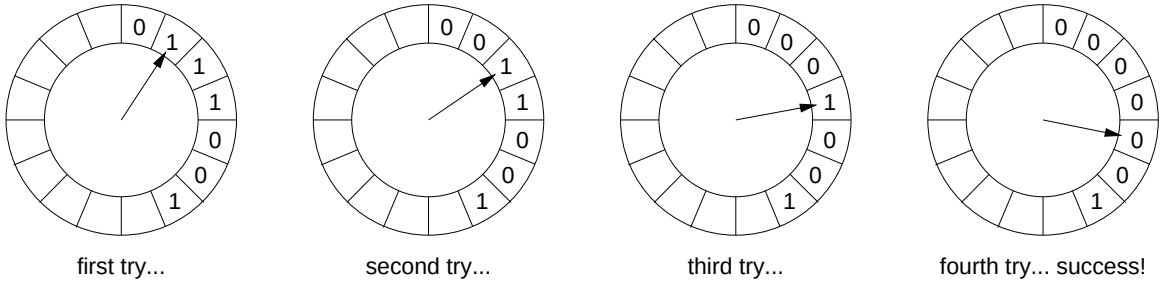
Regrettably, the LRU algorithm cannot be implemented in practice: it requires the system to know about the order in which all pages have been referenced. We need a level of simplification, and some hardware support.

LRU can be approximated by the clock algorithm

Most computers only provide minimal hardware support for page replacement. This is done in the form of *used bits*. Every frame of physical memory is associated with a single bit. This bit is set automatically by the hardware whenever the page mapped to this frame is accessed. It can be reset by the operating system.

The way to use these bits is as follows. Initially, all bits are set to zero. As pages are accessed, their bits are set to 1 by the hardware. When the operating system needs a frame, it scans all the pages in sequence. If it finds a page with its used bit still 0, it means that this page has not been accessed for some time. Therefore this page is a good candidate for eviction. If it finds a page with its used bit set to 1, it means that this page has been accessed recently. The operating system then resets the bit to 0, but does not evict the page. Instead, it gives the page a *second chance*. If the bit stays 0 until this page comes up again for inspection, the page will be evicted then. But if the page is accessed continuously, its bit will be set already when the operating system looks at it again. Consequently pages that are in active use will not be evicted.

It should be noted that the operating system scans the pages in a cyclic manner, always starting from where it left off last time. This is why the algorithm is called the "clock" algorithm: it can be envisioned as arranging the pages in a circle, with a hand pointing at the current one. When a frame is needed, the hand is moved one page at a time, setting the used bits to 0. When it finds a page whose bit is already 0, it stops and the page is evicted.



Exercise 110 *Is it possible to implement the clock algorithm without hardware support for used bits? Hint: remember that the operating system can turn on and off the present bit for each page.*

Using more than one bit can improve performance

The clock algorithm uses only one bit of information: whether the page was accessed since the last time the operating system looked at it. A possible improvement is for the operating system to keep a record of how the value of this bit changes over time. This allows it to differentiate pages that were not accessed a long time from those that were not accessed just now, and provides a better approximation of LRU.

Another improvement is to look at the *modify bit* too. Then unmodified ("clean") pages can be evicted in preference over modified ("dirty") pages. This saves the overhead of writing the modified pages to disk.

The algorithm can be applied locally or globally

So far, we have considered all the physical memory frames as candidates for page eviction. This approach is called *global paging*.

The alternative is *local paging*. When a certain process suffers a page fault, we only consider the frames allocated to that process, and choose one of the pages belonging to that process for eviction. We don't evict a page belonging to one process and give the freed frame to another. The advantage of this scheme is isolation: a process that needs a lot of memory is prevented from monopolizing multiple frames at the expense of other processes.

The main consequence of local paging is that it divorces the replacement algorithm from the issue of determining the resident set size. Each process has a static set of frames at its disposal, and its paging activity is limited to this set. The operating system then needs a separate mechanism to decide upon the appropriate resident set size for each process; for example, if a process does a lot of paging, its allocation of frames can be increased, but only subject to verification that this does not adversely affect other processes.

Regardless of algorithm, it is advisable to maintain a certain level of free frames

Another problem with paging is that the evicted page may be *dirty*, meaning that it was modified since being read off the disk. Dirty pages have to be written to disk when they are evicted, leading to a large delay in the service of page faults (first write one page, then read another). Service would be faster if the evicted page was clean, meaning that it was not modified and therefore can be evicted at once.

This problem can be solved by maintaining a certain level of free frames, or clean pages. Whenever the number of free frames falls below the predefined level, the operating system initiates the writing of pages to disk, in anticipation of future page faults. When the page faults indeed occur, free frames are available to serve them.

All algorithms may suffer from thrashing

All the page replacement algorithms cannot solve one basic problem. This problem is that if the sum of the sizes of the working sets of all the processes is larger than the size of the physical memory, all the pages in all the working sets cannot be in memory at the same time. Therefore every page fault will necessarily evict a page that is in some process's working set. By definition, this page will be accessed again soon, leading to another page fault. The system will therefore enter a state in which page faults are frequent, and no process manages to make any progress. This is called *thrashing*.

Thrashing is an important example of a threshold effect, where positive feedback causes system peformance to collapse: once it starts things deteriorate sharply. It is therefore imperative to employ mechanisms to prevent this and keep the system stable.

Using local paging reduces the effect of thrashing, because processes don't automatically steal pages from each other. But the only real solution to thrashing is to reduce the memory pressure by reducing the *multiprogramming level*. This means that we will have less processes in the system, so each will be able to get more frames for its pages. It is accomplished by swapping some processes out to disk, as described in Section 4.4 below.

Exercise 111 *Can the operating system identify the fact that thrashing is occurring? how?*

To read more: There is extensive literature on page replacement algorithms. A recent concise survey by Denning tells the story of locality [6]. Working sets were introduced by Denning in the late 1960's and developed till the 1980's [4, 7, 5]. A somewhat hard-to-read early work, noted for introducing the notion of an optimal algorithm and identifying the importance of use bits, is Belady's report of research at IBM [2]. The notion of stack algorithms was introduced by Mattson et al. [12].

4.3.5 Disk Space Allocation

One final detail we did not address is how the system allocates disk space to store pages that are paged out.

A simple solution, used in early Unix systems, is to set aside a partition of the disk for this purpose. Thus disk space was divided into two main parts: one used for paging, and the other for file systems. The problem with this approach is that it is inflexible. For example, if the system runs out of space for pages, but still has a lot of space for files, it cannot use this space.

An alternative, used e.g. in Windows 2000, is to use a paging file. This is a large pre-allocated file, that is used by the memory manager to store paged out pages. Pre-allocating a certain size ensures that the memory allocator will have at least that much space for paging. But if needed, and disk space is available, this file can grow

beyond its initial allocation. A maximal size can be set to prevent all disk space from being used for paging, leaving none for the file system.

4.4 Swapping

In order to guarantee the responsiveness of a computer system, processes must be preemptable, so as to allow new processes to run. But what if there is not enough memory to go around? This can also happen with paging, as identified by excessive thrashing.

Swapping is an extreme form of preemption

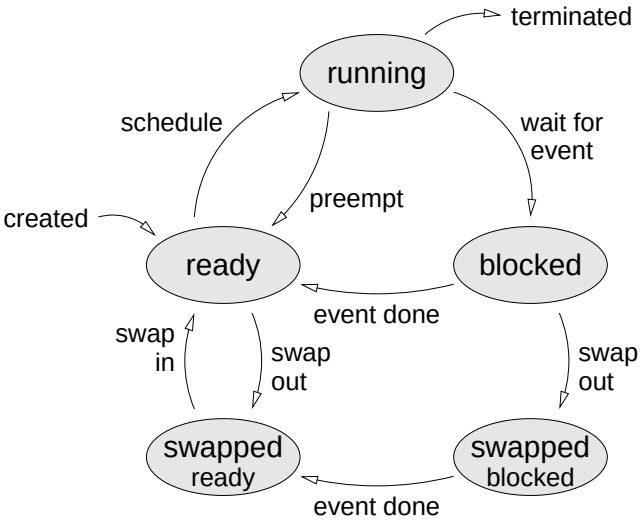
The solution is that when a process is preempted, its memory is also preempted. The contents of the memory are saved on secondary storage, typically a disk, and the primary memory is re-allocated to another process. It is then said that the process has been *swapped out*. When the system decides that the process should run again, its address space is *swapped in* and loaded into primary memory.

The decisions regarding swapping are called long-term or medium-term scheduling, to distinguish them from the decisions about scheduling memory-resident jobs. The criteria for these decisions are

- Fairness: processes that have accumulated a lot of CPU usage may be swapped out for a while to give other processes a chance.
- Creating a good job mix: jobs that compete with each other will be swapped out more than jobs that complement each other. For example, if there are two compute-bound jobs and one I/O-bound job, it makes sense to swap out the compute-bound jobs alternately, and leave the I/O-bound job memory resident.

"Swapped out" is a new process state

Swapped out processes cannot run, even if they are "ready". Thus adding swapping to a system enriches the process state graph:



Exercise 112 *Are all the transitions in the graph equivalent, or are some of them "heavier" (in the sense that they take considerably more time) ?*

And there are the usual sordid details

Paging and swapping introduce another resource that has to be managed: the disk space used to back up memory pages. This has to be allocated to processes, and a mapping of the pages into the allocated space has to be maintained. In principle, the methods described above can be used: either use a contiguous allocation with direct mapping, or divide the space into blocks and maintain an explicit map.

One difference is the desire to use large contiguous blocks in order to achieve higher disk efficiency (that is, less seek operations). An approach used in some Unix systems is to allocate swap space in blocks that are some power-of-two pages. Thus small processes get a block of say 16 pages, larger processes get a block of 16 pages followed by a block of 32 pages, and so on up to some maximal size.

4.5 Summary

Memory management provides one of the best examples in each of the following.

Abstractions

Virtual memory is one of the main abstractions supported by the operating system. As far as applications are concerned, they have large contiguous stretches of address space at their disposal. It is up to the operating system to implement this abstraction using a combination of limited, fragmented primary memory, and swap space on a slow disk.

Resource management

Memory management includes many themes related to resource management.

One is the classical algorithms for the allocation of contiguous resources — in our case, contiguous addresses in memory. These include First Fit, Best Fit, Next Fit, and the Buddy allocation scheme.

Another is the realization that it is better to break the resources into small fixed-size pieces — in this case, pages — rather than trying to allocate contiguous stretches. This is based on a level of indirection providing an associative mapping to the various pieces. With the use of such pieces comes the issue of how to free them. The most popular algorithm is LRU.

Finally, there is the distinction between implicit allocation (as in demand paging) and explicit allocation (as is needed when paging is done locally within the allocation of each process).

Needless to say, these themes are more general than just memory management.

Workload issues

Workloads typically display a high degree of locality. Without it, the use of paging to disk in order to implement virtual memory would simply not work. Locality allows the cost of expensive operations such as access to disk to be amortized across multiple memory accesses, and it ensures that these costly operations will be rare.

Hardware support

Memory management is probably where hardware support to the operating system is most prominent. Specifically, hardware address translation as part of each access is a prerequisite for paging. There are also various additions such as support for used and modify bits for each page.

Bibliography

[1] O. Babaoglu and W. Joy, "*Converting a swap-based system to do paging in an architecture lacking page-referenced bits*". In 8th *Symp. Operating Systems Principles*, pp. 78–86, Dec 1981.

[2] L. A. Belady, "*A study of replacement algorithms for a virtual-storage computer*". *IBM Syst. J.* **5(2)**, pp. 78–101, 1966.

[3] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.

[4] P. J. Denning, "*The working set model for program behavior*". *Comm. ACM* **11(5)**, pp. 323–333, May 1968.

[5] P. J. Denning, "*Working sets past and present*". *IEEE Trans. Softw. Eng.* **SE-6(1)**, pp. 64–84, Jan 1980.

[6] P. J. Denning, "*The locality principle*". *Comm. ACM* **48(7)**, pp. 19–24, Jul 2005.

[7] P. J. Denning and S. C. Schwartz, "*Properties of the working-set model*". *Comm. ACM* **15(3)**, pp. 191–198, Mar 1972.

[8] G. A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. MIT Press, 1992.

[9] B. Jacob and T. Mudge, "*Virtual memory: Issues of implementation*". *Computer* **31(6)**, pp. 33–43, Jun 1998.

[10] B. L. Jacob and T. N. Mudge, "*A look at several memory management units, TLB-refill mechanisms, and page table organizations*". In 8th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 295–306, Oct 1998.

[11] D. E. Knuth, *The Art of Computer Programming. Vol 1: Fundamental Algorithms*. Addison Wesley, 2nd ed., 1973.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "*Evaluation techniques for storage hierarchies*". *IBM Syst. J.* **9(2)**, pp. 78–117, 1970.

[13] W. Stallings, *Operating Systems*. Prentice-Hall, 2nd ed., 1995.

Chapter 5

File Systems

Support for files is an abstraction provided by the operating system, and does not entail any real resource management. If disk space is available, it is used, else the service cannot be provided. There is no room for manipulation as in scheduling (by timesharing) or in memory management (by paging and swapping). However there is some scope for various ways to organize the data when implementing the file abstraction.

5.1 What is a File?

The most important characteristics are being named and persistent

A file is a named persistent sequential (structured) data repository.

The attributes of being named and being persistent go hand in hand. The idea is that files can be used to store data for long periods of time, and specifically, for longer than the runtimes of the processes that create them. Thus one process can create a file, and another process may re-access the file using its name.

The attribute of being sequential means that data within the file can be identified by its offset from the beginning of the file.

As for structure, in Unix there simply is no structure. There is only one type of file, which is a linear sequence of bytes. Any interpretation of these bytes is left to the application that is using the file. Of course, some files are used by the operating system itself. In such cases, the operating system is the application, and it imposes some structure on the data it keeps in the file. A prime example is directories, where the data being stored is file system data; this is discussed in more detail below. Another example is executable files. In this case the structure is created by a compiler, based on a standard format that is also understood by the operating system.

Other systems may support more structure in their files. Examples include:

- IBM mainframes: the operating system supports lots of options, including fixed

or variable-size records, indexes, etc. Thus support for higher-level operations is possible, including "read the next record" or "read the record with key X".

- Macintosh OS: executables have two "forks": one containing code, the other labels used in the user interface. Users can change labels appearing on buttons in the application's graphical user interface (e.g. to support different languages) without access to the source code.
- Windows NTFS: files are considered as a set of attribute/value pairs. In particular, each file has an "unnamed data attribute" that contains its contents. But it can also have multiple additional named data attributes, e.g. to store the file's icon or some other file-specific information. Files also have a host of system-defined attributes.

The extreme in terms of structure is a full fledged database. As the organization and use of databases is quite different from that encountered in general-purpose systems, it is common to use a special database management system (DBMS) in lieu of the operating system. We shall not discuss database technology here.

The most important attributes are permissions and data layout

Given that files are abstract objects, one can ask what attributes are kept about them. While there are differences among systems, the main ones are

`Owner:` the user who owns this file.

`Permissions:` who is allowed to access this file.

`Modification time:` when this file was last modified.

`Size:` how many bytes of data are there.

`Data location:` where on the disk the file's data is stored.

As this is data *about* the file maintained by the operating system, rather than user data that is stored *within* the file, it is sometimes referred to as the file's *metadata*.

Exercise 113 *What are these data items useful for?*

Exercise 114 *And how about the file's name? Why is it not here?*

Exercise 115 *The Unix `stat` system call provides information about files (see man `stat`). Why is the location of the data on the disk not provided?*

The most important operations are reading and writing

Given that files are abstract objects, one can also ask what operations are supported on these objects. In a nutshell, the main ones are

`Open`: gain access to a file.

`Close`: relinquish access to a file.

`Read`: read data from a file, usually from the current position.

`Write`: write data to a file, usually at the current position.

`Append`: add data at the end of a file.

`Seek`: move to a specific position in a file.

`Rewind`: return to the beginning of the file.

`Set attributes`: e.g. to change the access permissions.

`Rename`: change the name of the file.

Exercise 116 *Is this set of operations minimal, or can some of them be implemented using others?*

5.2 File Naming

As noted above, an important characteristic of files is that they have names. These are typically organized in directories. But internally, files (and directories) are represented by a data structure containing the attributes listed above. Naming is actually a mapping from names — that is, strings given by human users — to these internal representations. In Unix, this data structure is called an inode, and we'll use this name in what follows as shorthand for "a file or directory's internal representation".

5.2.1 Directories

Directories provide a hierarchical structure

The name space for files can be *flat*, meaning that all files are listed in one long list. This has two disadvantages:

- There can be only one file with a given name in the system. For example, it is not possible for different users to have distinct files named "ex1.c".
- The list can be very long and unorganized.

The alternative is to use a hierarchical structure of directories. This structure reflects organization and logical relations: for example, each user will have his own private directory. In addition, files with the same name may appear in different directories.

Using directories also creates an opportunity for supporting collective operations on sets of related files: for example, it is possible to delete all the files in a given directory.

With a hierarchical structure files are identified by the path from the root, through several levels of directories, to the file. Each directory contains a list of those files and subdirectories that are contained in it. Technically, the directory maps the names of these files and subdirectories to the internal entities known to the file systems — that is, to their inodes. In fact, directories are just like files, except that the data stored in them is not user data but rather file system data, namely this mapping. The hierarchy is created by having names that refer to subdirectories.

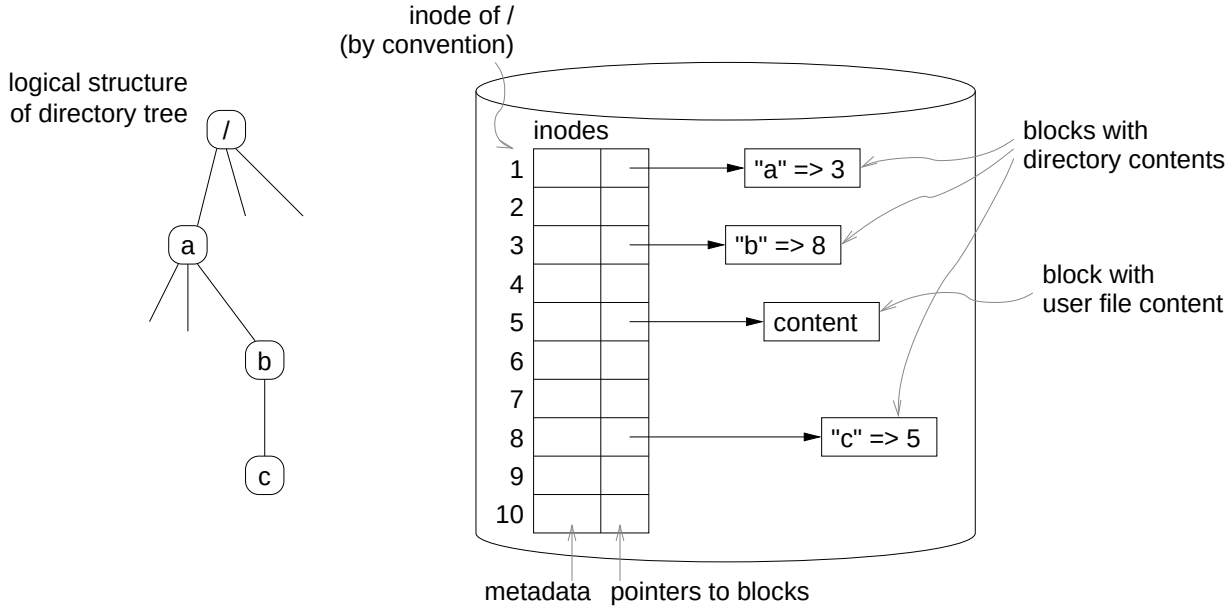
Exercise 117 *What happens if we create a cycle of directories? What is a simple way to prevent this?*

Names are mapped to inodes recursively

The system identifies files by their full *path*, that is, the file's name concatenated to the list of directories traversed to get to it. This always starts from a distinguished *root directory*.

Exercise 118 *So how does the system find the root directory itself?*

Assume a full path `/a/b/c` is given. To find this file, we need to perform the following:



1. Read the inode of the root / (assume it is the first one in the list of inodes), and use it to find the disk blocks storing its contents, i.e. the list of subdirectories and files in it.

2. Read these blocks and search for the entry `a`. This entry will map `/a` to its inode. Assume this is inode number 3.
3. Read inode 3 (which represents `/a`), and use it to find its blocks.
4. Read the blocks and search for subdirectory `b`. Assume it is mapped to inode 8.
5. Read inode 8, which we now know to represent `/a/b`.
6. Read the blocks of `/a/b`, and search for entry `c`. This will provide the inode of `/a/b/c` that we are looking for, which contains the list of blocks that hold this file's contents.
7. To actually gain access to `/a/b/c`, we need to read its inode and verify that the access permissions are appropriate. In fact, this should be done in each of the steps involved with reading inodes, to verify that the user is allowed to see this data. This is discussed further in Chapter 7.

Note that there are 2 disk accesses per element in the path: one for the inode, and the other for the contents.

A possible shortcut is to start from the current directory (also called the working directory) rather than from the root. This obviously requires the inode of the current directory to be available. In Unix, the default current directory when a user logs onto the system is that user's home directory.

Exercise 119 *Are there situations in which the number of disk accesses when parsing a file name is different from two per path element?*

Exercise 120 *The contents of a directory is the mapping from names (strings of characters) to inode (e.g. integers interpreted as an index into a table). How would you implement this? Recall that most file names are short, but you need to handle arbitrarily long names efficiently. Also, you need to handle dynamic insertions and deletions from the directory.*

Exercise 121 *In UNIX, the contents of a directory is simply a list of mappings from name to inode. An alternative is to use a hash table. What are the advantages and disadvantages of such a design?*

Note that the process described above may also fail. For example, the requested name may not be listed in the directory. Alternatively, the name may be there but the user may not have the required permission to access the file. If something like this happens, the system call that is trying to gain access to the file will fail. It is up to the application that called this system call to print en error message or do something else.

5.2.2 Links

Files can have more than one name

The mapping from a user-defined name string to an inode) is called a link. In principle, it is possible to have multiple strings mapped to the same inode. This causes the file to have multiple names. And if the names appear in different directories, it will have multiple different paths.

Exercise 122 *Why in the world would you want a file to have multiple names?*

Exercise 123 *Are there any disadvantages to allowing files to have multiple names? Hint: think about "..".*

Special care must be taken when deleting (unlinking) a file. If it has multiple links, and one is being removed, we should not delete the file's contents — as they are still accessible using the other links. The inode therefore has a counter of how many links it has, and the data itself is only removed when this count hits zero.

Exercise 124 *An important operation on files is renaming them. Is this really an operation on the file? How it is implemented?*

Soft links are flexible but problematic

The links described above, implemented as mappings in a directory, are called hard links. An alternative is soft links: a directory entry that is actually an indirection, not a real link (in Windows systems, this is called a "shortcut"). This means that instead of mapping the name to an inode, the name is mapped to an alternative path. When a soft link is encountered in the process of parsing a path name, the current path is replaced by the one indicated in the link.

Exercise 125 *What are advantages and dangers of soft links?*

5.2.3 Alternatives for File Identification

The reason for giving files names is to make them findable. But when you have lots of files, accumulated over many years, you might forget the name you chose. And names are typically constrained to be quite short (if not by the system, then by your desire not to type too much).

It might be better to identify files by association

An alternative to names is to identify files by association. Specifically, you will probably think of files in the context of what you were working on at the time. So an association with this context will make the desired files easy to find.

One simple interface suggested to accomplish this is "lifestreams". With this, files are shown in sequence according to when they were used, with the newest ones at the front. The user can use the mouse to move backwards or forward in time and view different files [5].

A more sophisticated interface is the calendarial file access mechanism developed at Ricoh Research Center [7]. The interface is a calendar, with a box for each day, organized into weeks, months, and years. Each box contains information about scheduled activities in that day, e.g. meetings and deadlines. It also contains thumbnail images of the first pages of document files accessed on that day. Thus when looking for a file, you can find it by searching for the meeting in which it was discussed. Importantly, and based on the fact that Ricoh is a manufacturer of photocopiers and other office equipment, this includes all documents you worked with, not only those you viewed or edited on your computer. In a 3-year usage trial, 38% of accesses using the system were to documents only one week old, showing that in many cases users preferred this interface to the conventional one even for relatively fresh documents.

Or to just search

Another recent alternative is to use keyword search. This is based on the success of web search engines, that index billions of web pages, and provide a list of relevant pages when queried. In principle, the same can be done for files in a file system. The problem is how to rank the files and show the most relevant on top.

To read more: The opposition to using file names, and preferring search procedures, is promoted by Raskin, the creator of the Mac interface [11].

5.3 Access to File Data

The main function of a file system is to store data in files. But files are an abstraction which needs to be mapped to and implemented with the available hardware. This involves the allocation of disk blocks to files, or, viewed the other way, the mapping of files into the disk. It also involves the actual read and write operations, and how to optimize them. One important optimization is to avoid disk access altogether by caching data in memory.

As in other areas, there are many options and alternatives. But in the area of file systems, there is more data about actual usage patterns than in other areas. Such data is important for the design and evaluation of file systems, and using it ensures that the selected policies do indeed provide good performance.

Background: direct memory access (DMA)
Getting the data on or off the disk is only one side of the I/O operation. The other side is accessing the appropriate memory buffer. The question of how this is done depends on the hardware, and has a great impact on system performance.

If only the processor can access memory the only option is *programmed I/O*. This means that the CPU (running operating system code) accepts each word of data as it comes off the disk, and stores it at the required address. This has the obvious disadvantage of keeping the CPU busy throughout the duration of the I/O operation, so no overlap with other computation is possible.

In modern systems, components who need it typically have Direct Memory Access (DMA), so they do not have to go through the CPU (this is true for both disks and network interfaces). Thus when the operating system wants to perform an I/O operation, it only has to activate the disk controller, and tell it what to do. The operating system then blocks the process that requested this I/O operation, and frees the CPU to run another ready process. In the meantime, the disk controller positions the heads, accesses the disk, and transfers the data between the disk and the memory. When the transfer is complete, the disk controller interrupts the CPU. The operating system interrupt handler unblocks the waiting process, and puts it on the ready queue.

5.3.1 Data Access

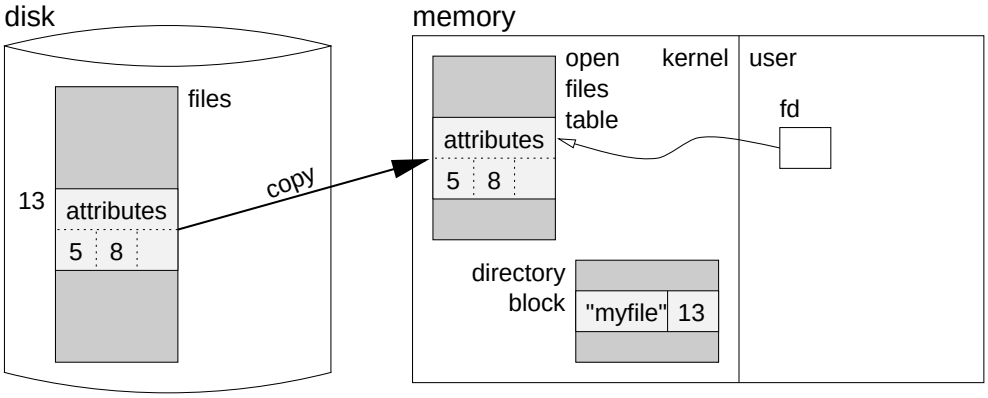
To use files, users use system calls that correspond to the operations listed in Section 5.1. This section explains how the operating system implements these operations.

Opening a file sets things up for future access

Most systems require files to be *opened* before they can be accessed. For example, using Unix-like notation, the process may perform the system call

```
fd=open("myfile",R)
```

This leads to the following sequence of actions:



1. The file system reads the current directory, and finds that "myfile" is represented internally by entry 13 in the list of files maintained on the disk.
2. Entry 13 from that data structure is read from the disk and copied into the kernel's open files table. This includes various file attributes, such as the file's owner and its access permissions, and a list of the file blocks.
3. The user's access rights are checked against the file's permissions to ascertain that reading (`R`) is allowed.
4. The user's variable `fd` (for "file descriptor") is made to point to the allocated entry in the open files table. This serves as a handle to tell the system what file the user is trying to access in subsequent `read` or `write` system calls, and to prove that permission to access this file has been obtained, but without letting user code obtain actual access to kernel data.

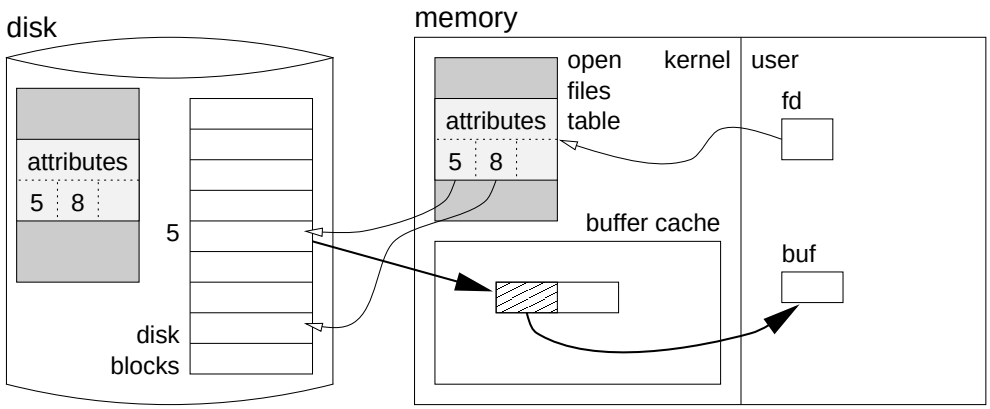
The reason for opening files is that the above operations may be quite time consuming, as they may involve a number of disk accesses to map the file name to its inode and to obtain the file's attributes. Thus it is desirable to perform them once at the outset, rather then doing them again and again for each access. `open` returns a handle to the open file, in the form of the file descriptor, which can then be used to access the file many times.

Access to disk blocks uses the buffer cache

Now the process performs the system call

```
read(fd,buf,100)
```

which means that 100 bytes should be read from the file indicated by `fd` into the memory buffer `buf`.



The argument `fd` identifies the open file by pointing into the kernel's open files table. Using it, the system gains access to the list of blocks that contain the file's data. In our example, it turns out that the first data block is disk block number 5. The file

system therefore reads disk block number 5 into its *buffer cache*. This is a region of memory where disk blocks are cached. As this takes a long time (on the order of milliseconds), the operating system typically blocks the operation waiting for it to complete, and does something else in the meanwhile, like scheduling another process to run.

When the disk interrupts the CPU to signal that the data transfer has completed, handling of the `read` system call resumes. In particular, the desired range of 100 bytes is copied into the user's memory at the address indicated by `buf`. If additional bytes from this block will be requested later, the block will be found in the buffer cache, saving the overhead of an additional disk access.

Exercise 126 *Is it possible to read the desired block directly into the user's buffer, and save the overhead of copying?*

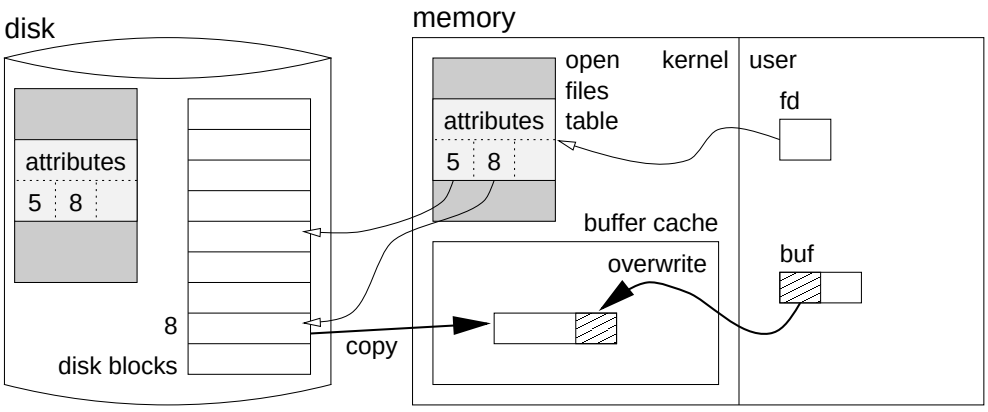
Writing may require new blocks to be allocated

Now suppose the process wants to write a few bytes. Let's assume we want to write 100 bytes, starting with byte 2000 in the file. This will be expressed by the pair of system calls

```
seek(fd,2000)
write(fd,buf,100)
```

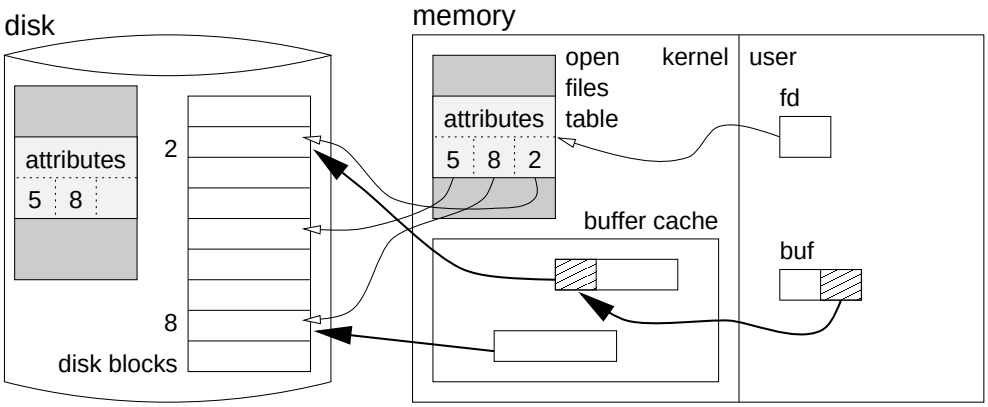
Let's also assume that each disk block is 1024 bytes. Therefore the data we want to write spans the end of the second block to the beginning of the third block.

The problem is that disk accesses are done in fixed blocks, and we only want to write part of such a block. Therefore the full block must first be read into the buffer cache. Then the part being written is modified by overwriting it with the new data. In our example, this is done with the second block of the file, which happens to be block 8 on the disk.



The rest of the data should go into the third block, but the file currently only has two blocks. Therefore a third block must be allocated from the pool of free blocks. Let's

assume that block number 2 on the disk was free, and that this block was allocated. As this is a new block, there is no need to read it from the disk before modifying it — we just allocate a block in the buffer cache, prescribe that it now represents block number 2, and copy the requested data to it. Finally, the modified blocks are written back to the disk.



Note that the copy of the file's inode was also modified, to reflect the allocation of a new block. Therefore this too must be copied back to the disk. Likewise, the data structure used to keep track of free blocks needs to be updated on the disk as well.

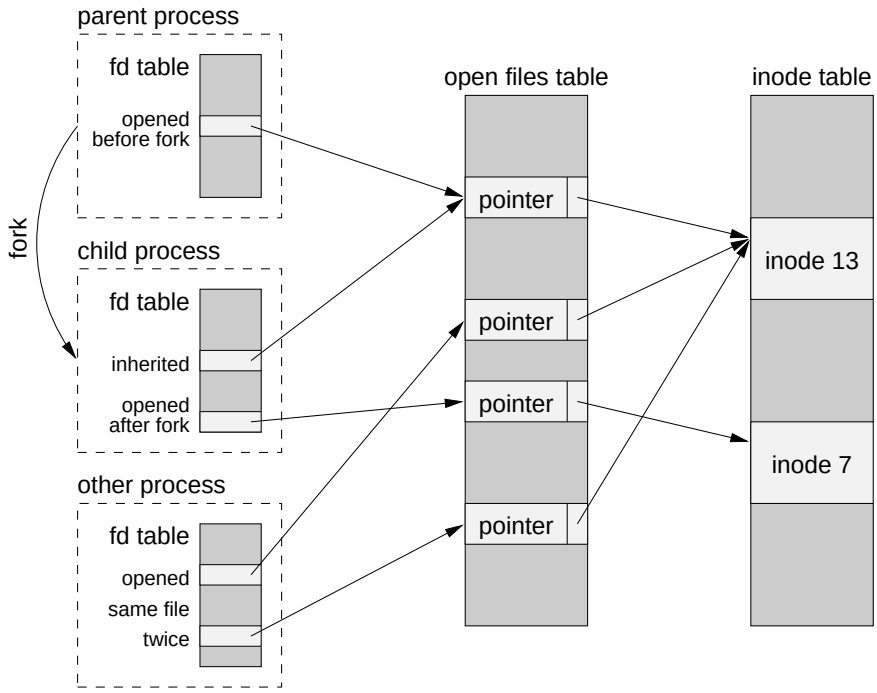
The location in the file is maintained by the system

You might have noticed that the `read` system call provides a buffer address for placing the data in the user's memory, but does not indicate the offset in the file from which the data should be taken. This reflects common usage where files are accessed sequentially, and each access simply continues where the previous one ended. The operating system maintains the current offset into the file (sometimes called the *file pointer*), and updates it at the end of each operation.

If random access is required, the process can set the file pointer to any desired value by using the `seek` system call (random here means arbitrary, not indeterminate!).

Exercise 127 *What happens (or should happen) if you seek beyond the current end of the file, and then write some data?*

> **Example: Unix uses three tables**
> The above is actually a somewhat simplified generic description. So let's look at a few more details as implemented in classic Unix systems. This implementation uses a sequence of 3 tables to hold data about open files.
>
> The first is the in-core inode table, which contains the inodes of open files (recall that an inode is the internal structure used to represent files, and contains the file's metadata as outlined in Section 5.1). Each file may appear at most once in this table. The data is essentially the same as in the on-disk inode, i.e. general information about the file such as its owner, permissions, modification time, and listing of disk blocks.



The second table is the open files table. An entry in this table is allocated every time a file is opened. These entries contain three main pieces of data:

- An indication of whether the file was opened for reading or for writing
- An offset (sometimes called the "file pointer") storing the current position within the file.
- A pointer to the file's inode.

A file may be opened multiple times by the same process or by different processes, so there can be multiple open file entries pointing to the same inode. The data here is used by `read` and `write` system calls, first to make sure that the access is permitted, and then to find (and update) the offset at which it should be performed.

The third table is the file descriptors table. There is a separate file descriptor table for each process in the system. When a file is opened, the system finds an unused slot in the opening process's file descriptor table, and uses it to store a pointer to the new entry it creates in the open files table. The index of this slot is the return value of the `open` system call, and serves as a handle to get to the file. The first three indices are by convention pre-allocated to standard input, standard output, and standard error.

Exercise 128 *Why do we need the file descriptors table? Couldn't* `open` *just return the index into the open files table?*

Another important use for the file descriptors table is that it allows file pointers to be shared. For example, this is useful in writing a log file that is shared by several

processes. If each process has its own file pointer, there is a danger that one process will overwrite log entries written by another process. But if the file pointer is shared, each new log entry will indeed be added to the end of the file. To achieve sharing, file descriptors are inherited across forks. Thus if a process opens a file and then forks, the child process will also have a file descriptor pointing to the same entry in the open files table. The two processes can then share the same file pointer by using these file descriptors. If either process opens a file *after* the fork, the associated file pointer is not shared.

Exercise 129 *Given that multiple file descriptors can point to the same open file entry, and multiple open file entries can point to the same inode, how are entries freed? Specifically, when a process closes a file descriptor, how can the system know whether it should free the open file entry and/or the inode?*

5.3.2 Caching and Prefetching

Disk I/O is substantially slower than processing, and the gap is growing: CPUs are becoming faster much faster than disks. This is why it makes sense to perform a context switch when waiting for I/O. It also means that some effort should be invested in making I/O faster.

Caching is instrumental in reducing disk accesses

As mentioned above, operating systems typically place a buffer cache between the disk and the user. All file operations pass through the buffer cache. The use of a buffer cache is required because disk access is performed in predefined blocks, that do not necessarily match the application's requests. However, there are a few additional important benefits:

- If an application requests only a small part of a block, the whole block has to be read anyway. By caching it (rather than throwing it away after satisfying the request) the operating system can later serve additional requests from the same block without any additional disk accesses. In this way small requests are aggregated and the disk access overhead is amortized rather than being duplicated.
- In some cases several processes may access the same disk block; examples include loading an executable file or reading from a database. If the blocks are cached when the first process reads them off the disk, subsequent accesses will hit them in the cache and not need to re-access the disk.
- A lot of data that is written to files is actually transient, and need not be kept for long periods of time. For example, an application may store some data in a temporary file, and then read it back and delete the file. If the file's blocks are initially stored in the buffer cache, rather than being written to the disk

immediately, it is possible that the file will be deleted while they are still there. In that case, there is no need to write them to the disk at all. The same holds for data that is overwritten after a short time.

- Alternatively, data that is not erased can be written to disk later (delayed write). The process need not be blocked to wait for the data to get to the disk. Instead the only overhead is just to copy the data to the buffer cache.

Data about file system usage in working Unix 4.2 BSD systems was collected and analyzed by Ousterhout and his students [9, 2]. They found that a suitably-sized buffer cache can eliminate 65–90% of the disk accesses. This is attributed to the reasons listed above. Specifically, the analysis showed that 20–30% of newly-written information is deleted or overwritten within 30 seconds, and 50% is deleted or overwritten within 5 minutes. In addition, about 2/3 of all the data transferred was in sequential access of files. In files that were opened for reading or writing, but not both, 91–98% of the accesses were sequential. In files that were opened for both reading and writing, this dropped to 19–35%.

The downside of caching disk blocks is that the system becomes more vulnerable to data loss in case of a system crash. Therefore it is necessary to periodically flush all modified disk blocks to the disk, thus reducing the risk. This operation is called *disk synchronization*.

Exercise 130 *It may happen that a block has to be read, but there is no space for it in the buffer cache, and another block has to be evicted (as in paging memory). Which block should be chosen? Hint: LRU is often used. Why is this possible for files, but not for paging?*

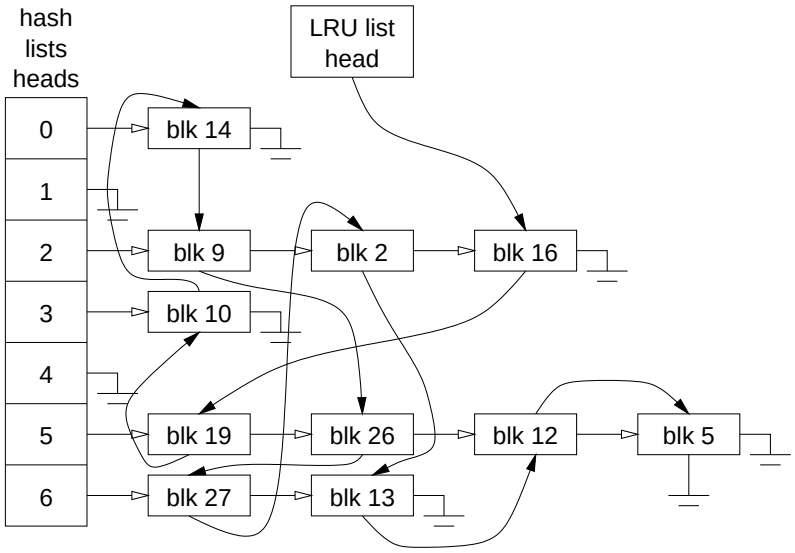
Example: the Unix Buffer Cache Data Structure

A cache, by definition, is associative: blocks are stored randomly (at least in a fully associative cache), but need to be accessed by their address. In a hardware cache the search must be performed in hardware, so costs dictate a sharp reduction in associativity (leading, in turn, to more conflicts and reduced utilization). But the buffer cache is maintained by the operating system, so this is not needed.

The data structure used by Unix[1] to implement the buffer cache is somewhat involved, because two separate access scenarios need to be supported. First, data blocks need to be accessed according to their disk address. This associative mode is implemented by hashing the disk address, and linking the data blocks according to the hash key. Second, blocks need to be listed according to the time of their last access in order to implement the LRU replacement algorithm. This is implemented by keeping all blocks on a global LRU list. Thus each block is actually part of two linked lists: one based on its hashed address, and the other based on the global access order.

For example, the following simplified picture shows a possible linking of 11 data blocks, where the hashing is done by taking the block number modulo 7.

[1] Strictly speaking, this description relates to older versions of the system. Modern Unix variants typically combine the buffer cache with the virtual memory mechanisms.



Prefetching can overlap I/O with computation

The fact that most of the data transferred is in sequential access implies locality. This suggests that if the beginning of a block is accessed, the rest will also be accessed, and therefore the block should be cached. But it also suggests that the operating system can guess that the *next* block will also be accessed. The operating system can therefore prepare the next block *even before is is requested*. This is called prefetching.

Prefetching does not reduce the number of disk accesses. In fact, it runs the risk of increasing the number of disk accesses, for two reasons: first, it may happen that the guess was wrong and the process does not make any accesses to the prefetched block, and second, reading the prefetched block into the buffer cache may displace another block that will be accessed in the future. However, it does have the potential to significantly reduce the time of I/O operations as observed by the process. This is due to the fact that the I/O was started ahead of time, and may even complete by the time it is requested. Thus prefetching overlaps I/O with the computation of the same process. It is similar to asynchronous I/O, but does not require any specific coding by the programmer.

Exercise 131 *Asynchronous I/O allows a process to continue with its computation, rather than being blocked while waiting for the I/O to complete. What programming interfaces are needed to support this?*

5.3.3 Memory-Mapped Files

An alternative to the whole mechanism described above, which is used by many modern systems, is to map files to memory. This relies on the analogy between handling file access and handling virtual memory with paging.

File layout is similar to virtual memory layout

As described in the previous pages, implementing the file abstraction has the following attributes:

- Files are continuous sequences of data.
- They are broken into fixed-size blocks.
- These blocks are mapped to arbitrary locations in the disk, and the mapping is stored in a table.

But this corresponds directly to virtual memory, where continuous logical memory segments are broken into pages and mapped to memory frames using a page table. Instead of implementing duplicated mechanisms, it is better to use one to implement the other.

Mapping files to memory uses paging to implement I/O

The idea is simple: when a file is opened, create a memory segment and define the file as the disk backup for this segment. In principle, this involves little more than the allocation of a page table, and initializing all the entries to invalid (that is, the data is on disk and not in memory).

A `read` or `write` system call is then reduced to just copying the data from or to this mapped memory segment. If the data is not already there, this will cause a page fault. The page fault handler will use the inode to find the actual data, and transfer it from the disk. The system call will then be able to proceed with copying it.

Alternatively, the data can simply be accessed at the place where it is mapped, without copying it elsewhere in the address space. This is especially natural in applications like text editors, where the whole contents of the file can be mapped to a memory segment and edited according to the user's instructions.

For example, the following pseudo-code gives the flavor of mapping a file and changing a small part of it:

```
ptr=mmap("myfile",RW)
strncpy(ptr+2048,"new text",8)
```

The first line is the system call `mmap`, which maps the named file into the address space with read/write permissions. It returns a pointer to the beginning of the newly created segment. The second line over-writes 8 bytes at the beginning of the third block of the file (assuming 1024-byte blocks).

Memory mapped files are more efficient

An important benefit of using memory-mapped files is that this avoids the need to set aside some of the computer's physical memory for the buffer cache. Instead, the

buffered disk blocks reside in the address spaces of the processes that use them. The portion of the physical memory devoted to such blocks can change dynamically according to usage, by virtue of the paging mechanism.

Moreover, as noted above copying the data may be avoided. This not only avoids the overhead of the copying, but also completely eliminates the overhead of trapping into the system to perform the `read` or `write` system call. However, the overhead of disk access cannot be avoided — it is just done as a page fault rather than as a system call.

Exercise 132 *What happens if two distinct processes map the same file?*

To read more: See the man page for `mmap`.

5.4 Storing Files on Disk

The medium of choice for persistent storage is magnetic disks. This may change. In the past, it was tapes. In the future, it may be flash memory and optical disks such as DVDs. Each medium has its constraints and requires different optimizations. The discussion here is geared towards disks.

5.4.1 Mapping File Blocks

OK, so we know about accessing disk blocks. But how do we find the blocks that together constitute the file? And how do we find the right one if we want to access the file at a particular offset?

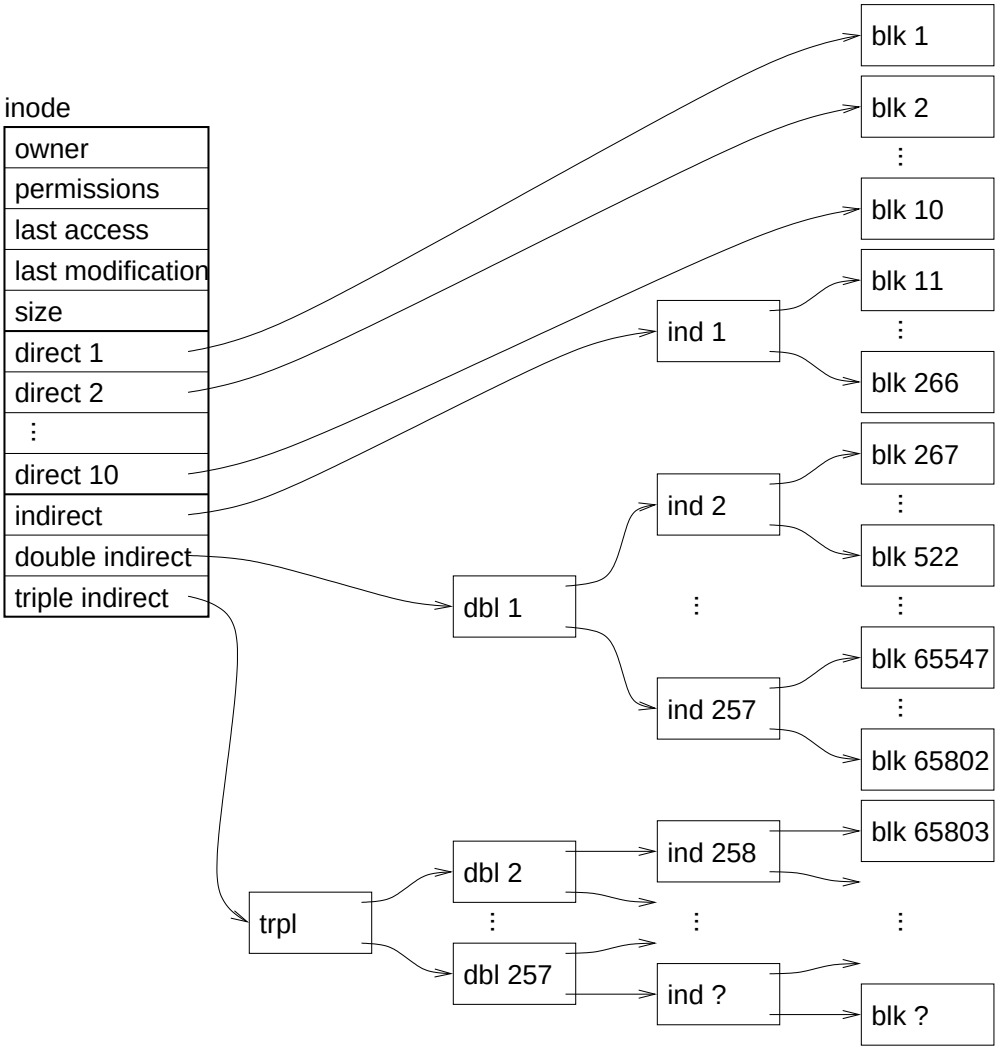
The Unix inode contains a hierarchical index

In Unix files are represented internally by a structure known as an inode. Inode stands for "index node", because apart from other file metadata, it also includes an index of disk blocks.

The index is arranged in a hierarchical manner. First, there are a few (e.g. 10) *direct* pointers, which list the first blocks of the file. Thus for small files all the necessary pointers are included in the inode, and once the inode is read into memory, they can all be found. As small files are much more common than large ones, this is efficient.

If the file is larger, so that the direct pointers are insufficient, the *indirect* pointer is used. This points to a whole block of additional direct pointers, which each point to a block of file data. The indirect block is only allocated if it is needed, i.e. if the file is bigger than 10 blocks. As an example, assume blocks are 1024 bytes (1 KB), and each pointer is 4 bytes. The 10 direct pointers then provide access to a maximum of 10 KB. The indirect block contains 256 additional pointers, for a total of 266 blocks (and 266 KB).

If the file is bigger than 266 blocks, the system resorts to using the *double* indirect pointer, which points to a whole block of indirect pointers, each of which point to an additional block of direct pointers. The double indirect block has 256 pointers to indirect blocks, so it represents a total of 65536 blocks. Using it, file sizes can grow to a bit over 64 MB. If even this is not enough, the *triple* indirect pointer is used. This points to a block of double indirect pointers.

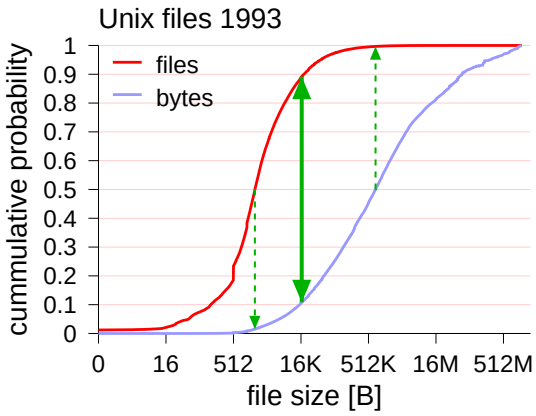


A nice property of this hierarchical structure is that the time needed to find a block is logarithmic in the file size. And note that due to the skewed structure, it is indeed logarithmic in the actual file size — not in the maximal supported file size. The extra levels are only used in large files, but avoided in small ones.

Exercise 133 *what file sizes are possible with the triple indirect block? what other constraints are there on file size?*

The Distribution of File Sizes

The distribution of file sizes is one of the examples of heavy-tailed distributions in computer workloads. The plot to the right shows the distribution of over 12 million files from over 1000 Unix file systems collected in 1993 [8]. Similar results are obtained for more modern file systems.

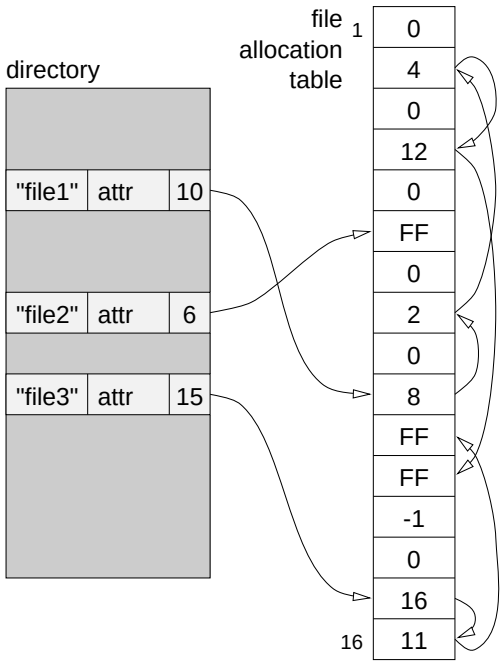


The distribution of files is the top line (this is a CDF, i.e. for each size x it shows the probability that a file will be no longer than x). For example, we can see that about 20% of the files are up to 512 bytes long. The bottom line is the distribution of bytes: for each file size x, it shows the probability that an arbitrary byte *belong to* a file no longer than x. For example, we can see that about 10% of the bytes belong to files that are up to 16 KB long.

The three vertical arrows allow us to characterize the distribution [4]. The middle one shows that this distribution has a "joint ratio" of 11/89. This means that the top 11% of the files are so big that together they account for 89% of the disk space. At the same time, the bottom 89% of the files are so small that together they account for only 11% of the disk space. The leftmost arrow shows that the bottom *half* of the files are so small that they only account for 1.5% of the disk space. The rightmost arrow shows that the other end of the distribution is even more extreme: half of the disk space is accounted for by only 0.3% of the files, which are each very big.

FAT uses a linked list

A different structure is used by FAT, the original DOS file system (which has the dubious distinction of having contributed to launching Bill Gates's career). FAT stands for "file allocation table", the main data structure used to allocate disk space. This table, which is kept at the beginning of the disk, contains an entry for each disk block (or, in more recent versions, each cluster of consecutive disk blocks that are allocated as a unit). Entries that constitute a file are linked to each other in a chain, with each entry holding the number of the next one. The last one is indicated by a special marking of all 1's (FF in the figure). Unused blocks are marked with 0, and bad blocks that should not be used also have a special marking (-1 in the figure).



Exercise 134 *Repeat Ex. 127 for the Unix inode and FAT structures: what happens if you seek beyond the current end of the file, and then write some data?*

Exercise 135 *What are the pros and cons of the Unix inode structure vs. the FAT structure? Hint: consider the distribution of file sizes shown above.*

> **FAT's Success and Legacy Problems**
> The FAT file system was originally designed for storing data on floppy disks. Two leading considerations where therefore simplicity and saving space. As a result file names were limited to the 8.3 format, where the name is no more than 8 characters, followed by an extension of 3 characters. In addition, the pointers were 2 bytes, so the table size was limited to 64K entries.
>
> The problem with this structure is that each table entry represents an allocation unit of disk space. For small disks it was possible to use an allocation unit of 512 bytes. In fact, this is OK for disks of up to $512 \times 64\text{K} = 32\text{MB}$. But when bigger disks became available, they had to be divided into the same 64K allocation units. As a result the allocation units grew considerably: for example, a 256MB disk was allocated in units of 4K. This led to inefficient disk space usage, because even small files had to allocate at least one unit. But the design was hard to change because so many systems and so much software were dependent on it.

5.4.2 Data Layout on the Disk

The structures described above allow one to find the blocks that were allocated to a file. But which blocks should be chosen for allocation? Obviously blocks can be chosen

at random — just take the first one you find that is free. But this can have adverse effects on performance, because accessing different disk blocks requires a physical movement of the disk head. Such physical movements are slow relative to a modern CPU — they can take milliseconds, which is equivalent to millions of instructions. This is why a process is blocked when it performs I/O operations. The mechanics of disk access and their effect on file system layout are further explained in Appendix C.

Placing related blocks together improves performance

The traditional Unix file system is composed of 3 parts: a superblock (which contains data about the size of the system and the location of free blocks), inodes, and data blocks. The conventional layout is as follows: the superblock is the first block on the disk, because it has to be at a predefined location[2]. Next come all the inodes — the system can know how many there are, because this number appears in the superblock. All the rest are data blocks.

The problem with this layout is that it entails much seeking. Consider the example of opening a file named `/a/b/c`. To do so, the file system must access the root inode, to find the blocks used to implement it. It then reads these blocks to find which inode has been allocated to directory `a`. It then has to read the inode for `a` to get to its blocks, read the blocks to find `b`, and so on. If all the inodes are concentrated at one end of the disk, while the blocks are dispersed throughout the disk, this means repeated seeking back and forth.

A possible solution is to try and put inodes and related blocks next to each other, in the same set of cylinders, rather than concentrating all the inodes in one place (a cylinder is a set of disk tracks with the same radius; see Appendix C). This was done in the Unix fast file system. However, such optimizations depend on the ability of the system to know the actual layout of data on the disk, which tends to be hidden by modern disk controllers [1]. Modern systems are therefore limited to using logically contiguous disk blocks, hoping that the disk controller indeed maps them to physically proximate locations on the disk surface.

Exercise 136 *The superblock contains the data about all the free blocks, so every time a new block is allocated we need to access the superblock. Does this entail a disk access and seek as well? How can this be avoided? What are the consequences?*

Log structured file systems reduce seeking

The use of a large buffer cache and aggressive prefetching can satisfy most read requests from memory, saving the overhead of a disk access. The next performance bottleneck is then the implementation of small writes, because they require much seeking to get to the right block. This can be solved by not writing the modified blocks

[2]Actually it is usually the second block — the first one is a boot block, but this is not part of the file system.

in place, but rather writing a single continuous log of all changes to all files and metadata. In addition to reducing seeking, this also improves performance because data will tend to be written sequentially, thus also making it easier to read at a high rate when needed.

Of course, this complicates the system's internal data structures. When a disk block is modified and written in the log, the file's inode needs to be modified to reflect the new location of the block. So the inode also has to be written to the log. But now the location of the inode has also changed, so this also has to be updated and recorded. To reduce overhead, metadata is not written to disk immediately every time it is modified, but only after some time or when a number of changes have accumulated. Thus some data loss is possible if the system crashes, which is the case anyway.

Another problem is that eventually the whole disk will be filled with the log, and no more writing will be possible. The solution is to perform garbage collection all the time: we write new log records at one end, and delete old ones at the other. In many cases, the old log data can simply be discarded, because it has since been overwritten and therefore exists somewhere else in the log. Pieces of data that are still valid are simply re-written at the end of the log.

To read more: Log structured file systems were introduced by Rosenblum and Ousterhout [12].

Logical volumes avoid disk size limitations

The discussion so far has implicitly assumed that there is enough space on the disk for the desired files, and even for the whole file system. With the growing size of data sets used by modern applications, this can be a problematic assumption. The solution is to use another layer of abstraction: logical volumes.

A logical volume is an abstraction of a disk. A file system is created on top of a logical volume, and uses its blocks to store metadata and data. In many cases, the logical volume is implemented by direct mapping to a physical disk or a disk partition (a part of the disk that is disjoint from other parts that are used for other purposes). But it is also possible to create a large logical volume out of several smaller disks. This just requires an additional level of indirection, which maps the logical volume blocks to the blocks of the underlying disks.

Solid-state disks introduce new considerations

The considerations cited above regarding block placement are based on the need to reduce seeking, because seeking (mechanically moving the head of a disk) takes a relatively long time. But with new solid state disks, based on flash memory, there is no mechanical movement. therefore such considerations become void, and we have full flexibility in using memory blocks.

However, other considerations must be made. In particular, current flash memory can only be re-written a limited number of times. It is therefore imperative not to

overwrite the same block too often. File systems designed for solid-state disks (e.g. the ubiquitous disk on key) therefore attempt to distribute block usage as equally as possible. If a certain block turns out to be popular and is re-written often, it will be re-mapped periodically to different locations on the flash device.

5.4.3 Reliability

By definition, the whole point of files to to store data *permanently*. This can run into two types of problems. First, the system may crash leaving the data structures on the disk in an inconsistent state. Second, the disks themselves sometimes fail. Luckily, this can be overcome.

Journaling improves reliability using transactions

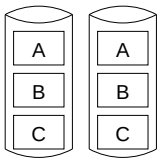
Surviving system crashes is done by journaling. This means that each modification of the file system is handled as a database transaction, implying all-or-nothing semantics.

The implementation involves the logging of all operations. First, a log entry describing the operation is written to disk. Then the actual modification is done. If the system crashes before the log entry is completely written, then the operation never happened. If it crashes during the modification itself, the file system can be salvaged using the log that was written earlier.

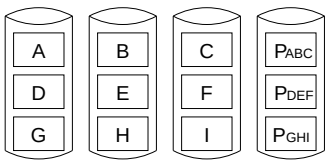
RAID improves reliability using redundancy

Reliability in the face of disk crashes can be improved by using an array of small disks rather than one large disk, and storing redundant data so that any one disk failure does not cause any data loss. This goes by the acronym RAID, for "redundant array of inexpensive (or independent) disks" [10]. There are several approaches:

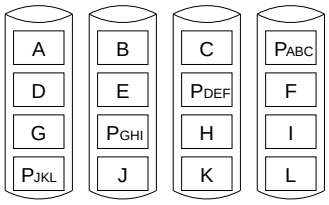
RAID 1: mirroring — there are two copies of each block on distinct disks. This allows for fast reading (you can access the less loaded copy), but wastes disk space and delays writing.



RAID 3: parity disk — data blocks are distributed among the disks in round-robin manner. For each set of blocks, a parity block is computed and stored on a separate disk. Thus if one of the original data blocks is lost due to a disk failure, its data can be reconstructed from the other blocks and the parity. This is based on the self-identifying nature of disk failures: the controller knows which disk has failed, so parity is good enough.



RAID 5: distributed parity — in RAID 3, the parity disk participates in every write operation (because this involves updating some block and the parity), and becomes a bottleneck. The solution is to store the parity blocks on different disks.



(There are also even numbers, but in retrospect they turned out to be less popular).

How are the above ideas used? One option is to implement the RAID as part of the file system: whenever a disk block is modified, the corresponding redundant block is updated as well. Another option is to buy a disk controller that does this for you. The interface is the same as a single disk, but it is faster (because several disks are actually used in parallel) and is much more reliable. While such controllers are available even for PCs, they are still rather expensive.

To read more: The definitive survey of RAID technology was written by Chen and friends [3]. An interesting advanced system is the HP AutoRAID [13], which uses both RAID 1 and RAID 5 internally, moving data from one format to the other according to space availability and usage. Another interesting system is Zebra, which combines RAID with a log structured file system [6].

5.5 Summary

Abstractions

Files themselves are an abstraction quite distant from the underlying hardware capabilities. The hardware (disks) provides direct access to single blocks of a given size. The operating system builds and maintains the file system, including support for files of arbitrary size, and their naming within a hierarchical directory structure.

the operating system itself typically uses a lower-level abstraction of a disk, namely logical volumes.

Resource management

As files deal with permanent storage, there is not much scope for resource management — either the required space is available for exclusive long-term usage, or it is not. The only management in this respect is the enforcement of disk quotas.

Disk scheduling, if still practiced, has a degree of resource management.

Implementation

Files and directories are represented by a data structure with the relevant metadata; in Unix this is called an inode. Directories are implemented as files that contain

the mapping from user-generated names to the respective inodes. Access to disk is mediated by a buffer cache.

Workload issues

Workload issues determine several aspects of file system implementation and tuning.

The distribution of file sizes determines what data structures are useful to store a file's block list. Given that a typical distribution includes multiple small files and few very large files, good data structures need to be hierarchical, like the Unix inode that can grow as needed by using blocks of indirect pointers. This also has an effect on the block size used: if it is too small disk access is less efficient, but if it is too big too much space is lost to fragmentation when storing small files.

Dynamic aspects of the workload, namely the access patterns, are also very important. The locality of data access and the fact that a lot of data is deleted or modified a short time after it is written justify (and even necessitate) the use of a buffer cache. The prevailing use of sequential access allows for prefetching, and also for optimizations of disk layout.

Hardware support

Hardware support exists at the I/O level, but not directly for files. One form of support is DMA, which allows slow I/O operations to be overlapped with other operations; without it, the whole idea of switching to another process while waiting for the disk to complete the transfer would be void.

Another aspect of hardware support is the migration of functions such as disk scheduling to the disk controller. This is actually implemented by firmware, but from the operating system point of view it is a hardware device that presents a simpler interface that need not be controlled directly.

Bibliography

[1] D. Anderson, "*You don't know Jack about disks*". *Queue* **1(4)**, pp. 20–30, Jun 2003.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "*Measurements of a distributed file system*". In 13th *Symp. Operating Systems Principles*, pp. 198–212, Oct 1991. Correction in *Operating Systems Rev.* **27(1)**, pp. 7–10, Jan 1993.

[3] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "*RAID: high-performance, reliable secondary storage*". *ACM Comput. Surv.* **26(2)**, pp. 145–185, Jun 1994.

[4] D. G. Feitelson, "*Metrics for mass-count disparity*". In 14th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[5] E. Freeman and D. Gelernter, "*Lifestreams: a storage model for personal data*". *SIGMOD Record* **25(1)**, pp. 80–86, Mar 1996.

[6] J. H. Hartman and J. K. Ousterhout, "*The Zebra striped network file system*". *ACM Trans. Comput. Syst.* **13(3)**, pp. 274–310, Aug 1995.

[7] J. J. Hull and P. E. Hart, "*Toward zero-effort personal document management*". *Computer* **34(3)**, pp. 30–35, Mar 2001.

[8] G. Irlam, "*Unix file size survey - 1993*". URL http://www.gordoni.com/ufs93.html.

[9] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "*A trace-driven analysis of the UNIX 4.2 BSD file system*". In 10th *Symp. Operating Systems Principles*, pp. 15–24, Dec 1985.

[10] D. A. Patterson, G. Gibson, and R. H. Katz, "*A case for redundant arrays of inexpensive disks (RAID)*". In *SIGMOD Intl. Conf. Management of Data*, pp. 109–116, Jun 1988.

[11] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[12] M. Rosenblum and J. K. Ousterhout, "*The design and implementation of a log-structured file system*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 26–52, Feb 1992.

[13] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "*The HP AutoRAID hierarchical storage system*". *ACM Trans. Comput. Syst.* **14(1)**, pp. 108–136, Feb 1996.

Appendix C

Mechanics of Disk Access

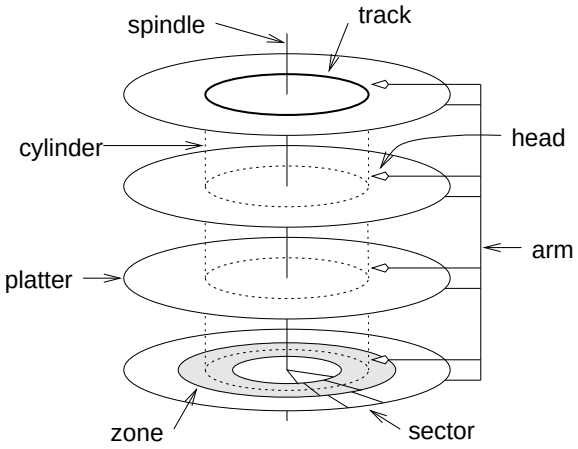
C.1 Addressing Disk Blocks

The various aspects of controlling disk operations are a good example of the shifting roles of operating systems and hardware. In the past, most of these things were done by the operating system, and thereby became standard material in the operating systems syllabus. But in modern systems, most of it is done by the disk controller. The operating system no longer needs to bother with the details.

To read more: A good description of the workings of modern disk drives is given by Ruemmler and Wilkes [3]. An update for even more advanced disk drives that have their own caches and reorder requests is given by Shriver et al. [4].

Addressing disk blocks is (was) based on disk anatomy

A modern disk typically has multiple (1–12) *platters* which rotate together on a common *spindle* (at 5400 or 7200 RPM). Data is stored on both surfaces of each platter. Each surface has its own *read/write head*, and they are all connected to the same *arm* and move in unison. However, typically only one head is used at a time, because it is too difficult to align all of them at once. The data is recorded in concentric *tracks* (about 1500–2000 of them). The set of tracks on the different platters that have the same radius are called a *cylinder*. This concept is important because accessing tracks in the same cylinder just requires the heads to be re-aligned, rather than being moved. Each track is divided into *sectors*, which define the minimal unit of access (each is 256–1024 data bytes plus error correction codes and an inter-sector gap; there are 100-200 per track). Note that tracks near the rim of the disk are much longer than tracks near the center, and therefore can store much more data. This is done by dividing the radius of the disk into (3–20) *zones*, with the tracks in each zone divided into a different number of sectors. Thus tracks near the rim have more sectors, and store more data.

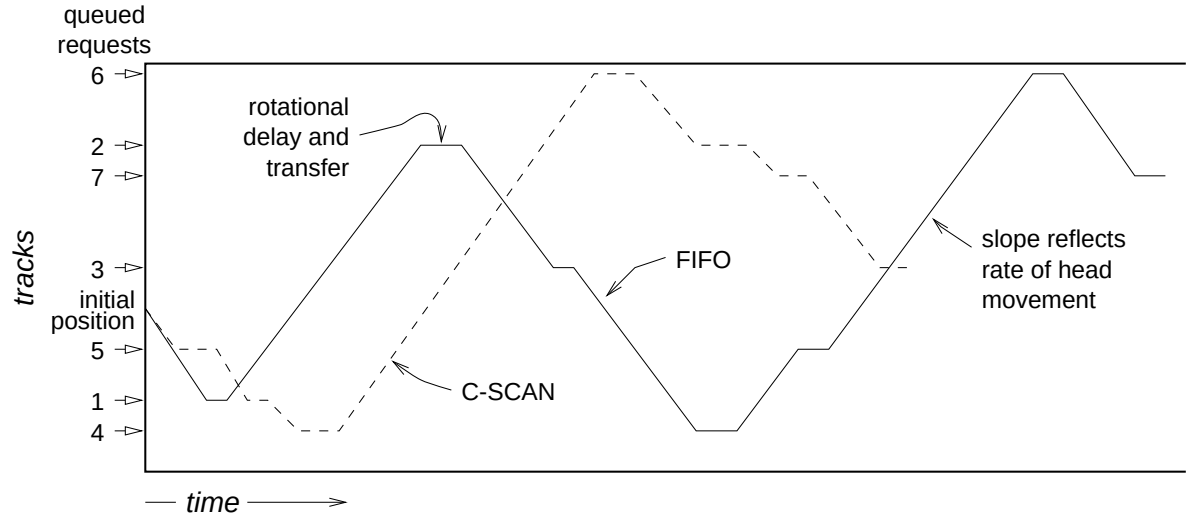


In times gone by, addressing a block of data on the disk was accomplished by specifying the surface, track, and sector. Contemporary disks, and in particular those with SCSI controllers, present an interface in which all blocks appear in one logical sequence. This allows the controller to hide bad blocks, and is easier to handle. However, it prevents certain optimizations, because the operating system does not know which blocks are close to each other. For example, the operating system cannot specify that certain data blocks should reside in the same cylinder [1].

C.2 Disk Scheduling

Getting the read/write head to the right track and sector involves mechanical motion, and takes time. Therefore reordering the I/O operations so as to reduce head motion improves performance. The most common optimization algorithms involve reordering the requests according to their tracks, to minimize the movement of the heads along the radius of the disk. Modern controllers also take the rotational position into account.

The base algorithm is FIFO (first in first out), that just services the requests in the order that they arrive. The most common improvement is to use the SCAN algorithm, in which the head moves back and forth across the tracks and services requests in the order that tracks are encountered. A variant of this is C-SCAN (circular SCAN), in which requests are serviced only while moving in one direction, and then the head returns as fast as possible to the origin. This improves fairness by reducing the maximal time that a request may have to wait.



As with addressing, in the past it was the operating system that was responsible for scheduling the disk operations, and the disk accepted such operations one at a time. Contemporary disks with SCSI controllers are willing to accept multiple outstanding requests, and do the scheduling themselves.

C.3 The Unix Fast File System

As noted in Chapter 5, the Unix fast file system included various optimizations that took advantage of knowledge regarding the geometry of the disk. The main one was to distribute the inodes in clusters across the disk, and attempt to allocate blocks in the same set of cylinders as the inode of their file. Thus reading the file which involves reading both the inode and the blocks will suffer less seeking.

While placing an inode and the blocks it points to together reduces seeking, it may also cause problems. Specifically, a large file may monopolize all the blocks in the set of cylinders, not leaving any for other inodes in the set.

Luckily, the list of file blocks is not all contained in the inode: for large files, most of it is in indirect blocks. The fast file system therefore switches to a new set of cylinders whenever a new indirect block is allocated, choosing a set that is less loaded than the average. Thus large files are indeed spread across the disk. The extra cost of the seek is relatively low in this case, because it is amortized against the accesses to all the data blocks listed in the indirect block.

However, this solution is also problematic. Assuming 10 direct blocks of size 4 KB each, the first indirect block is allocated when the file size reaches 40 KB. Having to perform a seek at this relatively small size is not amortized, and leads to a substantial reduction in the achievable bandwidth for medium-size files (in the range of 50–100 KB). The solution to this is to make the *first* indirect block a special case, that stays in the same set of cylinders as the inode [5].

While this solution improves the achievable bandwidth for intermediate size files, it does not necessarily improve things for the whole workload. The reason is that large files indeed tend to crowd out other files, so leaving their blocks in the same set of cylinders causes other small files to suffer. More than teaching us about disk block allocation, this then provides testimony to the complexity of analyzing performance implications, and the need to take a comprehensive approach.

Sequential layout is crucial for achieving top data transfer rates. Another optimization is therefore to place consecutive logical blocks a certain distance from each other along the track, called the *track skew*. The idea is that sequential access is common, so it should be optimized. However, the operating system and disk controller need some time to handle each request. If we know how much time this is, and the speed that the disk is rotating, we can calculate how many sectors to skip to account for this processing. Then the request will be handled exactly when the requested block arrives under the heads.

To read more: The Unix fast file system was originally described by McKusick and friends [2].

Bibliography

[1] D. Anderson, "*You don't know Jack about disks*". *Queue* **1(4)**, pp. 20–30, Jun 2003.

[2] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "*A fast file system for UNIX*". *ACM Trans. Comput. Syst.* **2(3)**, pp. 181–197, Aug 1984.

[3] C. Ruemmler and J. Wilkes, "*An introduction to disk drive modeling*". *Computer* **27(3)**, pp. 17–28, Mar 1994.

[4] E. Shriver, A. Merchant, and J. Wilkes, "*An analytic behavior model for disk drives with readahead caches and request reordering*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 182–191, Jun 1998.

[5] K. A. Smith and M. I. Seltzer, "*File system aging—increasing the relevance of file system benchmarks*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, Jun 1997.

Chapter 6

Review of Basic Principles

Now after we know about many specific examples of what operating systems do and how they do it, we can distill some basic principles.

policy / mechanism separation; example: mechanism for priority scheduling, policy to set priorities; mechanism for paging, policy to choose pages for replacement

flexibility in changing/optimizing policy, code reuse of mechanism implementation.

6.1 Virtualization

Virtualization is probably the most important tool in the bag of tricks used for system design. It means that the objects that the system manipulates and presents to its users are decoupled from the underlying hardware. Examples for this principle abound.

- Processes are actually a virtualization of the whole computer: they provide the context in which a program runs, including CPU, memory, and I/O options. While this is realized by direct execution on the underlying hardware most of the time, some of the resources are modified. For example, certain priviledged instructions are hidden and cannot be used.
- Threads, which provide the locus of computation in a program, may be considered as an abstraction of the CPU only.
- Virtual memory is the most explicit form of virtualization: each process is presented with an address space that is mapped to the underlying hardware as needed.
- Files can be considered as an idealized (and virtual) storage device: you can store and access your data randomly, without having to worry about the idiosyncrasies of actual devices. At a lower level, logical volumes are a virtualization of disks that does away with the specific size limitations of the hardware.

- Network connections are virtualized just like files, with one important restriction: data access is sequential, rather than being random.

Virtualization typically involves some mapping from the virtual objects to the physical ones. Thus each level of virtualization is equivalent to a level of indirection. While this has its price in terms of overhead, the benefits of virtualization outweigh the cost by far.

hide virtualization mapping using caching

Virtualization helps resource management

most famous in JVM; origins much older

virtual machines in MVS

virtual memory

Virtualization helps protection

virtual machines can't interfere with each other

sandbox for mobile code

Virtualization helps application construction

another example of operating system idea that is moving to applications

virtual tree structure for load balancing - gribble

all this is unrelated to virtual machines as in VMware – see sect. 9.5

6.2 Resource Management

We now turn to more concrete principles, starting with resource management. This is done in two dimensions: time and space.

Use small fixed chunks

Contiguous allocation suffers from fragmentation (in space) and is inflexible (in time). The solution is to partition the resource into small chunks of a fixed size for allocation, and use some mechanism to map the desired contiguous range onto non-contiguous chunks. Examples include:

- Memory is allocated in pages that are mapped to the address space using a page table.
- Disk space is allocated in blocks that are mapped to files using the file's index structure.

- Network bandwidth is allocated in packets that are numbered by the sender and reassembled at the destination

- In parallel systems, single processors can be allocated individually. They are then mapped to the application's logical structure by the communication mechanisms.

This replaces potentially large external fragmentation by limited internal fragmentation, and allows requests from multiple jobs to be interleaved and serviced at the same time.

Maintain flexibility

One of the reasons for using small fixed chunks is to increase flexibility. Thus to be flexible we should avoid making large fixed allocations.

A good example of this principle is the handling of both memory space and disk space. It is possible to allocate a fixed amount of memory for the buffer cache, to support file activity, but this reduces flexibility. It is better to use a flexible scheme in which the memory manager handles file I/O based on memory mapped files. This allows for balanced allocations at runtime between the memory activity and file activity.

Likewise, it is possible to allocate a fixed portion of the disk as backing store for paging, but this too reduces flexibility. It is better to do paging into a file, so that additional space can be added by the file system. Again, this allows for more balanced allocations based on acutal usage patterns.

Use hierarchies to span multiple sizes

In many cases the distribution of request sizes is highly skewed: there are very many small requests, some medium-sized requests, and few very large requests. Thus the operating system should optimize its performance for small requests, but still be able to support large requests efficienty. The solution is to use hierarchical structures that span several (binary) orders of magnitude. Examples are

- The Unix inode structure used to keep maps of file blocks, with its indirect, double indirect, and triple indirect pointers.

- Buddy systems used to allocate memory, disk space, or processors.

- Multi-level feedback queues, where long jobs receive increasingly longer time quanta at lower priorities.

Time is limitless

Physical resources are necessarily bounded. For example, there is only so much disk space; when it is used up, nobody can write any more data. But time is limitless. When coupled with the use of small fixed chunks (implying preemption), this leads to

- The possibility to serve more and more clients, by gracefully degrading the service each one receives. Examples are time slicing the CPU and sending messages in packets.

- The possibility of not having to know about resource requirements in advance in order to make reasonable allocation decisions. If you make a bad decision, you have a chance to correct it in the next round.

Avoid running out

Running out of a resource at an inopertune moment is worse than cleanly failing a request at the outset. It is best to always have a few instances of each resource at hand, and take measures to renew the supply when it runs low.

One example is using hysteresis to dampen oscilations. When a system operates at the end of its tether, it continuously runs out and has to solve this crisis by some quick action. The solution typically provides immediate relief, but does not really solve the basic problem, and the system soon finds itself in the same crisis situation. This is inefficient if the crisis-handling routine is expensive.

A better solution is to prepare in advance. When resource availability falls bellow a certain low-water mark, the system initiates an orderly procedure to obtain more. The important point is that this is continued until a higher level of availability is obtained: instead of just getting a bit more than the low-water mark, we achieve a significantly higher high-water mark. This creates breathing room and ensures that the costly resource reclamation procedure is not called again too soon.

The problem is of course that sometimes the load on the system indeed needs more resources than are available. The solution in this case is — if possible — to forcefully reduce the load. For example, this is the case when swapping is used to escape from thrashing, or when a system uses admission controls.

Define your goals

The most common approach to resource management is to strive for “fairness”. This is essentially equivalent to “best effort” and “graceful degradation”. The idea is that the system is not judgmental — it tries to give good service to all, regardless of their standing and behavior.

Recently there is more and more emphasis on *differentiated services*, meaning that different processes (or users) receive different levels of service. This may be based on

administrative considerations, as in fair-share scheduling, or on technical considerations, as when trying to meet the real-time needs of multimedia applications.

The distinction between these two approaches is important because it dictates the mechanisms that are used. If the goal is to serve all users, then approaches such as partitioning resources into ever smaller portions are justified. But if quality of service is paramount, there is no escape from using admission controls.

6.3 Reduction

Problems don't always have to be solved from basic principles. On the contrary, the best solution is usually to reduce the problem to a smaller problem that can be solved easily.

Use Conventions

Sometimes it is simply impossible to solve a problem from first principles. For example, the problem of finding any file or directory is solved by using a hierarchical directory structure. But this ceates a cycle because we need to be able to find the first directory in order to start the procedure. Thus we need some help from outside of the domain of directories. The simplest solution to this reduced problem (find the first directory) is to use conventions.

Such conventions are globally useful for priming a search:

- The inode of root is 2.
- Network address lookups start from the root DNS.
- Predefined port numbers represent well-known services.

Use Amplification

Amplification is a more general mechanism. The idea is again to use a simple solution to a simple problem, and amplify it in order to solve a larger problem.

- The atomic test-and-set instruction can be used to implement the more general abstraction of a semaphore.
- A name server is the only service you need to know about initially. You can then use it to gain knowledge about any other service you might need.
- When booting the machine, you don't start the full operating system in a single step. You first use a small program stored in ROM to load the boot sector from the disk. This contains a larger program that can start up the operating system.

6.4 Hardware Support and Co-Design

Some theoreticians like challenging models of computation in which the application considers the operating system and hardware to be adversaries, and has to do everything by itself. In real life, it is better to use all the help you can get.

The hardware is your friend

Specifically, operating systems have and will continue to be co-designed with hardware. Always look for the simple solution, and exploit whatever the hardware provides. If it does not yet provide what is needed, it might in the next generation (especially if you ask the hardware designers).

One example comes from the early support for paging and virtual memory on the VAX running Unix. Initially, hardware support for mapping was provided, but without use bits that can be used to implement replacement algorithms that approximate LRU. The creative solution was to mark the pages as absent. Accessing them would then cause a page fault and a trap to the operating system. But the operating system would know that the page was actually there, and would simply use this page fault to simulate its own used bits. In later generations, the used bit migrated to hardware.

Another example comes from concurrent programming. True, it is possible to devise clever algorithms that solve the mutual exclusion problem using only load and store operations on memory. But is is much easier with instructions such as test-and-set.

Finally, many aspects of I/O control, including caching and disk scheduling, are migrating from the operating system to the disk controllers.

As a side note, migration from the operating system to user applications is also important. Various abstractions invented within operating systems, such as semaphores, are actually useful for any application concerned with concurrent programming. These should (and have been) exposed at the systems interface, and made available to all.

There are numerous examples of hardware support

Here is a list of examples we have mentioned for the close interactions of hardware and the operating system:

- Kernel execution mode
  - The interrupt vector, enabling the registration of operating system functions to handle different interrupts
  - The trap instruction to implement system calls
- Clock interrupts to regain control and support timing functions
- Support for concurrent programming

  - Atomic operations such as test-and-set
  - The options of blocking interrupts
  - DMA to allow I/O to proceed in parallel with computation, and interrupts to alert the system when I/O operations have completed

- Support for memory mapping
  - Address translation that includes protection and page faults for unmapped pages
  - The ability to switch mappings easily using a register that points to the base address of the page/segment table
  - The TLB which serves as a cache for the page table
  - Updating the used/modified bits of pages when they are accessed

- Help in booting the machine

Part III

Crosscutting Issues

The classical operating system curriculum emphasizes major concepts and abstractions, and the intellectual achievements in their design. However, this is sometimes divorced from what real systems do.

First, a major concern in modern systems is security and authentication. This is most directly related to file access, but in reality it relates to all aspects of system usage. For example, not every user should have the option to kill an arbitrary process.

Second, there is the issue of supporting multiprocessor systems. In part II we limited the discussion to systems with a single processor, but this is increasingly anachronistic. Servers have employed symmetric multiprocessing for years, and in recent years even desktop machines are using hyperthreading and multicore processors. This provides the opportunity to review all what we learned with the added perspective of how to adjust it to support true parallelism.

Third, there is the question of structure. An operating system is a large and complex program. Therefore organization, bookkeeping, and structure are just as important as using great algorithms.

Fourth, there is the issue of performance. We therefore devote a chapter to performance evaluation, and introduces the background material needed to be able to evaluate the performance of operating systems. One of the issues covered is workloads — the "input" to the operating system. The dependence of performance on workload is a recurring theme throughout these notes.

Finally, there are lots of technical issues that simply don't have much lustre. However, you need to know about them to really understand how the system works.

Chapter 7

Identification, Permissions, and Security

Processes are the active elements in a computer system: they are the agents that perform various operations on system elements. For example, a process can create or delete a file, map a region of memory, and signal another process.

In most cases, a process performs its work on behalf of a human user. In effect, the process represents the user in the system. And in many cases, it is desirable to control what different users are allowed to do to certain system objects. This chapter deals with the issues of identifying a user with a process, and of granting or denying the rights to manipulate objects. In particular, it also covers security — denying certain rights despite the users best efforts to obtain them.

7.1 System Security

The operating system is omnipotent

The operating system can do anything it desires. This is due in part to the fact that the operating system runs in kernel mode, so all the computer's instructions are available. For example, it can access all the physical memory, bypassing the address mapping that prevents user processes from seeing the memory of other processes. Likewise, the operating system can instruct a disk controller to read any datda block off the disk, regardless of who the data belongs to.

The problem is therefore to prevent the system from performing such actions on behalf of the wrong user. Each user, represented by his processes, should be able to access only his private data (or data that is flaged as publicly available). The operating system, when acting on behalf of this user (e.g. during the execution of a system call), must restrain itself from using its power to access the data of others.

In particular, restricting access to file data is a service that the operating system

provides, as part of the file abstraction. The system keeps track of who may access a file and who may not, and only performs an operation if it is permitted. If a process requests an action that is not permitted by the restrictions imposed by the system, the system will fail the request rather than perform the action — even though it is technically capable of performing this action.

7.1.1 Levels of Security

Security may be all or nothing

It should be noted that when the operating system's security is breached, it typically means that *all* its objects are vulnerable. This is the holy grail of system hacking — once a hacker "breaks into" a system, he gets full control. For example, the hacker can impersonate another user by setting the user ID of his process to be that of the other user. Messages sent by that process will then appear to all as if they were sent by the other user, possibly causing that user some embarasement.

> **Example: the Unix superuser**
> In unix, security can be breached by managing to impersonate a special user known as the superuser, or "root". This is just a user account that is meant to be used by the system administrators. By logging in as root, system administrators can manipulate system files (e.g. to set up other user accounts), change system settings (e.g. set the clock) or clean up after users who had experienced various problems (e.g. when they create files and specify permissions that prevent them from later deleting them). The implementation is very simple: all security checks are skipped if the requesting process is being run by root.
>
> Windows was originally designed as a single-user system, so the user had full priviledges. The distinction between system administrators and other users was only introduced recently.

Alternatively, rings of security can be defined

An alternative is to design the system so that it has several security levels. Such a design was employed in the Multics system, where it was envisioned as a set of concentric rings. The innermost ring represents the highest security level, and contains the core structures of the system. Successive rings are less secure, and deal with objects that are not as important to the system's integrity. A process that has access to a certain ring can also access objects belonging to larger rings, but needs to pass an additional security check before it can access the objects of smaller rings.

7.1.2 Mechanisms for Restricting Access

Access can be denyed by hiding things

A basic tool used for access control is hiding. Whatever a process (or user) can't see, it can't access. For example, kernel data structures are not visible in the address space of any user process. In fact, neither is the data of other processes. This is achieved using the hardware support for address mapping, and only mapping the desired pages into the address space of the process.

A slightly less obvious example is that files can be hidden by not being listed in any directory. For example, this may be useful for the file used to store users' passwords. It need not be listed in any directory, because it is not desirable that processes will access it directly by name. All that is required is that the system know where to find it. As the system created this file in the first place, it can most probably find it again. In fact, its location can even be hardcoded into the system.

Exercise 137 *One problem with the original versions of Unix was that passwords were stored together with other data about users in a file that was readable to all (`/etc/passwd`). The password data was, however, encrypted. Why was this still a security breach?*

Conversely, permissions can be granted using opaque handles

Given that system security is an all or nothing issue, how can limited access be granted? In other words, how can the system create a situation in which a process gains access to a certain object, but cannot use this to get at additional objects? One solution is to represent objects using opaque handles. This means that processes can store them and return them to the system, where they make sense in a certain context. However, processes don't understand them, and therefore can't manipulate or forge them.

A simple example is the file descriptor used to access open files. The operating system, as part of the `open` system call, creates a file descriptor and returns it to the process. The process stores this descriptor, and uses it in subsequent `read` and `write` calls to identify the file in question. The file descriptor's value makes sense to the system in the context of identifying open files; in fact, it is an index into the process's file descriptors table. It is meaningless for the process itself. By knowing how the system uses it, a process can actually forge a file descriptor (the indexes are small integers). However, this is useless: the context in which file descriptors are used causes such forged file descriptors to point to other files that the process has opened, or to be illegal (if they point to an unused entry in the table).

To create un-forgeable handles, it is possible to embed a random bit pattern in the handle. This random pattern is also stored by the system. Whenever the handle is presented by a process claiming access rights, the pattern in the handle is compared

against the pattern stored by the system. Thus forging a handle requires the process to guess the random patterns that is stored in some other legitimate handle.

Exercise 138 *Does using the system's default random number stream for this purpose compromise security?*

7.2 User Identification

As noted above, users are represented in the system by processes. But how does the system know which user is running a particular process?

Users are identified based on a login sequence

Humans recognize each other by their features: their face, hair style, voice, etc. While user recognition based on the measurement of such features is now beginning to be used by computer systems, it is still not common.

The alternative, which is simpler for computers to handle, is the use of passwords. A password is simply a secret shared by the user and the system. Any user that correctly enters the secret password is assumed by the system to be the owner of this password. In order to work, two implicit assumptions must be acknowledged:

1. The password should be physically secured by the user. If the user writes it on his terminal, anyone with access to the terminal can fool the system.

2. The password should be hard to guess. Most security compromises due to attacks on passwords are based on the fact that many users choose passwords that are far from being random strings.

To read more: Stallings [1, sect. 15.3] includes a nice description of attacks against user passwords.

Processes may also transfer their identity

It is sometimes beneficial to allow other users to assume your identity. For example, the teacher of a course may maintain a file in which the grades of the students are listed. Naturally, the file should be accessible only by the teacher, so that students will not be able to modify their grades or observe the grades of their colleagues. But if users could run a program that just reads their grades, all would be well.

Unix has a mechanism for just such situations, called "set user ID on execution". This allows the teacher to write a program that reads the file and extracts the desired data. The program belongs to the teacher, but is executed by the students. Due to the set user ID feature, the process running the program assumes the teacher's user ID when it is executed, rather than running under the student's user ID. Thus it has access to the grades file. However, this does not give the student unrestricted access

to the file (including the option of improving his grades), because the process is still running the teacher's program, not a program written by the student.

Exercise 139 *So is this feature really safe?*

A more general mechanism for impersonation is obtained by passing opaque handles from one process to another. A process with the appropriate permissions can obtain the handle from the system, and send it to another process. The receiving process can use the handle to perform various operations on the object to which the handle pertains. The system assumes it has the permission to do so because it has the handle. The fact that it obtained the handle indirectly is immaterial.

Exercise 140 *Does sending a Unix file descriptor from one process to another provide the receiving process with access to the file?*

7.3 Controling Access to System Objects

System objects include practically all the things you can think of, e.g. files, memory, and processes. Each has various operations that can be performed on it:

Object	*Operations*
File	read, write, rename, delete
Memory	read, write, map, unmap
Process	kill, suspend, resume

In the following we use files for concreteness, but the same ideas can be applied to all other objects as well.

Access to files is restricted in order to protect data

Files may contain sensitive information, such as your old love letters or new patent ideas. Unless restricted, this information can be accessed by whoever knows the name of the file; the name in turn can be found by reading the directory. The operating system must therefore provide some form of protection, where users control access to their data.

There are very many possible access patterns

It is convenient to portray the desired restrictions by means of an *access matrix*, where the rows are users (or more generally, domains) and the columns are objects (e.g. files or processes). The i, j entry denotes what user i is allowed to do to object j. For example, in the following matrix file3 can be read by everyone, and moti has read/write permission on dir1, dir2, file2, and the floppy disk.

	objects							
users	dir1	dir2	file1	file2	file3	file4	floppy	tape
yossi	r		r		r			
moti	rw	rw		rw	r		rw	
yael		r			r			r
rafi					r			rw
tal	r		rw		r	rw	rw	
...								

Most systems, however, do not store the access matrix in this form. Instead they either use its rows or its columns.

You can focus on access rights to objects...

Using columns means that each object is tagged with an *access control list* (ACL), which lists users and what they are allowed to do to this object. Anything that is not specified as allowed is not allowed. Alternatively, it is possible to also have a list cataloging users and specific operations that are to be prevented.

Example: ACLs in Windows
Windows (starting with NT) uses ACLs to control access to all system objects, including files and processes. An ACL contains multiple *access control entries* (ACEs). Each ACE either allows or denies a certain type of access to a user or group of users.

Given a specific user that wants to perform a certain operation on an object, the rules for figuring out whether it is allowed are as follows.

- If the object does not have an ACL at all, it means that no access control is desired, and everything is allowed.
- If the object has a null ACL (that is, the ACL is empty and does not contain any ACEs), it means that control is desired and nothing is allowed.
- If the object has an ACL with ACEs, all those that pertain to this user are inspected. The operation is then allowed if there is a specific ACE which allows it, provided there is no ACE which forbids it. In other words, ACEs that deny access take precedence over those which allow access.

Exercise 141 *Create an ACL that allows read access to user yosi, while denying write access to a group of which yosi is a member. And how about an ACL that denies read access to all the group's members except yosi?*

... or on the capabilities of users

Using rows means that each user (or rather, each process running on behalf of a user) has a list of *capabilities*, which lists what he can do to different objects. Operations that are not specifically listed are not allowed.

A nice option is sending capabilities to another process, as is possible in the Mach system. This is done by representing the capabilities with opaque handles. For example, a process that gains access to a file gets an unforgeable handle, which the system will identify as granting access to this file. The process can then send this handle to another process, both of them treating it as an un-interpreted bit pattern. The receiving process can then present the handle to the system and gain access to the file, without having opened it by itself, and in fact, without even knowing which file it is!

Grouping can reduce complexity

The problem with both ACLs and capabilities is that they may grow to be very large, if there are many users and objects in the system. The solution is to group users or objects together, and specify the permissions for all group members at once. For example, in Unix each file is protected by a simplified ACL, which considers the world as divided into three: the file's owner, his group, and all the rest. Moreover, only 3 operations are supported: read, write, and execute. Thus only 9 bits are needed to specify the file access permissions. The same permission bits are also used for directories, with some creative interpretation: read means listing the directory contents, write means adding or deleting files, and execute means being able to access files and subdirectories.

Exercise 142 *what can you do with a directory that allows you read permission but no execute permission? what about the other way round? Is this useful? Hint: can you use this to set up a directory you share with your partner, but is effectively inaccessible to others?*

7.4 Summary

Abstractions

Security involves two main abstractions. One is that of a user, identified by a user ID, and represented in the system by the processes he runs. The other is that of an object, and the understanding that practically all entities in the system are such objects: table entries, memory pages, files, and even processes. This leads to the creation of general and uniform mechanisms that control all types of access to (and operations upon) system objects.

Resource management

Gaining access to an object is a prerequisite to doing anything, but does not involve resource management in itself. However, handles used to access objects can in some cases be resources that need to be managed by the system.

Workload issues

Because security if not involved with resource management, it is also not much affected by workload issues. But one can take a broader view of "workload", and consider not only the statistics of what operations are performed, but also what patterns are desired. This leads to the question of how rich the interface presented by the system should be. For example, the access permissions specification capabilities of Unix are rather poor, but in many cases they suffice. Windows NT is very rich, which is good because it is very expressive, but also bad because it can lead to to overhead for management and checking and maybe also to conflicts and errors.

Hardware support

Security is largely performed at the operating system level, and not in hardware. However, it does sometimes use hardware support, as in address mapping that hides parts of the memory that should not be accessible.

Bibliography

[1] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

Chapter 8

SMPs and Multicore

operating system developed in single-processor environment in the 1970s. now most servers and many desktops are SMPs. near future is chip multiprocessors, possibly heterogeneous.

8.1 Operating Systems for SMPs

8.1.1 Parallelism vs. Concurrency

with SMP, things really happen at the same time
    disabling interrupts doesn't help

8.1.2 Kernel Locking

single global lock
    fine grain locking

8.1.3 Conflicts

different processors may take conflicting actions, e.g. regarding allocation of pages.

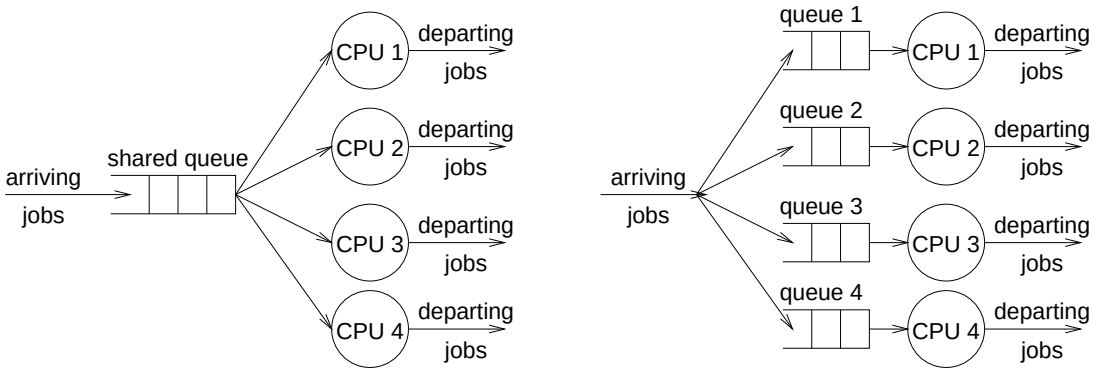
8.1.4 SMP Scheduling

8.1.5 Multiprocessor Scheduling

In recent years multiprocessor systems (i.e. machines with more than one CPU) are becoming more common. How does this affect scheduling?

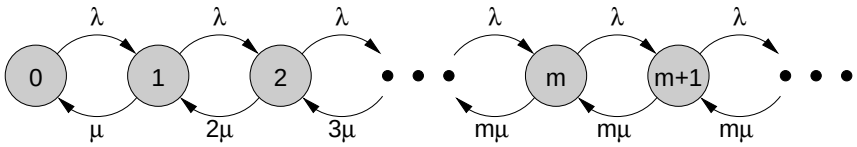
Queueing analysis can help

The question we are posing is the following: is it better to have one shared ready queue, or should each processor have its own ready queue? A shared queue is similar to common practice in banks and post offices, while separate queues are similar to supermarkets. Who is right?



The queueing model for multiple independent queues is simply to take m copies of the M/M/1 queue, each with an arrival rate of λ/m. This represents a system in which arrivals are assigned at random to one of the processors.

The queueing model for a shared queue is M/M/m, where m is the number of processors (servers). The state space is similar to that of an M/M/1 queue, except that the transitions from state i to $i-1$ depend on how many servers are active:



The mathematics for this case are a bit more complicated than for the M/M/1 queue, but follow the same principles. The result for the average response time is

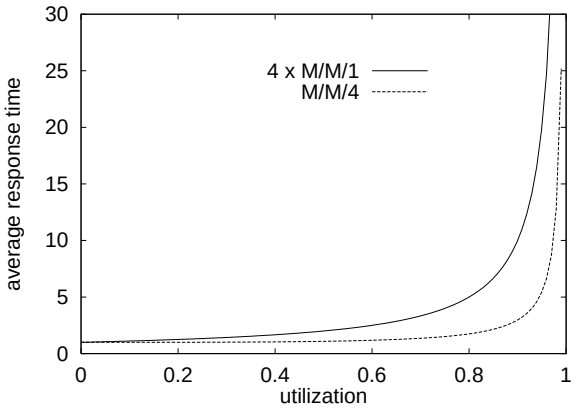
$$\bar{r} = \frac{1}{\mu}\left(1 + \frac{q}{m(1-\rho)}\right)$$

where q is the probability of having to wait in the queue because all m servers are busy, and is given by

$$q = \pi_0 \frac{(m\rho)^m}{m!(1-\rho)}$$

The expression for π_0 is also rather complicated...

The resulting plots of the response time, assuming $\mu = 1$ and $m = 4$, are



Obviously, using a shared queue is substantially better. The banks and post offices are right. And once you think about it, it also makes sense: with a shared queue, there is never a situation where a client is queued and a server is idle, and clients that require only short service do not get stuck behind clients that require long service — they simply go to another server.

Exercise 143 *Why is it "obvious" from the graphs that $M/M/4$ is better?*

But what about preemption?

The queueing analysis is valid for the non-preemptive case: each client sits in the queue until it is picked by a server, and then holds on to this server until it terminates.

With preemption, a shared queue is less important, because clients that only require short service don't get stuck behind clients that need long service. However, this leaves the issue of load balancing.

If each processor has its own local queue, the system needs explicit load balancing — migrating jobs from overloaded processors to underloaded ones. This ensures fairness and overall low response times. However, implementing process migration is not trivial, and creates overhead. It may therefore be better to use a shared queue after all. This ensures that there is never a case where some job is in the queue and some processor is idle. As jobs are not assigned to processors in this case, this is called "load sharing" rather than "load balancing".

The drawback of using a shared queue is that each job will probably run on a different processor each time it is scheduled. This leads to the corruption of cache state. It is fixed by *affinity scheduling*, which takes the affinity of each process to each processor into account, where affinity is taken to represent the possibility that relevant state is still in the cache. It is similar to simply using longer time quanta.

8.2 Supporting Multicore Environments

Chapter 9

Operating System Structure

An operating system does a lot of things, and therefore has many components. These can be organized in various ways. In addition, operating systems may need to control various hardware platforms, including distributed and parallel ones.

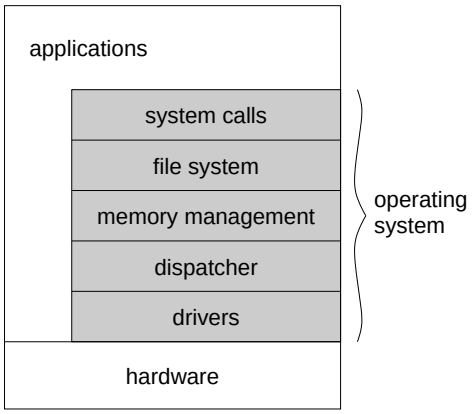
9.1 System Composition

Operating systems are built in layers

It is hard to build all the functionality of an operating system in a single program. Instead, it is easier to create a number of layers that each add some functionality, based on functionality already provided by lower layers. In addition, separate modules can handle different services that do not interact.

For example, one can implement thread management as a low level. Then the memory manager, file system, and network protocols can use threads to maintain the state of multiple active services.

The order of the layers is a design choice. One can implement a file system above a memory manager, by declaring a certain file to be the backing store of a memory segment, and then simply accessing the memory. If the data is not there, the paging mechanism will take care of getting it. Alternatively, one can implement memory management above a file system, by using a file for the swap area.



All the layers can be in one program, or not

The "traditional" way to build an operating system is to create a *monolithic* system, in which all the functionality is contained in the kernel. The kernel is then a big and complex thing. All the system tables and data structures are shared by different parts of the system — in effect, they are like global variables. The different parts of the system can access the data structures directly. This is what causes the mutual exclusion problems mentioned in Chapter 3.

An alternative approach is to use a *micro*kernel, that just provides the basic functionality. The rest of the operating system work is done by external servers. Each of these servers encapsulates the policies and data structures it uses. Other parts of the system can only communicate with it using a well-defined interface.

This distinction is also related to the issue of where the operating system code runs. A kernel can either run as a separate entity (that is, be distinct from all processes), or be structured as a collection of routines that execute as needed within the environment of user processes. External servers are usually separate processes that run at user level, just like user applications. Routines that handle interrupts or system calls run within the context of the current process, but typically use a separate kernel stack.

Some services are delegated to daemons

In any case, some services can be carried out by independent processes, rather than being bundled into the kernel. In Unix, such processes are caled *daemons*. Some daemons are active continuously, waiting for something to do. For example, requests to print a document are handled by the print daemon, and web servers are implemented as an http daemon that answers incoming requests from the network. Other daemons are invoked periodically, such as the daemon that provides the service of starting user applications at predefined times.

9.2 Monolithic Kernel Structure

Monolithic kernels may be layered, but appart from that, tend not to be very modular. Both the code and the data structures make use of the fact that everything is directly accessible.

9.2.1 Code Structure

The operating system has many entry points

Recall that an operating system is basically a reactive program. This implies that it needs to have many entry points, that get called to handle various events. These can

be divided into two types: interrupt handlers and system calls. But using these two types is rather different.

Interrupt handlers must be known by the hardware

Interrupts are a hardware event. When an interrupt occurs, the hardware must know what operating system function to call. This is supported by the interrupt vector. When the system is booted, the addresses of the relevant functions are stored in the interrupt vector, and when an interrupt occurs, they are called. The available interrupts are defined by the hardware, and so is the interrupt vector; the operating system must comply with this definition in order to run on this hardware.

While the entry point for handling the interrupt must be known by the hardware, it is not necessary to perform all the handling in this one function. In many cases, it is even unreasonable to do so. The reason is that interrupts are asynchronous, and may occur at a very inoportune moment. Thus many systems partition the handling of interrupts into two: the handler itself just stores some information about the interrupt that has occurred, and the actual handling is done later, by another function. This other functions is typically invoked at the next context switch. This is a good time for practically any type of operation, as the system is in an itermediate state after having stopped one process but before starting another.

System calls cannot be known by the hardware

The repertoire of system calls provided by an operating system cannot be known by the hardware. In fact, this is what distinguishes one operating system from another. Therefore a mechanism such as the interrupt vector cannot be used.

Instead, the choice of system calls is done by means of a single entry point. This is the function that is called when a trap instruction is issued. Inside this function is a large switch instructions, that branches according to the desired system call. For each system call, the appropriate internal function is called.

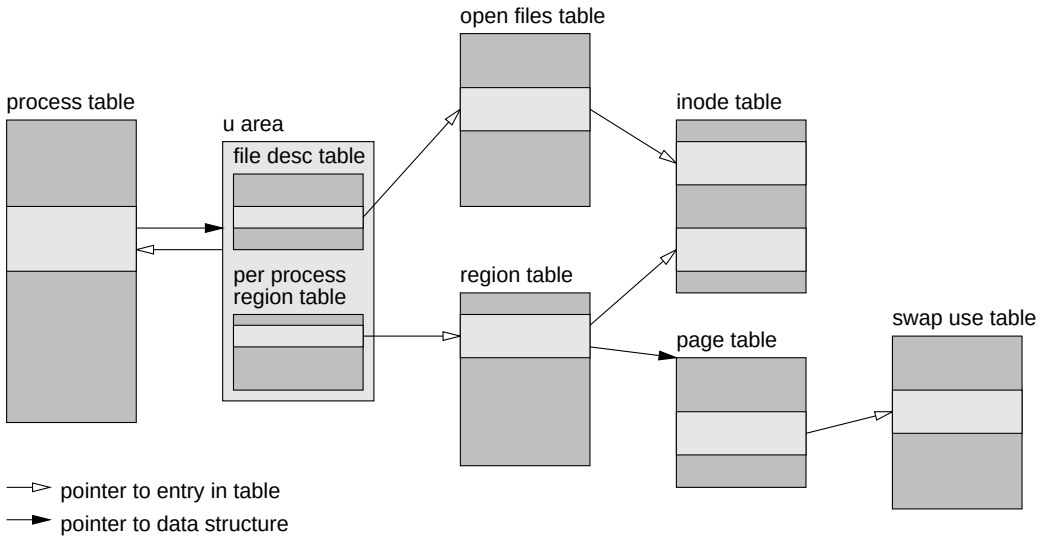
But initial branching does not imply modularity

The fact that different entry points handle different events is good — it shows that the code is partitioned according to its function. However, this does not guarantee a modular structure as a whole. When handling an event, the operating system may need to access various data structures. These data structures are typically global, and are accessed directly by different code paths. In monolithic systems, the data structures are not encapsulated in separate modules that are only used via predefined interfaces.

9.2.2 Data Structures

Tables are global and point to each other

Operating systems by nature require many different data structures, to keep track of the different entities that they manage. These include the process table, open files table, tables related to memory management, and more. In monolithic kernels, these are typically big global data structures. Moreover, they are intertwined by means of mutual pointing.

> **Example: major Unix tables**
>
> The following figure shows major tables in a classic Unix implementation.
>
> 
>
>
> **Process table** — this table contains an entry for each process. Additional data about each process is maintained in the u-area. The u-area and process table enry point to each other. In addition, entries in the process table point to each other. For example, each process points to its parent, and processes are linked to each other in the run queue and various wait queues.
>
> **File related tables** — the u-area of each process contains the file descriptors table for that process. Entries in this table point to entries in the open files table. These, in turn, point to entries in the inode table. The information in each inode contains the device on which the file is stored.
>
> **Memory related tables** — the u-area also contains a per-process region table, whose entries point to entries in the global region table, which describe different memory regions. Data maintained for a region includes a pointer to the inode used to initialize this region, and a page table for the region. This, in turn, contains a pointer to where the backup for each page is stored in the swap area.

problems: code instability and security risks due to global address space. any driver can modify core kernel data. need to lock correctly. (see linus vs. tanenbaum)

9.2.3 Preemption

preempting the operating system for better responsiveness [2]. fits with partitioning interrupt handling. must not wait for I/O.

9.3 Microkernels

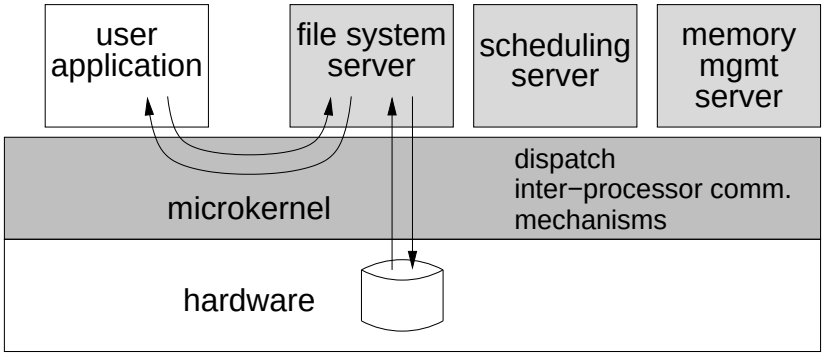
Microkernels take the idea of doing things outside of the kernel to the limit.

Microkernels separate mechanism from policy

The reasons for using microkernels are modularity and reliability. The microkernel provides the mechanisms to perform various actions, such as dispatching a process or reading a disk block. However, that is all it does. It does not decide *which* process to run — this is the work of an external scheduling server. It does not create named files out of disk blocks — this is the work of an external file system server. Thus it is possible to change policies and services by just replacing the external servers, without changing the microkernel. In addition, any pproblem with one of the servers will at worst crash that server, and not the whole system. And then the server can simply be restarted.

Microkernels are thus named because they are supposed to be much smaller than a conventional kernel. This is a natural result of the fact that a lot of functionality was moved out of the kernel. The main things that are left in the kernel are

- Multitasking and process or thread dispatch. The microkernel will do the actual context switch, and will also schedule interrupt handlers.
- Message passing among user processes. This is needed to pass systems calls from user applications to the servers that handle them.
- The mechanics of memory mapping and disk access



Note that the microkernel effectively encapsulates all hardware-specific code. Therefore porting the system to a new architecture only requires porting the microkernel. The servers are inherently portable by virtue of running above the microkernel, with no direct interaction with the hardware.

To read more: The concept of microkernels was developed in the Mach operating system [4]. The relative merits of microkernels vs. monolithic designs were hashed out in a well-known debate between Andrew Tanenbaum (professor and textbook author, creator of the micro-kernel based Minix aducational operating system) and Linux Torvalds (creator of the monolithic Linux kernel).

In particular, interrupt handling is divided into two parts

The distinction between the microkernel and the servers extends to interrupt handling. The interrupt handler invoked by the hardware is part of the microkernel. But this function typically cannot actually handle the interrupt, as this should be done subject to policy decisions of the external servers. Thus a microkernel architecture naturally creates the partitioning of interrupt handlers into two parts that was done artificially in monolithic systems.

Note that this also naturally inserts a potential preemption point in the handling of interrupts, and in fact of any kernel service. This is important for improved responsiveness.

External servers can have different personalities

The system's policies and the services it provides are what differentiates it from other systems. Once these things are handled by an external server, it is possible to support multiple system personalities at once. For example, Windows NT can support Windows applications with its Windows server, OS/2 applications with its OS/2 server, and Unix applications with its Unix server. Each of these servers supplies unique interfaces and services, which are implemented on top of the NT microkernel.

In addition, the fact that services are provided by independent processes makes it easier to debug them, and also makes it possible to add and remove services while the system is running.

But performance can be a problem

The price is overhead in communicating between the different modules. For example, when a user application wants to read a file, the system first has to switch to the file system server. It also has to route messages back and forth between the application and the server.

9.4 Extensible Systems

Extensibility is required in order to handle new devices

It may be surprising at first, but the fact is that operating systems have to be extended after they are deployed. There is no way to write a "complete" operating system that can do all that it will ever be asked to do. One reason for this is that new hardware devices may be introduced after the operating system is done. As this cannot be anticipated in advance, the operating system has to be extended.

Exercise 144 *But how does code that was compiled before the new device was added know how to call the routines that handle the new device?*

It can also be used to add functionality

The case of hardware devices is relatively simple, and is handled by modules called device drivers that are added to the kernel. But once the concept is there, it can be used for other things.

Several systems support *loadable modules*. This means that additional code can be loaded into a running system, rather than having to recompile the kernel and reboot the system.

Using loadable modules requires indirection: to add a system call or wrap an existing system call, one has to modify entries in the system call table of the trap handler. When wrapping an existing function, the wrapper calls the original function. To add functionality, add e.g. to a table of supported file systems. Such modules can be loaded automatically as needed, and unloaded when they falls out of use.

This leads to the option of application-specific customization

One of the functions of an operating system is to provide various useful abstractions to application programs. But it is impossible to anticipate all the needs of applications. Moreover, different applications may have conflicting needs.

A solution to this problem is to allow applications to *customize* the operating system services according to their needs. A good example is provided by the Choices research operating system [1]. This system is based on an object-oriented framework, and exports various classes to the user interface. Applications may write code that inherits part of the original operating system implementation, but replaces other parts with customized versions.

As a concrete example, consider the implementation of a semaphore. When a process performs the P operation, the operating system may need to block it if the semaphore is currently not available. Thus many processes may become queued in the semaphore. The question is how to maintain this queue. Different implementations can emphasize speed, fairness, or consideration of process priorities. Rather

than trying to anticipate and provide all these options, the operating system implements only one which seems generally useful. Applications that prefer another approach can substitute it with their own implementation.

Exercise 145 *Can an application be allowed to customize the system scheduler?*

9.5 Operating Systems and Virtual Machines

In recent years there is a growing trend of using *virtualization*. This means that it is possible to create multiple virtual copies of the machine (called *virtual machines*), and run a different operating system on each one. This decouples the issue of creating abstractions within a virtual machine from the provisioning of resources to the different virtual machines.

Virtual machines are transparent

The idea of virtual machines is not new. It originated with MVS, the operating system for the IBM mainframes. In this system, time slicing and abstractions are completely decoupled. MVS actually only does the time slicing, and creates multiple *exact copies* of the original physical machine. Then, a single-user operating system called CMS is executed in each virtual machine. CMS provides the abstractions of the user environment, such as a file system.

As each virtual machine is an exact copy of the physical machine, it was also possible to run MVS itself on such a virtual machine. This was useful to debug new versions of the operating system on a running system. If the new version is buggy, only its virtual machine will crash, but the parent MVS will not. This practice continues today, and VMware has been used as a platform for allowing students to experiment with operating systems.

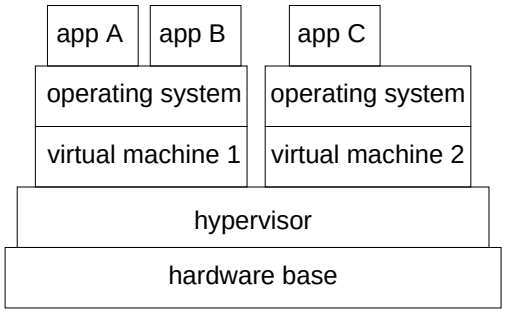
So what is the difference between a virtual machine and a process? A process also provides a virtualized execution environment for an application, giving it the impression of having the machine to itself. However, processes in fact "leak" information and only provide partial isolation between running applications. In addition, they can only run a single application. A virtual machine enjoys better isulation, and may run a full operating system.

> **Types of virtual machines**
> probably the best known virtualization technology is the Java virtual machine (JVM). The Java virtual machine is a runtime system, that provides a runtime environment to Java applications. It is used to disconnect the running application from the underlying operating system and hardware in the interest of portability. The virtual machines we are talking about are a direct emulation of the hardware. They need a guest operating system in order to run applications.

To read more: History buffs can read more about MVS in the book by Johnson [3].

Hypervisors perform some operating system functions

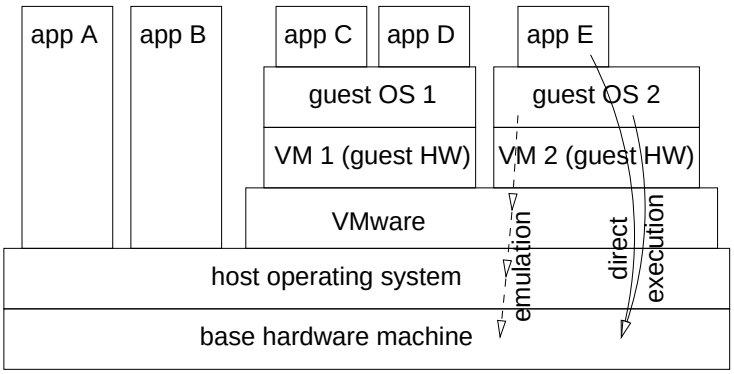
At its heart virtualization is similar to what operating systems do: it is involved with allocating resources to competing entities, and providing them with a semblance of isolation from each other. The software performing this task is called a hypervisor or virtual machine monitor (VMM).



Scheduling in a hypervisor is similar to scheduling in an operating system. However, the hypervisor typically has less information at its disposal. Likewise, allocation of memory to the different virtual machines is typically based on static allocations, leaving the guest operating systems to use the allocated space via paging.

There can be multiple levels of virtualization

A remarkable example is given by VMware. This is actually a user-level application, that runs on top of a conventional operating system such as Linux or Windows. It creates a set of virtual machines that mimic the underlying hardware. Each of these virtual machines can boot an independent operating system, and run different applications:



Execution of applications on the virtual machines is done directly on the hardware. However, their system calls are intercepted and redirected to the guest operating system. The guest operating system also may run directly on the hardware. However, when it performs a priviledged instruction or attempts to control the hardware, thys is

emulated by the underlying VMware system. The operating system itself is oblivious to this, and runs without knowing about or supporting the virtualization.

As demonstrated by VMware, the issue of what exactly constitutes the operating system can be murky. In principle, the operating system is the entity that uses the underlying hardware to provide the operating environment needed by the application. But what is this for, say, application D? Its environment is created by three distinct layers of software: the host operating system, the VMware system, and the operating system running on virtual machine 1.

Bibliography

[1] R. H. Campbell, N. Islam, D. Raila, P. Madany. Designing and implementing Choices: An object-oriented system in C++. *Comm. ACM*, 36(9):117–126, Sep 1993.

[2] A. Goel, L. Abeni, C. Krasic, J. Snow, J. Walpole. Supporting time-sensitive applications on a commodity OS. *Symp. Operating Systems Design & Implementation*, number 5, strony 165–180, Dec 2002.

[3] R. H. Johnson. *MVS: Concepts and Facilities*. McGraw-Hill, 1989.

[4] R. F. Rashid. Threads of a new system. *UNIX Review*, 4(8):37–49, Aug 1986.

Chapter 10

Performance Evaluation

Operating system courses are typically concerned with *how to do* various things, such as processor scheduling, memory allocation, etc. Naturally, this should be coupled with a discussion of *how well* the different approaches work.

10.1 Performance Metrics

Interestingly, the answer to how well a system works may depend on how you quantify "wellness", that is, on your metric for good performance.

Being fast is good

The most common metrics are related to time. If something takes less time, this is good for two reasons. From the user's or client's perspective, being done sooner means that you don't have to wait as long. From the system's perspective, being done sooner means that we are now free to deal with other things.

While being fast is good, it still leaves the issue of units. Should the task of listing a directory, which takes milliseconds, be on the same scale as the task of factoring large numbers, which can take years? Thus it is sometimes better to consider time relative to some yardstick, and use different yardsticks for different jobs.

Being productive is good

A system's client is typically only concerned with his own work, and wants it completed as fast as possible. But from the system's perspective the sum of all clients is typically more important than any particular one of them. Making many clients moderately happy may therefore be better than making one client very happy, at the expense of others.

The metric of "happy clients per second" is called throughput. Formally, this is the number of jobs done in a unit of time. It should be noted that these can be various

types of jobs, e.g. applications being run, files being transferred, bank accounts being updated, etc.

Exercise 146 *Are response time and throughput simply two facets of the same thing?*

Being busy is good

Finally, another common metric is utilization. This metric is especially common when discussing systems, as it represents the system owner's point of view: if the system is being utilized, we are getting our money's worth. However, as we'll see below, there is often a tradeoff between utilization and responsiveness. If the system is driven to very high utilization, the average response time (the time from when a request arrives until it is done) rises precipitously.

Being up is good

The above metrics are concerned with the amount of useful work that gets done. There is also a whole group of metrics related to getting work done at all. These are metrics that measure system availability and reliability. For example, we can talk about the mean time between failures (MTBF), or the fraction of time that the system is down.

A special case in this class of metrics is the supportable load. Every system becomes overloaded if it is subjected to extreme load conditions. The question is, first, at what level of load this happens, and second, the degree to which the system can function under these conditions.

Keeping out of the way is good

A major goal of the operating system is to run as little as possible, and enable user jobs to use the computer's resources. Therefore, when the operating system does run, it should do so as quickly and unobtrusively as possible. In other words, the operating system's *overhead* should be low. In addition, it should not hang or cause other inconveniences to the users.

Exercise 147 *Which of the following metrics is related to time, throughput, utilization, or reliability?*

- *The probability that a workstation is idle and can therefore execute remote jobs*
- *The response time of a computer program*
- *The probability that a disk fails*
- *The bandwidth of a communications network*
- *The latency of loading a web page*
- *The number of transactions per second processed by a database system*

10.2 Workload Considerations

Workload = sequence of things to do

The workload on a system is the stream of jobs submitted to the system. From the system's point of view, the important attributes of the workload are when each job arrives, and what resources it needs. Similar considerations apply to subsystems. For example, for the file system the workload is the sequence of I/O operations, and the most important attribute is the number of bytes accessed.

Exercise 148 *What characterizes the workload on a memory subsystem?*

For reactive systems, the workload is the input

In algorithm design, we know that sometimes there is a significant gap between the worst-case behavior and the average behavior. For example, quick-sort has a worst-case cost of $O(n^2)$, but on average it is $O(n \log n)$ with a rather low constant, making it better than other algorithms that have a worst-case cost of $O(n \log n)$. The observed behavior depends on the input in each instance.

Similarly, the performance of a system depends not only on the system design and implementation but also on the workload to which it is subjected. We will see that in some cases one design is better for one workload, while a different design is better for another workload.

Performance evaluation must therefore be done subject to the results of a detailed workload analysis. Usually this is based on past workload measurements, which are used to create a workload model. Alternatively, the recorded workload from the past can be used directly to drive the evaluation of a new design.

Exercise 149 *You are designing a kiosk that allows people to check their email. You are told to expect some 300 users a day, each requiring about 2 minutes. Based on this you decide to deploy two terminals (there are 600 minutes in 10 hours, so two terminals provide about double the needed capacity — a safe margin). Now you are told that the system will be deployed in a school, for use during recess. Should you change your design?*

There are few benchmarks

Computer architecture design also depends on detailed analysis that is based on the workload. In that case, the workload used is a canonized set of applications that are recognized as being representative of general workloads; such selected applications are called benchmarks. A well-known example is the SPEC benchmarks, which are updated every few years [17].

For operating systems there are few such agreed benchmarks. In addition, creating a representative workload is more difficult because it has a *temporal* component.

This means that in a typical system additional work arrives unexpectedly at different times, and there are also times at which the system is idle simply because it has nothing to do. Thus an important component of modeling the workload is modeling the arrival process (as in the bursty example in Exercise 149). By contradistinction, in computer architecture studies all the work is available at the outset when the application is loaded, and the CPU just has to execute all of it as fast as possible.

10.2.1 Statistical Characterization of Workloads

The workload is randomly sampled from a distribution

One view of the workload on a system is that there exists a population of possible jobs to do, and every item is sampled at random from this population. The population is characterized by one or more distributions. For example, an important attribute of jobs is their runtime, and we can consider jobs as being sampled from a distribution of possible runtimes. By characterizing the distribution (e.g. the distribution of runtimes), we characterize the workload.

In many cases, the distributions turn out to be "a lot of low values and a few high values". Moreover, the high values are sometimes *VERY* HIGH, and there are enough of them to make a difference. For example, consider the distribution of file sizes. Most files are very small, no more than a few dozen bytes. But some files are extremely big, spanning several gigabytes. While the probability of having a very large file is small, e.g. 0.0001, it is not negligible: in a system with 10,000 files we will probably have at least one such file. And because of its size, its disk space consumption is equivalent to that of a very large number of the smaller files.

Technically speaking, such distributions are said to posses a "fat tail". Examples include the Pareto distribution and the lognormal distribution. If you have not heard of them, it is probably due to the regrettable tendency of introductory probability and statistics courses to focus on distributions that have simple mathematical properties, rather than on distributions which are known to provide a good representation of real experimental data.

Details: fat-tailed distributions

Fat tailed distributions are somewhat counter-intuitive, so we will start with some examples. Note that this discussion relates to distributions on positive numbers, that only have a right tail; negative numbers are typically meaningless in our applications (a file cannot have a negative size, and a job cannot have a negative runtime).

The distribution of the tail is fatter than the distribution as a whole

Consider the distribution of job runtimes, and ask the following question: given that a job has already run for time t, how much longer do you expect it to run?

The answer depends on the distribution of runtimes. A trivial case is when all jobs run for exactly T seconds. In this case, a job that has already run for t seconds has another $T - t$ seconds to run. In other words, the longer you have waited already, and less additional time you expect to have to wait for the job to complete. This is true not only for this trivial case, but for all distributions that do not have much of a tail.

The boundary of such distributions is the exponential distribution, defined by the pdf $f(x) = \lambda e^{-\lambda x}$. This distribution has the remarkable property of being memoryless. Its mean is $1/\lambda$, and this is *always* the value of the time you can expect to wait. When the job just starts, you can expect to wait $1/\lambda$ seconds. 7 seconds later, you can expect to wait an additional $1/\lambda$ seconds. And the same also holds half an hour later, or the next day. In other words, if you focus on the tail of the distribution, its shape is identical to the shape of the distribution as a whole.

fat-tailed distributions are even more counter-intuitive. They are characterized by the fact that the more you wait, *the more additional time* you should expect to wait. In other words, if you focus on the tail of the distribution, its shape has a fatter tail than the distribution as a whole.

Exercise 150 *You are designing a system that handles overload conditions by actively killing one job, in the hope of reducing the load and thus being able to better serve the remaining jobs. Which job should you select to kill, assuming you know the type of distribution that best describes the job sizes?*

Formal definition of heavy tails

The above can be formalized mathematically using the definition of a *heavy tail*. A distribution is said to posses a heavy tail if its tail decays according to a power law. This means that the probability of seeing very large values grows smaller and smaller, but not as fast as an exponential reduction. The equation is

$$\Pr[X > x] \approx x^{-\alpha} \qquad 0 < \alpha < 2$$

The simplest example of such a distribution is the Pareto distribution, which we discuss below.

But real-life is not formal

There are two main problems with applying this formal definition to real-life situations.

First, the definition applies to values of x tending to infinity. In real life, everything is bounded to relatively small values. For example, we are typically not interested in jobs that run for more than a year (a mere 31,536,000 seconds), or in files of more than a terabyte or so; the highest values seen in most systems are considerably smaller.

Second, it is typically very hard to assess whether or not the tail really decays according to a power law: there are simply not enough observations. And from a practical point

of view, it typically doesn't really matter. Therefore we prefer not to get into arguments about formal definitions.

What does indeed matter is that there is a non-negligible probability to encounter extremely high values. For example, in an exponential distribution the probability of seeing a value that is 100 times (or more) larger than the mean is less than 10^{-43}. This is negligible, and can safely be ignored. In a fat-tailed distribution this probability can be as high as 10^{-5}. While still very small, this is non-negligible, and any non-trivial system can expect to encounter events with such a probability. In order to avoid formal arguments, we will therefore generally talk about fat-tailed distributions, and not claim that they necessarily conform with the formal definition of having a "heavy tail".

Example distributions

The Pareto distribution is defined on the range $x > 1$, and has a simple power-law CDF:

$$F(x) = 1 - x^{-a}$$

where a must be positive and is called the *shape parameter* — the lower a is, the heavier the tail of the distribution. In fact, the distribution only has a mean if $a > 1$ (otherwise the calculation of the mean does not converge), and only has a variance if $a > 2$ (ditto). The pdf is

$$f(x) = ax^{-(a+1)}$$

This is the simplest example of the group of heavy-tail distributions.

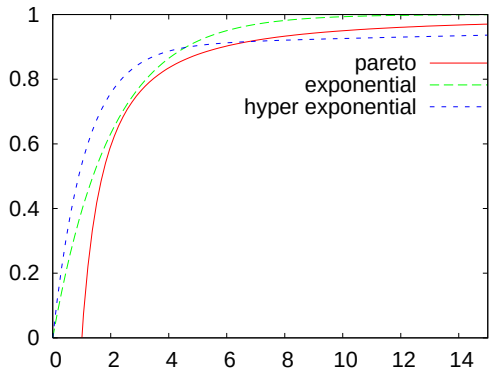
Exercise 151 *What is the mean of the Pareto distribution when it does exist?*

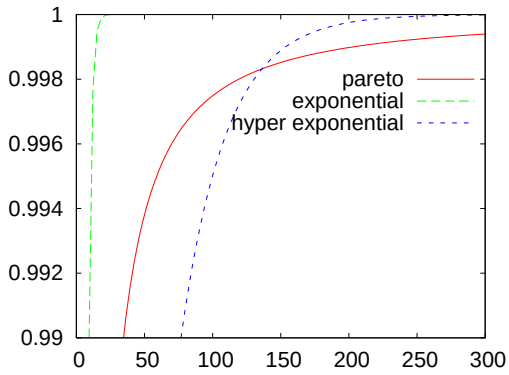
For modeling purposes, it is common to use a hyper-exponential distribution. This is the probabilistic combination of several exponentials, and can be tailored to mimic different tail shapes. For example, a two-stage hyper-exponential is

$$f(x) = p\lambda_1 e^{-\lambda_1 x} + (1-p)\lambda_2 e^{-\lambda_2 x}$$

for some $0 < p < 1$. By a judicious choice of λ_1, λ_2, and p, one can create a distribution in which the standard deviation is as large as desired relative to the mean, indicating a fat tail (as opposed to the exponential distribution, in which the standard deviation is always equal to the mean). However, this is not a heavy tail.

The following graphs compare the CDFs of the exponential, Pareto, and hyper-exponential distributions. Of course, these are special cases, as the exact shape depends on the parameters chosen for each one. The right graph focuses on the tail.





In many cases, the important characteristic of the workload is not the mathematical description of the tail, but the phenomenon of *mass-count disparity*. This means that there are many small elements, and a few huge ones, and moreover, that most of the "mass" is in the huge ones (technically, it means that if we integrate the distribution's pdf, much of the sum comes from the tail). Instances of workloads that are often found to be fat-tailed and exhibit significant mass-count disparity include the following:

- The distribution of job runtimes. In this case, most of the jobs are short, but most of the CPU time is spent running long jobs.
- The distribution of file sizes in a file system. Here most of the files are small, but most of the disk space is devoted to storing large files.
- The distribution of flow sizes on the Internet. As with files, most flows are short, but most bytes transferred belong to long flows.

We will consider the details of specific examples when we need them.

Exercise 152 *The distribution of traffic flows in the Internet has been characterized as being composed of "mice and elephants", indicating many small flows and few big ones. Is this a good characterization? Hint: think about the distribution's modes.*

A special case of fat tailed distributions is the Zipf distribution. This distribution applies to a surprising number of instances of ranking by popularity, e.g. the popularity of web pages.

Details: the Zipf distribution

If the items being considered are ordered by rank, Zipf's Law postulates that

$$\Pr(x) \propto \frac{1}{x}$$

that is, the probability of using each item is inversely proportional to its rank: if the most popular item is used k times, the second most popular is used $k/2$ times, the third $k/3$

times, and so on. Note that we can't express this in the conventional way using a probability density function, because $\int \frac{1}{x} dx = \infty$! The only way out is to use a normalizing factor that is proportional to the number of observations being made. Assuming N observations, the probability of observing the item with rank x is then

$$f(x) = \frac{1}{x \ln N}$$

and the CDF is

$$F(x) = \frac{\ln x}{\ln N}$$

To read more: Advanced books on performance evaluation typically include some treatment of the above distributions. Examples include those by Jain [7] and Law and Kelton [11]. The volume edited by Adler et al. contains a collection of chapters on the subject, but fails to provide a basic and cohesive introduction [1]. Relatively readable papers on heavy tails include those by Crovella [2] and Downey [4]. Mass-count disparity is described by Feitelson [5]. Zipf's Law was introduced in 1949 based on empirical evidence regarding the relative frequency in which different words occur in written language [18].

Workload attributes may be inter-dependent

A problem with characterizing the workload only by the distributions is that the different workload attributes may be correlated with each other. For example, it may be that jobs that run longer also use more memory.

The conventional probabilistic definition of correlation measures the *linear* dependence between two random variables: a high correlation indicates that when one is above its mean, so it the other, and by approximately the same degree. Again, in real life things can be more complicated and harder to define.

Exercise 153 *Would you expect CPU usage and I/O activity to be correlated with each other?*

10.2.2 Workload Behavior Over Time

Long-range dependence is common

A special case of dependence in workloads is dependence along time, rather than among different parameters. For example, the load on the system at time t may be correlated to the load at time $t + \delta$ for some δ. In fact, empirical evidence shows that such correlations are quite common, and exist over long time frames.

The results of such long-range dependence is that the workload is bursty. In other words, workloads are usually not spread out evenly, but rather come in bursts. This occurs at many time scales, from sub-second to days and weeks. As a result the fluctuation in the workload observed during short periods may look similar to the fluctuations over long periods, an effect known as *self-similarity*. This is discussed briefly in Appendix D.

And there may be a local structure

While random sampling is a convenient model, it does not capture the dynamics of real workloads. In a real system, the workload over a relatively short time frame may be substantially different from the average of a whole year. For example, the characteristics of the load on a server in the Computer Science department may change from week to week depending on the programming exercise that everyone is working on that week.

In addition, workloads often display a daily cycle as a result of being generated by human beings who go to sleep at night. And there are also weekly and even yearly cycles (e.g. the workload in late December may be much lower than average).

Exercise 154 *What is the significance of the daily cycle in each of the following cases?*

1. *Access to web pages over the Internet*
2. *Usage of a word processor on a privately owned personal computer*
3. *Execution of computationally heavy tasks on a shared system*

10.3 Analysis, Simulation, and Measurement

There are three main approaches to get a handle on performance issue:

Analysis — use mathematical analysis from first principles to evaluate the system.

Simulation — simulate the system's operation. This is akin to measuring a small-scale partial implementation.

Measurement — implement the system in full and measure its performance directly.

Analysis provides insights

The major benefit of using mathematical analysis is that it provides the best insights: the result of analysis is a functional expression that shows how the performance depends on system parameters. For example, you can use analysis to answer the question of whether it is better to configure a system with one fast disk or two slower disks. We will see an example of this below, in Section 10.5.

Exercise 155 *Consider performance metrics like job response time and network bandwidth. What are system parameters that they may depend on?*

The drawback of analysis is that it is hard to do, in the sense that it is not always possible to arrive at a closed-form solution. Even when it is possible, various approximations and simplifications may have to be made in the interest of mathematical tractability. This reduces the confidence in the relevance of the results, because the simplifications may create unrealistic scenarios.

Simulation is flexible

The main advantage of simulation is its flexibility: you can make various modifications to the simulation model, and check their effect easily. In addition, you can make some parts of the model more detailed than others, if they are thought to be important. In particular, you do not need to make simplifying assumptions (although this is sometimes a good idea in order to get faster results). For example, you can simulate the system with either one or two disks, at many different levels of detail; this refers both to the disks themselves (e.g. use an average value for the seek time, or a detailed model that includes the acceleration and stabilization of the disk head) and to the workload that they serve (e.g. a simple sequence of requests to access consecutive blocks, or a sequence of requests recorded on a real system).

The drawback of simulations is that they are often perceived to be unreliable. After all, this is not a full system implementation but only the parts you think are important. But what if there are some effects you didn't think about in advance?

Measurements are convincing

The most convincing approach is to measure a real system. However, this only provides a single data point, and is not useful to compare different approaches or different configurations. For example, you can get detailed data about how the time for an I/O operation depends on the number of bytes accessed, but only for your current system configuration.

The perception of being "the real thing" may also be misguided. It is not easy to measure the features of a complex system. In many cases, the performance is a function of many different things, and it is impossible to uncover each one's contribution. Moreover, it often happens that the measurement is affected by factors that were not anticipated in advance, leading to results that don't seem to make sense.

Exercise 156 *How would you go about measuring something that is very short, e.g. the overhead of trapping into the operating system?*

In the end, simulation is often the only viable alternative

It would be wrong to read the preceding paragraphs as if all three alternatives have equal standing. The bottom line is that simulation is often used because it is the only viable alternative. Analysis may be too difficult, or may require too many simplifying assumptions. Once all the assumptions are listed, the confidence in the relevance of the results drops to the point that they are not considered worth the effort. And difficulties arise not only from trying to make realistic assumptions, but also from size. For example, it may be possible to analyze a network of half a dozen nodes, but not one with thousands of nodes.

Measurement is sometimes impossible because the system does not exist or it would take too much time. In other cases it is irrelevant because we cannot change the configuration as desired.

10.4 Modeling: the Realism/Complexity Tradeoff

Except for direct measurements, performance evaluations depend on a model of the studied system. This model can be made as detailed as desired, so as to better reflect reality. However, this leads to two problems. First, a more detailed model requires more data about the workload, the environment, and the system itself, data that is not always available. Second, a more detailed model is naturally harder to evaluate.

Statics are simpler than dynamics

One way to simplify the model is to use a static workload rather than a dynamic one. For example, a scheduler can be evaluated by how well it handles a given job mix, disregarding the changes that occur when additional jobs arrive or existing ones terminate.

The justification is that the static workload is considered to be a snapshot of the real dynamic workload. By freezing the dynamic changes in the workload, one saves the need to explicitly model these dynamics. By repeatedly evaluating the system under different static workloads, one may endeavor to capture the behavior of the system as it would be in different instants of its dynamic evolution.

But real systems are typically dynamic

The problem with using static workloads is that this leads to lesser accuracy and less confidence in the evaluations results. This happens because incremental work, as in a dynamically evolving real system, modifies the conditions under which the system operates. Creating static workloads may miss such effects.

For example, the layout of files on a disk is different if they are stored in a disk that was initially empty, or in a disk that has been heavily used for some time. When storing data for the first time in a new file system, blocks will tend to be allocated consecutively one after the other. Even if many different mixes of files are considered, they will each lead to consecutive allocation, because each time the evaluation starts from scratch. But in a real file system after heavy usage — including many file deletions — the available disk space will become fragmented. Thus in a live file system there is little chance that blocks will be allocated consecutively, and evaluations based on allocations that start from scratch will lead to overly optimistic performance results. The solution in this case is to develop a model of the steady state load on a disk, and use this to prime each evaluation rather than starting from scratch [15].

And of course distributions are important too

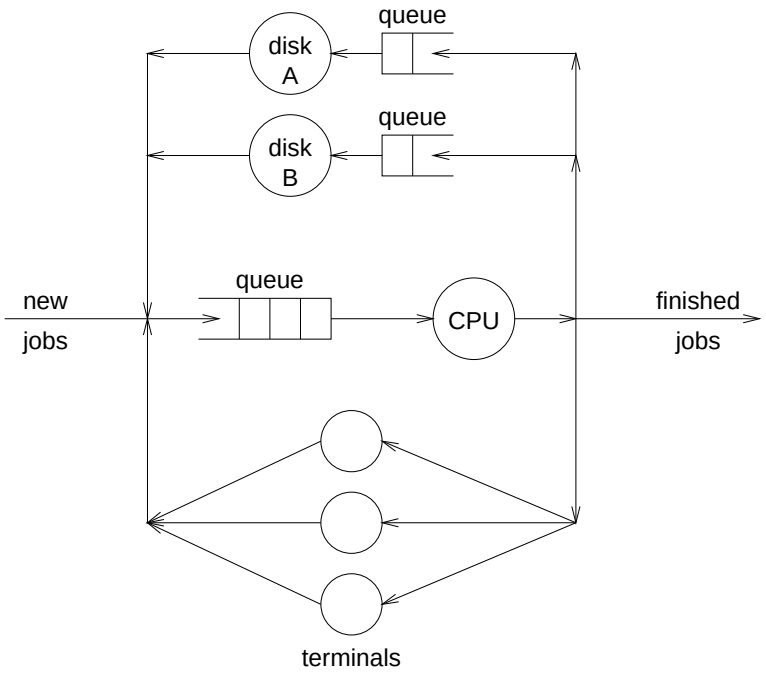
Another aspect of the realism/complexity tradeoff is the choice of distributions used to represent the workload. The whole previous section on workloads is actually about the need to create more complex and realistic models. If simple distributions are chosen, this may adversely affect the validity of the evaluation results.

10.5 Queueing Systems

10.5.1 Waiting in Queues

A system is composed of queues and servers

In real life, you often need to wait in queue for a service: there may be people ahead of you in the post office, the supermarket, the traffic light, etc. Computer systems are the same. A program might have to wait when another is running, or using the disk. It also might have to wait for a user to type in some input. Thus computer systems can be viewed as networks of queues and servers. Here is an example:



In this figure, the CPU is one service station, and there is a queue of jobs waiting for it. After running for some time, a job may either terminate, require service from one of two disks (where it might have to wait in queue again), or require input. Input terminals are modeled as a so called “delay center”, where you don’t have to wait in a queue (each job has its own user), but you do need to wait for the service (the user’s input).

Exercise 157 *Draw queuing networks describing queueing in a supermarket, a bank, and a cafeteria.*

Events happen at random times

It is most realistic to model arrival times and service times as random processes. This means that when we say that "on average, there are 3 jobs per minute", we do *not* mean that jobs arrive exactly 20 seconds apart. On the contrary, in some cases they may come 2 seconds apart, and in some cases 3 minutes may go by with no new job. The probability of this happening depends on the distribution of interarrival times.

Randomness is what makes you wait in queue

The random nature of arrivals and service times has profound implications on performance.

Consider an example where each job takes exactly 100 ms (that is, one tenth of a second). Obviously, if exactly one such job arrives every second then it will be done in 100 ms, and the CPU will be idle 90% of the time. If jobs arrive exactly half a second apart, they still will be serviced immediately, and the CPU will be 80% idle. Even if these jobs arrive each 100 ms they can still be serviced immediately, and we can achieve 100% utilization.

But if jobs take 100 ms *on average*, it means that some may be much longer. And if 5 such jobs arrive each second *on average*, it means that there will be seconds when many more than 5 arrive together. If either of these things happens, jobs will have to await their turn, and this may take a long time. It is not that the CPU cannot handle the load on average — in fact, it is 50% idle! The problem is that it cannot handle multiple jobs at once when a burst of activity happens at random.

Exercise 158 *Is it possible that a system will not be able to process all its workload, even if it is idle much of the time?*

We would therefore expect the average response time to rise when the load increases. But how much?

10.5.2 Queueing Analysis

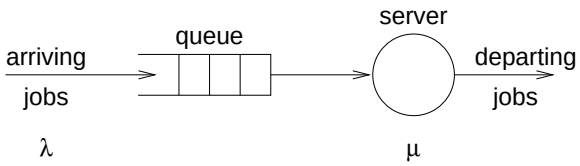
Queueing analysis is used to gain insights into the effect of randomness on waiting time, and show that these effects are derived from basic principles. Similar effects should be observed when using simulations and measurements, assuming they relate to the same system models.

To read more: The following is only a very brief introduction to the ideas of queueing theory. A short exposition on queueing analysis was given in early editions of Stallings [16, appendix A]. A more detailed discussion is given by Jain [7, Part VI]. Krakowiak [10, Chap. 8] bases the

general discussion of resource allocation on queueing theory. Then there are whole books devoted to queueing theory and its use in computer performance evaluation, such as Lazowska et al. [12] and Kleinrock [8, 9].

The simplest system has one queue and one server

Queueing analysis models the system as a collection of queues and servers. For example, a computer that runs jobs one after the other in the order that they arrive is modeled as a queue (representing the ready queue of waiting jobs) followed by a server (representing the CPU).



The main parameters of the model are the arrival rate and the service rate.

The *arrival rate*, denoted by λ, is the average number of clients (jobs) arriving per unit of time. For example, $\lambda = 2$ means that on average two jobs arrive every second. It also means that the average *interarrival time* is half a second.

The *service rate*, denoted by μ, is the average number of clients (jobs) that the server can service per unit of time. For example, $\mu = 3$ means that on average the server can service 3 jobs per second. It also means that the average *service time* is one third of a second.

The number of clients that the server actually serves depends on how many arrive. If the arrival rate is higher than the service rate, the queue will grow without a bound, and the system is said to be *saturated*. A *stable* system, that does not saturate, requires $\lambda < \mu$. The load or utilization of the system is $\rho = \lambda/\mu$.

Lots of interesting things can be measured

While only two numerical parameters are used, many different metrics can be quantified. A partial list of quantities that are commonly studied is

- The waiting time w.
- The service time s. According to our previous definition, $E[s] = 1/\mu$.
- The response time $r = w+s$. This is often the most direct metric for performance.
- the number of jobs in the system n (including those being serviced now). This is important in order to assess the size of system buffers and tables.

Note that the process by which jobs arrive at the queue and are serviced is a *random* process. The above quantities are therefore *random variables*. What we want to find out is usually the average values of metrics such as the response time and number of jobs in the system. We shall denote averages by a bar above the variable, as in $\bar{n}$.

Little's Law provides an important relationship

An important relationship between the above quantities, known as Little's Law, states that

$$\bar{n} = \lambda \cdot \bar{r}$$

Intuitively, this means that if λ jobs arrive each second, and they stay in the system for r seconds each, we can expect to see $\lambda \cdot r$ jobs in the system at any given moment. As a concrete example, consider a bar where 6 new customers arrive (on average) every hour, and each stays in the bar for (an average of) 3 hours. The number of customers we may expect to find in the bar is then 18: the 6 that arrived in the last hour, the 6 that arrived an hour earlier, and the 6 that arrived 2 hours ago. Any customers that arrived earlier than that are expected to have departed already.

This relationship is very useful, because if we know λ, and can find $\bar{n}$ from our analysis, then we can compute $\bar{r}$, the average response time, which is the metric for performance.

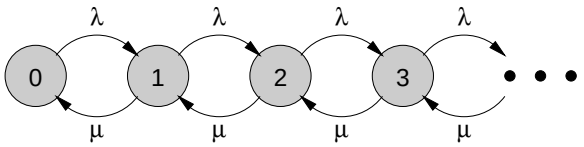
Exercise 159 *Can you derive a more formal argument for Little's Law? Hint: look at the cumulative time spent in the system by all the jobs that arrive and are serviced during a long interval T.*

The classic example is the M/M/1 queue

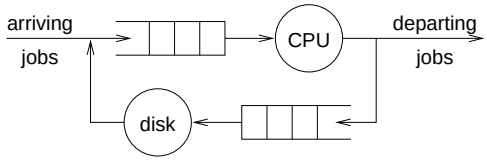
The simplest example is the so-called M/M/1 queue. This is a special case of the arrive-queue-server-done system pictured above. The first M means that interarrival times are exponentially distributed (the "M" stands for "memoryless"). The second M means that service times are also exponentially distributed. The 1 means that there is only one server.

Details of the analysis

The way to analyze such a queue (and indeed, more complicated queueing systems as well) is to examine its *state space*. For an M/M/1 queue, the state space is very simple. The states are labeled by integers 0, 1, 2, and so on, and denote the number of jobs currently in the system. An arrival causes a transition from state i to state $i + 1$. The average rate at which such transitions occur is simply λ, the arrival rate of new jobs. A departure (after a job is serviced) causes a transition from state i to state $i - 1$. This happens at an average rate of μ, the server's service rate.



Exercise 160 *What is the state space for a system with a CPU and a disk? Hint: think 2D.*



The nice thing about this is that it is a *Markov chain*. The probability of moving from state i to state $i+1$ or $i-1$ does not depend on the history of how you got to state i (in general, it may depend on which state you are in, namely on i. For the simple case of an M/M/1 queue, it does not even depend on i).

An important property of Markov chains is that there is a limiting distribution on the states. This means that if we observe the system for a very long time, we will find that it spends a certain fraction of the time in state 0, a certain fraction of the time in state 1, and so on[1]. We denote these long-term probabilities to be in the different states by π_i, i.e. π_0 will be the prbability to be in state 0, π_1 the probability to be in state 1, etc.

Given the fact that such long-term probabilities exist, we realize that the flow between neighboring states must be balanced. In other words, for every transition from state i to state $i+1$, there must be a corresponding transition back from state $i+1$ to state i. But transitions occur at a known rate, that only depends on the fact that we are in the given state to begin with. This allows us to write a set of equations that express the balanced flow between neighboring states. For example, the flow from state 0 to state 1 is

$$\lambda \, \pi_0$$

because when we are in state 0 the flow occurs at a rate of λ. Likewise, the flow back from state 1 to state 0 is

$$\mu \, \pi_1$$

The balanced flow implies that

$$\lambda \, \pi_0 = \mu \, \pi_1$$

or that

$$\pi_1 = \frac{\lambda}{\mu} \, \pi_0$$

Now let's proceed to the next two states. Again, balanced flow implies that

$$\lambda \, \pi_1 = \mu \, \pi_2$$

which allows us to express π_2 as

$$\pi_2 = \frac{\lambda}{\mu} \, \pi_1$$

Substituting the expression for π_1 we derived above then leads to

[1] Actually a Markov chain must satisfy several conditions for such limiting probabilities to exist; for example, there must be a path from every state to every other state, and there should not be any periodic cycles. For more on this see any book on probabilistic models, e.g. Ross [14]

$$\pi_2 = \left(\frac{\lambda}{\mu}\right)^2 \pi_0$$

and it is not hard to see that we can go on in this way and derive the general expression

$$\pi_i = \rho^i \, \pi_0$$

where $\rho = \frac{\lambda}{\mu}$.

Exercise 161 *What is the meaning of ρ? Hint: it is actually the utilization of the system. Why is this so?*

Given that the probabilities for being in all the states must sum to 1, we have the additional condition that

$$\sum_{i=0}^{\infty} \pi_0 \, \rho^i = 1$$

Taking π_0 out of the sum and using the well-known formula for a geometric sum, $\sum_{i=0}^{\infty} \rho^i = \frac{1}{1-\rho}$, this leads to

$$\pi_0 = 1 - \rho$$

This even makes sense: the probability of being in state 0, where there are no jobs in the system, is 1 minus the utilization.

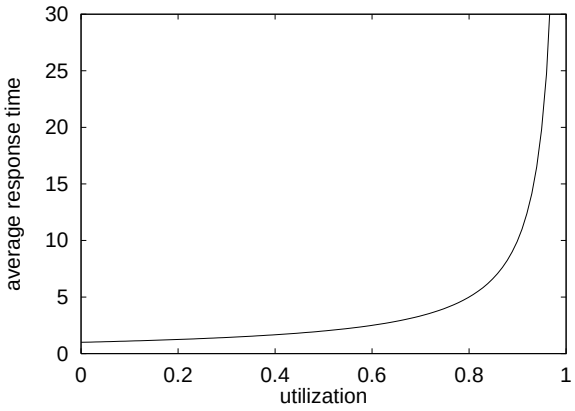
We're actually nearly done. Given the above, we can find the expected number of jobs in the system: it is

$$\begin{aligned} \bar{n} &= \sum_i i\pi_i \\ &= \sum_i i(1-\rho)\rho^i \\ &= \frac{\rho}{1-\rho} \end{aligned}$$

Finally, we use Little's Law to find the expected response time. It is

$$\begin{aligned} \bar{r} &= \frac{\bar{n}}{\lambda} \\ &= \frac{\rho}{\lambda(1-\rho)} \\ &= \frac{1}{\mu - \lambda} \end{aligned}$$

The end result of all this analysis looks like this (by setting $\mu = 1$ and letting λ range from 0 to 1):



For low loads, the response time is good. Above 80% utilization, it becomes very bad.

Exercise 162 *What precisely is the average response time for very low loads? Does this make sense?*

The shape of this graph is characteristic of practically all queueing analyses for open systems. It is a result of the randomness of the arrivals and service times. Because of this randomness, customers sometimes cluster together and have to wait a long time. As the utilization approaches 100%, the system is idle less, and the chances of having to wait increase. Reducing the variability in arrivals and/or service times reduces the average response time.

This analysis is based on a host of simplifying assumptions

You probably didn't notice it, but the above analysis is based on many simplifying assumptions. The most important ones are

- The interarrival times and service times are exponentially distributed.
- The service discipline is FCFS.
- The population of clients (or jobs) is infinite, and the queue can grow to unbounded size.
- At any instant only one arrival or one departure may occur.

Exercise 163 *Can you identify at which point in the derivation each assumption was used?*

Nevertheless, the resulting analysis demonstrates the way in which response time depends on load. It gives a mathematical explanation to the well-known phenomenon that as the system approaches saturation, the queue length grows to infinity. This means that if we want short response times, we must accept the fact that utilization will be less than 100%.

10.5.3 Open vs. Closed Systems

Open systems lack feedback

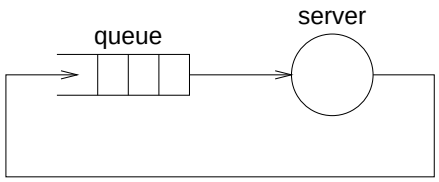
The M/M/1 queue considered above is an example of an *open system*: jobs arrive from an infinite population, pass through the system, and depart. In particular, there is no feedback from the service to the arrivals: new jobs continue to arrive at the same rate even if the service is abysmal.

The open system model is nevertheless quite useful, as situations in which arrivals are independent of the service quality indeed do exist. For example, this model may be very suitable for arrivals of requests at a web server, as such requests can be generated from around the world, and the users generating them may indeed not have prior experience with this server. Thus bad service to current requests does not translate into reduced future requests, because future requests are independent of the current ones. In addition, an open system model is the desired case when the system provides adequate service for the needs of its users.

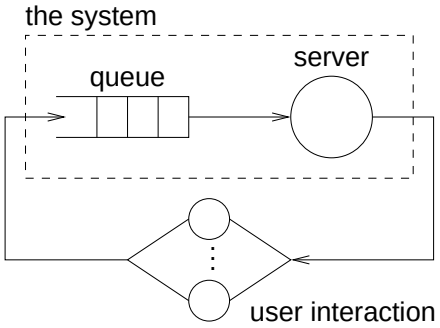
Closed systems model feedback explicitly

However, in many situations requests are not independent. For example, a local computer system may service a relatively small community of users. If these users learn (through experience or word of mouth) that the computer is not providing adequate service, they will probably stop submitting jobs.

Such scenarios are modeled by closed systems, in which the population of jobs is finite, and they continuously circulate through the system. In the basic, "pure" closed case, every job termination is immediately translated into a new job arrival:



A more realistic model is an interactive system, where the return path goes through a user-interaction component; in this case the number of jobs actually in the system may fluctuate, as different numbers may be waiting for user interaction at any given instance.



Exercise 164 *Consider the following scenarios. Which is better modeled as an open system, and which as a closed system?*

1. *A group of students have to complete an exercise in the operating systems course by 10 PM tonight.*
2. *A group of professors are trying to use the departmental server for non-urgent administrative chores related to courses they teach.*
3. *Tourists strolling by an information kiosk in a museum stop by to get some information.*

The metrics are different

As we saw, the performance of an open system like the M/M/1 queue can be quantified by the functional relationship of the response time on the load. For closed systems, this is irrelevant. A simple closed system like the one pictured above operates at full capacity all the time (that is, at the coveted 100% utilization), because whenever a job terminates a new one arrives to take its place. At the same time, it does not suffer from infinite response times, because the population of jobs is bounded.

The correct metrics for closed systems are therefore throughput metrics, and not response time metrics. The relevant question is how many jobs were completed in a unit of time, or in other words, how many cycles were completed.

10.6 Simulation Methodology

Analytical models enable the evaluation of simplified mathematical models of computer systems, typically in steady state, and using restricted workload models (e.g. exponential distributions). Simulations are not thus restricted — they can include whatever the modeler wishes to include, at the desired level of detail. Of course this is also a drawback, as they do not include whatever the modeler did not consider important.

To read more: Again, we only touch upon this subject here. Standard simulation methodology is covered in many textbooks, e.g. Jain [7, Part V], as well as textbooks dedicated to the topic such as Law and Kelton [11]. The issues discussed here are surveyed in [13].

10.6.1 Incremental Accuracy

One way to use simulations is to evaluate systems that cannot be solved analytically, but retaining the framework of steady state conditions. This means that the system is simulated until the measured features converge.

First you need to get into steady state

In order to simulate the system in a steady state, you must first ensure that it is in a steady state. This is not trivial, as simulations often start from an empty system. Thus the first part of the simulation is often discarded, so as to start the actual evaluation only when the system finally reaches the desired state.

For example, when trying to evaluate the response time of jobs in an open system, the first jobs find an empty system (no need to wait in queue), and fully available resources. Later jobs typically need to wait in the queue for some time, and the resources may be fragmented.

The decision regarding when steady state is achieved typically also depends on the convergence of the measured metrics.

The simulation length is determined by the desired accuracy

Once the steady state is reached, the simulation is continued until the desired accuracy is reached. As the simulation unfolds, we sample more and more instances of the performance metrics in which we are interested. For example, as we simulate more and more jobs, we collect samples of job response times. The average value of the samples is considered to be an estimator for the true average value of the metric, as it would be in a real system. The longer the simulation, the more samples we have, and the more confidence we have in the accuracy of the results. This derives from the fact that the size of the confidence interval (the range of values in which we think the true value resides with a certain degree of confidence) is inversely proportional to the square root of the number of samples.

Exercise 165 *What sort of confidence levels and accuracies are desirable in systems evaluation?*

10.6.2 Workloads: Overload and (Lack of) Steady State

While the simulation of systems at steady state is definitely useful, it is not clear that it captures the whole picture. In fact, are real systems ever in "steady state"?

Extreme conditions happen

It is well known that the largest strain on the worldwide telecommunications infrastructure occurs each year on Mother's Day. Likewise, extreme loads on the Internet have occurred during live-cast events ranging from rock concerts to Victoria's Secret fashion shows. Thus it would seem that systems actually have to be evaluated in two different modes: the steady state, in which "normal" conditions are studied, and the metrics are typically related to average response time, and overload conditions, in which the metrics are typically related to throughput. Note that paradoxically closed systems are the ones that model overload, despite the fact that overload results from

an effectively infinite population of clients that is unaware of the system conditions. This is so because whenever the system manages to get rid of a customer, another immediately comes in.

Naturally, one of the most important metrics for a system is the point at which it makes the transition from functioning to overloaded. This is also related to the question of the distribution of loads: is this a bimodal distribution, with most workloads falling in the well-behaved normal mode, and just some extreme discrete cases creating overload, or is there a continuum? The answer seems to be a combination of both. The load experienced by most systems is not steady, but exhibits fluctuations at many different time scales. The larger fluctuations, corresponding to the more extreme loads, occur much more rarely than the smaller fluctuations. The distinction into two modes is not part of the workload, but a feature of the system: above some threshold it ceases to handle the load, and the dynamics change. But as far as the workload itself is concerned, the above-threshold loads are just a more extreme case, which seems to be discrete only because it represents the tail of the distribution and occurs relatively rarely.

Simulations results are limited to a certain timeframe

What is the implication to simulations? The problem with extreme conditions is that they occur very rarely, because they are from the tail of the distribution of possible events. Thus if you observe the behavior of a real system over, say, one month, you *might* chance upon such an event, but most probably you will not. A good simulation would be the same: if you simulate a month, you might sample a high-load event, but most probably you will not. Thus your results are guaranteed to be wrong: if you do not have a high-load event, you fail to predict the outcome and effect of such events, and if you do, you fail to predict the normal behavior of the system.

The best solution to this problem is to acknowledge it. Simulations with bursty workloads should explicitly define a time horizon, and prohibit events that are not expected to occur within this time. The results are then at least relevant for an "average" time window of this duration [3].

Rare-event simulation techniques can be used to evaluate extreme cases

But what if the rare events are actually what we are interested in? For example, when evaluating communication networks, we are not interested only in the average latency of messages under normal conditions. We also want to know what is the probability of a buffer overflow that causes data to be lost. In a reasonable network this should be a rather rare event, and we would like to estimate just how rare.

Exercise 166 *What are other examples of such rare events that are important for the evaluation of an operating system?*

If, for example, the events in which we are interested occur with a probability of one in a million (10^{-6}), we will need to simulate billions of events to get a decent measurement of this effect. And 99.9999% of this work will be wasted because it is just filler between the events of interest. Clearly this is not an efficient way to perform such evaluations.

An alternative is to use rare event simulation techniques. This is essentially a combination of two parts: one is the assessment of how often the rare events occur, and the other is what happens when they do. By combining these two parts, one can determine the effect of the rare events on the system. The details of how to do this correctly and efficiently are non-trivial, and not yet part of standard methodology [6].

10.7 Summary

Performance is an important consideration for operating systems. It is true that the main consideration is functionality — that is, that the system will actually work. But it is also important that it will work efficiently and quickly.

The main points made in this chapter are:

- Performance evaluation must be done subject to realistic workloads. Workload analysis and modeling are indispensable. Without them, the results of a performance evaluation are largely meaningless.

  In particular, realistic workloads are bursty and are characterized by heavy tailed distributions. They often do not conform to mathematically nice distributions such as the exponential and normal distributions.

- In many senses, operating systems are queueing systems, and handle jobs that wait for services. In open systems, where the job population is very big and largely independent of system performance, this means that

  1. Randomness in arrival and service times causes the average response time to tend to infinity as the load approaches each sub-system's capacity.
  2. This is a very general result and can be derived from basic principles.
  3. You cannot achieve 100% utilization, and in fact might be limited to much less.

  In closed systems, on the other hand, there is a strong feedback from the system's performance to the creation of additional work. Therefore the average response time only grows linearly with the population size.

Disclaimer

Performance is paramount in system design, and research papers on computer systems always proudly exhibit a section professing the promising results of a performance evaluation. But all too often this is mainly a *characterization* of a proposed

system, not a deep evaluation of alternatives and tradeoffs. In addition, comparisons are hampered by the lack of agreed test conditions, metrics, and workloads.

In particular, the study of systems working in overloaded conditions, and of the characteristics and effect of workloads on a system's behavior, is in its infancy. Thus most of these notes do not make enough use of the issues discussed in this chapter — not enough research on these issues has been performed, and too little empirical data is available.

Bibliography

[1] R. J. Adler, R. E. Feldman, and M. S. Taqqu (eds.), *A Practical Guide to Heavy Tails*. Birkhäuser, 1998.

[2] M. E. Crovella, "*Performance evaluation with heavy tailed distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–10, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[3] M. E. Crovella and L. Lipsky, "*Long-lasting transient conditions in simulations with heavy-tailed workloads*". In *Winter Simulation conf.*, Dec 1997.

[4] A. B. Downey, "*Lognormal and Pareto distributions in the Internet*". *Comput. Commun.* **28(7)**, pp. 790–801, May 2005.

[5] D. G. Feitelson, "*Metrics for mass-count disparity*". In 14th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[6] P. Heidelberger, "*Fast simulation of rare events in queueing and reliability models*". *ACM Trans. Modeling & Comput. Simulation* **5(1)**, pp. 43–85, Jan 1995.

[7] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[8] L. Kleinrock, *Queueing Systems, Vol I: Theory*. John Wiley & Sons, 1975.

[9] L. Kleinrock, *Queueing Systems, Vol II: Computer Applications*. John Wiley & Sons, 1976.

[10] S. Krakowiak, *Principles of Operating Systems*. MIT Press, 1988.

[11] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw Hill, 3rd ed., 2000.

[12] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[13] K. Pawlikowski, "*Steady-state simulation of queueing processes: a survey of problems and solutions*". *ACM Comput. Surv.* **22(2)**, pp. 123–170, Jun 1990.

[14] S. M. Ross, *Introduction to Probability Models*. Academic Press, 5th ed., 1993.

[15] K. A. Smith and M. I. Seltzer, "*File system aging—increasing the relevance of file system benchmarks*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, Jun 1997.

[16] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[17] "*Standard performance evaluation corporation*". URL http://www.spec.org.

[18] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Appendix D

Self-Similar Workloads

Traditionally workload models have used exponential and/or normal distributions, mainly because of their convenient mathematical properties. Recently there is mounting evidence that fractal models displaying self-similarity provide more faithtul representations of reality.

D.1 Fractals

Fractals are geometric objects which have the following two (related) properties: they are self-similar, and they do not have a characteristic scale.

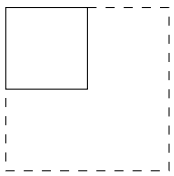
The fractal dimension is based on self-similarity

Being self similar means that parts of the object are similar (or, for pure mathematical objects, identical) to the whole. If we enlarge the whole object we end up with several copies of the original. That is also why there is no characteristic scale — it looks the same at every scale.

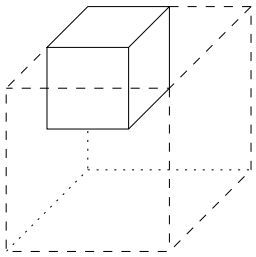
The reason they are called fractals is that they can be defined to have a fractional dimension. This means that these objects fill space in a way that is different from what we are used to. To explain this, we first need to define what we mean by "dimension".

Consider a straing line segment. If we double it, we get two copies of the original:

If we take a square, which is composed of 4 lines, and double the size of each of them, we get four copies of the original:



For a cube (which is composed of 12 line segments), doubling each one creates a structure that contains 8 copies of the original:

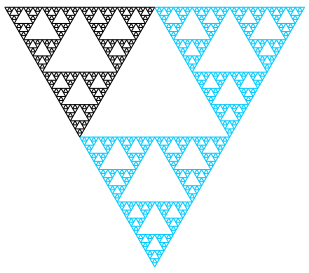


Let's denote the factor by which we increase the size of the line segments by f, and the number of copies of the original that we get by n. The above three examples motivate us to define dimensionality as

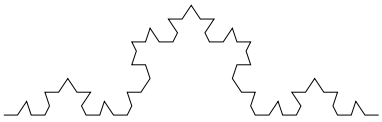
$$d = \log_f n$$

With this definition, the line segment is one-dimentional, the square is 2D, and the cube 3D.

Now apply the same definition to the endlessly recursive Sierpinski triangle. Doubling each line segment by 2 creates a larger triangle which contains 3 copies of the original. Using our new definitions, its dimention is therefore $\log_2 3 = 1.585$. It is a fractal.



Exercise 167 *What is the dimension of the Koch curve?*



Self-similar workloads have fractal characteristics

Self similarity in workloads is similar to the above: it means that the workload looks the same in different time scales.

More formally, consider a time series $x_1, x_2, x_3, \ldots x_n$. x_i can be the number of packets transfered on a network in a second, the number of files opened in an hour, etc. Now create a series of disjoint sums. For example, you can sum every 3 consecutive elements from the original series: $x_1^3 = x_1 + x_2 + x_3, x_2^3 = x_4 + x_5 + x_6, \ldots$. This can be done several times, each time summing consecutive elements from the previous series. For example, the third series will start with $x_1^9 = x_1^3 + x_2^3 + x_3^3 = x_1 + \ldots + x_9$. If all these series look the same, we say that the original series is self similar.

A major consequence of the self-similarity of workloads is that they have a bursty nature, which does not average out. They have random fluctuation at a fine timescale, and this turns into bursts of higher activity at the longer time scales. This is in stark contrast to workload models in which the random fluctuations average out over longer time scales.

D.2 The Hurst Effect

The description of fractals is very picturesque, but hard to quantify. The common way to quantify the degree of self similarity is based on the Hurst effect.

A random walk serves as a reference

A one-dimensional random walk is exemplified by a drunk on a sidewalk. Starting from a cartain lamppost, the drunk takes successive steps either to the left or to the right with equal probabilities. Where is he after n steps?

Let's assume the steps are independent of each other, and denote the drunk's location after i steps by x_i. The relationship between x_{i+1} and x_i is

$$x_{i+1} = \begin{cases} x_i + 1 & \text{with probability } 0.5 \\ x_i - 1 & \text{with probability } 0.5 \end{cases}$$

so on average the two options cancel out. But if we look at x_i^2, we get

$$x_{i+1}^2 = \begin{cases} (x_i+1)^2 = x_i^2 + 2x_i + 1 & \text{with probability } 0.5 \\ (x_i-1)^2 = x_i^2 - 2x_i + 1 & \text{with probability } 0.5 \end{cases}$$

leading to the relation $x_{i+1}^2 = x_i^2 + 1$ on average, and by induction $x_{i+1}^2 = i + 1$. The RMS distance of the drunk from the lamppost is therefore

$$|x_n| = n^{0.5}$$

Correlated steps lead to persistent processes

Now consider a drunk with inertia. Such a drunk tends to lurch several steps in the same direction before switching to the other direction. The steps are no longer independent — in fact, each step has an effect on following steps, which tend to be in the same direction. Overall, the probabilities of taking steps in the two directions are still the same, but these steps come in bunches.

Such a process, in which consecutive steps are positively correlated, is called a *persistent process*. The fact that it has longer bursts than would be expected had the steps been independent is called the *Joseph effect* (after the seven bad years and seven good years). However, there still are abrupt switches from one direction to another, which come at unpredictable times.

The opposite is also possible: a process in which consecutive steps tend to be inversely correlated. This is called an *anti-persistent process*.

The exponent characterizes the persistence of the process

Persistent and anti-persistent processes change the relationship between the distance travelled and the number of steps. In a persistent process, we would expect the drunk

to make more prograss, and get further away from the lamppost. Indeed, such processes are characterized by the relationship

$$x_n = cn^H$$

where H, the Hurst parameter, satisfies $0.5 < H < 1$.

In anti-persistent processes the drunk backtracks all the time, and makes less progress than expected. In such cases the Hurst parameter satisfies $0 < H < 0.5$.

And it is easy to measure experimentally

A nice thing about this formulation is that it is easy to verify. Taking the log of both sides of the equation $x_n = cn^H$ leads to

$$\log x_n = \log c + H \log n$$

We can take our data and find the average range it covers as a function of n. Then we plot it in log-log scales. If we get a straight line, our data is subject to the Hurst effect, and the slope of the line gives us the Hurst parameter H.

As it turns out, the Hurst effect is very common. Hurst himself showed this for various natural phenomena, including annual river flows, tree ring widths, and sunspot counts. It has also been shown for various aspects of computer workloads, ranging from network traffic to file server activity. Common values are in the range $0.7 < H < 0.9$, indicating persistent processes with high burstiness and self similarity.

To read more: Time series analysis is especially common in market analysis, and sevaral econometrics books deal with self-similarity in this context. A thorough introduction is given by Peters [3]. There is a growing number of papers which discuss self similarity in computer workloads [2, 1, 4].

Bibliography

[1] S. D. Gribble, G. S. Manku, D. Roselli, E. A. Brewer, T. J. Gibson, and E. L. Miller, "*Self-similarity in file systems*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 141–150, Jun 1998.

[2] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "*On the self-similar nature of Ethernet traffic*". *IEEE/ACM Trans. Networking* **2(1)**, pp. 1–15, Feb 1994.

[3] E. E. Peters, *Fractal Market Analysis*. John Wiley & Sons, 1994.

[4] D. Thiébaut, "*On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio*". *IEEE Trans. Comput.* **38(7)**, pp. 1012–1026, Jul 1989.

Chapter 11

Technicalities

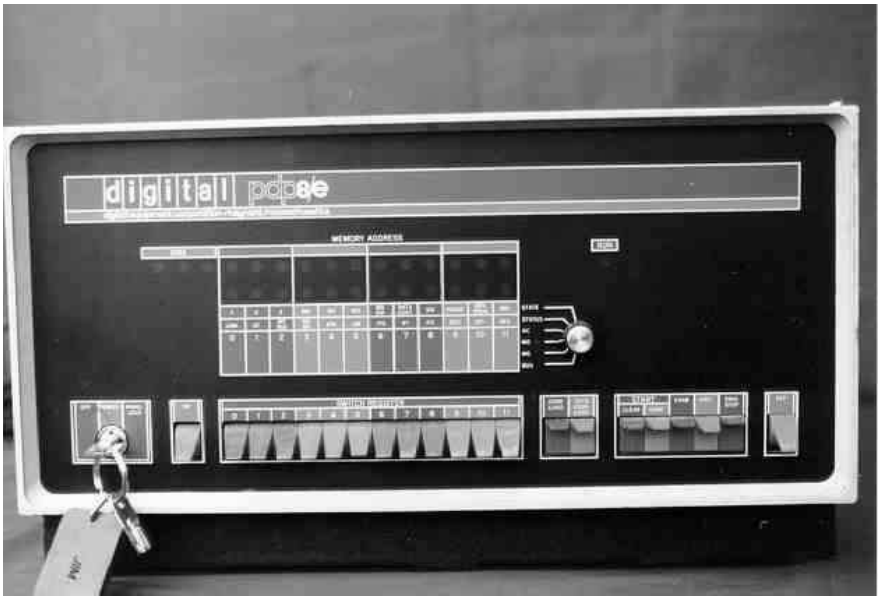
The previous chapters covered the classical operating system curriculum, which emphasizes abstractions, algorithms, and resource management. However, this is not enough in order to create a working system. This chapter covers some more technical issues and explains some additional aspects of how a computer system works. Most of it is Unix-specific.

11.1 Booting the System

Obviously, if the operating system is running, it can open the file containing the operating system executable and set things up to run it. But how does this happen the first time when the computer is turned on? Based on the analogy of getting out of the mud by pulling on your bootstraps, this process is called "booting the system".

The basic idea is to use amplification: write some very short program that can only do one thing: read a larger program off the disk and start it. This can be very specific in the sense that it expects this program to be located at a predefined location on the disk, typically the first sector, and if it is not there it will fail. By running this initial short program, you end up with a running longer program.

In the '60 the mechanism for getting the original short program into memory was to key it in manually using switches on the front panel of the computer: you literally set the bits of the opcode and arguments for successive instructions (in the PDP-8 in this picture, these are the row of switches at the bottom)



Contemporary computers have a non-volatile read-only memory (ROM) that contains the required initial short program. This is activated automatically upon power up.

The initial short program typically reads the boot sector from the disk. The boot sector contains a larger program that loads and starts the operating system. To do so, it has to have some minimal understanding of the file system structure.

Exercise 168 *How is address translation set up for the booting programs?*

Exercise 169 *How can one set up a computer such that during the boot process it will give you a choice of operating systems you may boot?*

One of the first things the operating system does whan it starts running is to load the *interrupt vector*. This is a predefined location in memory where all the entry points to the operating system are stored. It is indexed by interrupt number: the address of the handler for interrupt i is stored in entry i of the vector. When a certain interrupt occurs, the harwdare uses the interrupt number as an index into the interrupt vector, and loads the value found there into the PC, thus jumping to the appropriate handler (of course, it also sets the execution mode to kernel mode).

As a recap, it may be useful to review some of the interrupts we have seen in previous chapters, and some we didn't, roughly in priority order:

type	*interrupt*	*typical action*
panic	hardware error	emergency shutdown
	power failure	emergency shutdown
exception	illegal instruction (privileged or undefined)	kill process
	arithmetic exception (e.g. divide by zero)	kill process
	page fault	page in the missing page
device	clock interrupt	do bookkeeping, check timers, reschedule
	disk interrupt	wakeup waiting process, start new I/O operation
	terminal interrupt	wakeup waiting process
trap	software trap	switch on requested service

After setting up the interrupt vector, the system creates the first process environment. This process then forks additional system processes and the init process. The init process forks login processes for the various terminals connected to the system (or the virtual terminals, if connections are through a network).

11.2 Timers

providing time-related services. example: video/audio needs to occur at given rate in real time; old games would run faster when the PC was upgraded...

interface: ask for signal after certain delay. best effort service, may miss. using periodic clock sets the resolution. improved by soft timers [1], one-shot timers [5].

dependence on clock resolution [3]. Vertigo [4] — reduce hardware clock rate when not needed, to save power.

question of periodic timers, and need to keep control. if only one app, just let it run: control will return if something happens (e.g. network or terminal interrupt).

11.3 Kernel Priorities

considerations for priorities in kernel: the more low-level a process, and the more resources it holds, the higher its priority [2, p. 249]. for example, if waiting for disk I/O has higher priority than if waiting for a buffer, because the former already has a buffer and might finish and release it and other resources.

alternative of priorities based on device speeds – higher priority of waiting on slower device (=terminal =user). (check sun)

11.4 Logging into the System

11.4.1 Login

When you sit at a terminal connected to a Unix system, the first program you encounter is the *login* program. This program runs under the root user ID (the superuser), as it needs to access system information regarding login passwords. And in any case, it doesn't know yet which user is going to log in.

The login program prompts you for your login and password, and verifies that it is correct. After verifying your identity, the login program changes the user ID associated with the process to your user ID. It then execs a shell.

> **A trick**
> The login program need not necessarily exec the shell. For example, new student registration may be achieved by creating a login called "register", that does not require a password, and execs a registration program rather than a shell. New students can then initially login as "register", and provide their details to the system. When the registration program terminates they are automatically logged off.

11.4.2 The Shell

A *shell* is the Unix name for a command interpreter. Note that the shell runs in the context of the same process as the one that previously ran the login program, and therefore has the same user ID (and resulting priviledges). The shell accepts your commands and executes them. Some are builtin commands of the shell, and others are separate programs. Separate programs are executed by the fork/exec sequence described below.

The shell can also string a number of processes together using pipes, or run processes in the background. This simply means that the shell does not wait for their termination, but rather immediately provides a new prompt.

Exercise 170 *What happens if a background program needs to communicate with the user?*

11.5 Starting a Process

In all systems, processes can be started in two main ways: by a direct user command, or by another process.

In unix, processes are typically started by issuing a command to the shell. The shell then forks, and the resulting new process runs the desired program. But the option to fork is not restricted to the shell — every program can fork. Thus any program can create additional processes, that either continue to run the same program or convert to another program using the exec system call.

In Windows, a program is started by clicking on its icon on the desktop or by selecting it from a menu. It also has a very nice mechanism for invoking one application from within another, called "object linking and embedding" (OLE). For example, a text processor such as Word may embed a graphic created by a spreadsheet like Excel (this means that the graphic appears at a certain location in the text). When editing the text, it is then possible to click on the graphic and invoke the application that originally created it.

11.5.1 Constructing the Address Space

The exec system call constructs the new address space based on information contained in the executable file. Such a file starts with a predefined header structure which describes its contents. First comes a "magic number" that identifies this file as an executable of the right type (there is nothing magic about it — it is just a 16-bit pattern that hopefully will not appear by accident in other files). Next comes a list of sections contained in the file. Some of the sections correspond to memory segments, such as the text area for the process, and the initialized part of the data segment. Other sections contain additional information, such as a symbol table that may be used by a debugger. The header also contains the address of the entry point — the function (in the text area) where execution should begin.

Exercise 171 *Script files often start with a line of the form* `#!/bin/`⟨*interpreter*⟩. *What is the magic number in this case? Is it relevant?*

exec creates a text segment and initializes it using the text section of the executable. It creates a data segment and initializes it using another section. However, the created segment in this case is typically larger than the corresponding section, to account for uninitialized data. Stack and heap segments are also created, which have no corresponding section in the executable.

A special case occurs when shared libraries are used (shared libraries are called dynamic link libraries (DLLs) in Windows). Such libraries are not incorporated into the text segment by the compiler (or rather, the linker). Instead, an indication that they are needed is recorded. When the text segment is constructed, the library is included on the fly. This enables a single copy of the library code to serve multiple applications, and also reduces executable file sizes.

11.6 Context Switching

To switch from one process to another, the operating system has to store the hardware context of the current process, and restore that of the new process.

Rather than creating a special mechanism for this purpose, it is possible to use the existing mechanism for calling functions. When a function is called, the current

register values are typically stored on the stack, only to be restored when the function returns. The trick in using this for context switching is to swap stacks in the middle! Thus the current process calls the context switching function, causing all the hardware context to be stored on the kernel stack. This function loads the memory mapping registers with values of the new process, including those used to access the stack. At this instant, the CPU switches from executing the old process to the new one — which is also in the middle of the same function... The function completes already in the context of the new process, and returns, restoring the state of the new process as it was when it called the context switching function in order to stop running.

11.7 Making a System Call

To understand the details of implementing a system call, it is first necessary to understand how the kernel accesses different parts of the system's memory.

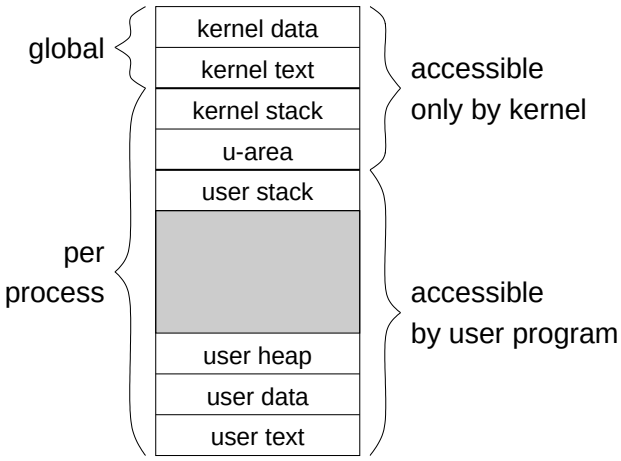
11.7.1 Kernel Address Mapping

Recall that system calls (and, for that matter, interrupt handlers) are individual entry points to the kernel. But in what context do these routines run?

The kernel can access everything

To answer this, we must first consider what data structures are required. These can be divided into three groups:

- Global kernel data structures, such as the process table, tables needed for memory allocation, tables describing the file system, and so forth.
- Process-specific data structures, in case the event being handled is specific to the current process. In Unix, the main one is the u-area. Access to the process's address space may also be needed.
- A stack for use when kernel routines call each other. Note that the kernel as a whole is re-entrant: a process may issue a system call and block in the middle of it (e.g. waiting for the completion of a disk operations), and then another process may also issue a system call, even the same one. Therefore a separate stack is needed for kernel activity on behalf of each process.



But different mappings are used

To enable the above distinction, separate address mapping data can be used. In particular, the kernel address space is mapped using distinct page tables. In total, the following tables are needed.

- Tables for the kernel text and global data. These are never changed, as there is a single operating system kernel. They are marked as being usable only in kernel mode, to prevent user code from accessing this data.

- Tables for per-process kernel data. While these are also flagged as being accessible only in kernel mode, they are switched on every context switch. Thus the currently installed tables at each instant reflect the data of the current process. Data for other processes can be found by following pointers in kernel data structures, but it cannot be accessed directly using the data structures that identify the "current process".

- Tables for the user address space. These are the segment and page tables discussed in Chapter 4.

Access to data in user space requires special handling. Consider a system call like read, that specifies a user-space buffer in which data should be deposited. This is a pointer that contains a virtual address. The problem is that the same memory may be mapped to a different virtual address when accessed from the kernel. Thus accessing user memory cannot be done directly, but must first undergo an address adjustment.

Interrupt handlers are special

A special case is the handling of interrupts, because interrupts happen asynchronously, and may not be related to any process (e.g. the clock interrupt). Some systems therefore have a special system context that is used for the handling of intrrupts. In Unix,

interrupt handling is done within the context of the current process, despite that fact that this process is probably unrelated to the interrupt.

Exercise 172 *Does the Unix scheme impose any restrictions on the interrupt handlers?*

11.7.2 To Kernel Mode and Back

The steps of performing a system call were outlined on page 1.3. Here we give some more details.

Recall that a process actually has two stacks: one is the regular stack that resides in the stack segment and is used by the user-level program, and the other is the kernel stack that is part of the per-process kernel data, together with the u-area. Making a system call creates frames on *both* stacks, and uses the u-area as a staging area to pass data from user mode to kernel mode and back.

System calls are actually represented by library functions. Thus making a system call starts by calling a normal library function, in user mode. This entails the creation of a call frame on the user stack, which contains the information required to return to the calling context. It may also contain the arguments passed to the library function.

The information regarding what system call is being requested, and what arguments are provided, is available to the user-level library function. The question is how to pass this information to the kernel-mode system call function. The problem is that the user-level function cannot access any of the kernel memory that will be used by the kernel-mode function. The simplest solution is therefore to use an agreed register to store the numberical code of the requested system call. After storing this code, the library function issues the trap instruction, inducing a transition to kernel mode.

The trap instruction creates a call frame on the kernel stack, just like the function-call instruction which creates a call frame on the user stack. It also loads the PC with the address of the system's entry point for system calls. When this function starts to run, it retrieves the system call code from the agreed register. Based on this, it knows how many arguments to expect. These areguments are then copied from the last call frame on the user stack (which can be identified based on the saved SP register value) to a designated place in the u-area. Once the arguments are in place, the actual function that implements the system call is called. All these funcitons retrieve their arguments from the u-area in the same way.

Exercise 173 *Why do the extra copy to the u-area? Each system call function can get the arguments directly from the user stack!*

When the system call function completes its task, it propagates its return value in the same way, by placing it in a designated place in the u-area. It then returns to its caller, which is the entry point of all system calls. This function copies the return value to an agreed register, where it will be found by the user-level library function.

11.8 Error Handling

The previous sections of this chapter were all about how basic operations get done. This one is about what to do when things go wrong.

It is desirable to give the application a chance to recover

The sad fact is that programs often ask the system to do something it cannot do. Sometimes it is the program's fault. For example, a program should verify that a value is not zero before dividing by this value. But what should the system do when the program does not check, and the hardware flags an exception? Other times it is not the program's fault. For example, when one process tries to communicate with another, it cannot be held responsible for misbehavior of the other process. After all, if it knew everything about the other process, it wouldn't have tocommunicate in the first place.

In either case, the simple way out is to kill the process that cannot proceed. However, this is a rather extreme measure. A better solution would be to punt the problem back to the application, in case it has the capability to recover.

System calls report errors via return values

Handling an error depends on when it happens. The easy case is if it happens when the operating system is working on behalf of the process, that is, during the execution of a system call. This is quite common. For example, the program might pass the system some illegal argument, like an invalid file descriptor (recall that in Unix a file descriptor is an index into the process's file descriptors table, which points to an entry that was allocated when a file was opened. The indexes of entries that have not been allocated, numbers larger than the table size, and negative numbers are all invalid). Alternatively, the arguments may be fine technically, but still the requested action is impossible. For example, a request to open a file may fail because the named file does not exist, or because the open files table is full and there would be no place to store the required information about the file had it been opened.

Exercise 174 *What are possible reasons for failure of the* `fork` *system call? How about* `write`*? And* `close`*?*

When a system call cannot perform the requested action, it is said to fail. This is typically indicated to the calling program by means of the system call's return value. For example, in Unix most system calls return -1 upon failure, and 0 or some non-negative number upon successful completion. It is up to the program to check the return value and act accordingly. If it does not, its future actions will probably run into trouble, because they are based on the unfounded assumption that the system call did what it was asked to do.

Exceptions require some asynchronous channel

The more difficult case is problems that occur when the application code is running, e.g. division by zero or issuing of an illegal instruction. In this case the operating system is notified of the problem, but there is no obvious mechanism to convey the information about the error condition to the application.

Unix uses signals

In Unix, this channel is signals. A signal is analogous to an interrupt in software. There are a few dozen pre-defined signals, including floating-point exception (e.g. division by zero), illegal instruction, and segmentation violation (attempt to access an invalid address). An application may register handlers for these signals if it so wishes. When an exception occurs, the operating system sends a signal to the application. This means that before returning to running the application, the handler for the signal will be called. If no handler was registered, some default action will be taken instead. In the really problematic cases, such as those mentioned above, the default action is to kill the process.

Exercise 175 *Where should the information regarding the delivery of signals to an application be stored?*

Once the signalling mechanism exists, it can also be used for other asynchronous events, not only for hardware exceptions. Examples include

- The user sends an interrupt from the keyboard.
- A timer goes off.
- A child process terminates.

Processes can also send signals to each other, using the `kill` system call.

A problem with signals is what to do if the signalled process is blocked in a system call. For example, it may be waiting for input from the terminal, and this may take a very long time. The solution is to abort the system call and deliver the signal instead, at least in system calls that may wait for an unlimited amount of time — such as waiting for terminal input, or waiting for another process to terminate.

Mach uses messages to a special port

Another example is provided by the Mach operating system. In this system, processes (called tasks in the Mach terminology) are multithreaded, and communicate by sending messages to ports belonging to other tasks. In addition to the ports created at run time for such communication, each task has a pre-defined port on which it can receive error notifications. A task is supposed to create a thread that blocks trying to receive messages from this port. If and when an error condition occurs, the opearting system sends a message with the details to this port, and the waiting thread receives it.

These mechanisms are used to implement language-level constructs, such as try/catch in Java

An example of the use of these mechanisms is the implementation of constructs such as the try/catch of Java, which expose exceptions at the language level. This construct includes two pieces of code: the normal code that should be executed (the "try" part), and the handler that should run if exceptions are caught (the "catch" part). To implement this on a Unix system, the catch part is turned into a handler for the signal representing the exception in question.

Exercise 176 *How would you implement a program that has sevaral different instances of catching the same type of exception?*

Bibliography

[1] M. Aron and P. Druschel, "*Soft timers: efficient microsecond software timer support for network processing*". *ACM Trans. Comput. Syst.* **18(3)**, pp. 197–228, Aug 2000.

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

[3] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[4] K. Flautner and T. Mudge, "*Vertigo: automatic performance-setting for Linux*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 105–116, Dec 2002.

[5] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "*Supporting time-sensitive applications on a commodity OS*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 165–180, Dec 2002.

Part IV

Communication and Distributed Systems

Practically all computers nowadays are connected to networks. The operating systems of these computers need to be able to interact. At a minimum, they should understand the same communication protocols. Using these protocols, it is possible to create various services *across* machines, such as

- Send and receive e-mail.
- Finger a user on a remote machine.
- "Talk" to a user on another machine interactively.

(Incidently, these services are handled by daemons.)

Beyond simple interaction, system functions may actually be distributed across a network. In this case, some computers are responsible for some functions, and only they handle these functions. If an application on another computer asks for such a service, the request is sent across the network to the responsible computer. The primary example is file servers, that contain large disks and store files for applications running on many other machines. In some distributed systems, it may be possible to migrate applications from a loaded machine to a less-loaded one, in order to improve response time. In this case, all the machines are "computation" servers.

This part in the notes explains how communication is performed, and how distributed systems are constructed.

Chapter 12

Interprocess Communication

Recall that an important function of operating systems is to provide abstractions and services to applications. One such service is to support communication among processes, in order to enable the construction of concurrent or distributed applications. A special case is client-server applications, which allow client applications to interact with server applications using well-defined interfaces.

This chapter discusses high-level issues in communication: naming, abstractions and programming interfaces, and application structures such as client-server. The next chapter deals with the low-level details of how bytes are actually transferred and delivered to the right place.

To read more: Communication is covered very nicely in Stallings [6], Chapter 13, especially Sections 13.1 and 13.2. It is also covered in Silberschatz and Galvin [4] Sections 15.5 and 15.6. Then, of course, there are whole textbooks devoted to computer communications. A broad introductory text is Comer [1]; more advanced books are tanenbaum [9] and stallings [5].

12.1 Naming

In order to communicate, processes need to know about each other

We get names when we are born, and exchange them when we meet other people. What about processes?

A basic problem with inter-process communication is *naming*. In general, processes do not know about each other. Indeed, one of the roles of the operating system is to keep processes separate so that they do not interfere with each other. Thus special mechanisms are required in order to enable processes to establish contact.

Inheritence can be used in lieu of naming

The simplest mechanism is through family relationships. As an analogy, if in a playground one kid hurts his knee and shouts "mommy!", the right mother will look up.

In computer systems such as Unix, if one process forks another, the child process may inherit various stuff from its parent. For example, various communication mechanisms may be established by the parent process before the fork, and are thereafter accessible also by the child process. One such mechanism is pipes, as described below on page 230 and in Appendix E.

Exercise 177 *Can a process obtain the identity (that is, process ID) of its family members? Does it help for communication?*

Predefined names can be adopted

Another simple approach is to use predefined and agreed names for certain services. In this case the name is known in advance, and represents a service, not a specific process. For example, when you call energency services by phone you don't care who specifically will answer, as long as he can help handle the emergency. Likewise, the process that should implement a service adopts the agreed name as part of its initialization.

Exercise 178 *What happens if no process adopts the name of a desired service?*

For example, this approach is the basis for the world-wide web. The service provided by web servers is actually a service of sending the pages requested by clients (the web browsers). This service is identified by the port number 80 — in essense, the port number serves as a name. This means that a browser that wants a web page from the server `www.abc.com` just sends a request to port 80 at that address. The process that implements the service on that host listens for incoming requests on that port, and serves them when they arrive. This is described in more detail below.

Names can be registered with a name server

A more general solution is to use a *name service*, maintained by the operating system. Any process can advertise itself by registering a chosen string with the name service. Other processes can then look up this string, and obtain contact information for the process that registered it. This is similar to a phonebook that maps names to phone numbers.

Exercise 179 *And how do we create the initial contact with the name service?*

A sticky situation develops if more than one set of processes tries to use the same string to identify themselves to each other. It is easy for the first process to figure

out that the desired string is already in use by someone else, and therefore another string should be chosen in its place. Of course, it then has to tell its colleagues about the new string, so that they know what to request from the name server. But it can't contact its colleagues, because the whole point of having a string was to establish the initial contact...

12.2 Programming Interfaces and Abstractions

Being able to name your partner is just a pre-requisite. The main issue in communication is being able to exchange information. Processes cannot do so directly — they need to ask the operating system to do it for them. This section describes various interfaces that can be used for this purpose.

We start with two mechanisms that are straightforward extensions to conventional programming practices: using shared memory, and remote procedure call. Then we turn to more specialized devices like message passing and streams.

12.2.1 Shared Memory

Shared access to the same memory is the least structured approach to communication. There are no restrictions on how the communicating processes behave. In particular, there is no a-priori guarantee that one process write the data before another attempts to read it.

Within the same system, processes can communicate using shared memory

Recall that a major part of the state of a process is its memory. If this is shared among a number of processes, they actually operate on the same data, and thereby communicate with each other.

Rather than sharing the whole memory, it is possible to only share selected regions. For example, the Unix shared memory system calls include provisions for

- Registering a name for a shared memory region of a certain size.
- Mapping a named region of shared memory into the address space.

The system call that maps the region returns a pointer to it, that can then be used to access it. Note that it may be mapped at different addresses in different processes.

To read more: See the man pages for `shmget` and `shmat`.

Exercise 180 *How would you implement such areas of shared memory? Hint: think about integration with the structures used to implement virtual memory.*

In some systems, it may also be possible to inherit memory across a fork. Such memory regions become shared between the parent and child processes.

distributed shared memory may span multiple machines

The abstraction of shared memory may be supported by the operating system even if the communicating processes run on distinct machines, and the hardware does not provide them direct access to each other's memory. This is called *distributed shared memory* (DSM).

The implementation of DSM hinges on playing tricks with the mechanisms of virtual memory. Recall that the page table includes a present bit, which is used by the operating system to indicate that a page is indeed present and mapped to a frame of physical memory. If a process tries to access a page that is *not* present, a page fault is generated. Normally this causes the operating system to bring the page in from permanent storage on a disk. In DSM, it is used to request the page from the operating system on another machine, where it is being used by another process. Thus DSM is only possible on a set of machines running the same system type, and leads to a strong coupling between them.

While the basic idea behind the implementation of DSM is simple, getting it to perform well is not. If only one copy of each page is kept in the system, it will have to move back and forth between the machines running processes that access it. This will happen even if they actually access different data structures that happen to be mapped to the same page, a situation known as *false sharing*. Various mechanisms have been devised to reduce such harmful effects, including

- Allowing multiple copies of pages that are only being read.
- Basing the system on objects or data structures rather than on pages.
- Partitioning pages into sub-pages and moving them independently of each other.

Exercise 181 *Is there a way to support multiple copies of pages that are also written, that will work correctly when the only problem is actually false sharing?*

To read more: A thorough discussion of DSM is provided by Tanenbaum [8], chapter 6. An interesting advanced system which solves the granularity problem is MilliPage [3].

Sharing memory leads to concurrent programming

The problem with shared memory is that its use requires synchronization mechanisms, just like the shared kernel data structures we discussed in Section 3.1. However, this time it is the user's problem. The operating system only has to provide the means by which the user will be able to coordinate the different processes. Many systems therefore provide semaphores as an added service to user processes.

Exercise 182 *Can user processes create a semaphore themselves, in a way that will block them if they cannot gain access? Hint: think about using pipes. This solution is good only among related processes within a single system.*

Files also provide a shared data space

An alternative to shared memory, that is supported with no extra effort, is to use the file system. Actually, files are in effect a shared data repository, just like shared memory. Moreover, they are persistent, so processes can communicate without overlapping in time. However, the performance characteristics are quite different — file access is much slower than shared memory.

Exercise 183 *Is it possible to use files for shared memory without suffering a disk-related performance penalty?*

12.2.2 Remote Procedure Call

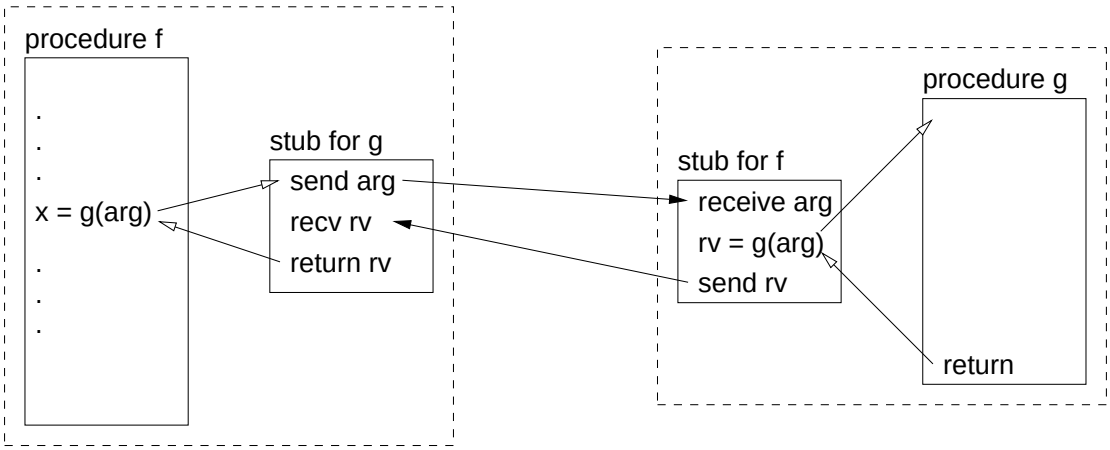
At the opposite extreme from shared memory is remote procedure call. This is the most structured approach to communications.

A natural extension to procedure calls is to call remote procedures

Remote procedure calls (RPC) extend the well-known procedure calling mechanism, and allow one process to call a procedure from another process, possibly on a different machine. This idea has become even more popular with the advent of object-oriented programming, and has even been incorporated in programming languages. An example is Java's remote method invocation (RMI). However, the idea can be implemented even if the program is not written in a language with explicit support.

The implementation is based on stub procedures

The implementation is based on stubs — crippled procedures that represent a remote procedure by having the same interface. The calling process is linked with a stub that has the same interface as the called procedure. However, this stub does not *implement* this procedure. Instead, it sends the arguments over to the stub linked with the other process, and waits for a reply.



The other stub mimics the caller, and calls the desired procedure locally with the specified arguments. When the procedure returns, the return values are shipped back and handed over to the calling process.

RPC is a natural extension of the procedure call interface, and has the advantage of allowing the programmer to concentrate on the logical structure of the program, while disregarding communication issues. The stub functions are typically provided by a library, which hides the actual implementation.

For example, consider an ATM used to dispense cash at a mall. When a user requests cash, the business logic implies calling a function that verifies that the account balance is sufficient and then updates it according to the withdrawal amount. But such a function can only run on the bank's central computer, which has direct access to the database that contains the relevant account data. Using RPC, the ATM software can be written as if it also ran directly on the bank's central computer. The technical issues involved in actually doing the required communication are encapsulated in the stub functions.

Exercise 184 *Can any C function be used by RPC?*

12.2.3 Message Passing

RPC organizes communication into structured and predefined pairs: send a set of arguments, and receive a return value. Message passing allows more flexibility by allowing arbitrary interaction patterns. For example, you can send multiple times, without waiting for a reply.

Messages are chunks of data

On the other hand, message passing retains the partitioning of the data into "chunks" — the messages. There are two main operations on messages:

`send(to, msg, sz)` — Send a message of a given size to the addressed recipient. `msg` is a pointer to a memroy buffer that contains the data to send, and `sz` is it's size.

`receive(from, msg, sz)` — Receive a message, possibly only from a specific sender. The arguments are typically passed by reference. `from` may name a specific sender. Alternatively, it may contain a "dontcare" value, which is then overwritten by the ID of the actual sender of the received message. `msg` is a pointer to a memory buffer where the data is to be stored, and `sz` is the size of the buffer. This limits the maximal message size that can be received, and longer messages will be truncated.

Exercise 185 *Should the arguments to `send` be passed by reference or by value?*

Receiving a message can be problematic if you don't know in advance what it's size will be. A common solution is to decree a maximal size that may be sent; recipients then always prepare a buffer of this size, and can therefore receive any message.

Exercise 186 *Is it possible to devise a system that can handle messages of arbitrary size?*

Sending and receiving are discrete operations, applied to a specific message. This allows for special features, such as dropping a message and proceeding directly to the next one. It is also the basis for collective operations, in which a whole set of processes participate (e.g. broadcast, where one process sends information to a whole set of other processes). These features make message passing attractive for communication among the processes of parallel applications. It is not used much in networked settings.

12.2.4 Streams: Unix Pipes, FIFOs, and Sockets

Streams just pass the data piped through them in sequential, FIFO order. The distinction from message passing is that they do not maintain the structure in which the data was created. This idea enjoys widespread popularity, and has been embodied in several mechanisms.

Streams are similar to files

One of the reasons for the popularity of streams is that their use is so similar to the use of sequential files. You can write to them, and the data gets accumulated in the order you wrote it. You can read from them, and always get the next data element after where you dropped off last time. It is therefore possible to use the same interfaces; in Unix this is file descriptors and `read` and `write` system calls.

Exercise 187 *Make a list of all the attributes of files. Which would you expect to describe streams as well?*

Pipes are FIFO files with special semantics

Once the objective is defined as inter-process communication, special types of files can be created. A good example is Unix *pipes*.

A pipe is a special file with FIFO semantics: it can be written sequentially (by one process) and read sequentially (by another). The data is buffered, but is not stored on disk. Also, there is no option to seek to the middle, as is possible with regular files. In addition, the operating system provides some special related services, such as

- If the process writing to the pipe is much faster than the process reading from it, data will accumulate unnecessarily. If this happens, the operating system blocks the writing process to slow it down.

- If a process tries to read from an empty pipe, it is blocked rather then getting an EOF.
- If a process tries to write to a pipe that no process can read (because the read end was closed), the process gets a signal. On the other hand, when a process tries to read from a pipe that no process can write, it gets an EOF.

Exercise 188 *How can these features be implemented efficiently?*

The problem with pipes is that they are unnamed: they are created by the `pipe` system call, and can then be shared with other processes created by a fork. They cannot be shared by unrelated processes. This gap is filled by FIFOs, which are essentially named pipes. This is a special type of file, and appears in the file name space.

To read more: See the man page for `pipe`. The mechanics of stringing processes together are described in detail in Appendix E. Named pipes are created by the `mknode` system call, or by the `mkfifo` shell utility.

In the original Unix system, pipes were implemented using the file system infrastructure, and in particular, inodes. In modern systems they are implemented as a pair of sockets, which are obtained using the `socketpair` system call.

Sockets support client-server computing

A much more general mechanism for creating stream connections among unrelated processes, even on different systems, is provided by *sockets*. Among other things, sockets can support any of a wide variety of communication protocols. The most commonly used are the internet protocols, TCP/IP (which provides a reliable stream of bytes), and UDP/IP (which provides unreliable datagrams).

Sockets are the most widely used mechanism for communication over the Internet, and are covered in the next section.

Concurrency and asynchrony make things hard to anticipate

Distrubuted systems and applications are naturally concurrent and asynchronous: many different things happen at about the same time, in an uncoordinated manner. Thus a process typically cannot know what will happen next. For example, a process may have opened connections with several other processes, but cannot know which of them will send it some data first.

The `select` system call is designed to help with such situations. This system call receives a set of file descriptors as an argument. It then blocks the calling process until any of the sockets represented by these file descriptors has data that can be read. Alternatively, a timeout may be set; if no data arrives by the timeout, the `select` will return with a failed status.

On the other hand, using streams does provide a certain built-in synchronization: due to the FIFO semantics, data cannot be read before it is written. Thus a process may safely try to read data from a pipe or socket. If no data is yet available, the process will either block waiting for data to become available or will receive an error notification — based on the precise semantics of the stream.

To read more: See the man pages for `socket`, `bind`, `connect`, `listen`, `accept`, and `select`.

12.3 Sockets and Client-Server Systems

12.3.1 Distributed System Structures

Using the interfaces described above, it is possible to create two basic types of distributed systems or applications: symmetrical or asymmetrical.

Symmetrical communications imply peer to peer relations

In symmetric applications or systems, all processes are peers. This is often the case for processes communicating via shared memory or pipes, and is the rule in parallel applications.

Peer-to-peer systems, such as those used for file sharing, are symmetrical in a different sense: in such systems, all nodes act both as clients and as servers. Therefore some of the differences between clients and servers described below do not hold.

Client-server communications are asymmetrical

The more common approach, however, is to use an asymmetrical structure: one process is designated as a server, and the other is its client. This is used to structure the application, and often matches the true relationship between the processes. Examples include

- A program running on a workstation is the client of a file server, and requests it to perform operations on files.
- A program with a graphical user interface running on a Unix workstation is a client of the X server running on that workstation. The X server draws things on the screen for it, and notifies it when input is events have occured in its window.
- A web browser is a client of a web server, and asks it for certain web pages.
- An ATM is a client of a bank's central computer, and asks it for authorization and recording of a transaction.

Client-server interactions can be programmed using any of the interfaces described above, but are especially convenient using RPC or sockets.

Servers typically outlive their clients

An interesting distinction between peer-to-peer communication and client-server communication is based on the temporal dimension: In peer-to-peer systems, all processes may come and go individually, or in some cases, they all need to be there for the interaction to take place. In client-server systems, on the other hand, the server often exists for as long as the system is up, while clients come and go.

The implications of this situation are twofold. First, the server cannot anticipate which clients will contact it and when. As a consequence, it is futile for the server to try and establish contact with clients; rather, it is up to the clients to contact the server. The server just has to provide a means for clients to find it, be it by registering in a name service or by listening on a socket bound to a well-known port.

Many daemons are just server processes

Unix daemons are server processes that operate in the background. They are used to provide various system services that do not need to be in the kernel, e.g. support for email, file spooling, performing commands at pre-defined times, etc. In particular, daemons are used for various services that allow systems to inter-operate across a network. In order to work, the systems have to be related (e.g. they can be different versions of Unix). The daemons only provide a weak coupling between the systems.

12.3.2 The Sockets Interface

The socket interface is designed for setting up client-server communications.

To read more: Writing client-server applications is covered in length in several books on TCP/IP programming, e.g. Commer [2]. Writing deamons correctly is an art involving getting them to be completely independent of anything else in the system. See e.g. Stevens [7, Sect. 2.6].

Basic connection is asymmetric

The first process, namely the *server*, does the following (using a somewhat simplified API).

`fd=socket()` First, it *creates a socket*. This means that the operating system allocates a data structure to keep all the information about this communication channel, and gives the process a file descriptor to serve as a handle to it.

`bind(fd, port)` The server then *binds* this socket (as identified by the file descriptor) to a port number. In effect, this gives the socket a name that can be used by clients: the machine's IP (internet) address together with the port are used to identify this socket (you can think of the IP address as a street address, and the port number as a door or suite number at that address). Common services have

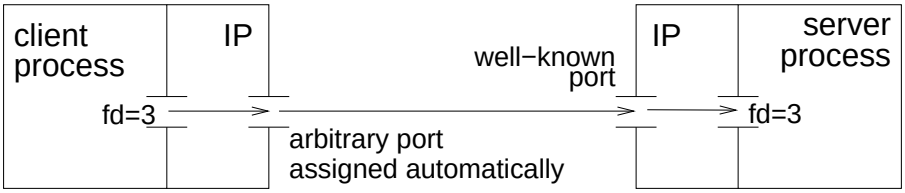
predefined port numbers that are well-known to all (to be described in Section 12.3.1). For other distributed applications, the port number is typically selected by the programmer.

`listen(fd)` To complete the setup, the server then *listens* to this socket. This notifies the system that communication requests are expected.

The other process, namely the *client*, does the following.

`fd=socket()` First, it also creates a socket.

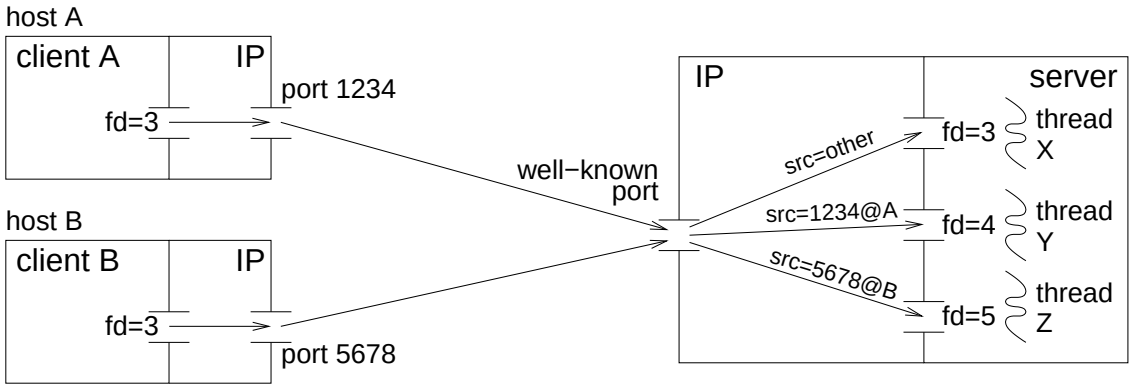
`connect(fd, addr, port)` It then *connects* this socket to the server's socket, by giving the server's IP address and port. This means that the server's address and port are listed in the local socket data structure, and that a message regarding the communication request is sent to the server. The system on the server side finds the server's socket by searching according to the port number.



To actually establish a connection, the server has to take an additional step:

`newfd=accept(fd)` After the `listen` call, the original socket is waiting for connections. When a client connects, the server needs to `accept` this connection. This creates a *new* socket that is accessed via a new file descriptor. This new socket (on the server's side) and the client's socket are now connected, and data can be written to and read from them in both directions.

The reason for using `accept` is that the server must be ready to accept additional requests from clients at any moment. Before calling `accept`, the incoming request from the client ties up the socket bound to the well-know port, which is the server's advertised entry point. By calling `accept`, the server re-routes the incoming connection to a new socket represented by another file descriptor. This leaves the original socket free to receive additional clients, while the current one is being handled. Moreover, if multiple clients arrive, each will have a separate socket, allowing for unambiguous communication with each one of them. The server will often also create a new thread to handle the interaction with each client, to furhter encapsulate it.



Note that the distinction among connections is done by the IP addresses and port numbers of the two endpoints. All the different sockets created by `accept` share the same port on the server side. But they have different clients, and this is indicated in each incoming communication. Communications coming from an unknown source are routed to the original socket.

Exercise 189 *What can go wrong in this process? What happens if it does?*

After a connection is accepted the asymmetry of setting up the connection is forgotten, and both processes now have equal standing. However, as noted above the original socket created by the server still exists, and it may accept additional connections from other clients.

Naming is based on conventions regarding port numbers

The addressing of servers relies on universal conventions regarding the usage of certain ports. In Unix systems, the list of well-known services and their ports are kept in the file `/etc/services`. Here is a short excerpt:

port	*usage*
21	ftp
23	telnet
25	smtp (email)
42	name server
70	gopher
79	finger
80	http (web)

Exercise 190 *What happens if the target system is completely different, and does not adhere to the port-usage conventions?*

As an example, consider the `finger` command. Issuing `finger joe@hostname.dom` causes a message to be sent to port 79 on the named host. It is assumed that when

that host booted, it automatically started running a finger daemon that is listening on that port. When the query arrives, this daemon receives it, gets the local information about joe, and sends it back.

Exercise 191 *Is it possible to run two different web servers on the same machine?*

Communication is done using predefined protocols

To contact a server, the client sends the request to a predefined port on faith. In addition, the data itself must be presented in a predefined format. For example, when accessing the finger daemon, the data sent is the login of the user in which we are interested. The server reads whatever comes over the connection, assumes this is a login name, and tries to find information about it. The set of message formats that may be used and their semantics are called a *communication protocol*.

In some cases, the protocols can be quite extensive and complicated. An example is NFS, the network file system. Communication among clients and servers in this system involves many operations on files and directories, including lookup and data access. The framework in which this works is described in more detail in Section 14.2.

12.4 Middleware

Unix daemons are an example of a convention that enables different versions of the same system to interact. To some degree other systems can too, by having programs listen to the correct ports and follow the protocols. But there is a need to generalize this sort of interoperability. This is done by middleware.

Heterogeneity hinders interoperability

As noted above, various communication methods such as RPC and sockets rely on the fact that the communicating systems are identical or at least very similar. But in the real world, systems are very heterogeneous. This has two aspects:

- Architectural heterogeneity: the hardware may have a different architecture. The most problematic aspect of different architectures is that different formats may be used to represent data. Examples include little endian or big endian ordering of bytes, twos-complement or ones-complement representation of integers, IEEE standard or proprietary representations of floating point numbers, and ASCII or EBCDIC representation of characters. If one machine uses one format, but the other expects another, the intended data will be garbled.
- System heterogeneity: different operating systems may implement key protocols slightly differently, and provide somewhat different services.

For example, consider an application that runs on a desktop computer and needs to access a corporate database. If the database is hosted by a mainframe that uses different data representation and a different system, this may be very difficult to achieve.

Middleware provides a common ground

The hard way to solve the problem is to deal with it directly in the application. Thus the desktop application will need to acknowledge the fact that the database is different, and perform the necessary translations in order to access it correctly. This creates considerable additional work for the developer of the aplication, and is specific for the systems in question.

A much better solution is to use a standard software layer that handles the translation of data formats and service requests. This is what middleware is all about.

CORBA provides middleware for objects

The most pervasive example of middleware is probably CORBA (common object request broker architecture). This provides a framework that enables the invokation of methods in remote objects and across heterogeneous platforms. Thus it is an extension of the idea of RPC.

The CORBA framework consists of sevaral components. One is the interface definition language (IDL). This enables objects to provide their specification in a standardized manner, and also enables clients to specify the interfaces that they seek to invoke.

The heart of the system is the object request broker (ORB). This is a sort of naming service where objects register their methods, and clients search for them. The ORB makes the match and facilitates the connection, including the necessary translations to compensate for the heterogeneity. Multiple ORBs can also communicate to create one large object space, where all methods are available for everyone.

12.5 Summary

Abstractions

Interprocess communication as described here is mainly about abstractions, each with its programming interface, properties, and semantics. For example, streams include a measure of synchronization among the communicating processes (you can't receive data that has not been sent yet), whereas shared memory does not include implied synchronization and therefore some separate synchronization mechanism may need to be used.

Resource management

A major resource in communications is the namespace. However, this is so basic that it isn't really managed; Instead, it is either used in a certain way by convention (e.g. well-known port numbers) or up for grabs (e.g. registration with a name server).

Workload issues

Workloads can also be interpreted as a vote of popularity. In this sense, sockets are used overwhelmingly as the most common means for interprocess communication. Most other means of communication enjoy only limited and specific uses.

Hardware support

As we didn't discuss implementations, hardware support is irrelevant at this level.

Bibliography

[1] D. E. Comer, *Computer Networks and Internets*. Prentice Hall, 2nd ed., 1999.

[2] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications*. Prentice Hall, 2nd ed., 1996.

[3] A. Schuster et al., "*MultiView and MilliPage — fine-grain sharing in page-based DSMs*". In 3rd *Symp. Operating Systems Design & Implementation*, pp. 215–228, Feb 1999.

[4] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[5] W. Stallings, *Data and Computer Communications*. Macmillan, 4th ed., 1994.

[6] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[7] W. R. Stevens, *Unix Network Programming*. Prentice Hall, 1990.

[8] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.

[9] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, 3rd ed., 1996.

Chapter 13

(Inter)networking

Interprocess communication within the confines of a single machine is handled by that machine's local operating system; for example, a Unix system can support two processes that communicate using a pipe. But when processes on different machines need to communicate, communication networks are needed. This chapter describes the protocols used to establish communications and pass data between remote systems.

13.1 Communication Protocols

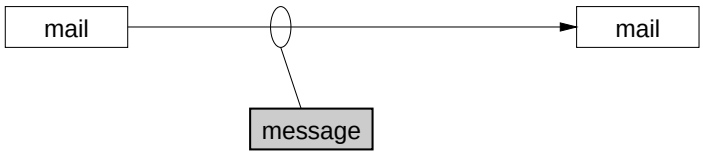
In order to communicate, computers need to speak a common language. Protocols provide this common language. They specify precisely what can be said, and under what circumstances.

13.1.1 Protocol Stacks

How is communication from one machine to another supported? Let's look at a specific (imaginary) example, and follow it from the top down. Our example will be sending an email message.

Sending email is like copying a file

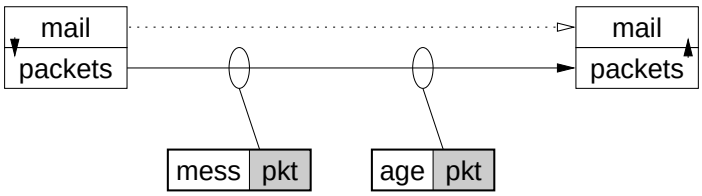
When you compose an email message, the email application saves what you write in a (temporary) file. At the top level, sending the email message is just copying this file from the sender to the receiver's incoming `mail` directory. This is conceptually very simple. The message text begins with a line saying "`To: yossi`", so the receiving email program knows to whom the message is addressed.



Long messages may need to be broken up into shorter ones

Because of limited buffer capacity in the communication hardware and other reasons, it is impossible to send arbitrarily long messages. But users don't know this, and may want to send a whole book in a single message. Therefore the email handling program may need to break up a long message into short *packets* on the sending side, and re-assemble them into a single long message at the receiving side.

As packetization is a useful service, we'll write a separate program to handle it. Thus the email program will not send the message directly to its counterpart on the other computer; instead, it will forward the message to its local packetization program. The packetization program will break the message up into packets, add a header with the packet number to each one, and send them to the packetization program on the other computer. There they will be reassembled and passed back up to the email program. *Logically* the message passes directly from the email application at the sender to the email application of the recipient, but in reality it takes a detour through the packetization service.



Two things are noteworthy. First, the packetization regards the data it receives as a single entity that has to be packetized; it does not interpret it, and does not care if it is actually composed of some higher-level headers attached to the "real" data. Second, the resulting packets need not be the same size: there is a maximal packet size, but if the remaining data is less that this, a smaller packet can be sent.

End-to-end control is required to ensure correct reassembly

The above description is based on the assumption that all packets arrive in the correct order. But what if some packet is lost on the way, or if one packet overtakes another and arrives out of sequence? Thus another responsibility of the packetization layer is to handle such potential problems.

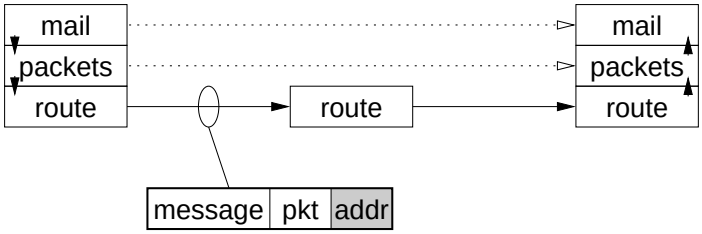
handling messages that arrive out of sequence is simple — as long as we can store the received packets, we can reassemble them based on their serial numbers in the correct order. And if one turns out to be missing, we can request the sender to re-send.

This leads to the notion of acknowledgment. Each received packet is acknowledged, and thus the sender can identify those that were not acknowledged as missing.

Routing is needed if there is no direct link

The notion that packets may be lost or reordered reflects the common case where there is not direct link between the two computers. The message (that is, all its packets) must then be routed through one or more intermediate computers.

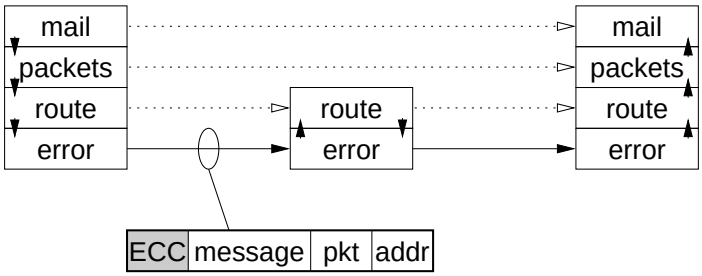
Again, routing is useful, so it will be handled separately. The packetizer then forwards the packets to the router, who adds an address or routing header, and determines where to send them. The router on the receiving node checks the address, and forwards the packets as appropriate. At the end the packets arrive at the designated destination computer. The router on that machine recognizes that it is the destination, and passes the arriving packets to the local packetizer for processing.



Error correction can be applied to each transmission

The above scenario is optimistic in the sense that it assumes that data arrives intact. In fact, data is sometimes corrupted during transmission due to noise on the communication lines. Luckily it is possible to devise means to recognize such situations. For example, we can calculate the parity of each packet and its headers, and add a parity bit at the end. If the parity does not match at the receiving side, en error is flagged and the sender is asked to re-send the packet. Otherwise, an acknowledgment is sent back to the sender to notify it that the data was received correctly.

Parity is a simple but primitive form of catching errors, and has limited recognition. Therefore real systems use more sophisticated schemes such as CRC. However, the principle is the same. And naturally, we want a separate program to handle the calculation of the error-correcting code (ECC) and the necessary re-sends.



Finally, bits have to be represented by some physical medium

At the bottom level the bits need to be transmitted somehow. for example, bits may be represented by voltage levels on a wire, or by pulses of light in a waveguide. This is called the physical layer and does not concern us here — we are more interested in the levels that are implemented by the operating system.

All this results in the creation of complex protocol stacks

As we saw, it is convenient to divide the work of performing communication into multiple layers. Each layer builds upon the layers below it to provide an additional service to the layers above it, and ultimately to the users. The data that is transmitted goes down through the layers, acquiring additional headers on the way, on the sending side. Then it is transmitted across the network. Finally it goes back up again on the receiving side, and the headers are peeled off.

The set of programs that the message goes through is called a *protocol stack*. Logically, each layer talks directly with its counterpart on the other machine, using some particular protocol. Each protocol specifies the format and meaning of the messages that can be passed. Usually this is a protocol-specific header and then a data field, that is not interpreted. For example, the packetization layer adds a header that contains the packet number in the sequence.

Exercise 192 *What is the protocol of the email application? look at your email and make an educated guess.*

The protocol also makes assumptions about the properties of the communication subsystem that it uses. For example, the email application assumes that the whole message is transferred with no errors. In reality, this abstraction of the communication subsystem is created by lower layers in the stack.

Standardization is important

A computer receiving a message acts upon it in blind faith. It *assumes* that the last byte is a checksum that testifies to the correctness of the data, and performs the required check. It *assumes* that the first byte is an address, and checks whether the data is addressed to itself or maybe has to be forwarded. It *assumes* that the next byte is a packet number, and so on. But what if the sending computer ordered the headers (and the corresponding layers in the protocol stack) differently?

In order to be able to communicate, computers must agree on the structure and functionality of their protocol stacks, and on the format of the headers used by each one. This leads to the creation of *open systems*, that can accept external data and assimilate it correctly.

An abstract model of how a stack of communications protocols should be structured was defined by the International Standards Organization, and is briefly described in

Appendix F. But the de-facto standard that serves as the basis for practically all communications, and in particular the Internet, is the TCP/IP protocol suite.

13.1.2 The TCP/IP Protocol Suite

The IP protocols are the de-facto standard on the Internet

IP, the Internet Protocol, is essentially concerned with routing data across multiple networks, thus logically uniting these networks into one larger network. This is called the "network" layer, and IP is a network protocol. However, the networks used may have their own routing facility, so IP may be said to correspond to the "top part" of the network layer. The lower layers ("link" layer and "physical" layer) are not part of the TCP/IP suite — instead, whatever is available on each network is used.

TCP and UDP provide end-to-end services (based on IP), making them "transport" protocols. In addition, they are responsible for delivering the data to the correct application. This is done by associating each application with a port number. These are the same port numbers used in the `bind` and `connect` system calls, as described above on page 233.

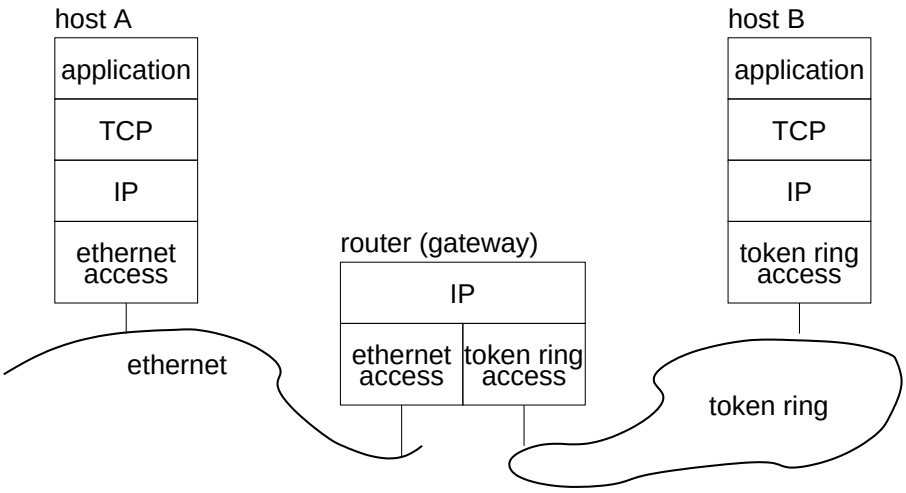
The top layer is the "application" layer. Some basic applications (such as ftp and telnet) are an integral part of the standard TCP/IP suite, and thus make these services available around the world.

An internet links many different networks

As mentioned, the main feature of the Internet Protocol (IP) is that it creates an *internet*: a network that is composed of networks.

Computers are typically connected to local-area networks (LANs). The most common LAN technology is Ethernet. Computers on a LAN identify each other by their MAC addresses — hardwired unique addresses that are embedded in the network interface card. In order to propagate information from one network to another, gateway computers are used. These are computers that are part of two or more networks. They can therefore receive a message on one network and forward it to the other network.

IP performs the routing of messages across the different networks, until they reach their final destination. For example, the message may go over an Ethernet from its origin to a router, and then over a token ring from the router to its destination. Of course, it can also go through a larger number of such hops. All higher levels, including the application, need not be concerned with these details.



Exercise 193 *Does IP itself have to know about all the different networks a message will traverse in advance?*

The worldwide amalgamation of thousands of public networks connecting millions of computers is called the Internet, with a capital I.

> **Details: the IP header**
> As noted above each transmission by a certain protocol is typically divided into two parts: a header and data. In the case of the IP protocol, the header is 5 words long, for a total of 20 bytes. This is composed of the following fields:
>
> - Protocol version (4 bits). Most of the Internet uses version 4 (IPv4), but some uses the newer version 6. The header described here is for version 4.
> - Header length, in words (4 bits).
> - Indication of quality of service desired, which may or may not be supported by routers (8 bits).
> - Packet length in bytes (16 bits). This limits a packet to a maximum of 64 KiB.
> - Three fields that deal with fragmentation, and the place of this fragment in the sequence of fragments that make up the packet (32 bits).
> - Time to live — how many additional hops this packet may propagate (8 bits). This is decremented on each hop, and when it hits 0 the packet is discarded.
> - Identifier of higher-level protocol that sent this packet, e.g. TCP or UDP (8 bits).
> - Header checksum, used to verify that the header has not been corrupted (16 bits).
> - Sender's IP address (32 bits)
> - Destination IP address (32 bits)

UDP provides direct access to datagrams

IP transfers data in datagrams — the "atomic" units in which data is transferred. The size of IP datagrams is not fixed, and can range up to 64 KB. Various networks,

however, have a smaller maximal transfer unit (MTU); for example, on an Ethernet the MTU is 1500 bytes. Therefore IP must fragment and reassemble datagrams along the way.

UDP (User Datagram Protocol) provides a relatively direct access to IP datagrams. It allows users to send datagrams of up to a fixed size, with no additional overhead for reliability. The datagrams are simply sent in a "best effort" style, with the hope that they will indeed arrive. However, a checksum is included, so that corrupted datagrams can be detected and discarded. UDP also has other features that do not exist in TCP, such as a broadcast capability.

Exercise 194 *Is UDP useful? What can you do with a service that may silently loose datagrams?*

TCP provides a reliable stream service

TCP (Transmission Control Protocol) uses IP to provide a reliable stream interface. This means that it creates a persistent connection between the communicating applications, and the applications can write data into the connection, and read data from it, in an unstructured manner, and in unlimited sizes. TCP breaks the data into datagrams, and ensures that they all arrive at the other end and are re-united into a sequence correctly. To do so it keeps each datagram in a buffer until it is acknowledged.

Exercise 195 *Why doesn't TCP provide broadcast?*

> **Details: the TCP header**
> The header used by the TCP protocol is 5–15 words long. 5 words are required, for a total of 20 bytes. This is composed of the following fields:
>
> - Source port number (16 bits). Port numbers are thus limited to the range up to 65,535. Of this the first 1024 are reserved for well-known services.
> - Destination port number (16 bits).
> - Sequence number of the first byte of data being sent in this packet (32 bits).
> - Acknowledgments, expressed as the sequence number of the next byte of data that the sender expects to receive from the recipient (32 bits).
> - Header size, so we will know where the data starts (4 bits). This is followed by 4 reserved bits, whose use has not been specified.
> - Eight single-bit flags, used for control (e.g. in setting up a new connection).
> - The available space that the sender has to receive more data from the recipient (16 bits).
> - Checksum on the whole packet (16 bits).
> - Indication of urgent data (16 bits)

And there are some useful applications too

Several useful applications have also become part of the TCP/IP protocol suite. These include

- SMTP (Simple Mail Transfer Protocol), with features like mailing lists and mail forwarding. It just provides the transfer service to a local mail service that takes care of things like editing the messages.
- FTP (File Transfer Protocol), used to transfer files across machines. This uses one TCP connection for control messages (such as user information and which files are desired), and another for the actual transfer of each file.
- Telnet, which provides a remote login facility for simple terminals.

To read more: The TCP/IP suite is described in Stallings [7] section 13.2, and very briefly also in Silberschatz and Galvin [5] section 15.6. It is naturally described in more detail in textbooks on computer communications, such as Tanenbaum [9] and Stallings [6]. Finally, extensive coverage is provided in books specific to these protocols, such as Comer [2] or Stevens [8].

13.2 Implementation Issues

So far we have discussed what has to be done for successful communication, and how to organize the different activities. But how do you actually perform these functions? In this section we describe three major issues: error correction, flow control, and routing.

13.2.1 Error Detection and Correction

The need for error correction is the result of the well-known maxim "shit happens". It may happen that a packet sent from one computer to another will not arrive at all (e.g. because a buffer overflowed or some computer en-route crashed) or will be corrupted (e.g. because an alpha-particle passed through the memory bank in which it was stored). We would like such occurrences not to affect our important communications, be they love letters or bank transfers.

If we know an error occurred, we can request a re-send

The simplest way to deal with transmission errors is to keep the data around until it is acknowledged. If the data arrives intact, the recipient sends an acknowledgment (ack), and the sender can discard the copy. If the data arrives with an error, the recipient sends a negative Acknowledgments (nack), and the sender sends it again (in TCP, a nack is expressed as a re-send of a previous ack; in other words, the recipient

just acknowledges whatever data it had received successfully). This can be repeated any number of times, until the data finally arrives safely at its destination. This scheme is called automatic repeat request (ARQ).

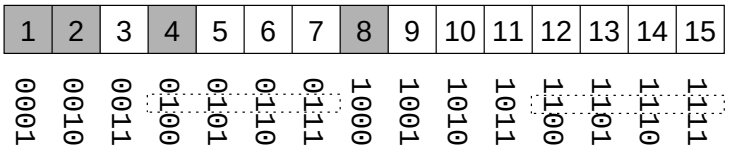
But how does the destination know if the data is valid? After all, it is just a sequence of 0's and 1's. The answer is that the data must be encoded in such a way that allows corrupted values to be identified. A simple scheme is to add a parity bit at the end. The parity bit is the binary sum of all the other bits. Thus if the message includes an odd number of 1's, the parity bit will be 1, and if it includes an even number of 1's, it will be 0. After adding the parity bit, it is guaranteed that the total number of 1's is even. A receiver that receives a message with an odd number of 1's can therefore be sure that it was corrupted.

Exercise 196 *What if two bits were corrupted? or three bits? In other words, when does parity identify a problem and when does it miss?*

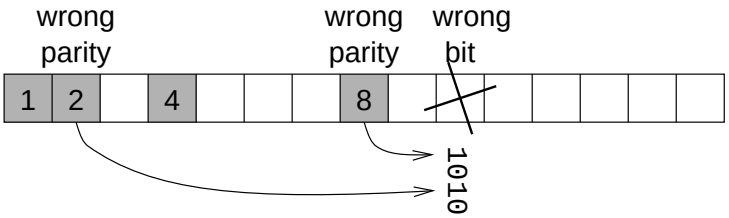
Or we can send redundant data to begin with

An alternative approach is to encode the data in such a way that we can not only detect that an error has occurred, but we can also correct the error. Therefore data does not have to be resent. This scheme is called forward error correction (FEC).

In order to be able to correct corrupted data, we need a higher level of redundancy. For example, we can add 4 parity bits to each sequence of 11 data bits. The parity bits are inserted in locations numbered by powers of two (counting from $2^0 = 1$). Each of these parity bits is then computed as the binary sum of the data bits whose position includes the parity bit position in its binary representation. For example, parity bit 4 will be the parity of data bits 5–7 and 12–15.



If any bit is corrupted, this will show up in a subset of the parity bits. The positions of the affected parity bits then provide the binary representation of the location of the corrupted data bit.



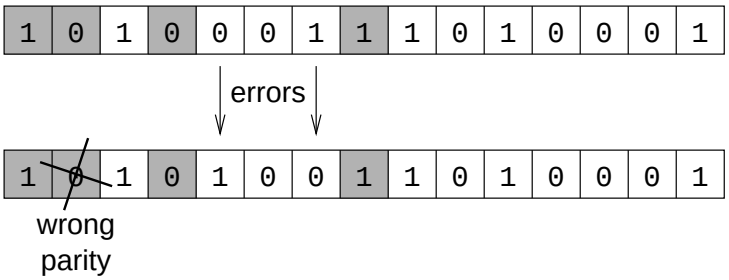
Exercise 197 *What happens if one of the parity bits is the one that is corrupted?*

Exercise 198 *Another way to provide error correction is to arrange n^2 data bits in a square, and compute the parity of each row and column. A corrupted bit then causes two parity bits to be wrong, and their intersection identifies the corrupt bit. How does this compare with the above scheme?*

The main reason for using FEC is in situations where ARQ is unwieldy. For example, FEC is better suited for broadcasts and multicasts, because it avoids the need to collect acknowledgments from all the recipients in order to verify that the data has arrived safely to all of them.

Sophisticated codes provide better coverage

The examples above are simple, but can only handle one corrupted bit. For example, if two data bits are corrupted, this may cancel out in one parity calculation but not in another, leading to a pattern of wrong parity bits that misidentifies a corrupted data bit.



The most commonly used error detection code is the cyclic redundancy check (CRC). This can be explained as follows. Consider the data bits that encode your message as a number. Now tack on a few more bits, so that if you divide the resulting number by a predefined divisor, there will be no remainder. The receiver does just that; if there is no remainder, it is assumed that the message is valid, otherwise it was corrupted.

To read more: There are many books on coding theory and the properties of the resulting codes, e.g. Arazi [1]. CRC is described in Stallings [6].

Timeouts are needed to cope with lost data

When using an error detection code (without correction capabilities) the sender must retain the sent message until it is acknowledged, because a resend may be needed. But what if the recipient does not receive it at all? In this case it will neither send an ack nor a nack, and the original sender may wait indefinitely.

The solution to this problem is to use timeouts. The sender waits for an ack for a limited time, and if the ack does not arrive, it assumes the packet was lost and retransmits it. But if the packet was only delayed, two copies may ultimately arrive! The transport protocol must deal with such situations by numbering the packets, and discarding duplicate ones.

Exercise 199 *What is a good value for the timeout interval?*

13.2.2 Buffering and Flow Control

We saw that the sender of a message must buffer it until it is acknowledged, because a resend may be needed. But the recipient and the routers along the way may also need to buffer it.

Flow control is needed to manage limited buffer space

The recipient of a message must have a buffer large enough to hold at least a single packet — otherwise when a packet comes in, there will be nowhere to store it. In fact, the need to bound the size of this buffer is the reason for specifying the maximal transfer unit allowed by a network.

Even when the available buffer space is more than a single packet, it is still bounded. If more packets arrive than there is space in the buffer, the buffer will *overflow*. The extra packets have nowhere to go, so they are dropped, meaning that for all intents and purposes they are lost. The situation in which the network is overloaded and drops packets is called *congestion*.

In order to avoid congestion, flow control is needed. Each sender has to know how much free space is available in the recipient's buffers. It will only send data that can fit into this buffer space. Then it will stop transmitting, until the recipient indicates that some buffer space has been freed. Only resends due to transmission errors are not subject to this restriction, because they replace a previous transmission that was already taken into account.

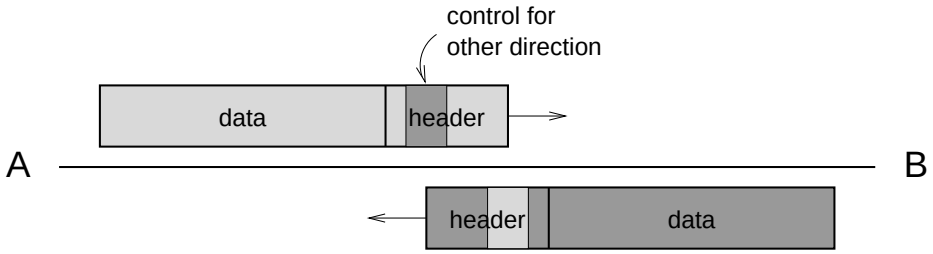
Exercise 200 *Is it realistic to know exactly how much buffer space is available? Hint: think of situations in which several different machines send data to the same destination.*

Piggybacking reduces overhead

We saw that the recipient needs to inform the sender of two unrelated conditions:

- That data has arrived correctly or not (ack or nack), so that the sender will know if it can discard old data or has to retransmit it.
- That buffer space is available for additional transmissions.

Instead of sending such control data in independent messages, it is possible to combine them into one message. Moreover, if the communication is bidirectional, the control data can *piggyback* on the data going in the other direction. Thus the headers of datagrams going from A to B contain routing and error correction for the data going from A to B, and control for data going from B to A. The opposite is true for headers of datagrams flowing from B to A.

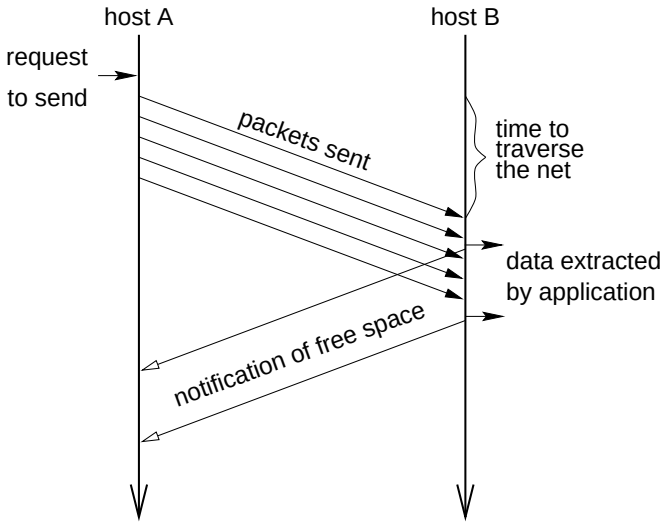


An example of how such control data is incorporated in the header of the TCP protocol is given above on page 245.

Large buffers improve utilization of network resources

If the buffer can only contain a single packet, the flow control implies that packets be sent one by one. When each packet is extracted by the application to which it is destined, the recipient computer will notify the sender, and another packet will be sent. Even if the application extracts the packets as soon as they arrive, this will lead to large gaps between packets, due to the time needed for the notification (known as the round-trip time (RTT)).

With larger buffers, a number of packets can be sent in sequence. This overlaps the transmissions with the waiting time for the notifications. In the extreme case when the propagation delay is the dominant part of the time needed for communication, the effective bandwidth is increased by a factor equal to the number of packets sent.



Exercise 201 *Denote the network bandwidth by B, and the latency to traverse the network by t_ℓ. Assuming data is extracted as soon as it arrives, what size buffer is needed to keep the network 100% utilized?*

Adaptivity can be used to handle changing conditions

Real implementations don't depend on a hardcoded buffer size. Instead, the flow control depends on a *sliding window* whose size can be adjusted.

The considerations for setting the window size are varied. On one hand is the desire to achieve the maximal possible bandwidth. As the bandwidth is limited by $BW \leq window/RTT$, situations in which the RTT is large imply the use of a large window and long timeouts. For example, this is the case when satellite links are employed. Such links have a propagation delay of more than 0.2 seconds, so the round-trip time is nearly half a second. The delay on terrestrial links, by contrast, is measured in milliseconds.

Exercise 202 *Does this mean that high-bandwidth satellite links are useless?*

On the other hand, a high RTT can be the result of congestion. In this case it is better to reduce the window size, in order to reduce the overall load on the network. Such a scheme is used in the flow control of TCP, and is described below.

An efficient implementation employs linked lists and I/O vectors

The management of buffer space for packets is difficult for two reasons:

- Packets may come in different sizes, although there is an upper limit imposed by the network.
- Headers are added and removed by different protocol layers, so the size of the message and where it starts may change as it is being processed. It is desirable to handle this without copying it to a new memory location each time.

The solution is to store the data in a linked list of small units, sometimes called *mbufs*, rather than in one contiguous sequence. Adding a header is then done by writing it in a separate mbuf, and prepending it to the list.

The problem with a linked list structure is that now it is impossible to define the message by its starting address and size, because it is not contiguous in memory. Instead, it is defined by an I/O vector: a list of contiguous chunks, each of which is defined by its starting point and size.

13.2.3 TCP Congestion Control

A special case of flow control is the congestion control algorithm used in TCP. As this is used throughout the Internet it deserves a section in itself.

Note that such congestion control is a very special form of resource management by an operating system. It has the following unique properties:

Cooperation — the control is not exercised by a single system, but rather flows from the combined actions of all the systems involved in using the Internet. It only works because they cooperate and follow the same rules.

External resource — the resource being managed, the bandwidth of the communication links, is external to the controlling systems. Moreover, it is shared with the other cooperating systems.

Indirection — the controlling systems do not have direct access to the controlled resource. They can't measure its state, and they can't affect its state. They need to observe the behavior of the communication infrastructure from the outside, and influence it from the outside. This is sometimes called "control from the edge".

Congestion hurts

The reason that congestion needs to be controlled (or rather, avoided) is that it has observable consequences. If congestion occurs, packets are dropped by the network (actually, by the routers that implement the network). These packets have to be re-sent, incurring additional overhead and delays. As a result, communication as a whole slows down.

Even worse, this degradation in performance is not graceful. It is *not* the case that when you have a little congestion you get a little degradation. Instead, you get a positive feedback effect that makes things progressively worse. The reason is that when a packet is dropped, the sending node immediately sends it again. Thus it *increases* the load on a network that is overloaded to begin with. As a result even more packets are dropped. Once this process starts, it drives the network into a state where most packets are dropped all the time and nearly no traffic gets through. The first time this happened in the Internet (October 1986), the effective bandwidth between two sites in Berkeley dropped from 32 Kb/s to 40 b/s — a drop of nearly 3 orders of magnitude [3].

Acks can be used to assess network conditions

A major problem with controlling congestion is one of information. Each transmitted packet may traverse many different links before reaching its destination. Some of these links may be more loaded than others. But the edge system, the one performing the original transmission, does not know which links will be used and what their load conditions will be.

The only thing that a transmitter knows reliably is when packets arrive to the receiver — because the receiver sends an ack. Of course, it takes the ack some time to arrive, but once it arrives the sender knows the packet went through safely. This implies that the network is functioning well.

On the other hand, the sender doesn't get any explicit notification of failures. The fact that a packet is dropped is *not* announced with a nack (only packets that arrive but have errors in them may be nack'd). Therefore the sender must infer that a packet was dropped by the absence of the corresponding ack. This is done with a timeout mechanism: if the ack did not arrive within the prescribed time, we assume that the packet was dropped.

The problem with timeouts is how to set the time threshold. If we wait and an ack fails to arrive, this could indicate that the packet was dropped. but it could just be the result of a slow link, that caused the packet (or the ack) to be delayed. The question is then how to distinguish between these two cases. The answer is that the threshold should be tied to the round-trip time (RTT): the higher the RTT, the higher the threshold should be.

Details: estimating the RTT

Estimating the RTT is based on a weighted average of measurements, similar to estimating CPU bursts for scheduling. Each "measurement" is just the time from sending a packet to receiving the corresponding ack. Given our current estimate of the RTT r, and a measurement m, the updated estimate will be

$$r_{\text{new}} = \alpha m + (1-\alpha)r$$

This is equivalent to using exponentially decreasing weights for more distant measurements. If $\alpha = \frac{1}{2}$, the weights are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$. If α is small, more old measurements are taken into account. If it approaches 1, more emphasis is placed on the most recent measurement.

The reason for using an average rather than just one measurement is that Internet RTTs have a natural variability. It is therefore instructive to re-write the above expression as [3]

$$r_{\text{new}} = r + \alpha(m-r)$$

With this form, it is natural to regard r as a predictor of the next measurement, and $m-r$ as an error in the prediction. But the error has two possible origins:

1. The natural variability of RTTs ("noise"), which is assumed to be random, so the noise in successive measurements will cancel out, and the estimate will converge to the correct value.
2. A bad prediction r, maybe due to insufficient data.

The factor α multiplies the *total* error. This means that we need to compromise. We want a large α to get the most out of the new measurement, and take a big step towards the true average value. But this risks amplifying the random error too much. It is therefore recommended to use relatively small values, such as $0.1 \leq \alpha \leq 0.2$. This will make the fluctuations of r be much smaller than the fluctuations in m, at the price of taking longer to converge.

In the context of setting the threshold for timeouts, it is important to note that we don't really want an estimate of the average RTT: we want the maximum. Therefore we also

need to estimate the variability of m, and use a threshold that takes this variability into account.

Start slowly and build up

The basic idea of TCP congestion control is to throttle the transmission rate: do not send more packets than what the network can handle. To find out how much the network can handle, each sender starts out by sending only one packet and waiting for an ack. If the ack arrives safely, two packets are sent, then four, etc.

In practical terms, controlling the number of packets sent is done using a window algorithm, just like in flow control. Congestion control is therefore just an issue of selecting the appropriate window size. In actual transmission, the system uses the minimum of the flow-control window size and the congestion control window size.

The "slow start" algorithm used in TCP is extremely simple. The initial window size is 1. As each ack arrives, the window size is increased by 1. That's it.

Exercise 203 *Adding 1 to the window size seems to lead to linear growth of the transmission rate. But this algorithm actually leads to* exponential *growth. How come?*

Hold back if congestion occurs

The problem with slow start is that it isn't so slow, and is bound to quickly reach a window size that is larger than what the network can handle. As a result, packets will be dropped and acks will be missing. When this happens, the sender enters "congestion avoidance" mode. In this mode, it tries to converge on the optimal window size. This is done as follows:

1. Set a threshold window size to half the current window size. This is the previous window size that was used, and worked OK.

2. Set the window size to 1 and restart the slow-start algorithm. This allows the network time to recover from the congestion.

3. When the window size reaches the threshold set in step 1, stop increasing it by 1 on each ack. Instead, increase it by $\frac{1}{w}$ on each ack, where w is the window size. This will cause the window size to grow much more slowly — in fact, now it will be linear, growing by 1 packet on each RTT. The reason to continue growing is twofold: first, the threshold may simply be too small. Second, conditions may change, e.g. a competing communication may terminate leaving more bandwidth available.

To read more: The classic on TCP congestion avoidance and control is the paper by Jacobson [3]. An example of recent developments in this area is the paper by Paganini et al. [4].

13.2.4 Routing

Suppose you are sending an email message to me to complain about these notes. You type in my address as `cs.huji.ac.il`. How does the email software find this machine?

The solution, like in many other cases, is to do this in several steps. The first is to translate the human-readable name to a numerical IP address. The second is to route the message from one router to another along the way, according to the IP address. This is done based on routing tables, so another question is how to create and maintain the routing tables.

Names are translated to IP addresses by DNS

Host naming and its resolution to network addresses is handled by the *Domain Name Server* (DNS). The idea is to use strings of meaningful names (that is, meaningful for humans), just like the names given to files in a file system. As with files, a hierarchical structure is created. The difference is that the names are represented in the reverse order, and dots are used to separate the components.

For example, consider looking up the host `cs.huji.ac.il`. The last component, `il`, is the top-level domain name, and includes all hosts in the domain "Israel". There are not that many top-level domains, and they hardly ever change, so it is possible to maintain a set of root DNS servers that know about *all* the top-level DNSs. Every computer in the world must be able to find its local root DNS, and can then query it for the address of the `il` DNS.

Given the address of the `il` DNS, it can be queried for the address of `ac.il`, the server of academic hosts in Israel (actually, it will be queried for the whole address, and will return the most specific address it knows; as a minimum, it should be able to return the address of `ac.il`). This server, in turn, will be asked for the address of `huji.ac.il`, the Hebrew University domain server. Finally, the address of the Computer Science host will be retrieved.

Exercise 204 *Assume the .il DNS is down. Does that mean that all of Israel is unreachable?*

Once an address is found, it should be cached for possible future use. In many cases a set of messages will be sent to the same remote computer, and caching its address saves the need to look it up for each one. For example, this happens when fetching a web page that has a number of graphical elements embedded in it.

> **Top-Level Domain Names**
> There are two types of top-level domain names: geographical and functional. In the first type, every country has its own top level domain name. Examples include:

.au	Australia
.es	Spain
.il	Israel
.it	Italy
.jp	Japan
.nl	Netherlands
.ru	Russia
.uk	United Kingdom

Somewhat surprisingly, one of the least used is the code for the United States (.us). This is because most US-based entities have preferred to use functional domains such as

.com	companies
.edu	educational institutions
.gov	government facilities
.org	non-profit organizations
.pro	professionals (doctors, lawyers)

As a sidenote, DNS is one of the major applications that use UDP rather than TCP.

IP addresses identify a network and a host

IP version 4 addresses are 32 bits long, that is four bytes. The most common way to write them is in dotted decimal notation, where the byte values are written separated by dots, as in 123.45.67.89. The first part of the address specifies a network, and the rest is a host on that network. In the past the division was done on a byte boundary. The most common were class C addresses, in which the first three bytes were allocated to the network part. This allows $2^{21} = 2097152$ networks to be identified (3 bits are used to identify this as a class C address), with a limit of 254 hosts per network (the values 0 and 255 have special meanings). Today classless addresses are used, in which the division can be done at arbitrary bit positions. This is more flexible and allows for more networks.

As noted above, IP is responsible for the routing between networks. Given an IP address, it must therefore find how to get the data to the network specified in the address.

Routing is done one step at a time

Routing is performed by routers — the computers that link different networks to each other. These routers typically do not know about all the other networks that can be reached, and the complete path by which to reach them. Instead, they know what is the next router in the right direction. The message is sent to that router, which then repeats the process and takes an additional step in the right direction. Each such step is called a hop.

Finding "the right direction" is based on a routing table that contains address prefixes, and corresponding next-hop addresses. Given an IP address, the router looks for the longest matching prefix, and sends the message to the next-hop router indicated in that entry. There is usually a default entry if nothing matches. The default entry transfers the message towards the largest routers in the backbone of the Internet. These routers have huge routing tables that should be able to resolve any request.

The routing tables are created by interactions among routers

The Internet protocols were designed to withstand a nuclear attack and not be susceptible to failures or malicious behavior. Routes are therefore created locally with no coordinated control.

When a router boots, it primes its routing table with data about immediate neighbors. It then engages in a protocol of periodically sending all the routes it knows about to its neighbors, with an indication of the distance to those destinations (in hops). When such a message arrives from one of the neighbors, the local routing table is updated with new routes or routes that are shorter than what the router knew about before.

The translation to physical addresses happens within each network

Routing tables contain the IP addresses of neighboring routers that can be used as the next hop in a routing operation. But the network hardware does not recognize IP addresses — it has its own notion of addressing, which is limited to the confines of the network. The final stage of routing is therefore the resolution of the IP address to a physical address in the same network. This is typically done using a table that contains the mapping of IP to physical addresses. If the desired address does not appear in the table, a broadcast query can be used to find it.

The same table is used to reach the final destination of a message, in the last network in the set.

13.3 Summary

Abstractions

Networking is all about abstractions: provide the abstraction that your computer is directly connected to every other computer in the world, with a reliable communication link. In reality, this is far from what the hardware provides. Many layers of software are used to bridge the gap, by providing error correction, routing, etc.

Resource management

Networking is unique in the sense that systems may participate in global resource management, rather than managing only their own resources. The prime example is TCP congestion control. The key idea there is for transmitting systems to throttle their transmissions in order to reduce the overall load on the network. Such throttling is an example of negative feedback (the higher the existing load, the more you reduce your contribution), in an effort to counter the natural positive feedback in the system (the higher the load, the more packets are dropped, leading to more retransmissions and an additional increase in load).

Workload issues

Communication workloads have several salient characteristics. One is the modal distribution of packet sizes, due to the packetization of different protocols or the MTUs of underlying hardware. Another is the self-similarity of the arrival process. In fact, networking is where self-similarity of computer workloads was first identified. In essence, self similarity means that the traffic is bursty in many different time scales, and does not tend to average out when multiple sources are aggregated. In practical terms, this means that large buffers may be needed to support the peaks of activity without losing packets.

Hardware support

The main requirement that networking places on a system is protocol processing. This may load the system's CPU at the expense of user applications. A possible solution is to provide an additional processor on the network interface card (NIC), and offload some of the protocol processing to that processor. For example, the NIC processor can handle routing and error correction without interrupting the main processor.

Bibliography

[1] B. Arazi, *A Commonsense Approach to the Theory of Error Correcting Codes*. MIT Press, 1988.

[2] D. E. Comer, *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture*. Prentice-Hall, 3rd ed., 1995.

[3] V. Jacobson, "*Congestion avoidance and control*". In *ACM SIGCOMM Conf.*, pp. 314–329, Aug 1988.

[4] F. Paganini, Z. Wang, J. C. Doyle, and S. H. Low, "*Congestion control for high performance, stability and fairness in general networks*". *IEEE/ACM Trans. Networking* **13(1)**, pp. 43–56, Feb 2005.

[5] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[6] W. Stallings, *Data and Computer Communications*. Macmillan, 4th ed., 1994.

[7] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[8] W. R. Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols*. Addison Wesley, 1994.

[9] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, 3rd ed., 1996.

Chapter 14

Distributed System Services

This chapter deals with aspects of system services that are unique to distributed systems: security and authentication, distributed shared file access, and computational load balancing.

14.1 Authentication and Security

A major problem with computer communication and distributed systems is that of trust. How do you know who is really sending you those bits over the network? And what do you allow them to do on your system?

14.1.1 Authentication

In distributed systems, services are rendered in response to incoming messages. For example, a file server may be requested to disclose the contents of a file, or to delete a file. Therefore it is important that the server know for sure who the client is. Authentication deals with such verification of identity.

Identity is established by passwords

The simple solution is to send the user name and password with every request. The server can then verify that this is the correct password for this user, and is so, it will respect the request. The problem is that an eavesdropper can obtain the user's password by monitoring the traffic on the network. Encrypting the password doesn't help at all: the eavesdropper can simply copy the encrypted password, without even knowing what the original password was!

Kerberos uses the password for encryption

The Kerberos authentication service is based on a secure authentication server (one that is locked in a room and nobody can temper with it), and on encryption. The server knows all passwords, but they are never transmitted across the network. Instead, they are used to generate encryption keys.

> **Background: encryption**
> Encryption deals with hiding the content of messages, so that only the intended recipient can read them. The idea is to apply some transformation on the text of the message. This transformation is guided by a secret key. The recipient also knows the key, and can therefore apply the inverse transformation. Eavesdroppers can obtain the transformed message, but don't know how to invert the transformation.

The login sequence is more or less as follows:

1. The client workstation where the user is trying to log in sends the user name U to the server.
2. The Kerberos server does the following:
   (a) It looks up the user's password p, and uses a one-way function to create an encryption key K_p from it. One way functions are functions that it is hard to reverse, meaning that it is easy to compute K_p from p, but virtually impossible to compute p from K_p.
   (b) It generates a new session key K_s for this login session.
   (c) It bundles the session key with the user name: $\{U, K_s\}$.
   (d) It uses its own secret encryption key K_k to encrypt this. We express such encryption by the notation $\{U, K_s\}_{K_k}$. Note that this is something that nobody can forge, because it is encrypted using the server's key.
   (e) It bundles the session key with the created unforgeable ticket, creating $\{K_s, \{U, K_s\}_{K_k}\}$.
   (f) Finally, the whole thing is encrypted using the user-key that was generated from the user's password, leading to $\{K_s, \{U, K_s\}_{K_k}\}_{K_p}$. This is sent back to the client.
3. The client does the following steps:
   (a) It prompts the user for his password p, immediately computes K_p, and erases the password. Thus the password only exists in the client's memory for a short time.
   (b) Using K_p, the client decrypts the message it got from the server, and obtains K_s and $\{U, K_s\}_{K_k}$.
   (c) It erases the user key K_p.

The session key is used to get other keys

Now, the client can send authenticated requests to the server. Each request is composed of two parts: the request itself, R, encrypted using K_s, and the unforgeable ticket. Thus the message sent is $\{R_{K_s}, \{U, K_s\}_{K_k}\}$. When the server receives such a request, it decrypts the ticket using its secret key K_k, and finds U and K_s. If this works, the server knows that the request indeed came from user U, because only user U's password could be used to decrypt the previous message and get the ticket. Then the server uses the session key K_s to decrypt the request itself. Thus even if someone spies on the network and manages to copy the ticket, they will not be able to use it because they cannot obtain the session key necessary to encrypt the actual requests.

However, an eavesdropper can copy the whole request message and retransmit it, causing it to be executed twice. Therefore the original session key K_s is not used for any real requests, and the Kerberos server does not provide any real services. All it does is to provide keys for *other* servers. Thus the only requests allowed are things like "give me a key for the file server". Kerberos will send the allocated key K_f to the client encrypted by K_s, and also send it to the file server. The client will then be able to use K_f to convince the file server of its identity, and to perform operations on files. An eavesdropper will be able to cause another key to be allocated, but will not be able to use it.

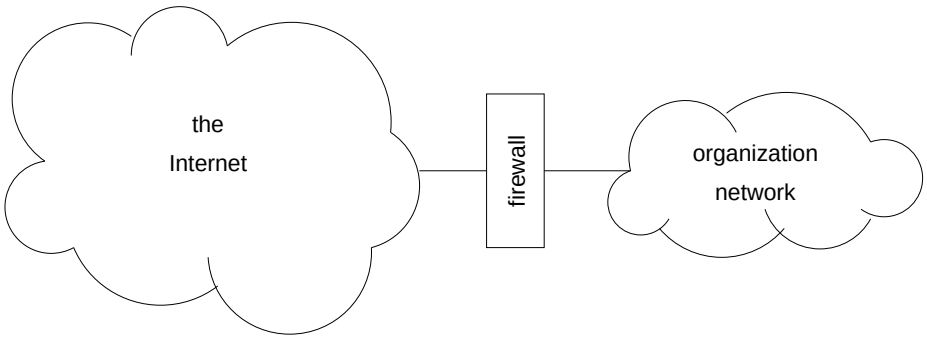
To read more: Kerberos is used in DCE, and is described in Tanenbaum's book on distributed operating systems [10, sect. 10.6].

14.1.2 Security

Kerberos can be used within the confines of a single organization, as it is based on a trusted third party: the authentication server. But on the Internet in general nobody trusts anyone. Therefore we need mechanisms to prevent malicious actions by intruders.

Security is administered by firewalls

An organization is typically connected to other organizations via a router. The router forwards outgoing communications from the internal network to the external Internet, and vice versa. This router is therefore ideally placed to control incoming packets, and stop those that look suspicious. Such a router that protects the internal network from bad things that may try to enter it is called a firewall.



The question, of course, is how to identify "bad things". Simple firewalls are just packet filters: they filter out certain packets based on some simple criterion. Criteria are usually expressed as rules that make a decision based on three inputs: the source IP address, and destination IP address, and the service being requested. For example, there can be a rule saying that datagrams from any address to the mail server requesting mail service are allowed, but requests for any other service should be dropped. As another example, if the organization has experienced break-in attempts coming from a certain IP address, the firewall can be programmed to discard any future packets coming from that address.

As can be expected, such solutions are rather limited: they may often block packets that are perfectly benign, and on the other hand they may miss packets that are part of a complex attack. A more advanced technology for filtering is to use *stateful inspection*. In this case, the firewall actually follows that state of various connections, based on knowledge of the communication protocols being used. This makes it easier to specify rules, and also supports rules that are more precise in nature. For example, the firewall can be programmed to keep track of TCP connections. If an internal host creates a TCP connection to an external host, the data about the external host is retained. Thus incoming TCP datagrams belonging to a connection that was initiated by an internal host can be identified and forwarded to the host, whereas other TCP packets will be dropped.

14.2 Networked File Systems

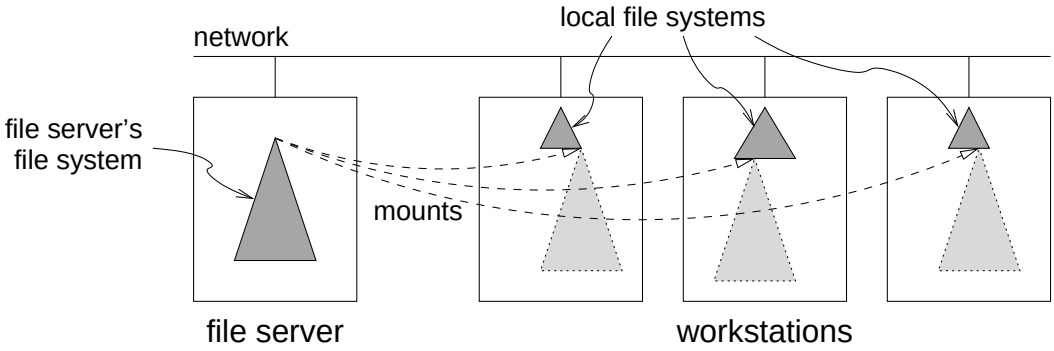
One of the most prominent examples of distributing the services of an operating system across a network is the use of networked file systems such as Sun's NFS. In fact, NFS has become a de-facto industry standard for networked Unix workstations. Its design also provides a detailed example of the client-server paradigm.

To read more: Distributed file systems are described in detail in Silberschatz chap. 17 [9] and Tanenbaum chapter 5 [10]. NFS is described by Tanenbaum [10, Sect. 5.2.5]. It is also described in detail, with an emphasis on the protocols used, by Comer [6, Chap. 23 & 24].

Remote file systems are made available by mounting

The major feature provided by NFS is support for the creation of a uniform file-system name space. You can log on to any workstation, and always see your files in the same way. But your files are not really there — they actually reside on a file server in the machine room.

The way NFS supports the illusion of your files being available wherever you are is by mounting the file server's file system onto the local file system of the workstation you are using. One only needs to specify the remoteness when mounting; thereafter it is handled transparently during traversal of the directory tree. In the case of a file server the same file system is mounted on all other machines, but more diverse patterns are possible.



Exercise 205 *The uniformity you see is an illusion because actually the file systems on the workstations are not identical — only the part under the mount point. Can you think of an example of a part of the file system where differences will show up?*

Mounting a file system means that the root directory of the mounted file system is identified with a leaf directory in the file system on which it is mounted. For example, a directory called `/mnt/fs` on the local machine can be used to host the root of the file server's file system. When trying to access a file called `/mnt/fs/a/b/c`, the local file system will traverse the local root and `mnt` directories normally (as described in Section 5.2.1). Upon reaching the directory `/mnt/fs`, it will find that this is a *mount point* for a remote file system. It will then parse the rest of the path by sending requests to access `/a`, `/a/b`, and `/a/b/c` to the remote file server. Each such request is called a *lookup*.

Exercise 206 *What happens if the local directory serving as a mount point (/mnt/fs in the above example) is not empty?*

Exercise 207 *Can the client send the whole path (/a/b/c in the above example) at once, instead of one component at a time? What are the implications?*

NFS servers are stateless

We normally consider the file system to maintain the state of each file: whether it is open, where we are in the file, etc. When implemented on a remote server, such an approach implies that the server is *stateful*. Thus the server is cognizant of sessions, and can use this knowledge to perform various optimizations. However, such a design is also vulnerable to failures: if clients fail, server may be stuck with open files that are never cleaned up; if a server fails and comes up again, clients will loose their connections and their open files. In addition, a stateful server is less scalable because it needs to maintain state for every client.

The NFS design therefore uses *stateless* servers. The remote file server does not maintain any data about which client has opened what file, and where it is in the file.

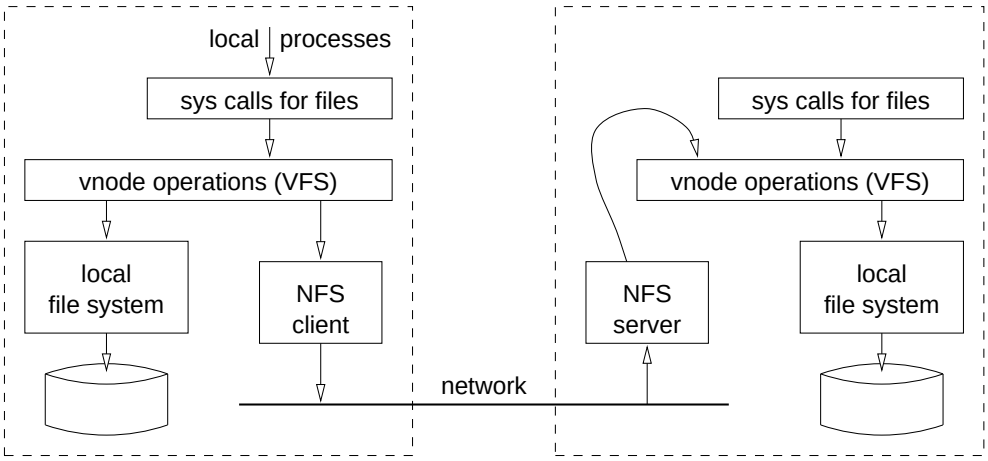
To interact with stateless servers, each operation must be self contained. In particular,

- There is no need for open and close operations at the server level. However, there is an open operation on the client, that parses the file's path name and retrieves a handle to it, as described below.
- Operations must be *idempotent*, meaning that if they are repeated, they produce the same result. For example, the system call to read the next 100 bytes in the file is not idempotent — if repeated, it will return a different set of 100 bytes each time. But the call to read the 100 bytes at offset 300 is idempotent, and will always return the same 100 bytes. The reason for this requirement is that a request may be sent twice by mistake due to networking difficulties, and this should not lead to wrong semantics.

The price of using stateless servers is a certain reduction in performance, because it is sometimes necessary to redo certain operations, and less optimizations are possible.

The virtual file system handles local and remote operations uniformly

Handling of local and remote accesses is mediated by the vfs (virtual file system). Each file or directory is represented by a *vnode* in the vfs; this is like a virtual inode. Mapping a path to a vnode is done by lookup on each component of the path, which may traverse multiple servers due to cascading mounts. The final vnode contains an indication of the server where the file resides, which may be local. If it is local, it is accessed using the normal (Unix) file system. If it is remote, the NFS client is invoked to do an RPC to the appropriate NFS server. That server injects a request to its vnode layer, where it is served locally, and then returns the result.



State is confined to the NFS client

For remote access, the NFS client contains a handle that allows direct access to the remote object. This handle is an *opaque object* created by the server: the client receives it when parsing a file or directory name, stores it, and sends it back to the server to identify the file or directory in subsequent operations. The content of the handle is only meaningful to the server, and is used to encode the object's ID. Note that using such handles does not violate the principle of stateless servers. The server generates handles, but does not keep track of them, and may revoke them based on a timestamp contained in the handle itself. If this happens a new handle has to be obtained. This is done transparently to the application.

The NFS client also keeps pertinent state information, such as the position in the file. In each access, the client sends the current position to the server, and receives the updated position together with the response.

The semantics are fuzzy due to caching

Both file attributes (inodes) and file data blocks are cached on client nodes to reduce communication and improve performance. The question is what happens if multiple clients access the same file at the same time. Desirable options are

Unix semantics: this approach is based on uniprocessor Unix systems, where a single image of the file is shared by all processes (that is, there is a single inode and single copy of each block in the buffer cache). Therefore modifications are immediately visible to all sharing processes.

Session semantics: in this approach each user gets a separate copy of the file data, that is only returned when the file is closed. It is used in the Andrew file system, and provides support for disconnected operation.

The semantics of NFS are somewhat less well-defined, being the result of implementation considerations. In particular, shared write access can lead to data loss. How-

ever, this is a problematic access pattern in any case. NFS does not provide any means to lock files and guarantee exclusive access.

Exercise 208 *Can you envision a scenario in which data is lost?*

14.3 Load Balancing

Computational servers provide cycles

Just like file servers that provide a service of storing data, so computational servers provide a service of hosting computations. They provide the CPU cycles and physical memory needed to perform the computation. The program code is provided by the client, and executed on the server.

Computation can be done on a server using mobile code, e.g. a Java application. But another method is to use process migration. In this approach, the process is started locally on the client. Later, if it turns out that the client cannot provide adequate resources, and that better performance would be obtained on a server, the process is moved to that server. In fact, in a network of connected PCs and workstations all of them can be both clients and servers: Whenever any machine has cycles (and memory) to spare, it can host computations from other overloaded machines.

Exercise 209 *Are there any restrictions on where processes can be migrated?*

Migration should be amortized by lots of computation

There are two major questions involved in migration for load balancing: which process to migrate, and where to migrate it.

The considerations for choosing a process for migration involve its size. This has two dimensions. In the space dimension, we would like to migrate small processes: the smaller the used part of the process's address space, the less data that has to be copied to the new location. In the time dimension, we would like to migrate long processes, that will continue to compute for a long time after they are migrated. Processes that terminate soon after being migrated waste the resources invested in moving them, and would have done better staying where they were.

Luckily, the common distributions of job runtimes are especially well suited for this. As noted in Section 2.3, job runtimes are well modeled by a heavy-tailed distribution. This means that a small number of processes dominate the CPU usage. Moving only one such process can make all the difference.

Moreover, it is relatively easy to identify these processes: they are the oldest ones available. This is because a process that has already run for a long time can be assigned to the tail of the distribution. The distribution of process lifetimes has the property that if a process is in its tail, it is expected to run for even longer, more so than processes that have only run for a short time, and (as far as we know) are not from the tail of the distribution. (This was explained in more detail on page 186.)

Details: estimating remaining runtime

A seemingly good model for process runtimes, at least for those over a second long, is that they follow a Pareto distribution with parameter near -1:

$$\Pr(r > t) = 1/t$$

The conditional distribution describing the runtime, given that we already know that it is more than a certain number T, is then

$$\Pr(r > t | r > T) = T/t$$

Thus, if the current age of a process is T, the probability that it will run for more than $2T$ time is about $1/2$. In other words, the median of the expected remaining running time grows linearly with the current running time.

To read more: The discussion of process lifetimes and their interaction with migration is based on Harchol-Balter and Downey [7].

Choosing a destination can be based on very little information

Choosing a destination node seems to be straightforward, but expensive. We want to migrate the process to a node that is significantly less loaded than the one on which it is currently. The simple solution is therefore to query that status of all other nodes, and choose the least loaded one (assuming the difference in loads is large enough).

A more economical solution is to randomly query only a subset of the nodes. It turns out that querying a rather small subset (in fact, a small constant number of nodes, independent of the system size!) is enough in order to find a suitable migration destination with high probability. An additional optimization is to invert the initiative: have underloaded nodes scour the system for more work, rather than having overloaded ones waste their precious time on the administrative work involved in setting up a migration.

Example: the Mosix system

The Mosix system is a modified Unix, with support for process migration. At the heart of the system lies a randomized information dispersal algorithm. Each node in the system maintains a short load vector, with information regarding the load on a small set of nodes [5]. The first entry in the vector is the node's own load. Other entries contain data that it has received about the load on other nodes.

At certain intervals, say once a minute, each node sends its load vector to some randomly chosen other node. A node that received such a message merges it into its own load vector. This is done by interleaving the top halves of the two load vectors, and deleting the bottom halves. Thus the retained data is the most up-to-date available.

Exercise 210 *Isn't there a danger that all these messages will overload the system?*

The information in the load vector is used to obtain a notion of the general load on the system, and to find migration partners. When a node finds that its load differs significantly from the perceived load on the system, it either tries to offload processes to some underloaded node, or solicits additional processes from some overloaded node. Thus load is moved among the nodes leading to good load balancing within a short time.

The surprising thing about the Mosix algorithm is that it works so well despite using very little information (the typical size of the load vector is only four entries). This turns out to have a solid mathematical backing. An abstract model for random assignment of processes is the throwing of balls into a set of urns. If n balls are thrown at random into n urns, one should not expect them to spread out evenly. In fact, the probability that an urn stay empty is $\left(1-\frac{1}{n}\right)^n$, which for large n tends to $1/e$ — i.e. more than a third of the urns stay empty! At the other extreme, the urn that gets the most balls is expected to have about $\log n$ balls in it. But if we do not choose the urns at random one at a time, but rather select two each time and place the ball in the emptier of the two, the expected number of balls in the fullest urn drops to $\log\log n$ [3]. This is analogous to the Mosix algorithm, that migrates processes to the less loaded of a small subset of randomly chosen nodes.

In addition to its efficient assignment algorithm, Mosix employs a couple of other heuristic optimizations. One is the preferential migration of "chronic forgers" — processes that fork many other processes. Migrating such processes effectively migrates their future offspring as well. Another optimization is to amortize the cost of migration by insisting that a process complete a certain amount of work before being eligible for migration. This prevents situations in which processes spend much of their time migrating, and do not make any progress.

The actual migration is done by stopping the process and restarting it

Negotiating the migration of a process is only part of the problem. Once the decision to migrate has been made, the actual migration should be done. Of course, the process itself should continue its execution as if nothing had happened.

To achieve this magic, the process is first blocked. The operating systems on the source and destination nodes then cooperate to create a perfect copy of the original process on the destination node. This includes not only its address space, but also its description in various system tables. Once everything is in place, the process is restarted on the new node.

Exercise 211 *Can a process nevertheless discover that it had been migrated?*

Exercise 212 *Does all the process's data have to be copied before the process is restarted?*

Special care is needed to support location-sensitive services

But some features may not be movable. For example, the process may have opened some files that are available locally on the original node, but are not available on

the new node. It might perform I/O to the console of the original node, and moving this to the console of the new node would be inappropriate. Worst of all, it might have network connections with remote processes somewhere on the Internet, and it is unreasonable to update all of them about the migration.

The solution to such problems is to maintain a presence on the original node, to handle the location-specific issues. Data is forwarded between the body of the process and its home-node representative as needed by the operating system kernel.

To read more: Full details about the Mosix system are available in Barak, Guday, and Wheeler [4]. More recent publications include [2, 8, 1].

Bibliography

[1] L. Amar, A. Barak, and A. Shiloh, "*The MOSIX direct file system access method for supporting scalable cluster file systems*". *Cluster Computing* **7(2)**, pp. 141–150, Apr 2004.

[2] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "*An opportunity cost approach for job assignment in a scalable computing cluster*". *IEEE Trans. Parallel & Distributed Syst.* **11(7)**, pp. 760–768, Jul 2000.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "*Balanced allocations*". In 26th *Ann. Symp. Theory of Computing*, pp. 593–602, May 1994.

[4] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993. Lect. Notes Comput. Sci. vol. 672.

[5] A. Barak and A. Shiloh, "*A distributed load-balancing policy for a multicomputer*". *Software — Pract. & Exp.* **15(9)**, pp. 901–913, Sep 1985.

[6] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications*. Prentice Hall, 2nd ed., 1996.

[7] M. Harchol-Balter and A. B. Downey, "*Exploiting process lifetime distributions for dynamic load balancing*". *ACM Trans. Comput. Syst.* **15(3)**, pp. 253–285, Aug 1997.

[8] R. Lavi and A. Barak, "*The home model and competitive algorithms for load balancing in a computing cluster*". In 21st *Intl. Conf. Distributed Comput. Syst.*, pp. 127–134, Apr 2001.

[9] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[10] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.

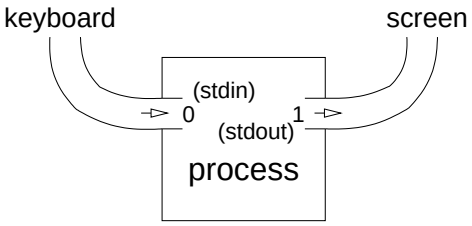
Appendix E

Using Unix Pipes

One of the main uses for Unix pipes is to string processes together, with the `stdout` (standard output) of one being piped directly to the `stdin` (standard input) of the next. This appendix explains the mechanics of doing so.

Appendix **??** described how Unix processes are created by the `fork` and `exec` system calls. The reason for the separation between these two calls is the desire to be able to manipulate the file descriptors, and string the processes to each other with pipes.

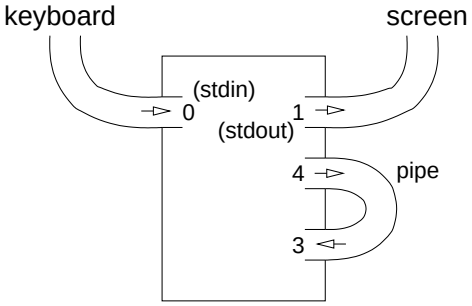
By default, a Unix process has two open files predefined: standard input (stdin) and standard output (stdout). These may be accessed via file descriptors 0 and 1, respectively.



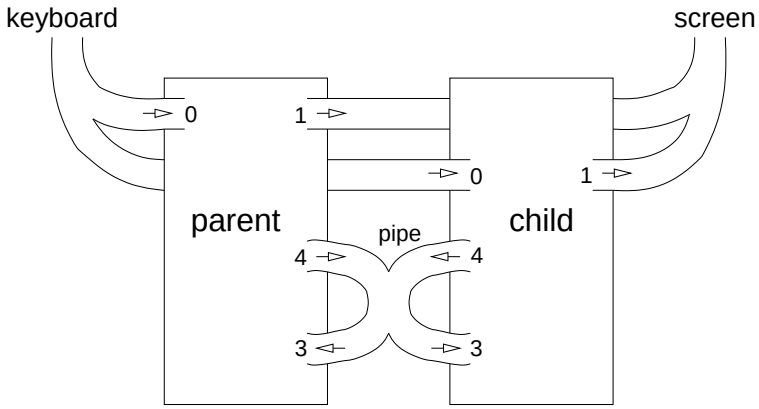
Again by default, both are connected to the user's terminal. The process can read what the user types by reading from stdin, that is, from file descriptor 0 — just like reading from any other open file. It can display stuff on the screen by writing to stdout, i.e. to file descriptor 1.

Using pipes, it is possible to string processes to each other, with the standard output of one connected directly to the standard input of the next. The idea is that the program does not have to know if it is running alone or as part of a pipe. It reads input from `stdin`, processes it, and writes output to `stdout`. The connections are handled *before* the program starts, that is, before the `exec` system call.

To connect two processes by a pipe, the first process calls the `pipe` system call. This creates a channel with an input side (from which the process can read) and an output side (to which it can write). They are available to the calling process as file descriptors 3 and 4 respectively (file descriptor 2 is the predefined standard error, `stderr`).



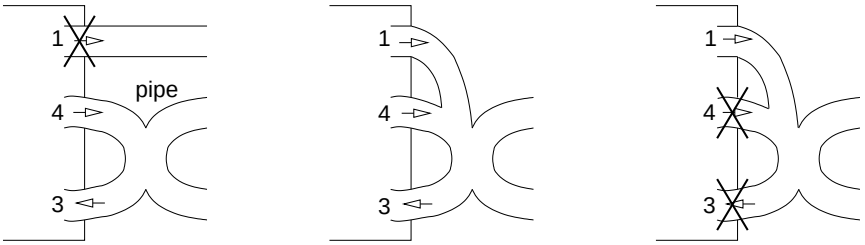
The process then forks, resulting in two processes that share their `stdin`, `stdout`, and the pipe.



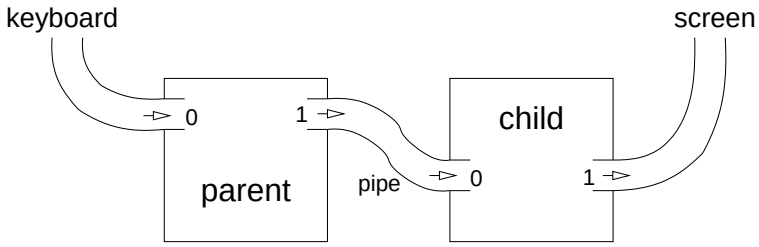
This is because open files are inherited across a fork, as explained on page 135.

Exercise 213 *What happens if the two processes actually read from their shared* `stdin`*, and write to their shared* `stdout`*? Hint: recall the three tables used to access open files in Unix.*

To create the desires connection, the first (parent) process now closes its original `stdout`, i.e. its connection to the screen. Using the `dup` system call, it then duplicates the output side of the pipe from its original file descriptor (4) to the `stdout` file descriptor (1), and closes the original (4). It also closes the input side of the pipe (3).



The second (child) process closes its original `stdin` (0), and replaces it by the input side of the pipe (3). It also closes the output side of the pipe (4). As a result the first process has the keyboard as `stdin` and the pipe as `stdout`, and the second has the pipe as `stdin` and the screen as `stdout`. This completes the desired connection. If either process now does an `exec`, the program will not know the difference.



Exercise 214 *What is the consequence if the child process does not close its output side of the pipe (file desceriptor 4)?*

Appendix F

The ISO-OSI Communication Model

As part of an effort to standardize comunnication protocols, The International Standards Organization (ISO) has defined a protocol stack with seven layers called the Open Systems Interconnect (OSI). The layers are

1. Physical: the network interface hardware, including connectors, signaling conventions, etc.

2. Data link: flow control over a single link, buffering, and error correction. Higher levels can assume a reliable communication medium.

3. Network: routing. This is the top layer used in intermediate routing nodes. Higher levels need not know anything about the network topology.

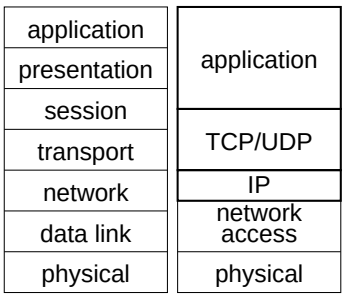
4. Transport: packetization and end-to-end verification. If a node fails along the way, the end-to-end checks will rectify the situation and re-send the required packet. Higher levels can assume a reliable connection.

5. Session: control over the dialog between end stations (often unused).

6. Presentation: handling the representation of data, e.g. compression and encryption.

7. Application: an interface for applications providing generally useful services, e.g. distributed database support, file transfer, and remote login.

In the context of operating systems, the most important are the routing and transport functions. In practice, the TCP/IP protocol suite became a de-facto standard before the OSI model was defined. It actually handles the central part of the OSI stack.



To read more: The OSI model is described briefly in Silberschatz and Galvin [1] section 15.6. Much more detailed book-length descriptions were written by Tanenbaum [3] and Stallings [2].

Bibliography

[1] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[2] W. Stallings, *Data and Computer Communications*. Macmillan, 4th ed., 1994.

[3] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, 3rd ed., 1996.

Answers to Exercises

Exercise 1: It can't. It needs hardware support in the form of a clock interrupt which happens regularly. On the other hand, it can be argued that the operating system does not really need to regain control. This is discussed in Section 11.2.

Exercise 3: System tables should be kept small because their storage comes at the expense of memory for users, and processing large tables generates higher overheads. The relative sizes of different tables should reflect their use in a "normal" workload; if one table is always the bottleneck and becomes full first, space in the other tables is wasted.

Exercise 4: First and foremost, privileged instructions that are used by the operating system to perform its magic. And maybe also low-level features that are hidden by higher-level abstractions, such as block-level disk access or networking at the packet level.

Exercise 5: Typically no. Instructions are executed by the hardware, and the operating system is not involved. However, it may be possible to emulate new instructions as part of handling an illegal-instruction exception.

Exercise 6: There are a number of ways. In most systems, an application can simply request to see a list of the processes running on the system. Even without this, an application can repeatedly read the system clock; when it notices a long gap between successive reading this implies that something else ran during this interval.

Exercise 7:

1. Change the program counter: Not privileged — done by every branch.
2. Halt the machine: privileged.
3. Divide by zero: division is not privileged, but division by zero causes an interrupt.
4. Change the execution mode: changing from user mode to kernel mode is done by the trap instruction, which is not protected. However, it has the side effect of also changing the PC to operating system code. Changing from kernel mode to user mode is only relevant in kernel mode.

Exercise 8: When the first privileged instruction is reached, the CPU will raise an illegal instruction exception. The operating system will then terminate your program.

Exercise 10: Some, that are managed by the operating system, are indeed limited to this resolution. For example, this applies to sleeping for a certain amount of time. But on some architectures (including the Pentium) it is possible to access a hardware cycle counter directly, and achieve much better resolution for application-level measurements.

Exercise 11: Assuming the interrupt handlers are not buggy, only an asynchronous interrupt can occur. If it is of a higher level, and not blocked, it will be handled immediately, causing a nesting of interrupt handlers. Otherwise the hardware should buffer it until the current handler terminates.

Exercise 12: The operating system, because they are defined by the services provided by the operating system, and not by the language being used to write the application.

Exercise 13: In principle, this is what happens in a branch: the "if" loads one next instruction, and the "else" loads a different next instruction.

Exercise 14: If the value is not the address of an instruction, we'll get an exception. Specifically, if the value specifies an illegal memory address, e.g. one that does not exist, this leads to a memory fault exception. If the value is an address within the area of memory devoted to the program, but happens to point into the middle of an instruction, this will most probably lead to an illegal instruction exception.

Exercise 15: Obviously each instruction in the text segment contains the addresses of its operands, but only those in the data segment can be given explicitly, because the others are not known at compile time (instead, indirection must be used). Pointer variables can hold the addresses of other variables, allowing data, heap, and stack to point to each other. There are also self-references: for example, each branch instruction in the text segment contains the address to which to go if the branch condition is satisfied.

Exercise 16: General purpose register contents are changed by instructions that use them as the target of a computation. Other registers, such as the PC and PSW, change as a side effect of instruction execution. The SP changes upon function call or return.

The text segment cannot normally be modified. The data, heap, and stack are modified by instructions that operate on memory contents. The stack is also modified as a side effect of function call and return.

The contents of system tables is only changed by the operating system. The process can cause such modifications by appropriate system calls. For example, system calls to open or close a file modify the open files table.

Exercise 17: The stack is used when this process calls a function; this is an internal thing that is part of the process's execution. The PCB is used by the operating system

when it performs a context switch. This is handy because it allows the operating system to restore register contents without having to delve into the process's stack.

Exercise 19: Real-time applications, especially those that need to handle periodic events that occur at a given rate. Examples: the application that draws the clock on your screen needs to wake up every minute to re-draw the minutes hand; a media player needs to wake up many times per second to display the next frame or handle the next audio sample.

Exercise 20: To share the CPU with other processes (multiprogramming).

Exercise 21: Because the process becomes unblocked as a result of an action taken by the operating system, e.g. by an interrupt handler. When this happens the operating system is running, so the process cannot start running immediately (there's only one CPU 8-). Thus the interrupt handler just makes the process ready to run, and it will actually run only when selected by the scheduler, which is another component of the operating system.

Exercise 22: Add a "suspended" state, with transitions from ready and blocked to suspended, and vice versa (alternatively, add two states representing suspended-ready and suspended-blocked). There are no transition from the running state unless a process can suspend itself; there is no transition to the running state because resumption is necessarily mediated by another process.

Exercise 23: Registers are part of the CPU. Global variables should be shared, so should not be stored in registers, as they might not be updated in memory when a thread switch occurs. The stack is private to each thread, so local variables are OK.

Exercise 24: Directly no, because each thread has its own stack pointer. But in principle it is possible to store the address of a local variable in a global pointer, and then all threads can access it. However, if the original thread returns from the function the pointer will be left pointing to an unused part of the stack; worse, if that thread subsequently calls *another* function, the pointer may accidentally point to something completely different.

Exercise 25: Yes, to some extent. You can set up a thread to prefetch data before you really need it.

Exercise 26: It depends on where the pointer to the new data structure is stored. If it is stored in a global variable (in the data segment) all the other threads can access it. But if it is stored in a private variable on the allocating thread's stack, the others don't have any access.

Exercise 27: Yes — they are not independent, and one may block the others. For example, this means that they cannot be used for asynchronous I/O.

Exercise 28: A relatively simple aporoach is to use several operating system threads as the basis for a user-level thread package. Thus when a user-level thread performs an I/O, only the operating system thread currently running it is blocked, and the other

operating system threads can still be used to execute the other user-level threads.

A more sophisticated approach can set aside a low-priority control thread, and use only one other thread to run all the user-level threads. This has the advantage of reduced overhead and interaction with the operating system. But if any user-level thread blocks, the low-priority control thread will run. It can then spawn a new kernel thread to serve the other user-level threads. When a user-level thread unblocks, it's kernel thread can be suspended. Thus kernel threads are only used for blocked user-level threads.

Exercise 30: they all terminate.

Exercise 31: In general, sleeping is useful in (soft) real-time applications. For example, a movie viewer needs to display frames at a rate of 30 frames per second. If it has finished decoding the displaying the current frame, it has nothing to do until the next frame is due, so it can sleep.

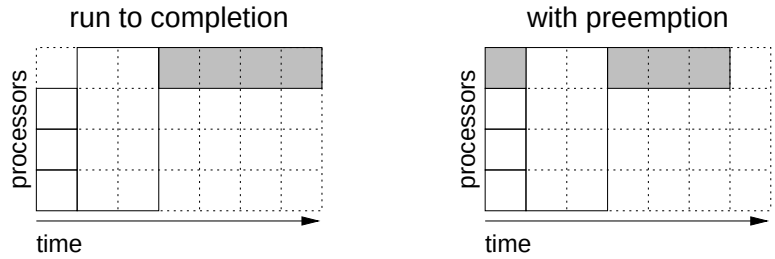
Exercise 33: The system models are quite different. The M/M/1 analysis is essentially a single server with FCFS scheduling. Here we are talking of multiple servers — e.g. the CPU and the disk, and allowing each to have a separate queue.

Exercise 34: If there is only one CPU and multiple I/O devices, including disks, terminals, and network connections. It can then be hoped that the different I/O-bound jobs are actually not identical, and will use different I/O devices. In a multiprocessor, multiple compute-bound jobs are OK.

Exercise 36:

metric	*range*	*preference*
response time	> 0	low is better
wait time	> 0	low is better
response ratio	> 1	low is better
throughput	> 0	high is better
utilization	$[0, 1]$	high is better

Exercise 37: For single-CPU jobs, the same argument holds. For multiple-CPU jobs, there are cases where an off-line algorithm will prefer using preemption over run-to-completion. Consider an example of 4 CPUs and 5 jobs: 3 jobs requiring one CPU for one unit of time, a job requiring one CPU for 4 units, and a job requiring all 4 CPUs for two units of time. With run to completion, the average response time is $\frac{13}{5} = 2.6$ units, but with preemption it is possible to reduce this to $\frac{12}{5} = 2.4$ units:



Exercise 38: When all jobs are available from the outset, yes.

Exercise 40: Just before, so that they run immediately (that is, at the end of the current quantum). Then only jobs that are longer than one quantum have to wait for a whole cycle.

Exercise 41: The shorter they are, the better the approximation of processor sharing. But context switching also has an overhead, and the length of the time slice should be substantially larger than the overhead to keep the relative overhead (percentage of time spent on overhead rather than on computation) acceptably low. In addition to the direct overhead (time spent to actually perform the context switch) there is an effect on the cache efficiency of the newly run process: when a process is scheduled to run it will need to reload its cache state, and suffer from many cache misses. The quantum should be long enough to amortize this. Typical values are between 0.01 and 0.1 of a second.

Exercise 42:

$$E_{n+1} = \alpha T_n + (1 - \alpha) E_n$$

Exercise 43: Yes. When waiting for user input, their priority goes up due to the exponential aging (recall that lower values reflect higher priority).

Exercise 45: If the process runs continuously, it will gain 100 points a second, but they will be halved on each subsequent second. The grand total will therefore approach 200.

Exercise 47: To a point. You can use multi-level feedback among the jobs in each group, with the allocation among groups being done by fair shares. You can also redistribute the tickets within a group to reflect changing priorities.

Exercise 48: It is surprising because it is used to verify access privileges, which are only relevant when the process is running and trying to access something. The reason is the desire to be able to list all the processes in the system (including those that are not running at the moment) with their owners (see man ps).

Exercise 49: A process only gets blocked when waiting for some service, which is requested using a system call, and therefore only happens when running in kernel mode.

Exercise 50: Because context switching is done in kernel mode. Thus a process is always scheduled in kernel mode, completes the end of the context-switch routine, and only then returns to user mode.

Exercise 51: Ready to run in kernel mode, as it is still in the `fork` system call.

Exercise 52: It is copied to kernel space and then reconstructed.

Exercise 54:

1. Another process runs first and starts to add another new item after current. Call this item `new2`. The other process performs `new2->next = current->next` and is interrupted.
2. Now "our" process links `new` after `current` with out any problems. So `current` now points to `new`.
3. The other process resumes and overwrites `current`, making it point to `new2`. `new` now points to whatever was after `current` originally, but nothing points to it.

Exercise 55: No. It may be preempted, and other processes may run, as long as they do not enter the critical section. Preventing preemptions is a sufficient but not necessary condition for mutual exclusion. Note that in some cases keeping hold of the processor is actually not desirable, as the process may be waiting for I/O as part of its activity in the critical section.

Exercise 56: Yes. As the case where both processes operate in lockstep leads to deadlock, we are interested in the case when one starts before the other. Assume without loss of generality that process 1 is first, and gets to its `while` loop before process 2 sets the value of `going_in_2` to `TRUE`. In this situation, process 1 had already set `going_in_1` to `TRUE`. Therefore, when process 2 gets to its `while` loop, it will wait.

Exercise 58: The problem is that the value of `current_ticket` can go down. Here is a scenario for three processes, courtesy of Grisha Chockler:

P1: choosing[1] = TRUE;
P1: my_ticket[1]=1;
P1: starts incrementing current_ticket: it reads the current value of current_ticket (which is 1) and goes to sleep;
P2: choosing[2] = TRUE;
P2: my_ticket[2]=1;
P2: succeeds to increment current_ticket exclusively: current_ticket = 2;
P2: choosing[2]=FALSE;
P3: choosing[3]=TRUE;
P3: my_ticket[3]=2;
P3: succeeds to increment current_ticket exclusively: current_ticket=3;
P3: choosing[3]=FALSE;
P1: Wakes up, increments what it thinks to be the last value of current_ticket, i.e., 1. So now current_ticket=2 again!

Now all three processes enter their for loops. At this point everything is still OK: They all see each other ticket: P1's ticket is (1,1), P2's ticket is (1,2), and P3's ticket is (2,3). So P1 enters the critical section first, then P2 and finally P3.

Now assume that P3 is still in the critical section, whereas P1 becomes hungry again and arrives at the bakery doorway (the entry section). P1 sees current_ticket=2 so it chooses (2,1) as its ticket and enters the for loop. P3 is still in the critical section, but its ticket (2,3) is higher than (2,1)! So P1 goes ahead and we have two processes simultaneously in the critical section.

Exercise 59: Once a certain process chooses its number, every other process may enter the critical section before it at most once.

Exercise 60:

1. At any given moment, it is possible to identify the next process that will get into the critical section: it is the process with the lowest ID of those that have the lowest ticket number.
2. Correctness follows from the serial ordering implied by the progress.
3. Again, when a process gets its ticket and increments the global counter, its place in the sequence is determined. Thereafter it can only be leapfrogged by a finite number of processes that managed to get the same ticket number, and have lower IDs.
4. Obviously.

Exercise 61: Initially set `guard=OPEN`. Then

```
while ( !  compare_and_swap( &guard, OPEN, CLOSED ) ) /*empty*/;
critical section
guard = OPEN;
```

Note the negation in the `while` condition: when the compare_and_swap succeeds, we want to *exit* the loop and proceed to the critical section.

Exercise 63: It is apparently wasteful because only one of these processes will succeed in obtaining the semaphore, and all the others will fail and block again. The reason to do it is that it allows the scheduler to prioritize them, so that the highest priority process will be the one that succeeds. If we just wake the first waiting process, this could happen to be a low-priority process, which will cause additional delays to a high-priority process which is further down in the queue.

Exercise 64: It is wrong to create a semaphore for each pair, because even when the A/B lists semaphore is locked, another process may nevertheless cause inconsistencies by locking the A/C semaphore and manipulating list A. The correct way to go is to have a semaphore per data structure, and lock all those you need: to move an item between lists A and B, lock both the A and B semaphores. However, this may cause deadlock problems, so it should be done with care.

Exercise 65: The solution uses 3 semaphores: `space` (initialized to the number of

available buffers) will be used to block the producer if all buffers are full, `data` (initially 0) will be used to block the consumer if there is no data available, and `mutex` (initially 1) will be used to protect the data. The pseudocode for the producer is

```
while (1) {
   /* produce new item */
   P(space)
   P(mutex)
   /* place item in next free buffer */
   V(mutex)
   V(data)
}
```

The pseudocode for the consumer is

```
while (1) {
   P(data)
   P(mutex)
   /* remove item from the first occupied buffer */
   V(mutex)
   V(space)
   /* consume the item */
}
```

Note, however, that this version prevents the producer and consumer from accessing the buffer at the same time even if they are accessing different cells. Can you see why? Can you fix it?

Exercise 66: The previous solution doesn't work because if we have multiple producers, we can have a situation where producer1 is about to place data in buffer[1], and producer2 is about to place data in buffer[2], but producer2 is faster and finishes first. producer2 then performs V(data) to signal that data is ready, but a consumer will try to get this data from the first occupied buffer, which as far as it knows is buffer[1], which is actually not ready yet. The solution is to add a semaphore that regulates the flow into and out of each buffer.

Exercise 67: Emphatically, Yes.

Exercise 68: This is exactly the case when a kernel function terminates and the process is returning to user mode. However, instead of running immediately, it has to wait in the ready queue because some other process has a higher priority.

Exercise 69: Preventing preemptions does not guarantee atomicity on multiprocessors. They guarantee that another process will not run (and therefore not access operating system data structures) on *this* processor, but if there are other processors available, a process can run on one of them and issue a system call. Early versions of Unix for multiprocessors therefore required all system calls to be processed on a

single designated processor.

Exercise 75: If they are located at the same place and have the same capabilities, they may be used interchangeably, and are instances of the same type. Usually, however, printers will be located in different places or have different capabilities. For example, "B/W printer on first floor" is a different resource type than "color printer on third floor".

Exercise 76: The system is deadlocked if the cycle involves all the instances of each resource participating in it, or — as a special case — if there is only one instance of each. This condition is also implied if the cycle contains all the processes in the system, because otherwise some request by some process could be fulfilled, and the cycle would be broken.

Exercise 77: No on both counts. If a lock is preempted, the data structure it protects may be left in an inconsistent state. And swapping out a process holding a lock will cause the lock to remain locked for an excessively long time.

Exercise 78: The resources should be ordered by a combination of popularity and average holding time. Popular resources that are held for a short time on average should be at the end of the list, because they will suffer more from "over holding".

Exercise 79: Because if a process can request an additional instance of a resource it already has, two such processes can create a deadlock.

Exercise 80: A uniform rule leads to deadlock. But with a traffic circle, giving right-of-way to cars already in the circle prevents deadlock, because it de-couples the cars coming from the different directions.

Exercise 84: Each resource (lock) has a single instance, and all processes may want to acquire all the resources. Thus once any one acquires a lock, we must let it run to completion before giving any other lock to any other process. In effect, everything is serialized.

Exercise 85: In principle yes — that the rules are being followed. For example, if the rule is that resources be requested in a certain order, the system should check that the order is not being violated. But if you are sure there are no bugs, and are therefore sure that the rules are being followed, you can leave it to faith.

Exercise 87: It's more like livelock, as the processes are active and not blocked. In principle, the operating system can detect it by keeping track of previous system calls, but this entails a lot of bookkeeping. Alternatively, it is possible to claim that as far as the operating system is concerned things are OK, as the processes are active and may eventually actually make progress. And if quotas are in place, a program may be killed when it exceeds its runtime quota, thus releasing its resources for others.

Exercise 88: Prevent deadlock by making sure runqueues are always locked in the same order, by using their addresses as unique identifiers.

Exercise 90: Yes. Locking takes the pessimistic view that problems will occur and takes extreme precautions to prevent them. Using compare_and_swap takes the optimistic approach. If the actual conflict occurs in a small part of the code, there is a good chance that things will actually work out without any undue serialization.

Moreover, the wait-free approach guarantees progress: if you need to try again, it is only because some other process changed the data concurrently with you. Your failure signals his success, so he made progress. In a related vein, with locks a process that is delayed while holding a lock leads to a cascading effect, where many other processes may be delayed too. With wait-free synchronization, the other processes will slip through, and the process that was delayed will have to try again.

Exercise 91: It is only possible if the application does not store virtual addresses and use them directly, because then the operating system cannot update them when the mapping is changed.

Exercise 93: If the bounds are not checked, the program may access storage that is beyond the end of the segment. Such storage may be unallocated, or may be part of another segment. The less-serious consequence is that the program may either produce wrong results or fail, because the accessed storage may be modified undeterministically via access to the other segment. The more serious consequence is that this may violate inter-process protection if the other segment belongs to another process.

Exercise 95: Did you handle the following special cases correctly?

1. allocation when all memory is free
2. allocation from a free region at address 0
3. an allocation that completely uses a free range, which should then be removed from the list
4. a deallocation that creates a new free range at address 0
5. a deallocation that is adjacent to a free region either before or after it, and should be united with it rather than creating a new free region
6. a deallocation that is adjacent to free regions both before and after it, so the three should be united into one

really good code would also catch bugs such as deallocation of a range that was actually free.

Exercise 96: Both algorithm allocate at the beginning of a free segment, so the only candidates are the segments at addresses 73, 86, and 96. Of these the segment of size 22 at address 96 is the only one that does not fit into free segments before it, so it is the only one that could be allocated by first fit. Any of the three could have been allocated last by best fit.

Exercise 97: Both claims are not true. Assume two free regions of sizes 110 and 100, and a sequence of requests for sizes 50, 60, and 100. First-fit will pack them perfectly, but best-fit will put the first request for 50 in the second free area (which leaves less

free space), the second request in the first free area (only option left), and then fail to allocate the third. Conversely, if the requests are for 100 and 110, best-fit will pack them perfectly, but first-fit will allocate 100 of 110 for the first request and fail to allocate the second.

Exercise 98: Next fit and its relatives suffer from external fragmentation. Buddy may suffer from both.

Exercise 99: External only. Internal fragmentation can only be reduced by using smaller chunks in the allocation.

Exercise 100: No. If the page is larger than a block used for disk access, paging will be implemented by several disk accesses (which can be organized in consecutive locations). But a page should not be smaller than a disk access, as that would cause unwanted data to be read, and necessitate a memory copy as well.

Exercise 101: This depends on the instruction set, and specifically, on the addressing modes. For example, if an instruction can operate on two operands from memory and store the result in a register, then normally three pages are required: one containing the instruction itself, and two with the operands. There may be special cases if the instruction spans a page boundary.

Exercise 102: No. It depends on hardware support for address translation.

Exercise 103: Handling cache misses is relatively fast, and does not involve policy decisions: the cache is at the disposal of the currently running process. Handling page faults is slow, as it involves a disk access, and should be masked with a context switch. In addition, it involves a policy decision regarding the allocation of frames.

Exercise 104: There are other considerations too. One is that paging allows for more flexible allocation, and does not suffer from fragmentation as much as contiguous allocation. Therefore memory mapping using pages is beneficial even if we do not do paging to disk. Moreover, using a file to back a paging system (rather than a dedicated part of the disk) is very flexible, and allows space to be used for files or paging as needed. Using only primary memory such dynamic reallocation of resources is impossible.

Exercise 105: No. With large pages (4MB in this example) you loose flexibility and suffer from much more fragmentation.

Exercise 106: No: some may land in other segments, and wreck havoc.

Exercise 107: There will be no external fragmentation, but instead there will be internal fragmentation because allocations are made in multiples of the page size. However, this is typically much smaller.

Exercise 108: Trying to access an invalid segment is a bug in the program, and the operating system can't do anything about it. But an invalid page can be paged in by the operating system.

Exercise 109: Yes: the number of segments is determined by how many bits are used, and they all have the same size (as defined by the remaining bits). If some segment is smaller, the leftover space cannot be recovered.

Exercise 110: Yes (see [1]). The idea is to keep a shadow vector of used bits, that is maintained by the operating system. As the pages are scanned, they are marked in the page table as *not* present (even though actually they are mapped to frames), and the shadow bits are set to 0. When a page is accessed for the first time, the hardware will generate a page fault, because it thinks the page is not present. The operating system will then set the shadow bit to 1, and simply mark the page as present. Thus the overhead is just a trap to the operating system for each page accessed, once in each cycle of the clock algorithm.

Exercise 113:

`Owner:` apply special access restrictions.

`Permissions:` check restrictions before allowing access.

`Modification time:` check file dependencies (make), identify situations where multiple processes modify the file simultaneously.

`Size:` know how much is used in last block, optimize queries about size and seeking to the end.

`Data location:` implement access.

All except the last are also used when listing files.

Exercise 115: Remember that files are abstract. The location on the disk is an internal implementation detail that should be hidden.

Exercise 116: Append can be implemented by a seek and a write, provided atomicity is not a problem (no other processes using the same file pointer). Rewind is a seek to position 0.

Exercise 117: It's bad because then a program that traverses the file system (e.g. `ls -R`) will get into an infinite loop. It can be prevented by only providing a mechanism to create new subdirectories, not to insert existing ones. In Unix it is prevented by not allowing links to directories, only to files.

Exercise 118: Some convention is needed. For example, in Unix files and directories are represented internally by data structures called inodes. The root directory is always represented by entry number 2 in the table of inodes (inode 0 is not used, and inode 1 is used for a special file that holds all the bad blocks in the disk, to prevent them from being allocated to a real file). In FAT, the root directory resides at the beginning of the disk, right after the file allocation table itself, and has a fixed size. In other words, you can get directly to its contents, and do not have to go through any mapping.

Exercise 119: If the directory is big its contents may span more than one block.

Exercise 121: the advantage is short search time, especially for very large directories. A possible disadvantage is wasted space, because hash tables need to be relatively sparse in order to work efficiently.

Exercise 124: Renaming actually means changing the name as it appears in the directory — it doesn't have to touch the file itself.

Exercise 125: An advantage is flexibility: being able to link to any file or directory, even on another file system. The dangers are that the pointed-to file may be deleted, leaving a dangling link.

Exercise 126: Only if the whole block is requested: otherwise it will erroneously overwrite whatever happens to be in memory beyond the end of buf.

Exercise 128: We need it to close a security loophole. If we allow processes to access open files using the index into the open files table, a process may use a wrong or forged index as the `fd` in a `read` or `write` call and thus access some file opened by another process.

Exercise 129: Unix solves this by maintaining reference counts. Whenever a file is opened, the count of open file entries pointing to the inode (which is maintained as part of the inode) is incremented. Likewise, the counts of file descriptors pointing to open file entries are updated when processes fork. These counts are decremented when files are closed, and the entries are only freed when the count reaches 0.

Exercise 130: It is best to choose blocks that will not be used in the future. As we don't know the future, we can base the decisions on the past using LRU.

LRU depends on knowing the order of all accesses. For memory, accesses are done by the hardware with no operating system intervention, and it is too difficult to keep track of all of them. File access is done using operating system services, so the operating system can keep track.

Exercise 131: The problem is how to notify the process when the I/O is done. The common solution is that the conventional read and write functions return a handle, which the process can then use to query about the status of the operation. This query can either be non-blocking (poll), returning "false" if the operation is not finished yet, or blocking, returning only when it is finished.

Exercise 132: It just becomes a shared memory segment. If they write to this file, they should use some synchronization mechanism such as a semaphore.

Exercise 133: Sorry — I get confused by large numbers. But the more serious constraint on file sizes is due to the size of the variables used to store the file size. An unsigned 32-bit number can only express file sizes up to 4GB. As 64-bit architectures become commonplace this restriction will disappear.

Exercise 136: The superblock is so useful that it is simply kept in memory all the time. However, in case of a system crash, the copy of the superblock on the disk may not reflect some of the most recent changes. This can result in the loss of some files

or data.

Exercise 137: Because the file cold be copied and cracked off-line. This was typically done by trying to encrypt lots of dictionary words, and checking whether the results matched any of the encrypted passwords.

Exercise 138: Yes. A user can create some legitimate handles to see what random numbers are being used, and try to forge handles by continuing the pseudo-random sequence from there.

Exercise 139: No. Programs may have bugs that cause security leaks.

Exercise 140: No. A file descriptor is only valid within the context of the process that opened the file; it is not a global handle to the file.

Exercise 141: The only way to deny access to all the group except one member is to list all the members individually.

Exercise 142: A directory with execute permissions and no read permissions allows you to grant access to arbitrary users, by telling them the names of the files in some secret way (and using hard-to-guess names). Users that did not get this information will not be able to access the files, despite the fact that they actually have permission to do so, simply because they cannot name them.

Exercise 144: In Unix, devices are identified by special files in the `/dev` directory. To add a new device, you add a new special file to represent it. The special file's inode contains a *major number*, which identifies the device type. This is used as an index into a table of function pointers, where pointers to functions specific to this device type are stored. Adding a device driver therefore boils down to storing the appropriate function pointers in a new entry in the table.

Exercise 146: In general, no. They would be if jobs were run to completion one after the other. But jobs may overlap (one uses the disk whole the other runs on the CPU) leading to a more complex interaction. As a simple example, consider two scenarios: in one jobs run for one minute each, and arrive at 2 minute intervals; the throughput is then 30 jobs/hour, and the response time is 1 minute each. but if the 30 jobs arrive all together at the beginning of each hour, the throughput would still be 30 jobs/hour, but their average response time would be 15 minutes.

Exercise 148: The access pattern: which addresses are accessed one after the other.

Exercise 149: Yes, definitely! those 300 users will not be spread evenly through the day, but will come in big groups at the end of each study period.

Exercise 150: If it is a light-tailed distribution, kill a new job. If it is memoryless, choose one at random. If it is fat-tailed, select the oldest one.

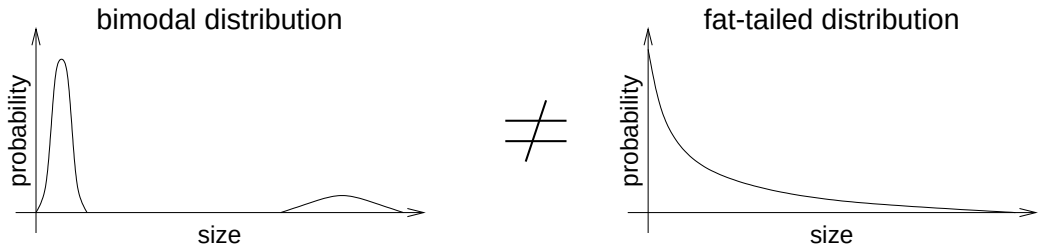
Exercise 151: The integral is

$$\int xf(x)dx = a\int x^{-a}dx$$

For $a \leq 1$ this does not converge. For $a > 1$, the result is

$$a \int_1^\infty x^{-a} dx = \frac{a}{a-1}$$

Exercise 152: Mice and elephants indicate a strict bimodality: it is a combination of two bell-like distributions, one for mice (centered at around 25 grams) and the other for elephants (at around 3 metric tons for Indian elephants or 6 metric tons for African elephants), with nothing in between. Real workloads more often have a steady decline in probability, not distinct modes.



Exercise 153: In the past, it was more common to observe an inverse correlation: applications tended to be either compute-bound (with little I/O), or I/O-bound (with little computation per data element). But modern multimedia applications (such as computer games) can be both compute and I/O intensive.

Exercise 154:

1. The peak load (typically during the day or evening) is much higher than the average (which also includes low-load periods).
2. No significance.
3. Computational tasks can be delayed and executed at night so as to allow for efficient interactive work during the day.

Exercise 155: Response time obviously depends on the CPU speed. But it is also affected by cache size, memory speed, and contention due to other jobs running at the same time. Network bandwidth is likewise limited by the raw capabilities of the networking hardware, but also depends on the CPU speed, memory, and communication protocol being used.

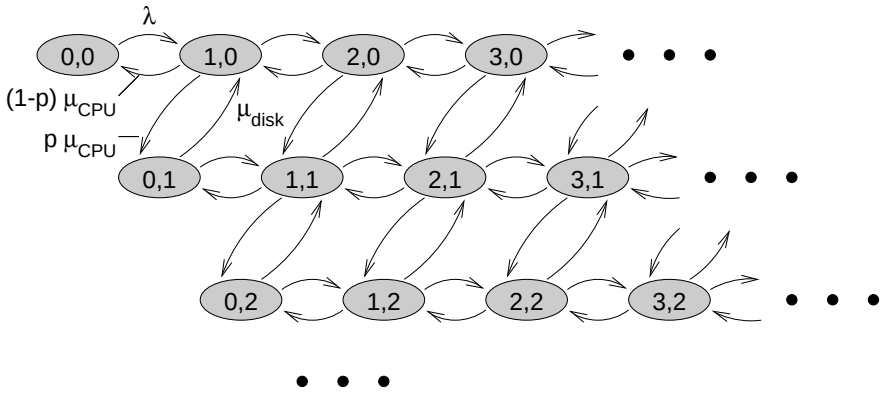
Exercise 156: Measure the time to call a very simple system call many times in a loop, and divide by the number of times. Improved accuracy is obtained by unrolling the loop, and subtracting the time for the loop overhead (measured by an empty loop). The number of iterations should be big enough to be measurable with reasonable accuracy, but not so big that context switching effects start having an impact.

Exercise 158: Yes! one simple case is when too much work arrives at once, and the queue overflows (i.e. there is insufficient space to store all the waiting jobs). A more complex scenario is when the system is composed of multiple devices, each of which is not highly utilized, but their use cannot be overlapped.

Exercise 159: The gist of the argument is as follows. Consider a long interval T. During this time, N jobs arrive and are processed. Focusing on job i for the moment, this job spends r_i time in the system. The cumulative time spent by all the jobs can be denoted by $X = \sum_i r_i$. Using these notations, we can approximate the arrival rate as $\lambda = N/T$, the average number of jobs in the system as $\bar{n} = X/T$, and the average response time as $\bar{r} = X/N$. Therefore

$$\bar{n} = \frac{X}{T} = \frac{N}{T} \cdot \frac{X}{N} = \lambda \cdot \bar{r}$$

Exercise 160: This is a 2-D state space, where state (x, y) means that there are x jobs waiting and being serviced by the CPU, and y jobs waiting and being serviced by the disk. For each y, the transition $(x, y) \rightarrow (x + 1, y)$ means a new job has arrived, and depends on λ. Denote by p the probability that a job needs service by the disk after being served by the CPU. The alternative is that it terminates, with probability $1 - p$. Transitions $(x, y) \rightarrow (x - 1, y)$ therefore occur at a rate proportional to $(1 - p)\mu_{CPU}$, and transitions $(x, y) \rightarrow (x - 1, y + 1)$ at a rate proportional to $p\mu_{CPU}$. Finally, transitions $(x, y) \rightarrow (x + 1, y - 1)$, which mean that a disk service has completed, occur at a rate proportional to μ_{disk}.



Exercise 161: If jobs arrive at a rate of λ, the average interarrival time is $1/\lambda$. Likewise, the service rate is μ, so the average service time is $1/\mu$. The utilization is the fraction of time that the system is busy, or in other words, the fraction of time from one arrival to the next that the system is busy serving the first arrival:

$$U = \frac{1/\mu}{1/\lambda} = \frac{\lambda}{\mu}$$

which is ρ.

Exercise 162: It is $1/\mu$, the expected service time, because under low load there is no waiting.

Exercise 164:

1. Students with a deadline: probably closed. Due to the looming deadline, the students will tend to stay in the edit-compile-execute cycle till they are done.
2. Professors doing administration: more like open. If the system doesn't respond, the professors will probably decide to do this some other time.
3. Tourist at information kiosk: open, due to the large population of random users. But if there is a long line, the tourists will probably wander off, so this is a system that may lose clients without giving them service.

Exercise 165: Typically in the range of 90–99% confidence that the true value is within 5–10% of the measured mean.

Exercise 167: Enlarging each line segment by a factor of 3 leads to 4 copies of the original, so the dimention is $\log_3 4 = 1.262$.

Exercise 168: It isn't. They use physical addresses directly.

Exercise 169: The program in the boot sector doesn't load the operating system directly. Instead, it checks all the disks (and disk partitions) to see which contain boot blocks for different operating systems, and creates a menu based on its findings. Choosing an entry causes the relevant boot block to be loaded, and everything continues normally from there.

Exercise 170: In old systems, where communication was directly through a terminal, it would get stuck. But nowadays it is common for each program to have a distinct window.

Exercise 171: It is the first 16 bits, which are the ASCII codes for '#' and '!'. This allows exec to identify the file as a script, and invoke the appropriate interpreter rather than trying to treat it as a binary file.

Exercise 172: Yes. for example, they may not block, because this will cause a random unrelated process to block. Moreover, they should not tamper with any of the "current process" data.

Exercise 173: Yes, but this may cause portability problems. For example, in some architectures the stack grows upwards, and in others downwards. The design described above requires only one function to know about such details. By copying the arguments to the u-area, they are made available to the rest of the kernel in a uniform and consistent manner.

Exercise 175: A bit-vector with a bit for each signal type exists in the process table

entry. This is used to note the types of signals that have been sent to the process since it last ran; it has to be in the process table, because the signals have to be registered when the process is not running. The handler functions are listed in an array in the u-area, because they need to run in the context of the process, and are therefore invoked only when the process is scheduled and the u-area is available.

Exercise 176: The problem is that you can only register one handler. The simplistic approach is to write a specific handler for each instance, and register the correct one at the beginning of the try code. However, this suffers from overhead to change the handlers all the time when the application is actually running normally. A better approach is to define a global variable that will contain an ID of the current try, and write just one handler that starts by performing a switch on this ID, and then only runs the desired code segment.

Exercise 177: The parent gets the identity of its child as the return value of the fork. The child process can obtain the identity of its parent from the operating system, usiong the `getppid` system call (this is not a typo: it stands for "**get p**arent **p**rocess **id**"). This is useful only for very limited communication by sending signals.

Exercise 179: The common approach is that it should be known in advance. One example is the original name service on Unix, that was identified by port 42. This was replaced by the domain name-server network, that is entered via an advertised list of root name servers (see 255).

As an alternative, the name service is sometimes embeded into the service that uses it, and is not handled by a separate process. An example is the naming of shared memory segments (Section 12.2.1).

Exercise 180: The area of shared memory will be a separate segment, with its own page table. This segment will appear in the segment tables of both processes. Thus they will both share the use of the page table, and through it, the use of the same pages of physical memory.

Exercise 181: You can save a shadow page with the original contents, and use it to produce a diff indicating exactly what was changed locally and needs to be updated on the master copy of the page.

Exercise 182: The first process creates a pipe and write one byte to it. It then forks the other processes. The P operation is implemented by reading a byte from the pipe; if it is empty it will block. The V operation is implemented by writing a byte to the pipe, thus releasing one blocked process.

Exercise 183: Yes, by mapping the same file into the address spaces of multiple processes.

Exercise 184: The function can not have side effects in the caller's context — it has to be a pure function. But it can actually affect the state at the callee, for example affecting the bank's database in the ATM scenario.

Exercise 185: Only `msg` is typically passed by reference, simply to avoid copying the data.

Exercise 186: There are two ways. One is to break up long messages into segments with a predefined length, and reassemble them upon receipt. The other is to use a protocol whereby messages are always sent in pairs: the first is a 4-byte message that contains the required buffer size, and the second is the real message. This allows the recipient to prepare a buffer of the required size before posting the receive.

Exercise 188: In the original Unix systems, the contents of a pipe were maintained using only the immediate pointers in the inode, in a circular fashion. If a process tried to write additional data when all the blocks were full, the process was blocked until some data was consumed. In modern Unix systems, pipes are implemented using socket pairs, which also have a bounded buffer. The other features are simple using counters of processes that have the pipe open.

Exercise 190: Either there will be no program listening to the port, and the original sender will be notified of failure, or there will be some unrelated program there, leading to a big mess.

Exercise 191: Yes, but not on the same port. This is why URLs sometimes include a port, as in `http://www.name.dom:8080`, which means send to port 8080 (rather than the default port 80) on the named host.

Exercise 192: It specifies the use of a text file, in which the first few lines constitute the header, and are of the format "⟨keyword⟩: ⟨value⟩"; the most important one is the one that starts with "To:", which specifies the intended recipient.

Exercise 193: No. IP on any router only has to deal with the networks to which that router is attached.

Exercise 194: Oh yes. For example it can be used for video transmission, where timeliness is the most important thing. In such an application UDP is preferable because it has less overhead, and waiting for retransmissions to get everything may degrade the viewing rather than improve it.

Exercise 195: It is messy to keep track of all the necessary acknowledgments.

Exercise 197: It still works — the corrupt parity bit identifies itself!

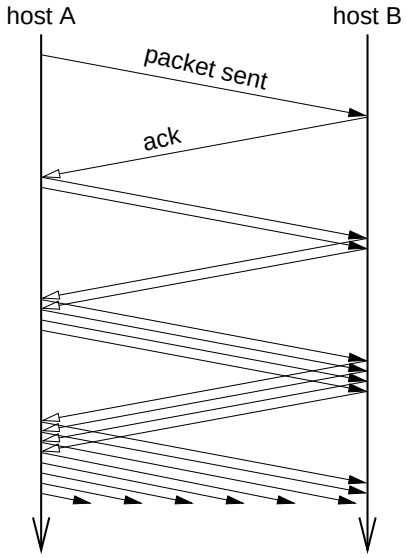
Exercise 198: It is also good for only one corrupt bit, but has higher overhead: for N data bits we need $2\sqrt{N}$ parity bits, rather than just $\lg N$ parity bits.

Exercise 199: Somewhat more than the round-trip time. We might even want to adjust it based on load conditions.

Exercise 201: The time for the notification to arrive is the round trip time, which is approximately $2t_\ell$. In order to keep the network fully utilized we need to transmit continuously during this time. The number of bytes sent will therefore be $2Bt_\ell$, or twice the bandwidth-delay product.

Exercise 202: no — while each communication might be limited, the link can be used to transfer many multiplexed communications.

Exercise 203: When each ack arrives, *two* packets are sent: one because the ack indicates that space became available at the receiver, and the other because the window was enlarged. Thus in each round-trip cycle the number of packets sent is doubled.



Exercise 204: Not necessarily. Some addresses may be cached at various places. Also, there are typically at least two servers for each domain in order to improve availability.

Exercise 205: `/tmp` is usually local, and stores temporary files used or created by users who logged on to each specific workstation.

Exercise 206: Its contents would become inaccessible, so this is not allowed.

Exercise 207: In principle this optimization is possible, but only among similar systems. NFS does it one component at a time so that only the client has to parse the local file name. For example, on a Windows system the local file name may be \a\b, which the server may not understand if it uses '/' as a delimiter rather than '\'.

Exercise 209: Sure. For example, both origin and destination machines should belong to the same architecture, and probably run the same operating system. And there may also be various administrative or security restrictions.

Exercise 210: Not for reasonable system designs. In modern switched networks, the capacity scales with the number of nodes.

Exercise 211: Maybe it can query the local host name, or check its process ID. But such queries can also be caught, and the original host name or ID provided instead of the current one.

Exercise 212: No — demand paging (from the source node) can be used to retrieve only those pages that are really needed. This has the additional benefit of reducing the latency of the migration.

Exercise 214: When the parent process terminates, the child process will not receive an EOF on the pipe as it should, because there is still a process that can write to the pipe: itself!

Index