Learner Guide

Faculty of Information Technology

Operating Systems 600

Year 3 *Semester 2*

RICHFIELD

richfield.ac.za

Registered with the Department of Higher Education as a Private Higher Education Institution under the Higher Education Act, 1997.
Registration Certificate No. 2000/HE07/008

FACULTY OF INFORMATION TECHNOLOGY

LEARNER GUIDE

MODULE: OPERATING SYSTEMS 600 (2ND SEMESTER)

PREPARED ON BEHALF OF
RICHFIELD GRADUATE INSTITUTE OF TECHNOLOGY (PTY) LTD

GRADUATE INDTITUTE OF TECHNOLOGY (PTY) LTD
Registration Number: 2000/000757/07

Chapter 1: Introducing Operating Systems

1.1 What Is an Operating System?
1.2 Operating System Software
1.3 Main Memory Management
1.4 User Interface
1.5 1.5 Cooperation Issues
1.6 An Evolution of Computing Hardware
1.7 Types of Operating Systems

Chapter 2: Memory Management Includes Virtual Memory

2.1 Best-Fit and First-Fit Allocation

2.2 Paged Memory Allocation

2.3 Page Replacement Policies and Concepts

2.4 The Importance of the Working Set

2.5 Segmented Memory Allocation

Chapter 3: Process Management Synchronization and Concurrent

3.1 Scheduling Sub-managers

3.2 Job and Process States

3.3 Scheduling Policies and Algorithms

3.4 Consequences of Poor Synchronization

3.5 Necessary Conditions for Deadlock

3.6 Strategies for Handling Deadlocks

3.7 Introduction to Multi-Core Processors

3.8 Concurrent Programming

Chapter 4: Device Management

4.1 Device management involves

4.2 Management of I/O Requests

4.3 Communication among Devices

Chapter 5: File Management

5.1 The File Manager

5.2 Physical File Organization

5.3 Contiguous Storage

5. 4 Access Methods

5.5 Capability Lists

Chapter 6: Network and Security

6.1 Network Topologies

6.2 Network Types

6.3 Circuit Switching

6.4 NOS Development

6.5 Memory Management

6.6 File Management

6.7 Role of the Operating System in Security

Chapter 7: System Management

7.1 Evaluating an Operating System

7.2 Role of Network Management

7.3 Software to Manage Deployment

Chapter 8: Operating Systems in Practice

8.1 The Evolution of UNIX

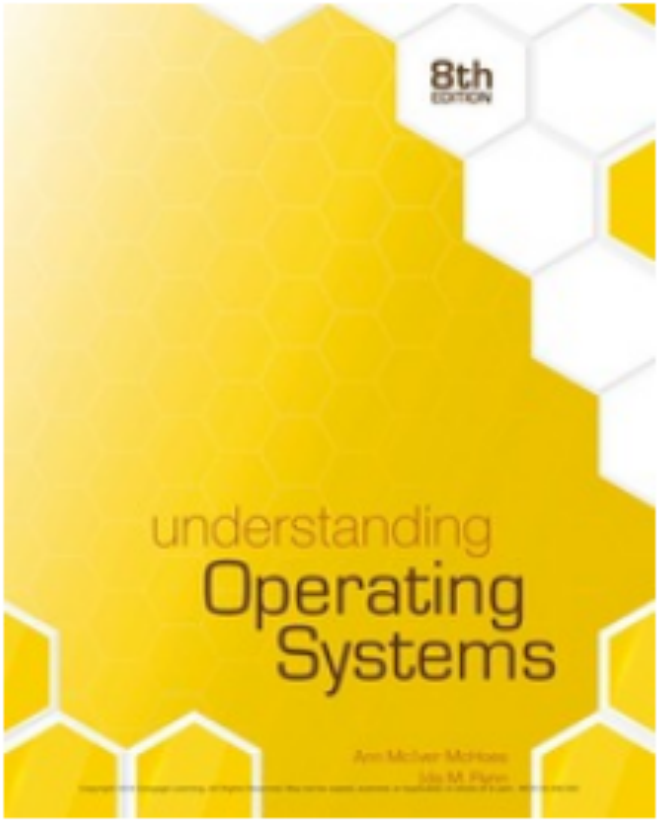
8.2 Process Synchronization

8.3 File Management

8.4 User Interfaces

8.5 Virtual Memory Implementation

8.6 The design goals for the Android™ operating system



Understanding Operating Systems

by Ann McHoes

- **Print**
  ISBN: 9781305674257, 1305674251
- **eText**
  ISBN: 9781337517539, 1337517534
- **Edition: 8th**
- **Copyright year: 2018**

LEARNING OUTCOMES

After reading this Section of the guide, the learner should be able to:

- **Able to demonstrate Component-based technology and Service-oriented technology**

Learning Objectives

- **How operating systems have evolved through the decades**
- **The basic role of an operating system**
- **How operating system software manages it subsystems**
- **The role of computer system hardware on the development of its operating system**
- **How operations systems are adapted to serve batch, interactive, real-time, hybrid, and embedded systems**
- **How operating systems designers envision their role and plan their work**

To understand an operating system is to begin to understand the workings of an entire computer system, because the operating system software manages each and every piece of hardware and software. In the pages that follow, we explore what operating systems are, how they work, what they do, and why.

1.1 What Is an Operating System?

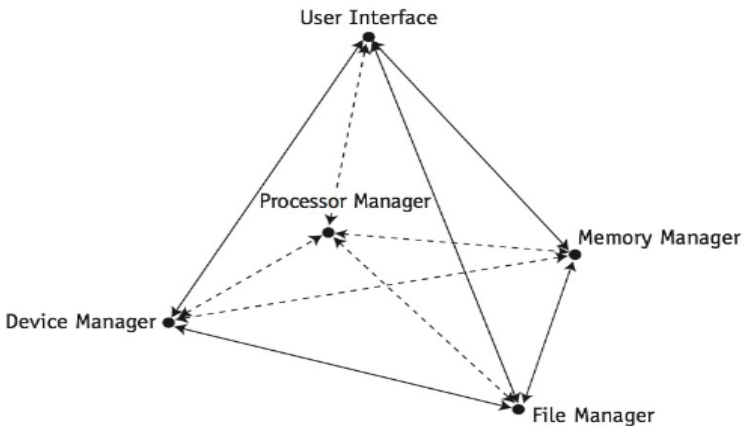
A computer system typically consists of software (programs) and hardware (the tangible machine and its electronic components). The operating system is the most important software— it's the portion of the computing system that manages all of the hardware and all of the other software. To be specific, the operating system software controls every file, every device, every section of main memory, and every moment of processing time. It controls who can use the system and how. In short, the operating system is the boss.

Therefore, each time the user sends a command, the operating system must make sure that the command is executed; or, if it's not executed, it must arrange for the user to get a message explaining the error. This doesn't necessarily mean that the operating system executes the command or sends the error message, but it does control the parts of the system that do.

1.2 Operating System Software

The pyramid shown in Figure 1.1 is an abstract representation of the operating system in its simplest form, and demonstrates how its major components typically work together.

At the base of the pyramid are the four essential managers of every major operating system: Memory Manager, Processor Manager, Device Manager, and File Manager. These managers, and their interactions, are discussed in detail in Chapters 1 through 8 of this book. Each manager works closely with the other managers as each one performs its unique role. At the top of the pyramid is the User Interface, which allows the user to issue commands to the operating system. Because this component has specific elements, in both form and function, it is often very different from one operating system to the next—sometimes even between different versions of the same operating system.



(figure 1.1)

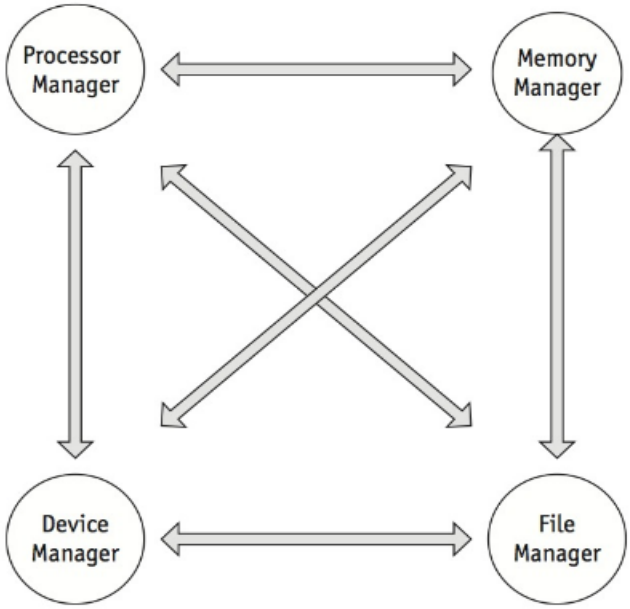
This pyramid represents an operating system on a stand-alone computer unconnected to a network. It shows the four subsystem managers and the User Interface.

Regardless of the size or configuration of the system, the four managers, illustrated in Figure 1.2, must, at a minimum, perform the following tasks while collectively keeping the system working smoothly:

- Monitor the system's resources
- Enforce the policies that determine what component gets what resources, when, and how much
- Allocate the resources when appropriate
- Deallocate the resources when appropriate

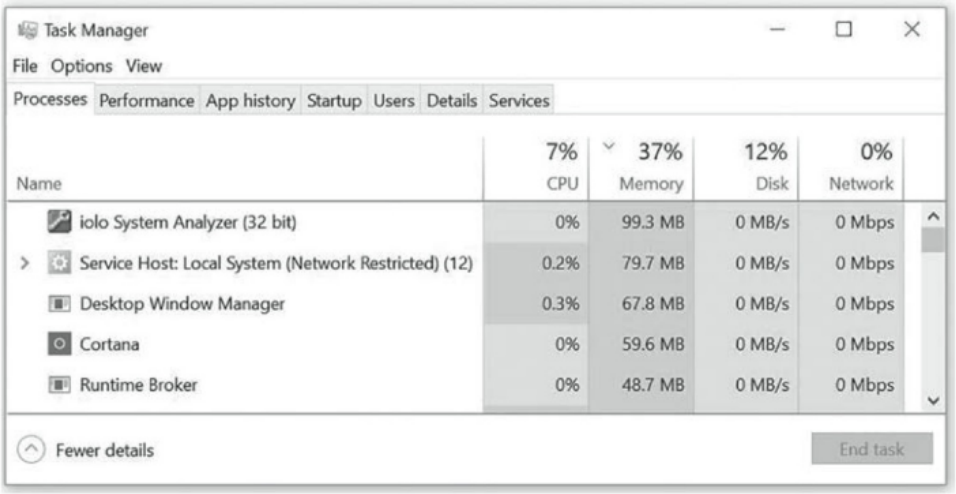
(figure 1.2)

Each manager at the base of the pyramid takes responsibility for its own tasks while working harmoniously with every other manager.



For example, the Memory Manager must keep track of the status of the computer system's main memory space, allocate the correct amount of it to incoming processes, and deallocate that space when appropriate—all while enforcing the policies that were established by the designers of the operating system.

An additional management task, networking, has not always been an integral part of operating systems. Today the vast majority of major operating systems incorporate a Network Manager, see Figure 1.3, to coordinate the services required for multiple sys- tems to work cohesively together. For example, the Network Manager must coordinate the workings of the networked resources, which might include shared access to memory space, processors, printers, databases, monitors, applications, and more. This can be a complex balancing act as the number of resources increases, as it often does.



(figure 1.3)

The Windows 10 Task Manager displays a snapshot of the system's CPU, main memory, disk, and network activity.

1.3 Main Memory Management

The Memory Manager is in charge of main memory, widely known as RAM (short for random access memory). The Memory Manager checks the validity of each request for memory space, and if it is a legal request, allocates a portion of memory that isn't already in use. If the memory space becomes fragmented, this manager might use policies established by the operating system's designers to reallocate memory to make more useable space available for other jobs that are waiting. Finally, when the job or process is finished, the Memory Manager deallocates its allotted memory space.

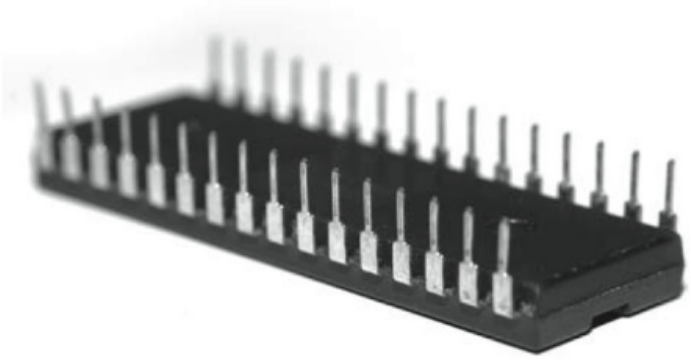
A key feature of RAM chips—the hardware that comprises computer memory—is that they depend on the constant flow of electricity to hold data. If the power fails or is turned off, the contents of RAM is wiped clean. This is one reason why computer system designers attempt to build elegant shutdown procedures, so that the contents of RAM can be stored on a non-volatile device, such as a hard drive, before the main memory chips lose power during computer shutdown.

A critical responsibility of the Memory Manager is to protect all of the space in main memory, particularly the space occupied by the operating system itself—it can't allow any part of the operating system to be accidentally or intentionally altered because that would lead to instability or a system crash.

Another kind of memory that's critical when the computer is powered on is read-only memory (often shortened to ROM), shown in Figure 1.4. This ROM chip holds soft- ware called firmware: the programming code that is used to start the computer and perform other necessary tasks. To put it in simplest form, it describes, in prescribed steps, when and how to load each piece of the operating system after the power is turned on, up to the point that the computer is ready for use. The contents of the ROM chip are non-volatile, meaning that they are not erased when the power is turned off, unlike the contents of RAM.

(figure 1.4)

A computer's relatively small ROM chip contains the firmware (unchanging software) that prescribes the system's initialization every time the system's power is turned on.



1.3.1 Processor Management

The Processor Manager decides how to allocate the central processing unit (CPU); an important function of the Processor Manager is to keep track of the status of each job, process, thread, and so on. We will discuss all of these in the chapters that follow, but for this overview, let's limit our discussion to a process and define it as a program's "instance of execution." A simple example could be a request to solve a mathematical equation: This would be a single job consisting of several processes, with each process performing a part of the overall equation.

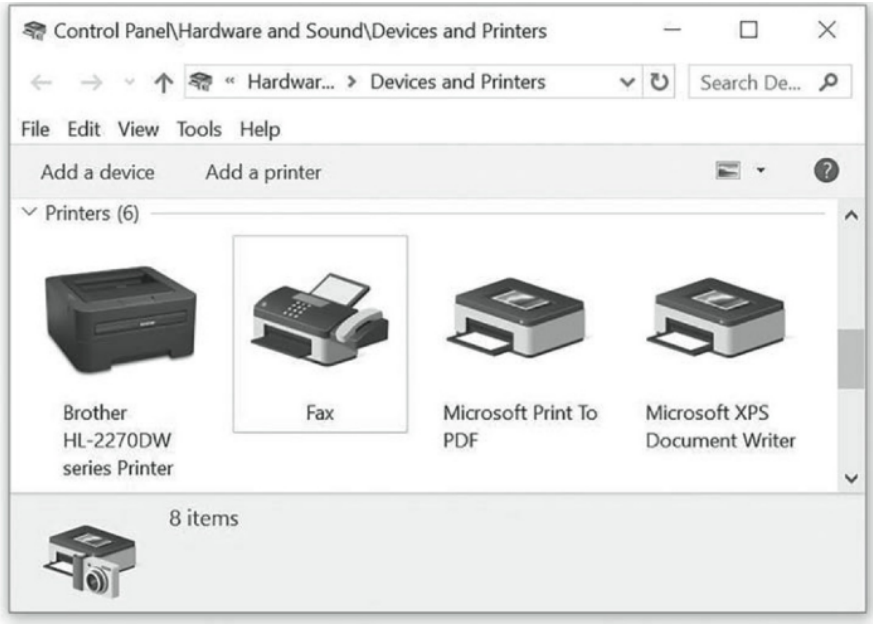
The Processor Manager is required to monitor the computer's CPU to see if it's busy executing a process or sitting idle as it waits for some other command to finish execution. Generally, systems are more efficient when their CPUs are kept busy. The Processor

Manager handles each process's transition, from one state of execution to another, as it moves from the starting queue, through the running state, and, finally, to the finish line (where it then tends to the next process). Therefore, this manager can be compared to a traffic controller. When the process is finished, or when the maximum amount of computation time has expired, the Processor Manager reclaims the CPU so it can allocate it to the next waiting process. If the computer has multiple CPUs, as with a multicore system, the Process Manager's responsibilities are greatly complicated.

1.3.2 Device Management

The Device Manager is responsible for connecting with every device that's available on the system, and for choosing the most efficient way to allocate each of these printers, ports, disk drives, and more, based on the device scheduling policies selected by the designers of the operating system.

Good device management requires that this part of the operating system uniquely iden- tify each device, start its operation when appropriate, monitor its progress, and, finally, deallocate the device to make the operating system available to the next waiting process. This isn't as easy as it sounds because of the exceptionally wide range of devices that can be attached to any system, such as the system shown in Figure 1.5.



(Figure 1.5) This computer, running the Windows 10 operating system, has device drivers loaded for all of the printing devices shown here.

For example, let's say you're adding a printer to your system. There are several kinds of printers commonly available (laser, inkjet, inkless thermal, etc.) and they're made by manufacturers that number in the hundreds or thousands. To complicate things, some devices can be shared, while some can be used by only one user or one job at a time. Designing an operating system to manage such a wide range of printers (as well as monitors, keyboards, pointing devices, disk drives, cameras, scanners, and so on) is a daunting task. To do so, each device has its own software, called a device

driver, which contains the detailed instructions required by the operating system to start that device, allocate it to a job, use the device correctly, and deallocate it when it's appropriate.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 1

1.4 User Interface

The user interface—the portion of the operating system that users interact with directly—is one of the most unique and most recognizable components of an operating system. Two primary types are the graphical user interface (GUI), shown in Figure 1.6, and the command line interface. The GUI relies on input from a pointing device, such as a mouse or the touch of your finger. Specific menu options, desktops, and formats often vary widely from one operating system to another, and, sometimes, from one version to another.

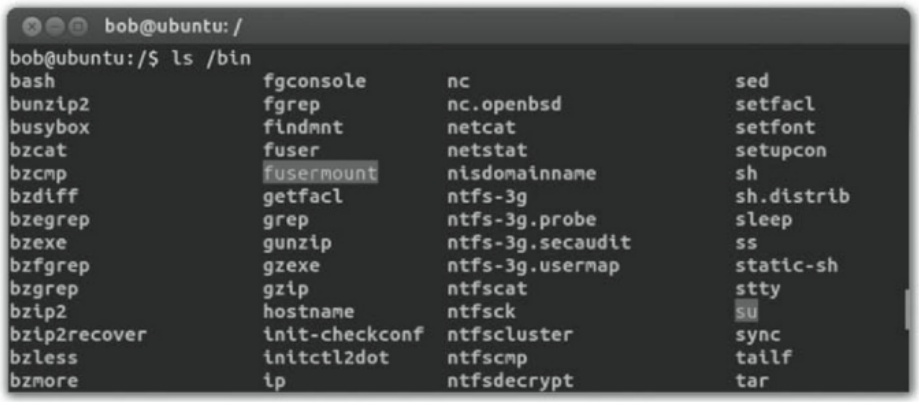
(figure 1.6)

An example of the graphical user interface (GUI) for the Ubuntu Linux operating system.

The alternative to a GUI is a command line interface, which responds to specific com- mands typed on a keyboard and displayed on the monitor, as shown in Figure 1.7. These interfaces accept typed commands, and offer skilled users powerful additional control because, typically, the commands can be linked together (concatenated) to perform complex tasks with a single multifunctional command that would require many mouse clicks to duplicate using a graphical interface.

(figure 1.7)

This is the Linux command line user interface showing a partial list of valid commands for this operating system. Many menu-driven operating systems also support a command-line interface similar to this one.



While a command structure offers powerful functionality, it has strict requirements for every command: Each must be typed accurately, each must be formed in the correct syntax, and combinations of commands must be assembled correctly. In addition, users need to know how to recover gracefully from any errors they encounter. These command line interfaces were once standard for operating systems and are still favored by power users, but have largely been supplemented with simple, forgiving, graphical user interfaces.

1.5 Cooperation Issues

None of the elements of an operating system can perform its individual tasks in isolation—each must also work harmoniously with every other manager. To illustrate this using a very simplified example, let's follow the steps as someone chooses a menu option to open a program. The following series of major steps are typical of the discrete actions that would occur in fractions of a second as a result of this choice:

1. The Device Manager receives the electrical impulse caused by a click of the mouse, decodes the command by calculating the location of the cursor, and sends that information through the User Interface, which identifies the requested command. Immediately, it sends the command to the Processor Manager.

2.The Processor Manager then sends an acknowledgment message (such as "waiting" or "loading") to be displayed on the monitor so that the user knows that the command has been sent successfully.

3. The Processor Manager determines whether the user request requires that a file (in this case a program file) be retrieved from storage, or whether it is already in memory.

4. If the program is in secondary storage (perhaps on a disk), the File Manager calculates its exact location on the disk and passes this information to the Device Manager, which retrieves the program and sends it to the Memory Manager.

5. If necessary, the Memory Manager Finds space for the program file in main memory and records its exact location. Once the program file is in memory, this manager keeps track of its location in memory.

6. When the CPU is ready to run it, the program begins execution via the Processor Manager. When the program has finished executing, the Processor Manager relays this information to the other managers.

7. The Processor Manager reassigns the CPU to the next program waiting in line. If the file was modified, the File Manager and Device Manager cooperate to store the results in secondary storage. If the file was not modified, there's no need to change the stored version of it.

8. The Memory Manager releases the program's space in main memory and gets ready to make it available to the next program that requires memory.

9. Finally, the User Interface displays the results and gets ready to take the next command.

1.6 An Evolution of Computing Hardware

To appreciate the role of the operating system (which is software), it may help to under- stand the computer system's hardware, which is the tangible, physical machine and its electronic components, including memory chips, the central processing unit (CPU), the input/output devices, and the storage devices.

• Main memory (RAM) is where the data and instructions must reside to be processed.

• The central processing unit (CPU) is the "brains" of the computer. It has the circuitry to control the interpretation and execution of instructions. All storage references, data manipulations, and input/output operations are initiated or performed by the CPU.

• Devices, sometimes called I/O devices for input/output devices, include every peripheral unit attached to the computer system, from printers and monitors to magnetic disks, optical disc drives, flash memory, keyboards, and so on.

A few of the operating systems that can be used on a variety of platforms are shown in Table 1.1.

(table 1.1)

A brief list of platforms, and a few of the operating systems designed to run on them, listed here in alphabetical order.

Platform	Operating System
Laptops, desktops	Linux, Mac OS X, UNIX, Windows
Mainframe computers	Linux, UNIX, Windows, IBM z/OS
Supercomputers	Linux, UNIX
Telephones, tablets	Android, iOS, Windows
Workstations, servers	Linux, Mac OS X Server, UNIX, Windows

1.7 Types of Operating Systems

Operating systems fall into several general categories distinguished by the speed of their response, and the method used to enter data into the system. The five categories are batch, interactive, real-time, hybrid, and embedded systems.

Batch systems feature jobs that are entered as a whole, and in sequence. That is, only one job can be entered at a time, and once a job begins processing, then no other job can start processing until the resident job is finished. These systems date from early computers, when each job consisted of a stack of cards—or reels of magnetic tape—for input, and were entered into the system as a unit, called a batch. The efficiency of a batch system is measured in throughput which is the number of jobs completed in a given amount of time (usually measured in minutes, hours, or days.)

Interactive systems allow multiple jobs to begin processing, and return results to users with better response times than batch systems; but interactive systems are slower than the real-time systems that we will talk about next. Early versions of these operating systems allowed each user to interact directly with the computer system via commands entered from a typewriter-like terminal. The operating system used complex algorithms to share processing power (often with a single processor) among the jobs awaiting processing. Interactive systems offered huge improvements in responsiveness with turn- around times in seconds or minutes, instead of the hours or days of batch-only systems.

Hybrid systems, widely used today, are a combination of batch and interactive. They appear to be interactive because individual users can enter multiple jobs or processes into the system and get fast responses, but these systems also accept and run batch programs in the background when the interactive load is light. A hybrid system takes advantage of the free time between high-demand usage of the system and low-demand times.

Real-time systems are used in time-critical environments where reliability and deadlines are critical. This time limit need not be ultra-fast, though it often is; however, system response time must meet the deadline because there are significant consequences for not doing so. They also need to provide contingencies to fail gracefully—that is, preserving as much of the system's capabilities and data as possible, throughout system failure, to facilitate recovery. Examples of real-time systems are those used for spacecraft, airport traffic control, fly-by-wire aircraft, critical industrial processes, and medical systems, to name a few. There are two types of real-time systems, depending on the consequences of missing the deadline: hard and soft systems.

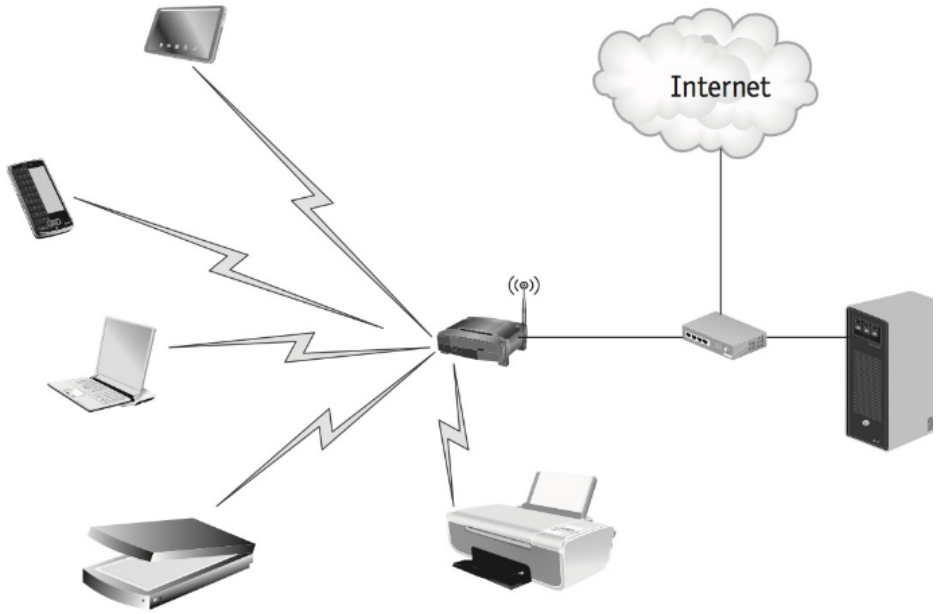
• Hard real-time systems risk total system failure if the predicted deadline is missed.

• Soft real-time systems suffer performance degradation, but not total system failure, as a consequence of a missed deadline.

Although it's theoretically possible to convert a general-purpose operating system into a real-time system by merely establishing a deadline, the need to be extremely predictable is not part of the design criteria for most operating systems; so they can't provide the guaranteed response times and graceful failure that real-time performance requires. Therefore, most embedded systems (described in the following paragraphs) and real-time environments require operating systems that are specially designed to meet real-time needs.

Networks allow users to manipulate resources that may be located over a wide geo- graphical area. Network operating systems were originally similar to single-processor operating systems in that each machine ran its own local operating system and served its own local user group. Now, network operating systems make up a special class of software that allows users to perform their tasks using few, if any, local resources. One example of this phenomenon is cloud computing.

As shown in below picture, wireless networking capability is a standard feature in many computing devices: cell phones, tablets, and other handheld Web browsers.



Conclusion

In this chapter, we looked at the definition of an operating system as well as its overall function. What began as hardware-dependent operations software has evolved to run increasingly complex computers and, like any complicated subject, there's much more detail to explore, such as the role of the main memory resource, the CPU (processor), the system's input and storage devices, and its numerous files. Each of these functions needs to be managed seamlessly, as does the cooperation among them and other system essentials, such as the network it's connected to. As we'll see in the remainder of this text, there are many ways to perform every task, and it's up to the designers of the operating system to choose the policies that best match the environment and its users.

Key Terms

Batch system: a type of computing system that executes programs, each of which is submitted in its entirety, can be grouped into batches, and is executed without external intervention.

Central processing unit (CPU): a component with circuitry that controls the interpreta-tion and execution of instructions. See also processor.

Process: an instance of execution of a program that is identifiable and controllable by the operating system.

Processor: (1) another term for the CPU (central processing unit); (2) any component in a computing system capable of performing a sequence of activities. It controls the interpretation and execution of instructions.

Processor Manager: a composite of two sub managers, the Job Scheduler and the Process Scheduler, that decides how to allocate the CPU.

RAM: short for random access memory. See main memory.

Real-time system: a computing system used in time-critical environments that require guaranteed response times. Examples include navigation systems, rapid transit systems, and industrial control systems.

Server: a node that provides clients with various network services, such as file retrieval, printing, or database access services.

Storage: the place where data is stored in the computer system. Primary storage is main memory. Secondary storage is non-volatile media, such as disks and flash memory.

User interface: the portion of the operating system that users interact with directly—is one of the most unique and most recognizable components of an operating system.

Chapter Review

To Explore More

For additional background on a few of the topics discussed in this chapter, begin a search with these terms.

- **Embedded computers aboard the International Space Station**
- **Operating systems for mainframes**
- **UNIX operating system in Apple devices**
- **Windows systems for sports stadiums**
- **Android operating system for phones**

Research Topics

A. **Write a one-page review of an article about the subject of operating systems that appeared in a recent computing magazine or academic journal. Give a summary of the article, including the primary topic, your own summary of the information presented, and the author's conclusion. Give your personal evaluation of the article, including topics that made the article interesting to you (or not) and its relevance to your own experiences. Be sure to cite your sources.**

Review Questions

1. Give a real-world example of a task that you perform everyday via cloud computing. Explain how you would cope if the network connection suddenly became unavailable.
2. In your opinion, what would be the consequences if the Memory Manager and the File Manager stopped communicating with each other?

3. Name five current operating systems (other than those mentioned in Table 1.1) and identify the computers, platforms, or configurations where each is used.
4. Many people confuse main memory and secondary storage. Explain why this might happen, and describe how you would explain the differences to class- mates so they would no longer confuse the two.
5. Name the five key concepts about an operating system that you think a typical user needs to know and understand.
6. Explain the impact of the continuing evolution of computer hardware and the accompanying evolution of operating systems software.
7. List three tangible, physical, resources that can be found on a typical computer system.

Advanced Exercises

8. Draw a system flowchart illustrating the steps performed by an operating sys- tem as it executes the instruction to back up a disk on a single-user computer system. Begin with the user typing the command on the keyboard or choosing an option from a menu, and conclude with the result being displayed on the monitor.
9. In a multiprogramming and time-sharing environment, several users share a single system at the same time. This situation can result in various security problems. Name two such problems. Can we ensure the same degree of security in a time-share machine as we have in a dedicated machine? Explain your answers.

Chapter 2: Memory Management Includes Virtual Memory

LEARNING OUTCOMES

After reading this Section of the guide, the learner should be able to:

• Able to demonstrate Component-based technology and Service-oriented technology

- **How two memory allocation systems work: best-fit and first-fit**
- **How page allocation methods spawned virtual memory**
- **How several page replacement policies compare in function and best uses**
- **How the concept of the working set is used in memory allocation schemes**
- **• How cache memory issued by the operating system to improve response time**

Introduction

One of the most critical segments of the operating system is the part that manages the main memory. In fact, for early computers, the performance of the entire system was judged by the quantity of the available main memory resources, and how that memory was optimized while jobs were being processed. Back in those days, jobs were submitted serially, one at a time; and managing memory centred on assigning that memory to each incoming job, and then reassigning it when the job was finished.

Main memory has gone by several synonyms, including random access memory or RAM, core memory, and primary storage (in contrast with secondary storage, where data is stored in a more permanent way). This chapter discusses four types of memory allocation schemes: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Let's begin with the simplest memory management scheme: the one used with early computer systems.

2.1 Best-Fit and First-Fit Allocation

For both fixed and dynamic memory allocation schemes, the operating system must keep lists of each memory location, noting which are free and which are busy. Then, as new jobs come into the system, the free partitions must be allocated fairly, according to the policy adopted by the programmers who designed and wrote the operating system.

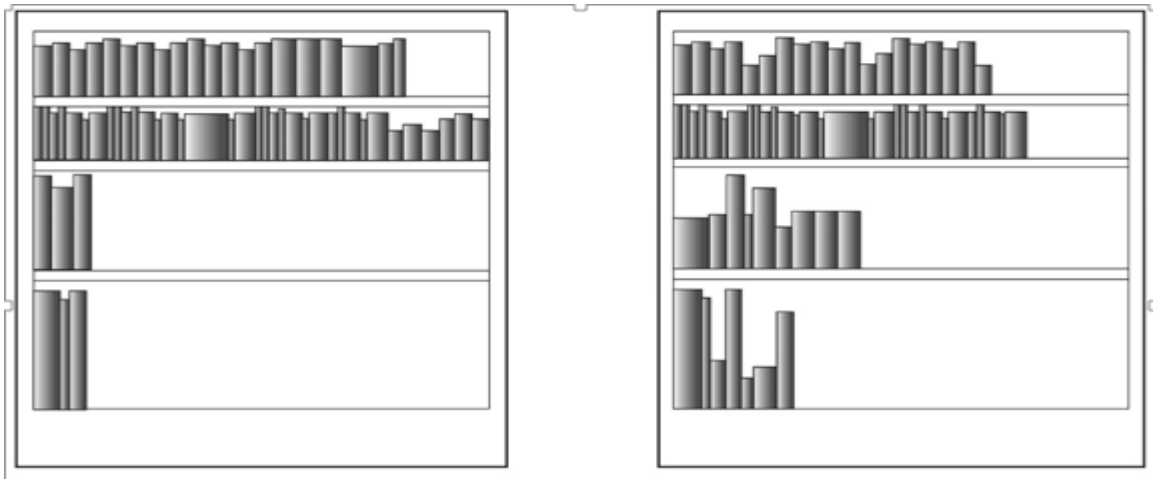
For both fixed and dynamic memory allocation schemes, the operating system must keep lists of each memory location, noting which are free and which are busy. Then, as new jobs come into the system, the free partitions must be allocated fairly, according to the policy adopted by the programmers who designed and wrote the operating system.

To understand the trade-offs, imagine that you are in charge of a small library. You have books of all shapes and sizes, and there's a continuous stream of people taking books out and bringing them back, so that someone is always waiting. It's clear that your library is a popular place and you'll always be busy, without any time to rearrange the books on the shelves.

You need a system. Your shelves have fixed partitions with a few tall spaces for over- sized books, several shelves for paperbacks and DVDs, and lots of room for textbooks. You'll need to keep track of which spaces on the shelves are full and where you have spaces for more. For the purposes of our example, we'll keep two lists: a free list show- ing all the available spaces, and a busy list showing all the occupied spaces. Each list will include the size and location of each space.

As each book is removed from its place on the shelf, you'll update both lists by removing the space from the busy list and adding it to the free list. Then, as your books are returned and placed back on a shelf, the two lists will be updated again.

There are two ways to organize your lists: by size or by location. If they're organized by size, the spaces for the smallest books are at the top of the list, and those for the largest are at the bottom, as shown in Figure 2 When they're organized by location,



(figure 2)

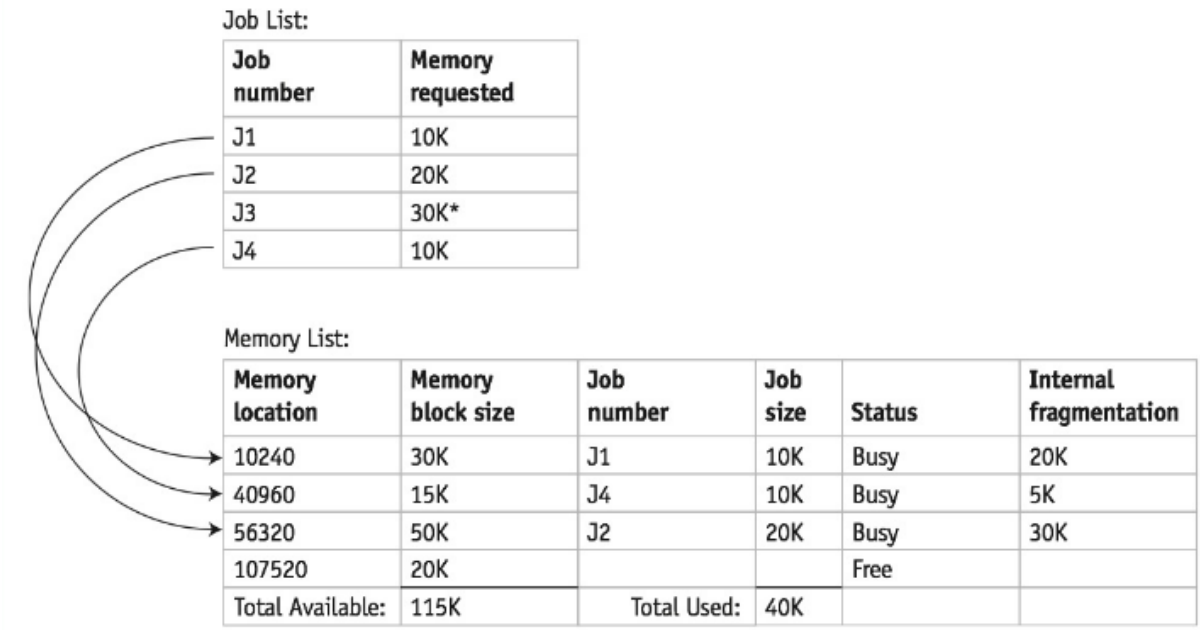
The bookshelf on the left uses the best-fit allocation with books stacked on shelves according to their height, that is, all the tall books, and no short books, are grouped together on shelves that can accommodate them. The bookshelf on the right is first-fit allocation, where some tall books share space with small books.

If the lists are organized by size, you're optimizing your shelf space: as books arrive, you'll be able to put them in the spaces that fit them best. This is a best-fit scheme. If a paperback is returned, you'll place it on a shelf with the other paperbacks or at least with other small books. Similarly, oversized books will be shelved with other large books. Your lists make it easy to find the smallest available empty space where the book can fit. The disadvantage of this system is that you're wasting time looking for the best space. Your other customers have to wait for you to put each book away, so you won't be able to process as many customers as you could with the other kind of list.

In the second case, a list organized by shelf location, you're optimizing the time it takes you to put books back onto the shelves by using a first-fit scheme. This system ignores the size of the book that you're trying to put away. If the same paperback book arrives, you can quickly find it an empty space. In fact, any nearby empty space will suffice if it's large enough, and if it's close to your desk. In this case, you are optimizing the time it takes you to shelve the books.

This is the fast method of shelving books, and if speed is important, it's the better of the two alternatives. However, it isn't a good choice if your shelf space is limited, or if many large books are returned, because large books must wait for the large spaces. If all of your large spaces are filled with small books, the customers returning large books must wait until a suitable space becomes available. Eventually you'll need time to rearrange the books and compact your collection.

Figure 2.1. shows how a large job can have problems with a first-fit memory alloca- tion list. Jobs 1, 2, and 4 are able to enter the system and begin execution; Job 3 has to wait even though there would be more than enough room to accommodate it if all of the fragments of memory were added together. First-fit offers fast allocation, but it isn't always efficient. On the other hand, the same job list using a best-fit scheme would use memory more efficiently, as shown in Figure 2.2. In this particular case, a best-fit scheme would yield better memory utilization.



Job List:

Job number	Memory requested
J1	10K
J2	20K
J3	30K*
J4	10K

Memory List:

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10240	30K	J1	10K	Busy	20K
40960	15K	J4	10K	Busy	5K
56320	50K	J2	20K	Busy	30K
107520	20K			Free	
Total Available:	115K	Total Used:	40K		

(figure 2.1)

Using a first-fit scheme, Job 1 claims the first available space. Job 2 then claims the first partition large enough to accommodate it, but, by doing so, it takes the last block large enough to accommodate Job 3. Therefore, Job 3 (indicated by the asterisk) must wait until a large block becomes available, even though there's 75Kof unused memory space (internal fragmentation). Notice that the memory list is ordered according to memory location.

Memory use has been increased, but the memory allocation process takes more time. What's more, while internal fragmentation has been diminished, it hasn't been completely eliminated.

The first-fit algorithm (included in Appendix A) assumes that the Memory Manager Keeps two lists, one for free memory blocks and one for busy memory blocks. The operation consists of a simple loop that compares the size of each job to the size of each memory block until a block is found that's large enough to fit the job. Then the job is stored into that block of memory, and the manager fetches the next job from the entry queue. If the entire list is searched in vain, then the job is placed into a waiting queue. The Memory Manager then fetches the next job and repeats the process.

Job List:

Job number	Memory requested
J1	10K
J2	20K
J3	30K
J4	10K

Memory List:

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
40960	15K	J1	10K	Busy	5K
107520	20K	J2	20K	Busy	None
10240	30K	J3	30K	Busy	None
56320	50K	J4	10K	Busy	40K
Total Available:	115K	Total Used:	70K		

(Figure 2.2)

Best-fit free scheme. Job 1 is allocated to the closest- fitting free partition, as are Job2andJob3.Job4 is allocated to the only available partition although it isn't the best-fitting one. In this scheme, all four jobs are served without waiting. Notice that the memory list is ordered according to memory size. This scheme uses memory more efficiently, but it's slower to implement.

In Table 1.1 a request for a block of 200 spaces has just been given to the Memory Manager. The spaces may be words, bytes, or any other unit the system handles. Using the first-fit scheme and starting from the top of the list, the manager locates the first block of memory large enough to accommodate the job, which is at loca-tion 6785. The job is then loaded, starting at location 6785 and occupying the next 200 spaces. The next step is to adjust the free list to indicate that the block of free memory now starts at location 6985 (not 6785 as before), and that it contains only 400 spaces (not 600 as before).

Status Before Request		Status After Request	
Beginning Address	**Free Memory Block Size**	**Beginning Address**	**Free Memory Block Size**
4075	105	4075	105
5225	5	5225	5
6785	600	*6985	400
7560	20	7560	20
7600	205	7600	205
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

(Table 1.1 these two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 6785, using the first-fit algorithm. (Note: All values are in decimal notation unless otherwise indicated.)

The best-fit algorithm is slightly more complex because the goal is to find the smallest memory block into which the job will fit. One of the problems with this is that the entire table must be searched before an allocation can be made, because the memory blocks are physically stored in sequence according to their location in the memory, and not by memory block sizes (as shown in Figure 1.3). The system could execute an algorithm to continuously rearrange the list in ascending order by memory block size, but that would add more overhead, and might not be an efficient use of processing time in the long run.

The best-fit algorithm (included in Appendix A) is illustrated showing only the list of free memory blocks. Table 2.2 shows the free list before and after the best-fit block has been allocated to the same request presented in Table 1.1.

Status Before Request		Status After Request	
Beginning Address	Free Memory Block Size	Beginning Address	Free Memory Block Size
4075	105	4075	105
5225	5	5225	5
6785	600	6785	600
7560	20	7560	20
7600	205	*7800	5
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

(Table 2.2)These two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 7600 (shaded), using the best-fit algorithm.

In Table 2.2, a request for a block of 200 spaces has just been given to the Memory Manager. Using the best-fit algorithm and starting from the top of the list, the manager searches the entire list and locates a block of memory starting at location 7600, which is the block that's the smallest one that's also large enough to accommodate the job. The choice of this block minimizes the wasted space. Only 5 spaces are wasted, which is less than in the four alternative blocks. The job is then stored, starting at location 7600 and occupying the next 200 spaces. Now the free list must be adjusted to show that the block of free memory starts at location 7800 (not 7600 as before) and that it contains only 5 spaces (not 205 as before). But it doesn't eliminate wasted space. Note that while the best-fit resulted in a better fit, it also resulted, as it does in the general case, in a smaller free space (5 spaces), which is known as a sliver.

Which is best: first-fit or best-fit? For many years there was no way to answer such a general question because performance depended on the job mix. In recent years, access times have become so fast that the scheme that saves the more valuable resource, memory space, may be the best. Research continues to focus on finding the optimum allocation scheme.

2.2 Paged Memory Allocation

Paged memory allocation is based on the concept of dividing jobs into units of equal size. Each unit is called a page. Some operating systems choose a page size that is the exact same size as a section of main memory, which is called a page frame. Likewise, the sections of a magnetic disk are called sectors or blocks. The paged memory alloca- tion scheme works quite efficiently when the pages, sectors, and page frames are all the same size. The exact size—the number of bytes that can be stored in each of them—is usually determined by the disk's sector size. Therefore, one sector will hold one page of job instructions (or data), and will fit into a single page frame of memory; but because this is the smallest addressable chunk of disk storage, it isn't subdivided, even for very tiny jobs.

Before executing a program, a basic Memory Manager prepares it by

1. determining the number of pages in the program,
2. locating enough empty page frames in main memory,
3. Loading all of the program's pages into those frames.

When the program is initially prepared for loading, its pages are in logical sequence—the first pages contain the first instructions of the program and the last page has the last instructions. We refer to the program's instructions as "bytes" or "words."

The loading process is different from the schemes because the pages do not have to be contiguous—loaded in adjacent memory blocks. In fact, each page can be stored in any available page frame anywhere in main memory.

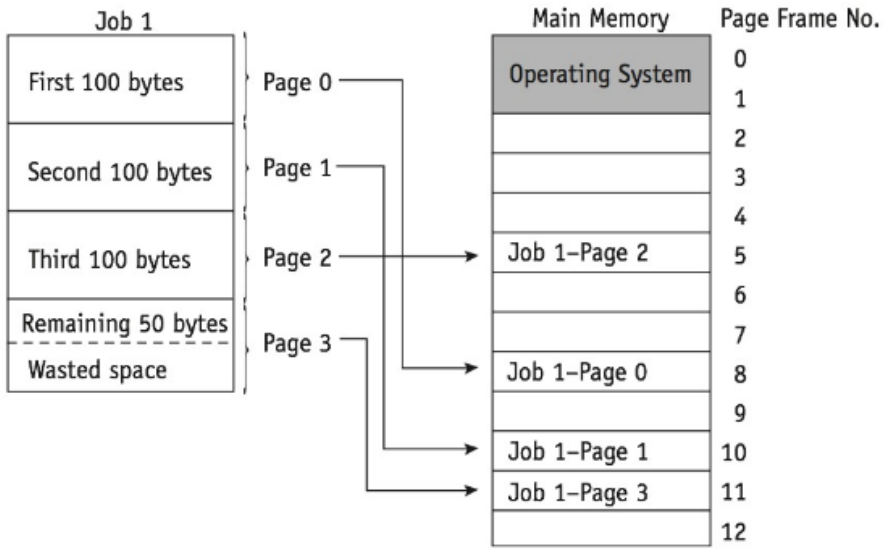
The primary advantage of storing programs in non-contiguous page frames is that main memory is used more efficiently, because an empty page frame can be used by any page of any job. In addition, the compaction scheme used for relocatable partitions is eliminated because there is no external fragmentation between page frames.-

However, with every new solution comes a new problem. Because a job's pages can be located anywhere in main memory, the Memory Manager now needs a mechanism to keep track of them—and this means enlarging the size, complexity, and overhead of the operating system software.

The simplified example in Figure 3.1 shows how the Memory Manager keeps track of a program that is four pages long. To simplify the arithmetic, we've arbitrarily set the page size at 100 bytes. Job 1 is 350 bytes long and is being readied for execution. The Page Map Table for the first job is shown later in Table 3.2.

Notice in Figure 3.1 that the last page frame (Page Frame 11) is not fully utilized because Page 3 is less than 100 bytes—the last page uses only 50 of the 100 bytes available. In fact, very few jobs perfectly fill all of the pages, so internal fragmentation is still a problem, but only in the job's last page frame.

Figure 3.1 has seven free page frames, therefore, the operating system can accommodate an incoming job of up to 700 bytes because it can be stored in the seven empty page frames. But a job that is larger than 700 bytes cannot be accommodated until Job 1 ends its execution and releases the four page frames it occupies. Any job larger than 1100 bytes will never fit into the memory of this tiny system. Therefore, although paged memory allocation offers the advantage of non-contiguous storage, it still requires that the entire job be stored in memory during its execution.



(figure 3.1) Job 1 In this example, each page frame can hold 100 bytes. This job, at 350 bytes long, is divided among four page frames. The resulting internal fragmen- tation (wasted space) is Main Memory associated with Page 3,Page 3 loaded into Page Frame 11.

Figure 3.1 uses arrows and lines to show how a job's pages fit into page frames in mem- ory, but the Memory Manager uses tables to keep track of them. There are essentially three tables that perform this function: Job Table, Page Map Table, and Memory Map Table. Although different operating systems may have different names for them, the tables provide the same service regardless of the names they are given. All three tables reside in the part of main memory that is reserved for the operating system.

As shown in Table 3.1, the Job Table (JT) contains two values for each active job: the size of the job (shown on the left), and the memory location where its Page Map Table is stored (on the right). For example, the first job has a size of 400 and is at location 3096 within memory. The Job Table is a dynamic list that grows as jobs are loaded into the system, and shrinks, as shown in (b) in Table 3.1, as they are completed later.

Job Table

Job Size	PMT Location
400	3096
200	3100
500	3150

(a)

Job Table

Job Size	PMT Location
400	3096
500	3150

(b)

Job Table

Job Size	PMT Location
400	3096
700	3100
500	3150

(c)

(Table 2.1) Three snapshots of the Job Table. Initially the Job Table has one entry for each job (a). When the second job ends (b), its entry in the table is released. Finally, it is replaced by the entry for the next job (c).

Each active job has its own Page Map Table (PMT), which contains the vital informa- tion for each page—the page number and its corresponding memory address to the page frame. Actually, the PMT includes only one entry per page. The page numbers are sequential (Page 0, Page 1, Page 2, and so on), so it isn't necessary to list each page num- ber in the PMT. That is, the first entry in the PMT always lists the page frame memory address for Page 0, the second entry is the address for Page 1, and so on.

A simple Memory Map Table (MMT) has one entry for each page frame, and shows its location and its free or busy status.

At compilation time, every job is divided into pages. Using job 1 from Figure 2.1, we can see how this works:

• Page 0 contains the first hundred bytes, 0-99

• Page 1 contains the second hundred bytes, 100-199

• Page 2 contains the third hundred bytes, 200-299

• Page 3 contains the last 50 bytes, 300-349

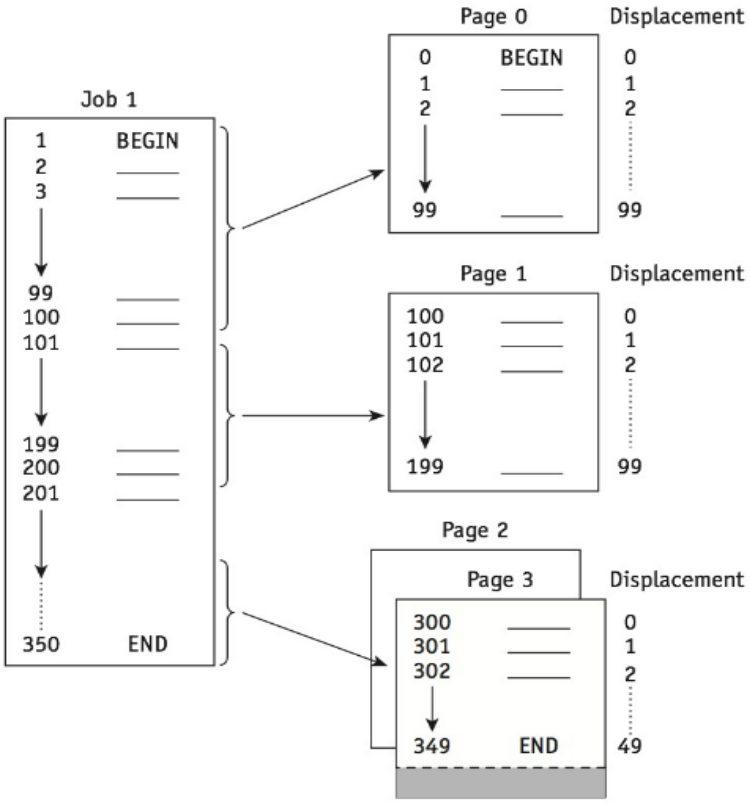
As you can see, the program has 350 bytes; but when they are stored, the system numbers them starting from 0 through 349. Therefore, the system refers to them as Byte 0 through Byte 349.

Page Displacement

How far away is a certain byte from the beginning of its page frame? This value is called the displacement (also called the offset), and it is a relative factor that's used to locate a certain byte within its page frame.

Here's how it works: In the simplified example shown in Figure 3.2, Bytes 0, 100, 200, and 300 are the first bytes for pages 0, 1, 2, and 3, respectively, so that each has a displacement of zero. To state it another way, the distance from the beginning of the page frame to Byte 200 is zero bytes. Likewise, if the Memory Manager needs to access Byte 314, it can first go to the page frame containing Page 3 and then go to Byte 14 (the fifteenth) in that page frame.

Because the first byte of each page has a displacement of zero, and the last byte has a displacement of 99, the operating system can access the correct bytes by using its relative position from the beginning of its page frame.



(Figure 2.3) This job is 350 bytes long and is divided into four pages of 100 bytes each that are loaded into four page frames in memory. Notice the internal frag- mentation at the end of Page 3.

In this example, it is easy for us to see intuitively that all bytes numbered less than 100 will be on Page 0; all bytes numbered greater than, or equal to, 100, but less than 200, will be on Page 1; and so on. This is the advantage of choosing a convenient fixed page size, such as 100 bytes. The operating system uses an algorithm for calculating the page and displacement using a simple arithmetic calculation:

BYTE_NUMBER_TO_BE_LOCATED / PAGE_SIZE = PAGE_NUMBER

However, this calculation gives only the page number, not the actual location on that page where our desired data resides.

The previous example used a very convenient page size, but now let's do the calculation with a more likely page size, where each page is 512 bytes each. The data to be retrieved is located at byte 1076. To find the displacement, one could use traditional long division or a modulo operation. This is a mathematical operation that displays the remainder that results when the dividend is divided by the divisor.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 3 page 64

Pages Versus Page Frames

The displacement calculation gives the location of an instruction with respect to the job's pages. However, these page numbers are ordered logically, not physically. In actuality, each page is stored in a page frame that can be located anywhere in available main memory. Therefore, the paged memory allocation algorithm needs to be expanded to find the exact location of the byte in main memory. To do so, we need to correlate each of the job's pages with its page frame number using the job's Page Map Table.

For example, if we look at the PMT for Job 1 from Figure 2.1, we see that it looks like the data in Table 2.2.

Page Number	Page Frame Number
0	8
1	10
2	5
3	11

(Table 3.2) Page Map Table for Job 1 in Figure 3.1.

Let's say we are looking for an instruction with a displacement of 76 on Page 2, assum- ing a page size and page frame size of 100. To find its exact location in memory, the operating system (or the hardware) conceptually has to perform the following four steps.

STEP 1 Do the arithmetic computation just described to determine the page number and displacement of the requested byte: 276/100 5 2 with a remainder of 76.

• The page number equals the integer resulting from the division of the job space address by the page size: 2

• Displacement equals the remainder from the page number division: 76.

STEP 2 Refer to this job's PMT (shown in Table 3.2) and find out which page frame contains Page 2.

PAGE 2 is located in PAGE FRAME 5.

STEP 3 Get the address of the beginning of the page frame by multiplying the page frame number (5) by the page frame size (100).

ADDRESS_PAGE-FRAME = NUMBER_PAGE-FRAME * SIZE_PAGE-FRAME ADDRESS_PAGE-FRAME = 5 * 100 = 500

STEP 4 Now add the displacement (calculated in Step 1) to the starting address of the page frame to compute the precise location of the instruction in the memory:

INSTR_ADDRESS_IN_MEMORY = ADDRESS_PAGE-FRAME + DISPLACEMENT
INSTR_ADDRESS_IN_MEMORY = 500 + 76 = 576

The result of these calculations tells us exactly where (location 576) the requested byte (Byte 276) is located in main memory.

Demand Paging Memory Allocation

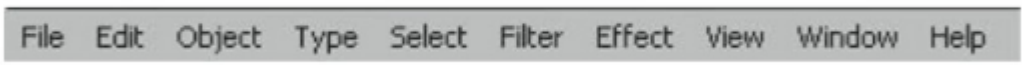
Demand paging introduces the concept of loading only a part of the program into memory for processing. It was the first widely used scheme that removed the restric- tion of having to have the entire job in memory from the beginning to the end of its processing. With demand paging, jobs are still divided into equally sized pages, which initially reside in secondary storage. When the job begins to run, its pages are brought into memory only as they are needed, and if they're never needed, they're never loaded.

Demand paging takes advantage of the fact that programs are written sequentially so that, while one section or module is processed, other modules may be idle. Not all the pages are accessed at the same time, or even sequentially. For example:

• Instructions to handle errors are processed only when a specific error is detected during execution. For instance, these instructions can indicate that input data was incorrect or that a computation resulted in an invalid answer. If no error occurs, and we hope this is generally the case, these instructions are never processed and never need to be loaded into memory.

• Many program modules are mutually exclusive. For example, while the program is being loaded—when the input module is active—then the processing module is inactive because it is generally not performing calculations during the input stage. Similarly, if the processing module is active, then the output module (such as printing) may be idle.

• Certain program options are either mutually exclusive or not always accessible. For example, when a program gives the user several menu choices, as shown in Figure 3.4, it allows the user to make only one selection at a time. If the user selects the first option (File), then the module with those program instructions is the only one that is being used; therefore, this is the only module that needs to be in memory at this time. The other modules remain in secondary storage until they are "called." For example, many tables may be assigned a large fixed amount of address space, even if only a fraction of the table is actually used. One instance of this is a symbol table that might be prepared to handle 100 symbols, but if only 10 symbols are used, then 90 percent of the table remains unused.



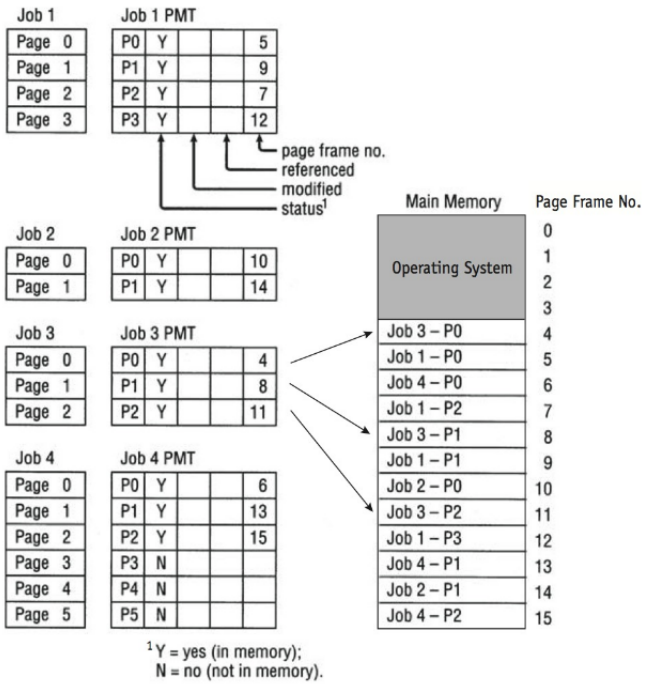
(figure 3.4) When you choose one option from the menu of an application program, such as this one, the other mod- ules, which aren't currently required such as Help, don't need to be moved into memory immediately, and, typically, are loaded only when requested.

The demand paging scheme allows the user to run jobs with less main memory than is required if the operating system were using the paged memory allocation scheme (described earlier). In fact, a demand paging scheme can give the appearance of vast amounts of physical memory when, in reality, physical memory is significantly less than vast.

The key to the successful implementation of this scheme is the use of a high-speed direct access storage device (often shortened to DASD), which will be discussed in Chapter 7. Examples of DASDs include hard drives or flash memory. High-speed access is vital because this scheme requires pages to be passed quickly from secondary storage into main memory and back again as they are needed.

How and when the pages are passed (also called swapped) between main memory and secondary storage depend on predefined policies that determine when to make room for needed pages, and how to do so. The operating system relies on tables, such as the Job Table, the Page Map Tables, and the Memory Map Table, to implement the algorithm. These tables are basically the same as for paged memory allocation. With demand pag- ing, there are three new fields for each page in each PMT: one to determine if the page being requested is already in memory, a second to determine if the page contents have been modified while in memory, and a third to determine if the page has been referenced most recently. The fourth field, which we discussed earlier in this chapter, shows the page frame number, as shown at the top of Figure 3.5.

The memory field tells the system where to find each page. If it is already in memory (shown here with a Y), the system will be spared the time required to bring it from second- ary storage. As one can imagine, it's faster for the operating system to scan a table located in main memory than it is to retrieve a page from a disk or other secondary storage device.



(Figure 3.5) Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified or recently referenced, and the page frame number for each page currently in main memory. (Note: For this illustration, the Page Map Tables have been simplified. See Table 3.3 for more detail.)

The modified field, noting whether or not the page has been changed, is used to save time when pages are removed from main memory and returned to secondary storage. If the contents of the page haven't been modified, then the page doesn't need to be rewrit- ten to secondary storage. The original, the one that is already there, is correct.

The referenced field notes any recent activity, and is used to determine which pages show the most processing activity and which are relatively inactive. This information is in main memory and which used by several page-swapping policy schemes to determine which pages should remain should be swapped out when the system needs to make room for other pages being requested.

The last field shows the page frame number for that page.

For example, in Figure 3.5 the number of total job pages is 15, and the number of total page frames available to jobs is 12. The operating system occupies the first four of the 16-page frames in main memory.

Assuming the processing status illustrated in Figure 3.5, what happens when Job 4 requests that Page 3 be brought into memory, given that there are no empty page frames available?

In order to move in a new page, one of the resident pages must be swapped back into secondary storage. Specifically, this includes copying the resident page to the disk (if it was modified) and writing the new page into the newly available page frame.

2.3 Page Replacement Policies and Concepts

As we just learned, the policy that selects the page to be removed, the page replacement policy, is crucial to the efficiency of the system, and the algorithm to do that must be carefully selected.

Several such algorithms exist, and it is a subject that enjoys a great deal of theoretical attention and research. Two of the most well-known algorithms are first-in first-out, and least recently used. The first-in first-out (FIFO) policy is based on the assumption that the best page to remove is the one that has been in memory the longest. The least recently used (LRU) policy chooses the page least recently accessed to be swapped out.

To illustrate the difference between FIFO and LRU, let us imagine a dresser drawer filled with your favorite sweaters. The drawer is full, but that didn't stop you from buying a new sweater. Now you have to put it away. Obviously it won't fit in your sweater drawer unless you take something out—but which sweater should you move to the storage closet? Your decision will be based on a "sweater removal policy."

You could take out your oldest sweater (i.e., the one that was first-in), figuring that you probably won't use it again—hoping you won't discover in the following days that it is your most used, most treasured, possession. Or, you could remove the sweater that you haven't worn recently and has been idle for the longest amount of time (i.e., the one that was least recently used). It is readily identifiable because it is at the bottom of the drawer. But just because it hasn't been used recently doesn't mean that a once-a-year occasion won't demand its appearance soon!

First-In First-Out

The first-in first-out (FIFO) page replacement policy will remove the pages that have been in memory the longest, that is, those that were first in. The process of swapping pages is illustrated in Figure 3.7.

• Step 1: Page A is moved into the first available page frame.

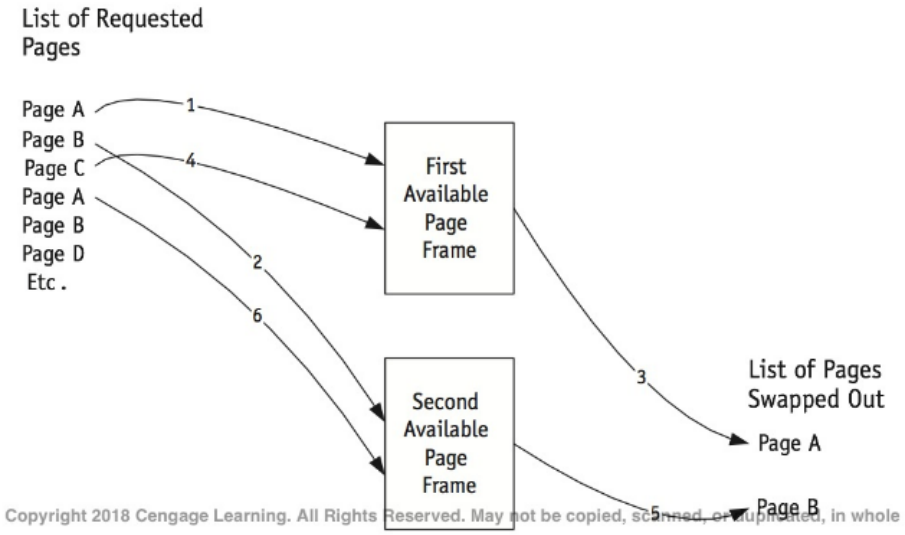
• Step 2: Page B is moved into the second available page frame.

• Step 3: Page A is swapped into secondary storage.

• Step 4: Page C is moved into the first available page frame.

• Step 5: Page B is swapped into secondary storage.

• Step 6: Page A is moved into the second available page frame.



(Figure 3.7) First, Pages A and B are loaded into the two avail- able page frames. When Page C is needed, the first page frame is emptied so that C can be placed there. Then Page B is swapped out so that Page A can be loaded there.

Figure 3.8 demonstrates how the FIFO algorithm works by following a job with four pages (A, B, C, and D) as it is processed by a system with only two available page frames. The job needs to have its pages processed in the following order: A, B, A, C, A, B, D, B, A, C, D.

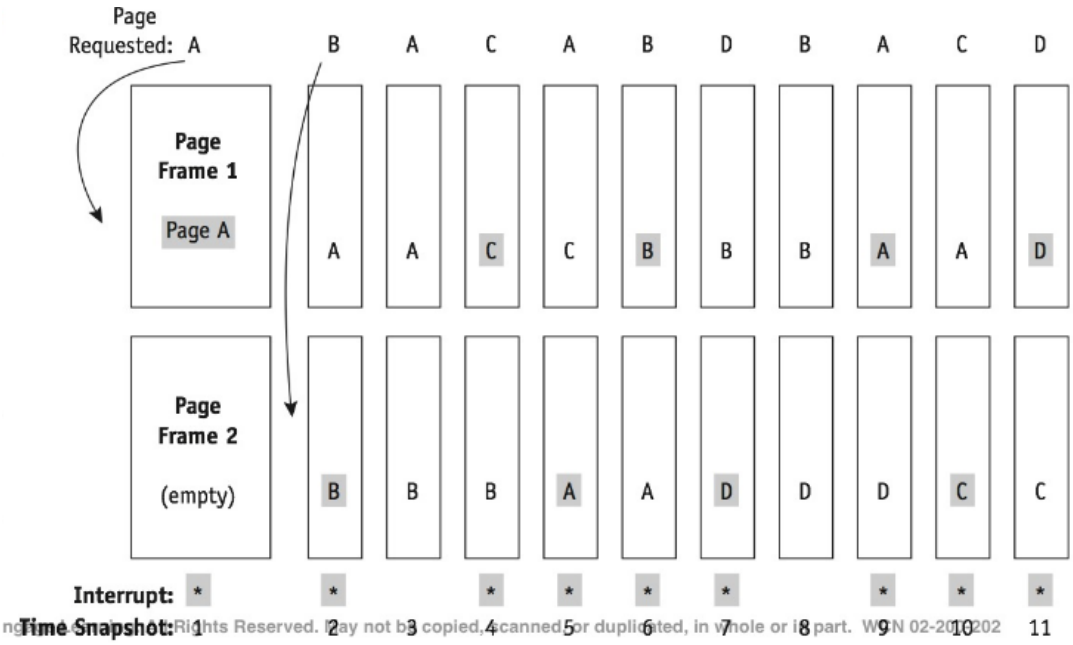
When both page frames are occupied, each new page that is brought into memory will cause an interrupt and a page swap into secondary storage. A page interrupt, which we identify with an asterisk (*), is generated anytime a page needs to be loaded into memory, whether a page is swapped out or not. Then, we count the number of page interrupts and compute the failure rate and the success rate.

The efficiency of this configuration is dismal—due to the limited number of page frames available, there are 9 page interrupts out of 11 page requests. To calculate the failure rate, we divide the number of interrupts by the number of page requests:

Failure-Rate = Number_of_Interrupts / Page_Requests_Made

The failure rate of this system is 9/11, which is 82 percent. Stated another way, the success rate is 2/11, or 18 percent. A failure rate that is this high is often unacceptable.

We are not saying FIFO is bad. We chose this example to show how FIFO works, not to diminish its appeal as a page replacement policy. The high-failure rate here was caused by both the limited amount of memory available, and the order in which pages were requested by the program. Because the job dictates the page order, this order can't be changed by the system, although the size of main memory can be changed. However, buying more memory may not always be the best solution—especially when you have many users and each one wants an unlimited amount of memory. In other words, as explained later in this chapter, there is no guarantee that buying more memory will always result in better performance.



(Figure 3.8) Using a FIFO policy, this page trace analysis shows how each page requested is swapped into the two available page frames. When the program is ready to be processed, all four pages are in secondary stor- age. When the program calls a page that isn't already in memory, a page interrupt is issued, as shown by the gray boxes and asterisks.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 2 page 74 up to 77

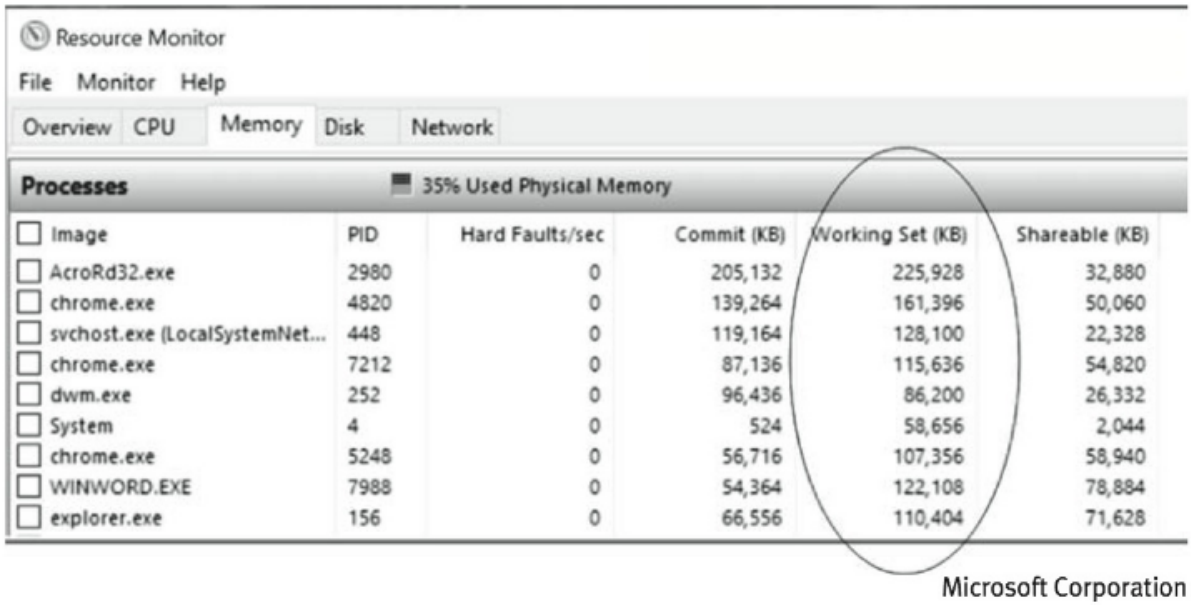
2.4 The Importance of the Working Set

One innovation that improved the performance of demand paging schemes was the concept of the working set, which is the collection of pages residing in memory that can be accessed directly without incurring a page fault. Typically, a job's working set changes as the job moves through the system: one working set could be used to initialize the job, another could work through repeated calculations, another might interact with output devices, and a final set could close the job.

For example, when a user requests an execution of a program, the first page is loaded into memory and execution continues as more pages are loaded: those containing variable declarations, others containing instructions, others containing data, and so on. After a while, most programs reach a fairly stable state, and processing continues smoothly with very few additional page faults. At this point, one of the job's working

sets is in memory, and the program won't generate many page faults until it gets to another phase requiring a different set of pages to do the work—a different working set. An example of working sets in Windows is shown in Figure 3.12.

sets is in memory, and the program won't generate many page faults until it gets to another phase requiring a different set of pages to do the work—a different working set. An example of working sets in Windows is shown in Figure 3.12.



(figure 3.12) This user opened the Resource Monitor and clicked on the Memory tab to see the working set for each open application. This image was captured from the Windows 10 operating system.

Fortunately, most programs are structured, and this leads to a locality of reference during the program's execution, meaning that, during any phase of its execution, the program references only a small fraction of its pages. If a job is executing a loop, then the instructions within the loop are referenced extensively while those outside the loop may not be used at all until the loop is completed—that's locality of reference. The same concept applies to sequential instructions, subroutine calls (within the subroutine), stack implementations, access to variables acting as counters or sums, or multidimensional variables, such as arrays and tables, where only a few of the pages are needed to handle the references.

It would be convenient if all of the pages in a job's working set were loaded into memory at one time to minimize the number of page faults and to speed up processing, but that is easier said than done. To do so, the system needs definitive answers to some difficult questions: How many pages comprise the working set? What is the maximum number of pages the operating system will allow for a working set?

The second question is particularly important in networked or time-sharing systems, which regularly swap jobs, or pages of jobs, into memory and back to secondary storage to accommodate the needs of many users. The problem is this, every time a job is reloaded back into memory (or has pages swapped), it has to generate several page faults until its working set is back in memory and processing can continue. It is a time- consuming task for the CPU, as shown in Figure 3.13.

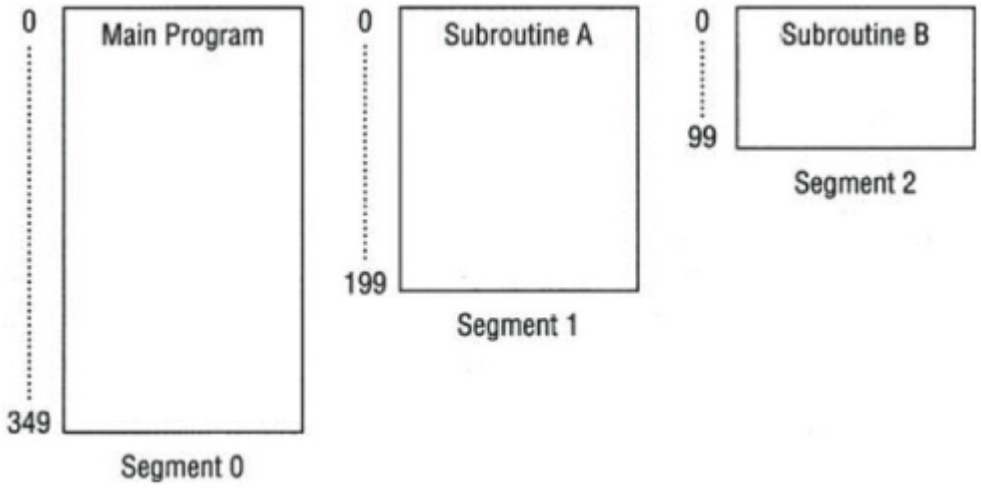
One solution adopted by many paging systems is to begin by identifying each job's working set and then loading it into memory in its entirety before allowing execution to begin. In a time-sharing or networked system, this means the operating system must keep track of the size and identity of every working set, making sure that the jobs des- tined for processing at any one time won't exceed the available memory. Some operating systems use a variable working set size, and either increase it when necessary (i.e., the job requires more processing) or decrease it when necessary. This may mean that the number of jobs in memory will need to be reduced if, by doing so, the system can ensure the completion of each job and the subsequent release of its memory space.

2.5 Segmented Memory Allocation

The concept of segmentation is based on the common practice by programmers of structuring their programs in modules—logical groupings of code. With segmented memory allocation, each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. A subroutine is an example of one such logical group. Segmented memory allocation was designed to reduce page faults, such as those that result from having a segment's loop split over two or more pages. Segmented memory allocation is fundamentally different from a paging scheme, which divides the job into several same sized pages, each of which often contains pieces from more than one program module.

A second important difference of segmented memory allocation is that its main memory is no longer divided into page frames, because the size of each segment is different, rang- ing from quite small to large. Therefore, as with the dynamic partition allocation scheme discussed in Chapter 2, memory is allocated in a dynamic manner.

When a program is compiled or assembled, the segments are set up according to the pro- gram's structural modules. Each segment is numbered and a Segment Map Table (SMT) is generated for each job; it contains the segment numbers, their lengths, access rights, status, and, when each is loaded into memory, its location in memory. Figures 3.14 and 3.15 show the same job that's composed of a main program and two subroutines with its Segment Map Table and main memory allocation. One subroutine calculates the normal pay rate, and a second one calculates the overtime pay or commissions. Notice that the segments need not be located adjacent or even near each other in main memory. The referenced and modified bits are used in segmentation, and appear in the SMT even though they aren't included in Figure 3.15.



(figure 3.14) Segmented memory allocation. Job 1 includes a main program and two subroutines. It is a single job that is structurally divided into three segments of different sizes.

The Memory Manager needs to keep track of the segments in memory. This is done with three tables, combining aspects of both dynamic partitions and demand paging memory management:

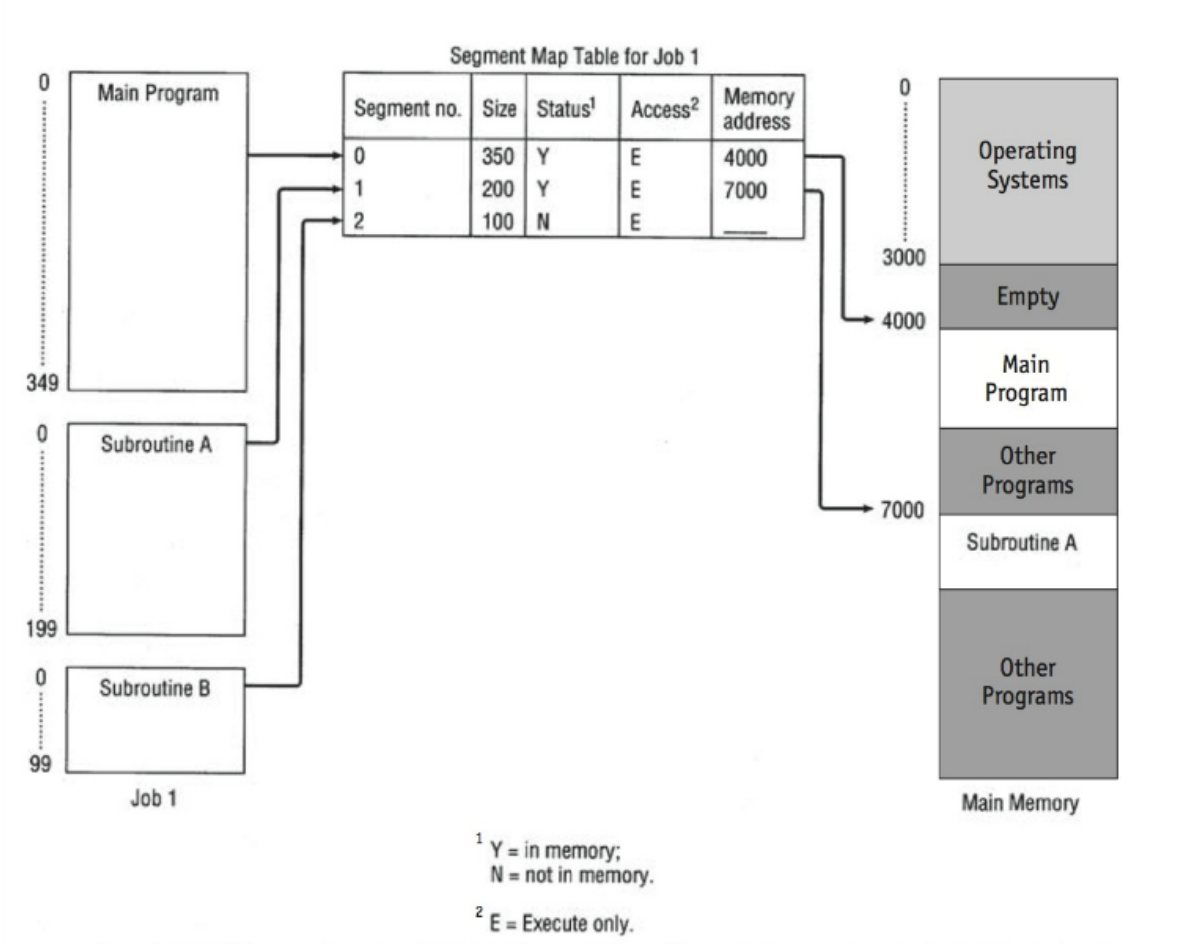
• The Job Table lists every job being processed (one for the whole system).

• The Segment Map Table lists details about each segment (one for each job).

• The Memory Map Table monitors the allocation of main memory (one for the whole system).

Like demand paging, the instructions within each segment are ordered sequentially, but the segments don't need to be stored contiguously in memory. We just need to know where each segment is stored. The contents of the segments themselves are contiguous in this scheme.

To access a specific location within a segment, we can perform an operation similar to the one used for paged memory management. The only difference is that we work with segments instead of pages. The addressing scheme requires the segment number and the displacement within that segment; and because the segments are of different sizes, the displacement must be verified to make sure it isn't outside of the segment's range.

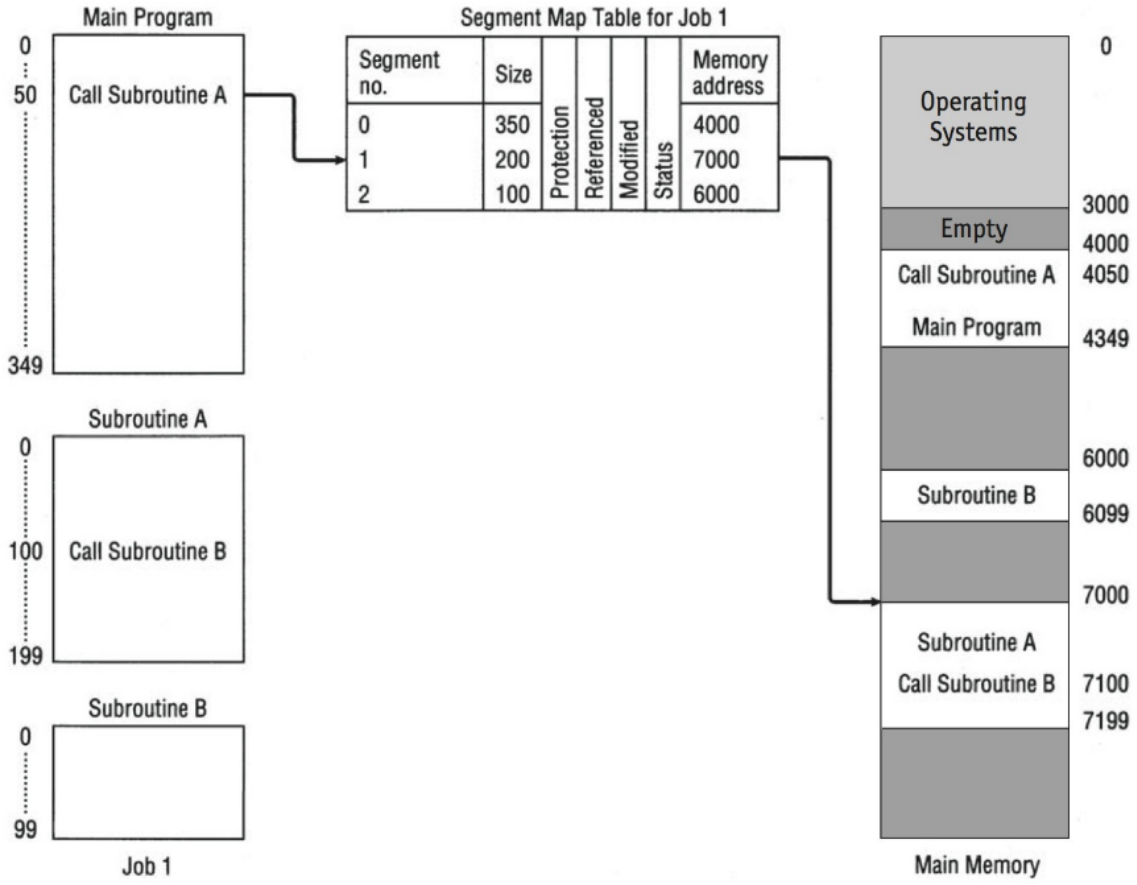


(Figure 3.15) The Segment Map Table tracks each segment for this job. Notice that Subroutine B has not yet been loaded into memory.

In Figure 3.16, Segment 1 includes all of Subroutine A, so the system finds the beginning address of Segment 1, 7000, and it begins there.

If the instruction requested that processing begin at byte 100 of Subroutine A (which is possible in languages that support multiple entries into subroutines) then, to locate that item in memory, the Memory Manager would need to add 100 (the displacement) to 7000 (the beginning address of Segment 1).

Can the displacement be larger than the size of the segment? No, not if the program is coded correctly; however, accidents and attempted exploits do happen, and the Memory Manager must always guard against this possibility by checking the displacement against the size of the segment, verifying that it is not out of bounds.



(Figure 3.16) During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.

To access a location in memory, when using either paged or segmented memory man- agement, the address is composed of two values: the page or segment number and the displacement. Therefore, it is a two-dimensional addressing scheme—that is, it has two elements:

SEGMENT_NUMBER and DISPLACEMENT.

The disadvantage of any allocation scheme in which memory is partitioned dynamically is the return of external fragmentation. Therefore, if that schema is used, re-compaction of available memory is necessary from time to time.

As you can see, there are many similarities between paging and segmentation, so they are often confused. The major difference is a conceptual one: pages are physical units that are invisible to the user's program and consist of fixed sizes; segments are logical units that are visible to the user's program and consist of variable sizes.

Virtual Memory

Virtual memory gives users the illusion that the computer has much more memory than it actually has. That is, it allows very large jobs to be executed even though they require large amounts of main memory that the hardware does not possess. A form of virtual memory first became possible with the capability of moving pages at will between main memory for processing and secondary storage while it awaits processing. This technique effectively removed restrictions on maximum program size.

With virtual memory, even though only a portion of each program is stored in memory at any given moment, by swapping pages into and out of memory, it gives users the appearance that their programs are completely loaded into main memory during their entire processing time—a feat that would require an incredibly large amount of main memory.

Virtual memory can be implemented with both paging and segmentation, as seen in Table 3.6.

Virtual Memory with Paging	Virtual Memory with Segmentation
Allows internal fragmentation within page frames	Doesn't allow internal fragmentation
Doesn't allow external fragmentation	Allows external fragmentation
Programs are divided into equal-sized pages	Programs are divided into unequal-sized segments that contain logical groupings of code
The absolute address is calculated using the page number and displacement	The absolute address is calculated using the segment number and displacement
Requires Page Map Table (PMT)	Requires Segment Map Table (SMT)

(Table 3.6) Comparison of the advantages and disadvantages of virtual memory with paging and segmentation.

Segmentation allows users to share program code. The shared segment contains: (1) an area where unchangeable code (called reentrant code) is stored, and (2) several data areas, one for each user. In this case, users share the code, which cannot be modified, but they can modify the information stored in their own data areas as needed, without affecting the data stored in other users' data areas.

Before virtual memory, sharing meant that copies of files were stored in each user's account. This allowed them to load their own copy and work on it at any time. This kind of sharing created a great deal of unnecessary system cost—the I/O overhead in loading the copies and the extra secondary storage needed. With virtual memory, these costs are substantially reduced because shared programs and subroutines are loaded on demand, satisfactorily reducing the storage requirements of main memory, although this is accomplished at the expense of the Memory Map Table.

The use of virtual memory requires cooperation between the Memory Manager, which tracks each page or segment; and the processor hardware, which issues the interrupt and resolves the virtual address. For example, when a page that is not already in memory is needed, a page fault is issued and the Memory Manager chooses a page frame, loads the page, and updates entries in the Memory Map Table and the Page Map Tables.

Virtual memory works well in a multiprogramming environment because most pro- grams spend a lot of time waiting—they wait for I/O to be performed; they wait for pages to be swapped in or out; and they wait for their turn to use the processor. In a multiprogramming environment, the waiting time isn't wasted because the CPU simply moves to another job.

Virtual memory has increased the use of several programming techniques. For instance, it aids the development of large software systems, because individual pieces can be developed independently and linked later on.

Virtual memory management has several advantages:

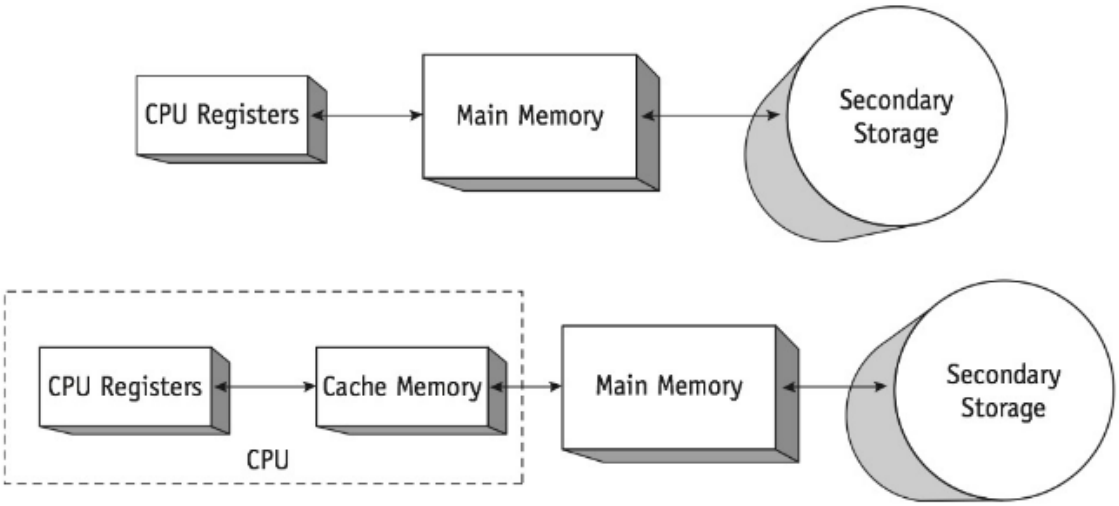
• A job's size is no longer restricted to the size of main memory (or worse, to the free space available within main memory).
• Memory is used more efficiently because the only sections of a job stored in memory are those needed immediately, while those not needed remain in secondary storage.
• It allows an unlimited amount of multiprogramming, which can apply to many jobs, as in dynamic and static partitioning; or to many users.
• It allows the sharing of code and data.
• It facilitates the dynamic linking of program segments.

The advantages far outweigh the following disadvantages:

• Increased processor hardware costs
• Increased overhead for handling paging interrupts
• Increased software complexity to prevent thrashing

Cache Memory

Cache memory is based on the concept of using a small, fast, and expensive memory to supplement the workings of main memory. Because the cache is usually small in capacity (compared to main memory), it can use more expensive memory chips. These are five to ten times faster than main memory and match the speed of the CPU. Therefore, if data or instructions that are frequently used are stored in cache memory, memory access time can be cut down significantly; and the CPU can execute those instructions faster, thus, raising the overall performance of the computer system. It's similar to the role of a frequently called list of telephone numbers in a telephone. By keeping those numbers in an easy-to-reach place, they can be called much faster than those in a long contact list.



(Figure 3.19) The traditional path used by early computers was direct: from secondary storage to main memory to the CPU registers, but speed was slowed by the slow connections (top). With cache memory directly accessible from the CPU registers (bottom), a much faster response is possible.

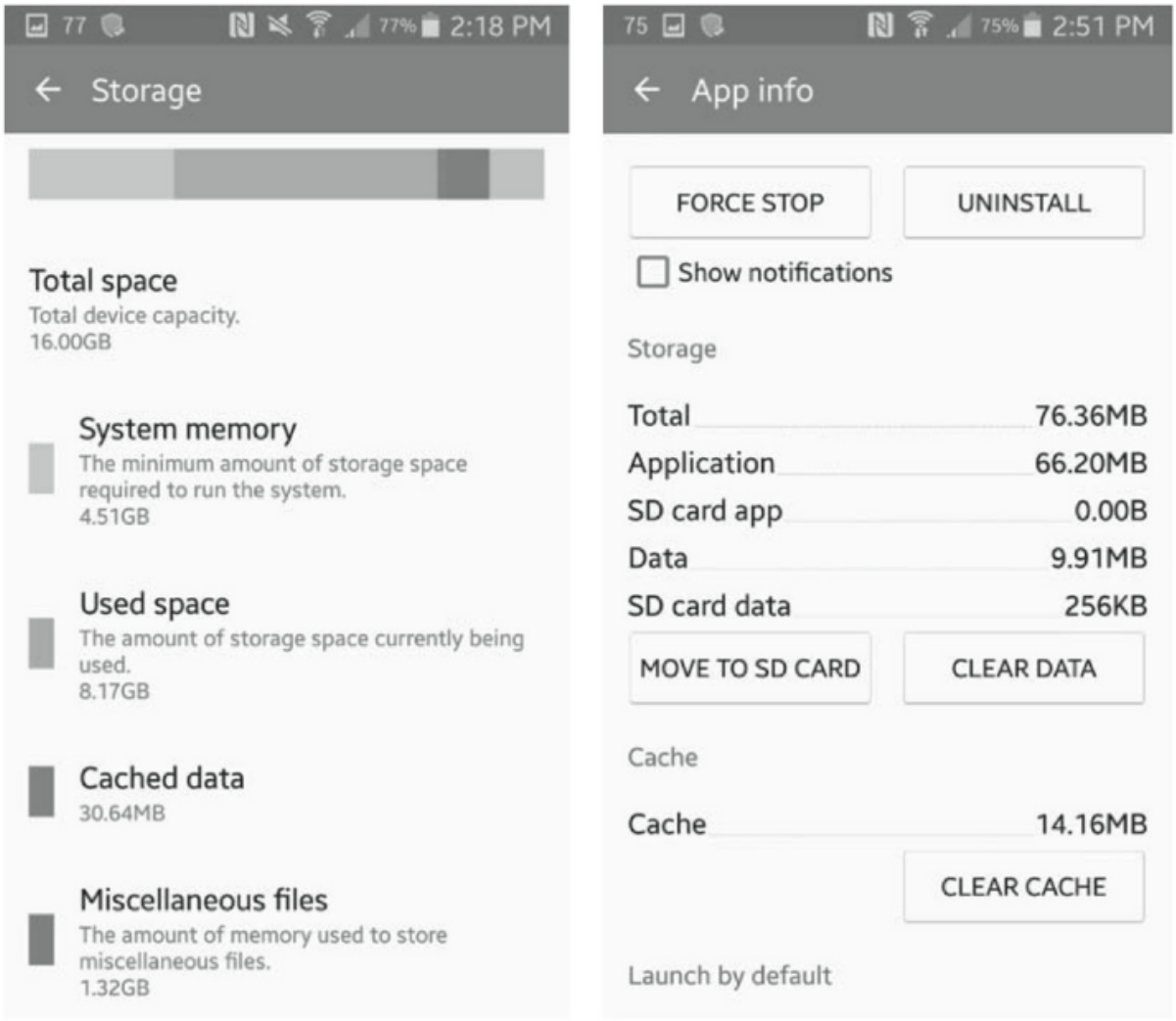
As shown in Figure 3.19, early computers were designed to have data and instructions transferred from secondary storage to main memory, and then to special-purpose registers for processing—this path was sufficient for inherently slow computing systems. However, because the same instructions were being used repeatedly in most programs, computer system designers thought it would be more efficient if the system did not use a complete memory cycle every time an instruction or data value was required. To speed up processing, designers found that this could be done if they placed repeatedly used data in general-purpose registers instead of in main memory.

Currently, computer systems automatically store frequently used data in an intermediate memory unit called cache memory. This adds a middle layer to the original hierarchy. Cache memory can be thought of as an intermediary between main memory and the special-purpose registers, which are the domain of the CPU, as shown in Figure 3.19.

A typical microprocessor has two or more levels of caches, such as Level 1 (L1), Level 2 (L2), and Level 3 (L3), as well as specialized caches.

In a simple configuration with only two cache levels, information enters the processor through the bus interface unit, which immediately sends one copy to the Level 2 cache, which is an integral part of the microprocessor and is directly connected to the CPU. A second copy is sent to one of two Level 1 caches, which are built directly into the chip. One of these Level 1 caches stores instructions, while the other stores data to be used by the instructions. If an instruction needs additional data, the instruction is put on hold while the processor looks for the missing data—first in the data Level 1 cache, and then in the larger Level 2 cache before searching main memory. Because the Level 2 cache is an integral part of the microprocessor, data moves two to four times faster between the CPU and the cache than between the CPU and main memory.

To understand the relationship between main memory and cache memory, consider the relationship between the size of the Web and the size of your private browser bookmark file. Your bookmark file is small and contains only a tiny fraction of all the available addresses on the Web; but the chance that you will soon visit a Web site that is in your bookmark file is high. Therefore, the purpose of your bookmark file is to keep your most frequently accessed addresses easy to reach so that you can access them quickly. Similarly, the purpose of cache memory is to keep the most recently accessed data and instructions handy, so that the CPU can access them repeatedly without wasting time.



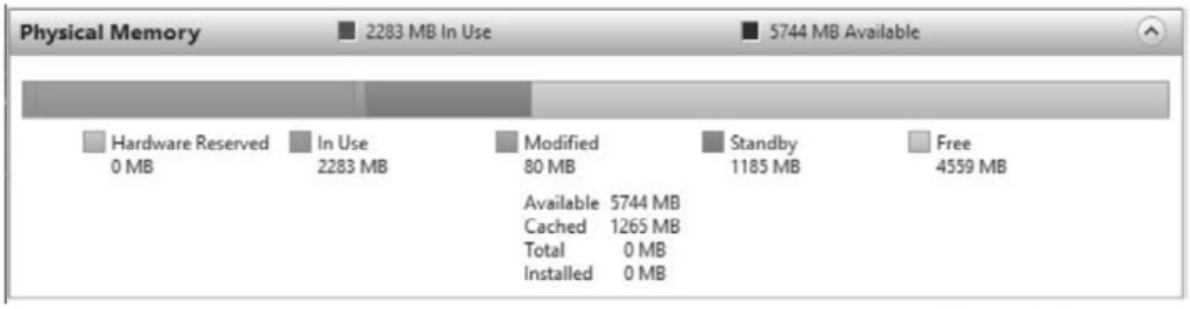
Microsoft Corporation

(Figure 3.20) This cell phone, which is running the Android operating system, shows (a) how the device's available memory and storage space is shared among the system, apps, cached data, and other miscellaneous files; and (b) how a single app has stored 14 MB in the phone's cache.

The movement of data or instructions from main memory to cache memory uses a method similar to that used in paging algorithms. First, cache memory is divided into blocks of equal size, called slots. Then, when the CPU first requests an instruction or data from a location in main memory, the requested instruction, along with several others around it, are transferred from main memory to cache memory, where they are stored in one of the free slots. Moving a block at a time is based on the principle of locality of reference, which states that it is very likely that the next CPU request will be physically close to the one just requested. In addition to the block of data transferred, the slot also contains a label that indicates the main memory address from which the block was copied. When the CPU requests additional information from that location in main memory, cache memory is accessed first; and if the contents of one of the labels in a slot matches the address requested, then access to main memory is not required.

The algorithm to execute one of these transfers from main memory is simple to imple- ment (a pseudocode algorithm can be found in Appendix A).

These steps become more complicated when there are no free cache slots, which can occur because the size of cache memory is smaller than that of main memory—in this case, individual slots cannot be permanently allocated to blocks. To address this con- tingency, the system needs a policy for block replacement, which could be similar to those used in page replacement.



(Figure 3.21) A Windows 10 screen- shot of the Resource Monitor shows this device's available memory and cache size.

When designing cache memory, one must take into consideration the following four factors:

• Cache size. Studies have shown that having any cache, even a small one, can sub- stantially improve the performance of the computer system.

• Block size. Because of the principle of locality of reference, as block size increases, the ratio of the number of references found in the cache to the total number of ref- erences will tend to increase.

• Block replacement algorithm. When all the slots are busy and a new block has to be brought into the cache, the block selected for replacement should be one that is least likely to be used in the near future. However, as we saw in paging, this is nearly impossi- ble to predict, so a reasonable course of action is to select a block that has not been used for a long time. Therefore, LRU is the algorithm that is often chosen for block replace- ment, which requires a hardware mechanism to determine the least recently used slot.

• Rewrite policy. When the contents of a block residing in cache are changed, it must be written back to main memory before it is replaced by another block. A rewrite policy must be in place to determine when this writing will take place. One alterna- tive is to do this every time that a change occurs, which would increase the number of memory writes, possibly increasing overhead. A second alternative is to do this only when the block is replaced or the process is finished, which would minimize overhead, but would leave the block in main memory in an inconsistent state. This would create problems in multiprocessor environments, and in cases where I/O modules can access main memory directly.

The optimal selection of cache size and replacement algorithm can result in 80 to 90 percent of all requests being made in the cache, making for a very efficient memory system. This measure of efficiency, called the cache hit ratio, is used to determine the performance of cache memory and, when shown as a percentage, represents the percent- age of total memory requests that are found in the cache. One formula is this:

$$\text{HitRatio} = \frac{\text{5 number of requests found in the cache}}{\text{total number of requests}} * 100$$

For example, if the total number of requests is 10, and 6 of those are found in cache memory, then the hit ratio is 60 percent, which is reasonably good and suggests improved system performance.

$$\text{HitRatio} = (6/10) * 100 = 60\%$$

Likewise, if the total number of requests is 100, and 9 of those are found in cache memory, then the hit ratio is only 9 percent.

$$\text{HitRatio} = (9/10) * 100 = 9\%$$

Another way to measure the efficiency of a system with cache memory, assuming that the system always checks the cache first, is to compute the average memory access time (Avg_Mem_AccTime) using the following formula:

Avg_Mem_AccTime = Avg_Cache_AccessTime + (1-HitRatio)*Avg_MainMem_AccTime

For example, if we know that the average cache access time is 200 nanoseconds (nsec) and the average main memory access time is 1000 nsec, then a system with a hit ratio of 60 percent will have an average memory access time of 600 nsec:

$$\text{AvgMemAccessTime} = 200 + (1 - 0.60) * 1000 = 600 \text{ nsec}$$

A system with a hit ratio of 9 percent will show an average memory access time of 1110 nsec:

$$\text{AvgMemAccessTime} = 200 + (1 - 0.60) * 1000 = 1110 \text{ nsec}$$

Because of its role in improving system performance, cache is routinely added to a wide variety of main memory configurations as well as devices.

Conclusion

Four memory management techniques were presented in this chapter: single-user sys- tems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Although they have differences, they all share the requirement that the entire program (1) beloaded into memory, (2) be stored contiguously, and (3) remain in memory until the entire job is completed. Consequently, each puts severe restrictions on the size of execut- able jobs because each one can only be as large as the biggest partition in memory. These schemes laid the groundwork for more complex memory management techniques that allowed users to interact more directly, and more often, with the system.

The memory management portion of the operating system assumes responsibility for allocating memory storage (in main memory, cache memory, and registers) and, equally important, for deallocating it when execution is completed.

The Memory Manager is only one of several managers that make up the operating system. After the jobs are loaded into memory using a memory allocation scheme, the Processor Manager takes responsibility for allocating processor time to each job, and every process and thread in that job, in the most efficient manner possible.

Key Terms

Address: a number that designates a particular memory location.
Best-fit memory allocation method: a main memory allocation scheme that considers all free blocks, and selects for allocation the one that will result in the least amount of wasted space.
Bounds register: a register used to store the highest location in memory legally accessible by each program.
Compaction of memory: the process of collecting fragments of available memory space into contiguous blocks by relocating programs and data in a computer's memory.
Deallocation: the process of freeing an allocated resource, whether memory space, a device, a file, or a CPU.
Dynamic partition: a memory allocation scheme in which jobs are given as much mem- ory as they request when they are loaded for processing, thus, creating their own parti- tions in the main memory.
External fragmentation: a situation in which the dynamic allocation of memory creates unusable fragments of free memory between blocks of busy, or allocated, memory.
First-fit memory allocation method: a main memory allocation scheme that searches from the beginning of the free block list, and selects to allocate the first block of memory large enough to fulfill the request.
Fixed partitions: a memory allocation scheme in which main memory is sectioned with one partition assigned to each job.
Internal fragmentation: a situation in which a partition is only partially used by the
Program; the remaining space within the partition is unavailable to any other job and is, therefore, wasted.
Cache memory: a small, fast memory used to hold selected data and to provide fast access.
FIFO anomaly or the Belady Anomaly: an unusual circumstance through which add- ing more page frames causes an increase in page interrupts when using a FIFO page replacement policy.

First-in first-out (FIFO) policy: a page replacement policy that removes from main mem- ory the pages that were first brought in. Job Table (JT): a table in main memory that contains two values for each active job— the size of the job, and the memory location where its Page Map Table is stored.

Chapter Review

To Explore More

For additional background on a few of the topics discussed in this chapter, begin a search with these terms.

- Memory fragmentation
- Memory relocation
- Best-fit memory allocation
- Memory partitions
- Virtual Memory Principles
- Cache Memory Management
- Working Set

Research Topics

A. New research on topics concerning operating systems and computer science are published frequently by professional societies; and academic libraries offer access to many of these journals. Research the Internet or current literature to identify at least two prominent professional societies with peer-reviewed computer science journals, then list the advantages of becoming a member and the dues for students. Also provide a one-sentence summary of a published paper concerning operating systems. Cite your sources.

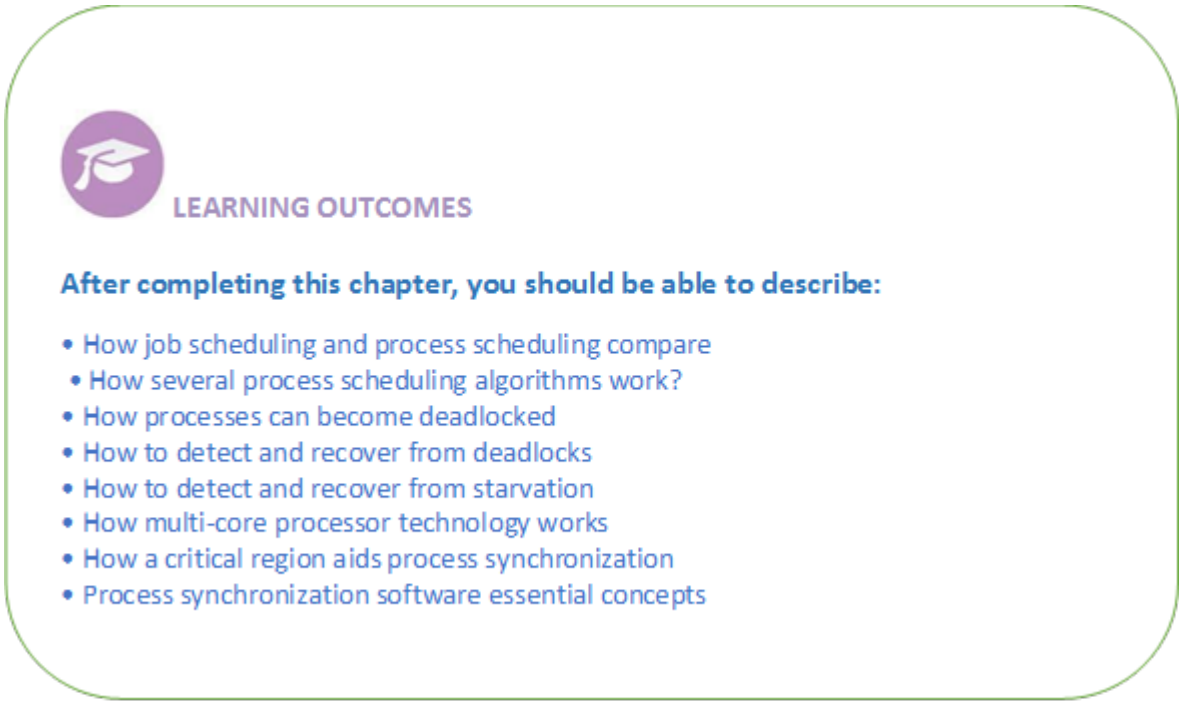
B. Research the maximum and minimum memory capacity of two models of laptop computers. Be sure not to confuse the computer's memory capacity with its secondary storage. Explain why memory capacity and secondary storage are often confused. Cite the sources of your research.

Review Questions

1. Consider the first and last memory allocation scheme described in this chapter. Describe their respective advantages and disadvantages.
2. In your own words, how often should memory compaction/relocation be per- formed? Describe the advantages and disadvantages of performing it even more often than recommended.
3. Give an example of computing circumstances that would favor first-fit allocation over best-fit. Explain your answer.
4. Compare and contrast internal fragmentation and external fragmentation. Explain the circumstances where one might be preferred over the other.
5. Explain the function of the Page Map Table in all of the memory allocation schemes described in this chapter that make use of it. Explain your answer by describing how the PMT is referenced by other pertinent tables in each scheme.
6.If a program has 471 bytes and will be loaded into page frames of 126 bytes each, assuming the job begins loading at the first page (Page 0) in memory, and the instruction to be used is at byte 132, answer the following questions:
a. How many pages are needed to store the entire job?
b. Compute the page number and exact displacement for each of the byte addresses where the desired data is stored.
7. Let's say a program has 1010 bytes and will be loaded into page frames of 256 bytes each, assuming the job begins loading at the first page (Page 0) in memory, and the instruction to be used is at Byte 577, answer the following questions:
a. How many pages are needed to store the entire job?
b. Compute the page number and exact displacement for the byte addresses where the data is stored.

8.Given that main memory is composed of only three page frames for public use, and that a seven-page program (with pages a, b, c, d, e, f, g) requests pages in the following order:
a, b, c, b, d, a, e, f, b, e, d, f
answer the following questions.
a. Using the FIFO page removal algorithm, indicate the movement of the pages into and out of the available page frames (called a page trace analysis), indicating each page fault with an asterisk (*). Then compute the failure ratio and success ratio.
b. Using FIFO again, increase the size of memory so that it contains four page frames for public use. Using the same page requests as above, do another page trace analysis and compute the failure and success ratios.
c. What general statement can you make from this example? Explain your answer.

Chapter 3: Process Management Synchronization and Concurrent



The Processor Manager is responsible for allocating the processor to execute the instruc- tions for incoming jobs. In this chapter, we'll see how an operating system's Processor Manager typically reaches these goals. Although we examine single-central processing unit (CPU) systems in this chapter, the same concepts are used in multi-CPU systems, with a higher-level scheduler that coordinates each CPU.

3.1 Scheduling Sub-managers

The Processor Manager is a composite of at least two sub-managers: one in charge of job scheduling and the other in charge of process scheduling. They're known as the Job Scheduler and the Process Scheduler.

Jobs can be viewed either as a series of global job steps—compilation, loading, and execution—or as one all-encompassing step—execution. However, the scheduling of jobs is actually handled on two levels by most operating systems. If we return to the example presented earlier, we can see that a hierarchy exists between the Job Scheduler and the Process Scheduler.

The scheduling of the two jobs (bicycle assembly and first-aid administration) was on the basis of priority. Each job that you started was initiated by the Job Scheduler based on certain criteria. Once a job was selected for execution, the Process Scheduler deter- mined when each step, or set of steps, was executed—a decision that was also based on certain criteria. When you started to assemble the bike, each step in the assembly instructions was selected for execution by the Process Scheduler.

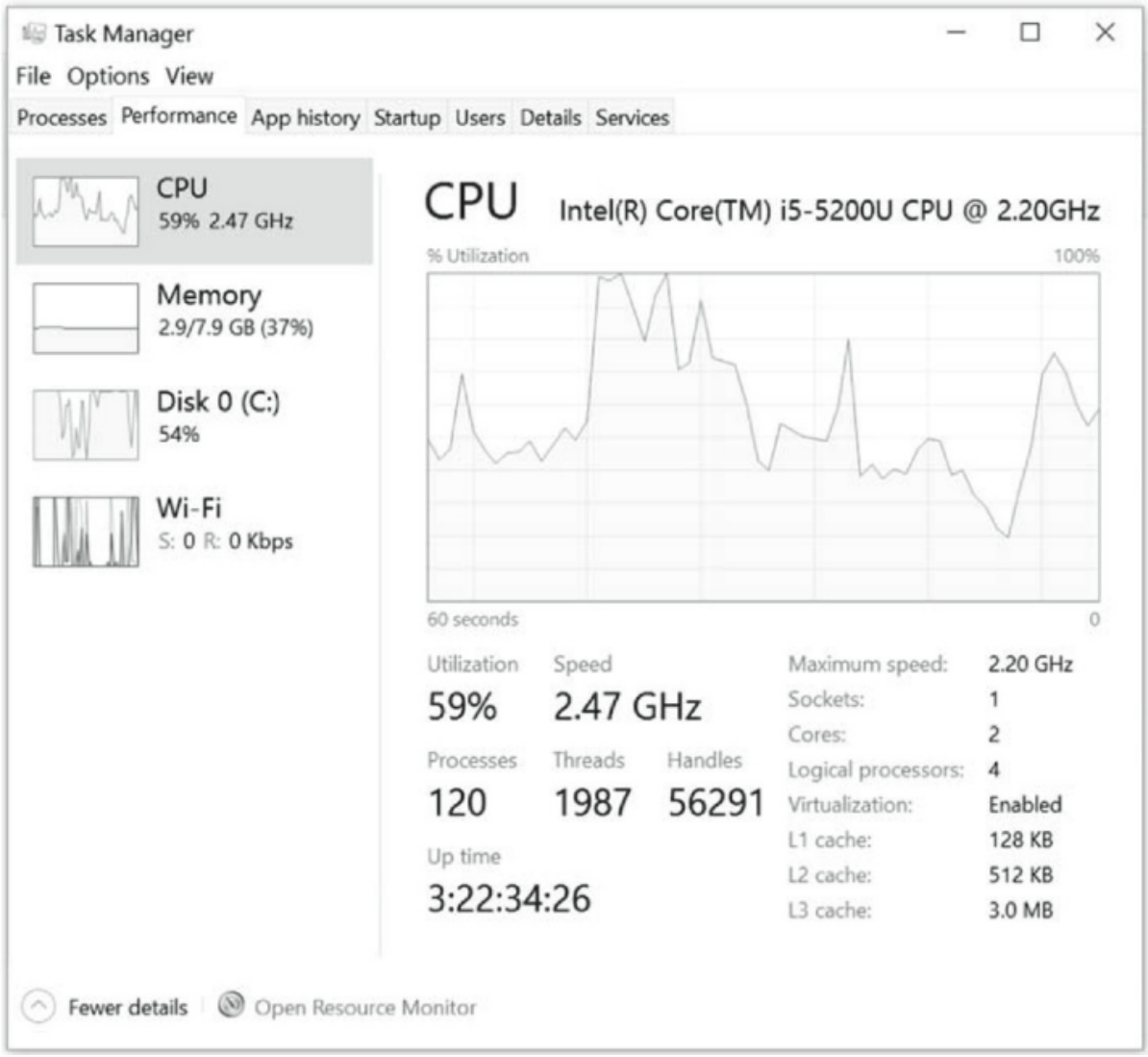
The same concepts apply to computer systems, where each job (sometimes called the pro- gram) passes through a hierarchy of managers. Since the first one it encounters is the Job Scheduler, this is also called the high-level scheduler. It is concerned only with selecting jobs from a queue of incoming jobs and placing them in the process queue based on each job's characteristics. The Job Scheduler's goal is to put the jobs (as they still reside on the disk) in a sequence that best meets the designers or administrator's goals; one such goal might be using the system's resources as efficiently as possible. This is an important function.

For example, if the Job Scheduler selected several jobs to run consecutively and each had numerous requests for input and output (often abbreviated as I/O), then those I/O devices would be kept very busy. The CPU might be busy handling the I/O requests (if an I/O controller were not used), resulting in the completion of very little computation. On the other hand, if the Job Scheduler selected several consecutive jobs with a great deal of computation requirements, then the situation would be reversed: The CPU would be very busy computing, and the I/O devices would remain idle waiting for requests. Therefore, a major goal of the Job Scheduler is to create an order for the incoming jobs that has a balanced mix of I/O interaction and computation requirements, thus, balancing the system's resources. As we might expect, one of the Job Scheduler's goals is to keep most of the components of the computer system busy most of the time, as shown in Figure 4-1.

Process Scheduler

After a job has been accepted by the Job Scheduler to run, the Process Scheduler takes over that job; and if the operating systems support threads, the Process Scheduler takes responsibility for this function as well. The Process Scheduler determines which processes will get the CPU, when, and for how long. It also decides what to do when processing is interrupted; it determines which waiting lines (called queues) the job should be allocated to during its execution; and it recognizes when a job has concluded and should be terminated.

The Process Scheduler is a low-level scheduler that assigns the CPU to execute the indi- vidual actions for those jobs placed on the READY queue by the Job Scheduler. This becomes crucial when the processing of several jobs has to be orchestrated—just as when you had to set aside your assembly and rush to help your neighbour.



(Figure 4.1) Screenshot using Task Manager in Windows 10 to monitor activity of the CPU as well as other system resources.

To schedule the CPU, the Process Scheduler takes advantage of a common trait among most computer programs: They alternate between CPU cycles and I/O cycles. Notice that the job on the next page begins with a brief I/O cycle followed by one relatively long CPU cycle, followed by another very brief I/O cycle.

Although the duration and frequency of CPU cycles vary from program to program, there are some general tendencies that can be exploited when selecting a scheduling algorithm. Two of these are I/O-bound jobs, such as printing a series of documents, that have long input or output cycles and brief CPU cycles; and CPU-bound jobs, such as finding the first 300,000 prime numbers, that have long CPU cycles and shorter I/O cycles. The total effect of all CPU cycles, from both I/O-bound and CPU-bound jobs, approximates a curve, as shown in Figure 4.2.

This extremely simple program is collecting data from the user, performing calculations, printing results on the screen, and ending.

1. Ask the user for the first number
2. Retrieve the first number that's entered: Input #1
3. Ask the user for the second number
4. Retrieve the second number entered: Input #2

} Brief I/O Cycle

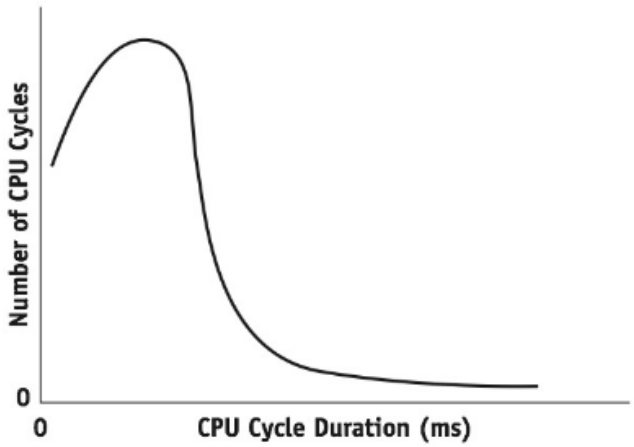
5. Add the two numbers: Input #1 + Input #2
6. Divide the sum from the previous calculation by 2 to get the average of the two inputs: (Input #1 + Input #2)/2 = Output #1
7. Multiply the result of the previous calculation by 12 to get the average for the year: Output #1*12 = Output #2

} CPU Cycle

8. Print the calculated average: Output #1
9. Print the average for the year: Output #2

} Brief I/O Cycle

10. End

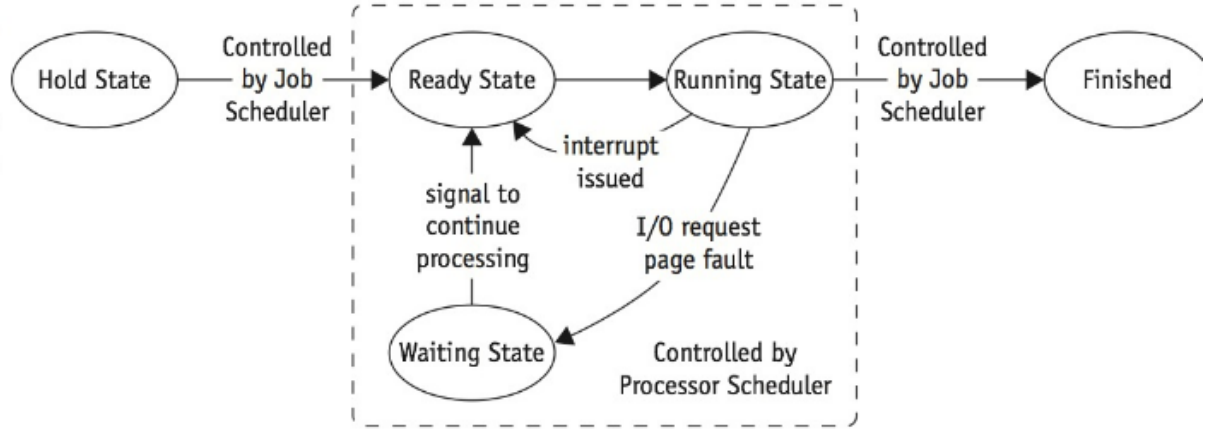


(Figure 4.2) Distribution of CPU cycle times. This distribution shows a greater number of jobs requesting short CPU cycles (the frequency peaks close to the low end of the CPU cycle axis), and fewer jobs requesting long CPU cycles.

In a highly interactive environment, there's also a third layer of the Processor Man- ager called the middle-level scheduler. In some cases, especially when the system is overloaded, the middle-level scheduler finds it is advantageous to remove active jobs from memory to reduce the degree of multiprogramming, which allows other jobs to be completed faster. The jobs that are swapped out and eventually swapped back in are managed by this middle-level scheduler.

3.2 Job and Process States

As a job, process, or thread moves through the system, its status changes, often from HOLD, to READY, to RUNNING, to WAITING, and, eventually, to FINISHED, as shown in Figure 4.3. These are called the job status, process status, or thread status, respectively.



(Figure 4.3) A typical job (or process) changes status as it moves through the system from HOLD to FINISHED.

Here's how a job status can change when a user submits a job into the system: When the job is accepted by the system, it's put on HOLD and placed in a queue. In some systems, the job spooler (or disk controller) creates a table with the characteristics of each job in the queue, and notes the important features of the job, such as an estimate of CPU time, priority, special I/O devices required, and maximum memory required. This table is used by the Job Scheduler to decide which job is to be run next.

The job moves to READY after the interrupts have been resolved. In some systems, the job (or process) might be placed on the READY queue directly. RUNNING, of course, means that the job is being processed. In a single processor system, this is one job or process. WAITING means that the job can't continue until a specific resource is allocated or an I/O operation has finished, and then it can move back to READY. Upon completion, the job is FINISHED and returned to the user.

The transition from one job status, job state, to another is initiated by the Job Scheduler, and the transition from one process or thread state to another is initiated by the Process Scheduler. Here's a simple example:

• The job's transition from HOLD to READY is initiated by the Job Scheduler, according to a policy that's predefined by the operating system's designers. At this point, the availability of enough main memory and any requested devices is checked.

• The transition from READY to RUNNING is handled by the Process Scheduler according to a predefined algorithm (this is discussed shortly).

• The transition from RUNNING back to READY is handled by the Process Sched- uler according to a predefined time limit or other criterion, such as a priority interrupt.

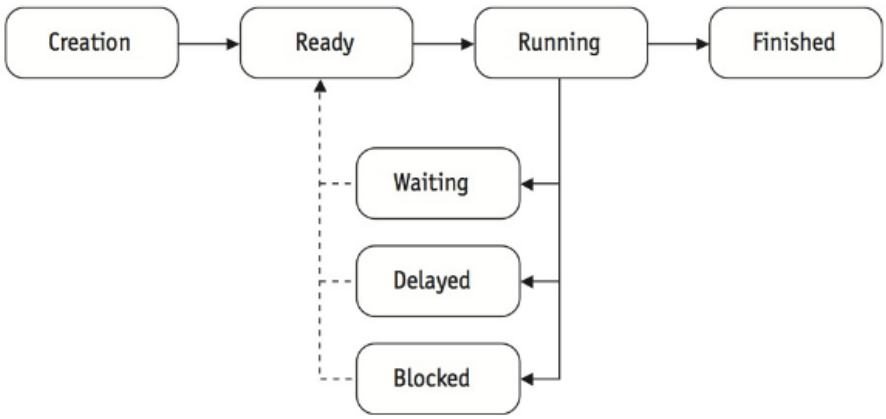
• The transition from RUNNING to WAITING is handled by the Process Sched- uler and is initiated in response to an instruction in the job, such as a command to READ, WRITE, or other I/O request.

• The transition from WAITING to READY is handled by the Process Scheduler, and is initiated by a signal from the I/O device manager that the I/O request has been satisfied and the job can continue. In the case of a page fetch, the page fault handler will signal that the page is now in memory and the process can be placed back in the READY queue.

• Eventually, the transition from RUNNING to FINISHED is initiated by the Process Scheduler, or the Job Scheduler, when the job is successfully completed and it ends execution, or, when the operating system indicates that an error has occurred and the job must be terminated prematurely.

Thread States

As a thread moves through the system it is in one of five states (not counting its creation and finished states) as shown in Figure 4.4. When an application creates a thread, it is made ready by allocating to it the needed resources and placing it in the READY queue. The thread state changes from READY to RUNNING when the Process Scheduler assigns it to a processor. In this chapter we consider systems with only one processor. See Chapter 6 for systems with multiple processors.



(Figure 4.4) A typical thread changes states several times as it moves from READY to FINISHED.

A thread transitions from RUNNING to WAITING when it has to wait for an event outside its control to occur. For example, a mouse click can be the trigger event for a thread to change states, causing a transition from WAITING to READY. Alternatively, another thread, having completed its task, can send a signal indicating that the waiting thread can continue to execute.

When an application has the capability of delaying the processing of a thread by a specified amount of time, it causes the thread to transition from RUNNING to DELAYED.

When the prescribed time has elapsed, the thread transitions from DELAYED to READY. For example, when using a word processor, the thread that periodically saves a current document can be delayed for a period of time after it has completed the save. After the time has expired, it performs the next save and then is delayed again. If the delay was not built into the application, this thread would be forced into a loop that would continuously test to see if it was time to do a save, wasting processor time and reducing system performance.

A thread transitions from RUNNING to BLOCKED when an I/O request is issued. After the I/O is completed, the thread returns to the READY state. When a thread transitions from RUNNING to FINISHED, all of its resources are released; it then exits the system or is terminated and ceases to exist.

In this way, the same operations are performed on both processes and threads. There- fore, the operating system must be able to support

- creating new threads;
- setting up a thread so it is ready to execute;
- delaying, or putting to sleep, threads for a specified amount of time;
- blocking, or suspending, threads that are waiting for i/o to be completed;
- setting threads to a wait state if necessary until a specific event has occurred;
- scheduling threads for execution;
- synchronizing thread execution using semaphores, events, or conditional variables;
- terminating a thread and releasing its resources.

To do so, the operating system needs to track the critical information for each thread.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 3 page 113 up to 115

3.3 Scheduling Policies and Algorithms

In a multiprogramming environment, there are usually more objects (jobs, processes, and threads) to be executed than could possibly be run at one time. Before the operating system can schedule them, it needs to resolve three limitations of the system:

• There are a finite number of resources.
• Some resources, once they're allocated, cannot be shared with another job.
• Some resources require operator intervention—that is, they can't be reassigned automatically from job to job.
What's a good scheduling policy?

Several goals come to mind, but notice in the follow- ing list that some goals contradict each other:

• Maximize throughput. Run as many jobs as possible in a given amount of time. This could be accomplished easily by running only short jobs or by running jobs without interrupting them.
• Minimize response time. Quickly turn around interactive requests. This could be done by running only interactive jobs and letting all other jobs wait until the interactive load ceases.
• Minimize turnaround time. Move entire jobs in and out of the system quickly. This could be done by running all batch jobs first, because batch jobs are put into groups so they can run more efficiently than interactive jobs.
• Minimize waiting time. Move jobs out of the READY queue as quickly as possible. This can be done only by reducing the number of users allowed on the system so that the CPU can be available immediately whenever a job entered the READY queue.
• Maximize CPU efficiency. Keep the CPU busy 100 percent of the time. This could be done by running only CPU-bound jobs (not I/O-bound jobs).
• Ensure fairness for all jobs. Give everyone an equal amount of CPU and I/O time. This could be done by running jobs as they arrive and not giving special treatment to any job, regardless of its CPU or I/O processing characteristics, or high or low priority.

As we can see from this list, if the system favors one type of user, then it hurts another, or it doesn't efficiently use its resources. The final decision rests with the system designer or administrator, who must determine which criteria are most important for that specific system. For example, you might want to maximize CPU utilization while minimizing response time and balancing the use of all system components through a mix of I/O- bound and CPU-bound jobs. In order to do this, you would select the scheduling policy that most closely satisfies these goals.

Although the Job Scheduler selects jobs to ensure that the READY and I/O queues remain balanced, there are instances when a job claims the CPU for a very long time before issuing an I/O request. If I/O requests are being satisfied (this is done by an I/O controller and is discussed later), this extensive use of the CPU will build up the READY queue while emptying out the I/O queues, creating an imbalance in the system, which is often perceived as unacceptable.

To solve this problem, the Process Scheduler uses a timing mechanism and periodically interrupts running processes when a predetermined slice of time has expired. When this happens, the scheduler suspends all activity on the job currently running and reschedules it into the READY queue; it will be continued later. The CPU is now allocated to another job that runs until one of three things happen: The timer goes off, the job issues an I/O command, or the job is finished. Then the job moves to the READY queue, the WAIT queue, or the FINISHED queue, respectively.

An I/O request is called a natural wait in multiprogramming environments. It allows the processor to be allocated to another job until the I/O request is completed. For example, if the I/O request is to print out the fourth chapter of a book, it could take several minutes for this request to be satisfied—time that allows other waiting processes access to the CPU

A scheduling strategy that interrupts the processing of a job and transfers the CPU to another job is called a pre-emptive scheduling policy. The alternative, of course, is a non- pre-emptive scheduling policy, which functions without external interrupts (interrupts external to the job). Therefore, once a job captures the processor and begins execution, it remains in the RUNNING state uninterrupted until it issues an I/O request (natural wait) or until it is finished. In the case of an infinite loop, the process can be stopped and moved to the FINISHED queue whether the policy is pre-emptive or non-pre-emptive.

Scheduling Algorithms

The Process Scheduler relies on a scheduling algorithm, based on a specific scheduling policy, in order to allocate the CPU in the best way for moving jobs through the system efficiently. Most systems place an emphasis on fast user-response time.

We refer to these algorithms as process scheduling algorithms, though they are also used to schedule threads. Here are several algorithms that have been used extensively for scheduling both processes and threads.

First-Come, First-Served

First-come, first-served (FCFS) is a non-pre-emptive scheduling algorithm that handles all incoming objects according to their arrival time: The earlier they arrive, the sooner they're served. It's a very simple algorithm to implement because it uses a First In, First Out (FIFO) queue. This algorithm is commonly used for batch systems, but it is un-acceptable for many interactive systems because interactive users expect quick response times.

With FCFS, as a new job enters the system, its PCB is linked to the end of the READY queue, and it is removed from the front of the READY queue after the job no longer needs the processor.

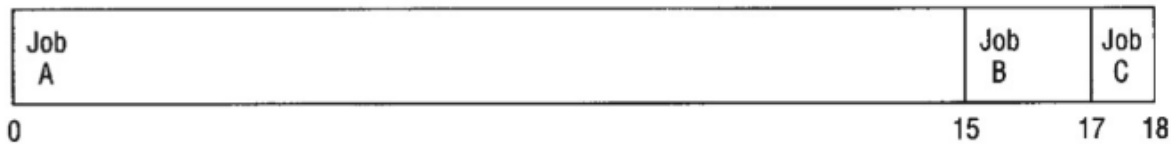
In a strictly FCFS system, there are no WAIT queues because each job is run to its completion, although there may be systems in which control (context) is switched on a natural wait (I/O request), with the job resuming upon I/O completion. Assuming there's no multiprogramming, turnaround time—the time required to execute a job and return the output to the user—is unpredictable with the strict FCFS policy. For example, consider the following three jobs with a run time that includes the jobs' CPU cycles and I/O cycles:

• Job A has a run time of 15 milliseconds (ms).

• Job B has a run time of 2 ms.

• Job C has a run time of 1 ms.

The timeline shown in Figure 4.7 shows an FCFS algorithm with an arrival sequence of A, B, and C.

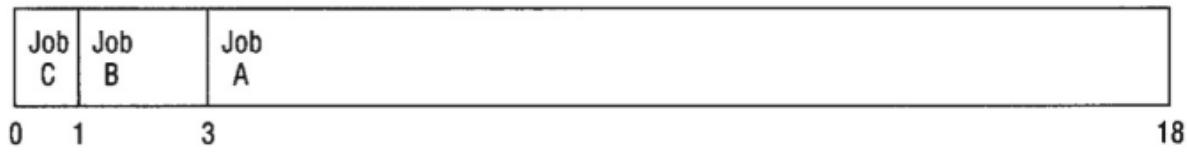


(Figure 4.7)Timeline for job sequence A, B, and C using the non-pre-emptive FCFS algorithm.

If all three jobs arrive in very quick succession (almost simultaneously at Time 0), we can calculate that the turnaround time—each job's finish time minus arrival time—for Job A is 15, for Job B is 17, and for Job C is 18. So the average turnaround time is:

$$\frac{(15-0)+(17-0)+(18-0)}{3} = 16.67$$

However, if the jobs arrived in a different order, say C, B, and A under the same condi- tions, almost simultaneously, then the results, using the same FCFS algorithm, would be very different, as shown in Figure 4.8.



(Figure 4.8) Timeline for job sequence C, B, and A using the same FCFS algorithm.

In this example, the turnaround time for Job A is 18, for Job B is 3, and for Job C is 1, and the average turnaround time, shown here in the order in which they finish, Job C, Job B, and Job A, is:

$$\frac{(1-0)+(3-0)+(18-0)}{3} = 7.3$$

That's quite an overall improvement from the first sequence. Unfortunately, these two examples illustrate the primary disadvantage of using the FCFS algorithm: the average turnaround times vary widely and are seldom minimized. In fact, when there are three jobs in the READY queue, the system has only a 1 in 6 chance of running the jobs in the most advantageous sequence (C, B, and A, versus A, B, and C). With four jobs, the odds fall to 1 in 24, and so on.

If one job monopolizes the system, the extent of its overall effect on system performance depends on the scheduling policy, and whether the job is CPU-bound or I/O-bound. While a job with a long CPU cycle (in this example, Job A) is using the CPU, the other jobs in the system are waiting for processing or finishing their I/O requests (if an I/O controller is used), and joining the READY queue to wait for their turn to use the pro- cessor. If the I/O requests are not being serviced, the I/O queues remain stable while the READY queue grows with new arrivals. In extreme cases, the READY queue could fill to capacity while the I/O queues remained empty, or stable, with the I/O devices sitting idle.

On the other hand, if the job is processing a lengthy I/O cycle, such as printing a book at a very high resolution, the I/O queues quickly build to overflowing, and the CPU would be sitting idle (if an I/O controller were used). This situation is eventually resolved when the I/O-bound job finishes its cycle and the queues start moving again, allowing the system to recover from the bottleneck.

In a strictly FCFS algorithm, neither situation occurs. However, the turnaround time is unpredictable. For this reason, FCFS is a less attractive algorithm than another that would serve the shortest job first, as the next scheduling algorithm does, even in an environment that does not support multiprogramming.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 3 page 118 to 120

3.4 Consequences of Poor Synchronization

In computer systems, a deadlock is a system-wide tangle of resource requests that begins when two or more jobs are put on hold, each waiting for a vital resource to become available. However, if the resources needed by those jobs are the resources held by other jobs, which are also waiting to run but cannot because they're waiting for other unavail- able resources, then the tangled jobs come to a standstill. The deadlock is complete if the remainder of the system comes to a standstill as well. When the situation can't be resolved by the operating system, then intervention is required because resources are being tied up and the entire system—not just a few programs—can be affected. There's no simple and immediate solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances, or everyone behind moves back. Fixing the situation requires outside intervention, and only then can the deadlock be resolved.

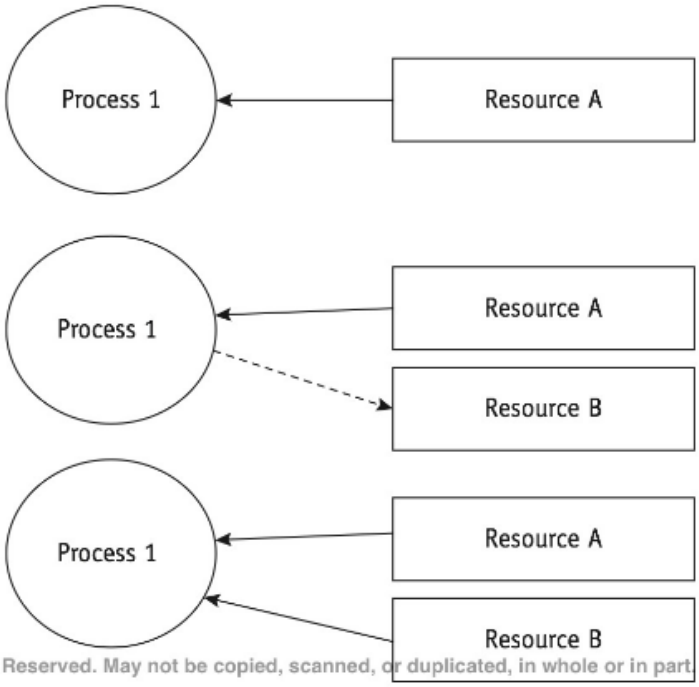
A deadlock is most easily illustrated with an example that we will return to throughout the chapter—a narrow staircase in a building. The staircase was built as a fire escape route, but people working in the building often take the stairs instead of waiting for the slow elevators. Traffic on the staircase moves well unless two people, traveling in opposite directions, need to pass on the stairs—there's room for only one person on each step. There's a landing at each floor that is wide enough for two people to share, but the stairs themselves are not wide enough for more than one person at a time. In this example, the staircase is the system and the steps and landings are the resources. Problems occur when someone going up the stairs meets someone coming down, and each refuses to retreat to a wider place. This creates a deadlock, but there's a type of deadlock in which processes do not come to a standstill.

If two people on a landing try to pass each other but cannot do so because as one steps to the right, the other steps to the left, and they continue moving in sync, back and forth; neither ever moves forward, this is called livelock.

On the other hand, if a few patient people wait on the landing for a break in the oppos- ing traffic, and that break never comes, they could wait there forever. This results in starvation, an extreme case of indefinite postponement.

Modeling Deadlocks with Directed Graphs

In 1972, Richard Holt published a visual tool to show how deadlocks can be mod- eled (illustrated) using directed graphs. Directed graphs visually represent the system's resources and processes, and show how they are deadlocked. Holt's directed graphs use circles to represent processes and squares to represent resources, as shown in Figure 5.2.



(Figure 5.2) Three snapshots show- ing Process 1 requesting and receiving resources. At first, it has been allocated only Resource A (top). Process 1 then requests another resource, Resource B (middle). Finally, after Process 1 is allocated both resources (bottom), it can run to completion.

A solid line with an arrow that runs from a resource to a process (running from a rectangle to a circle), as shown in Figure 5.2, indicates that the process requested a resource and was granted it. When a process requests another resource, that request is indicated by a dashed line running from the process to the resource. When, and if, the request is fulfilled, the dashed line becomes a solid line and the vector is reversed so that it points to the process.

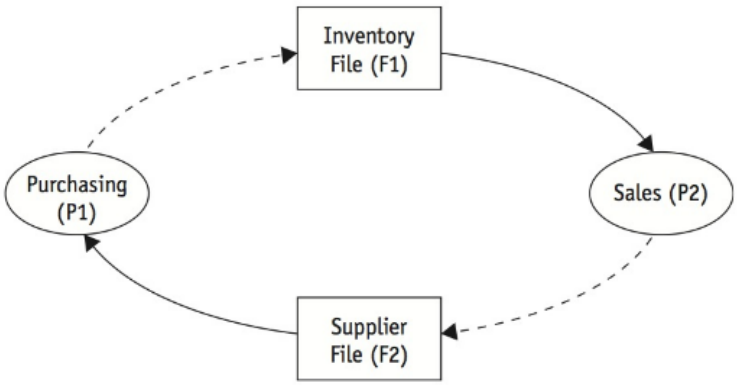
Figure 5.2 (top) shows that the resource has already been allocated to that process; in other words, the process is holding that resource and no other process can use it until the holding process relinquishes it. A dashed line with an arrow going from a process to a resource (a circle to a rectangle), as shown in Figure 5.2 (middle), means that the process has requested a resource and is waiting for it. The directions of the arrows are critical because they indicate flow. If for any reason there is an interruption in the flow, then there is no deadlock. On the other hand, if cyclic flow is shown in the graph, (as illustrated in Figure 5.2 (bottom)), then there exists a deadlock involving the processes and the resources shown in the cycle.

Several Examples of a Deadlock

A deadlock can occur when non-sharable and non-pre-emptable resources, such as printers or scanners, are allocated to jobs that eventually require other non-sharable, non-pre-emptive resources—resources that have been allocated to other jobs. However, deadlocks aren't restricted to printers and scanners. They can also occur on sharable resources that are locked, such as files, disks, and databases.

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a dead- lock can occur, as the simplified directed graph shown in Figure 5.3 illustrates.



(Figure 5.3) Case 1. These two processes, shown as circles, are each waiting for a resource, shown as a rectangle that has already been allocated to the other process. If neither surrenders its resource, a deadlock will result.

For example, consider the case of a home construction company with two application programs, Purchasing and Sales, which are active at the same time. Both need to access two separate files, called Inventory and Suppliers, to read and write transactions. One day the system deadlocks when the following sequence of events takes place:

1. The Purchasing process accesses the Suppliers file to place an order for more lumber.

2. The Sales process accesses the Inventory file to reserve the parts that will be required to build the home ordered that day.

3. The Purchasing process doesn't release the Suppliers file, but it requests the Inventory file so that it can verify the quantity of lumber on hand before plac- ing its order for more. However, Purchasing is blocked because the Inventory file is being held by Sales.

4. Meanwhile, the Sales process doesn't release the Inventory file, because it needs it, and also requests the Suppliers file to check the schedule of a subcontractor. At this point, the Sales process is also blocked because the Suppliers file is already being held by the Purchasing process.

In the meantime, any other programs that require the Inventory or Suppliers files will be put on hold as long as this situation continues. This deadlock will remain until one of the two programs is closed or forcibly removed, and its file is released. Only then can the other program continue and the system return to normal.

Case 2: Deadlocks in Databases

A deadlock can also occur if two processes access and lock records in a database.

To appreciate the following scenario, remember that database queries and transactions are often relatively brief processes that either search or modify parts of a database. Requests usually arrive at random and may be interleaved arbitrarily.

Database locking is a technique used to guarantee the integrity of the data through which the user locks out all other users while working with the database. Locking can be done at three different levels: the entire database can be locked for the duration of the request; a subsection of the database can be locked; or only the individual record can be locked. Locking the entire database, the most extreme and most successful solution, prevents a deadlock from occurring, but it restricts access to the database to one user at a time and, in a multiuser environment, response times are significantly slowed; this is normally an unacceptable solution. When the locking is performed on only one part of the database, access time is improved, but the possibility of a deadlock is increased because different processes sometimes need to work with several parts of the database at the same time.

Here's a system that locks each record in the database when it is accessed and keeps it locked until that process is completed. Let's assume there are two processes, the sales process and the address process, each of which needs to update a different record, the final exam record and the current address record, and the following sequence leads to a deadlock:

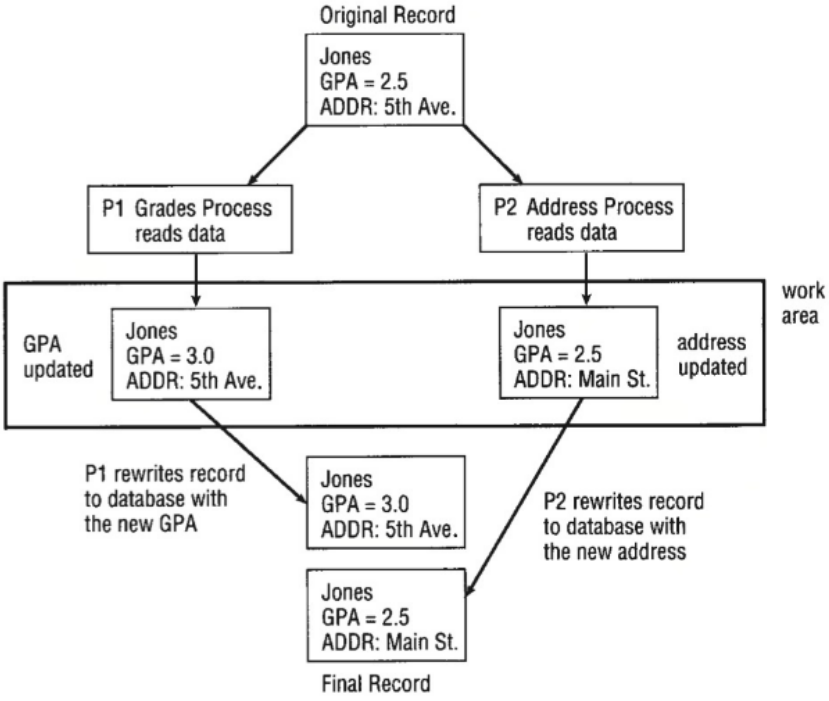
1. The sales process accesses the first quarter record and locks it.

2. The address process accesses the current address record and locks it.

3. The sales process requests the current address record, which is locked by the address process.

4. The address process requests the first quarter record, which is locked by the sales process.

If locks are not used to preserve database integrity, the resulting records in the database might include only some of the data—and their contents would depend on the order in which each process finishes its execution. Known as a race between processes, this is illustrated in the example shown in Figure 5.4. Leaving all database records unlocked is an alternative, but that leads to other difficulties.



(Figure 5.4) Case 2. P1 finishes first and wins the race but its version of the record will soon be overwritten by P2. Regardless of which process wins the race, the final version of the data will be incorrect.

Let's say that you are a student of a university that maintains most of its files on a data- base that can be accessed by several different programs, including one for grades and another listing home addresses. You've just moved at the end of the fall term, therefore, you send the university a change of address form. It happens to be shortly after grades are submitted. One fateful day, both programs race to access a single record in the database:

1. The grades process (P1) is the first to access your college record (R1), and it copies the record to its own work area.
2. Almost simultaneously, the address process (P2) accesses the same record (R1) and copies it to its own work area. Now the same record is in three different places: the database, the work area for grade changes, and the work area for address changes.
3. The grades process changes your college record by entering your grades for the fall term and calculating your new grade average.
4. The address process changes your college record by updating the address field.
5. The grades process finishes its work first and writes its version of your college record back to the database. Now, your record has updated grades, but your address hasn't changed.
6. The address process finishes and rewrites its updated version of your record back to the database. This version has the new address, but it replaced the ver- sion that contains your grades! At this point, according to the database, your new address is recorded, but you have no grades for this term.

If we reverse the order and say that the address process wins the race, your grades will be updated but not your address. Depending on your success in the classroom, you might prefer one mishap over the other, but from the operating system designer's point of view, both alternatives are unacceptable, because both are incorrect and allow the database to become corrupted. A successful operating system can't allow the integrity of the database to depend on luck or a random sequence of events.

Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as two audio recorders, can also deadlock the system. Remember that dedicated devices cannot be shared.

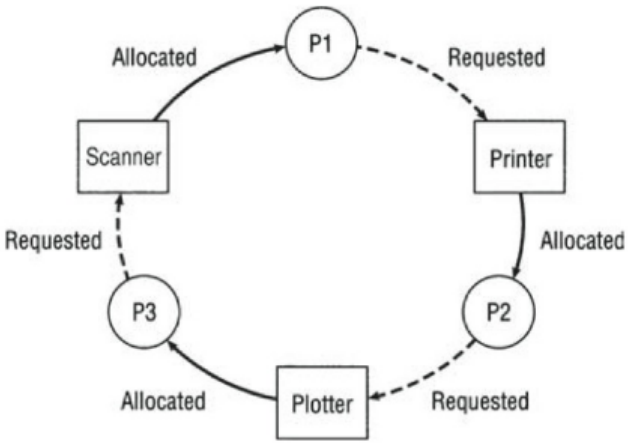
Let's say two administrators, Chris and Jay, from the local board of education are each running evening programs. Chris's Process (P1) and Jay's Process (P2) will each need to use both of the school's projectors (Resource 1 and Resource 2) for two different school board events. Both projectors are available at the time when both administrators make the request; the system's allocation policy states that each projector is to be allocated on an as requested basis. Soon the following sequence transpires:

1. Chris (P1) requests the projector closest to the office, Resource #1, and gets it.
2. Jay (P2) requests the projector closest to the printer, Resource #2, and gets it.
3. Chris (P1) then requests Resource #2, but is blocked and has to wait for Resource #2 (held by Jay) to become available.
4. Jay (P2) requests Resource #1, but is blocked because Chris is holding it and the wait begins here, too.

Neither Chris' nor Jay's process can continue because each is holding one projector while waiting for the other process to finish and release its resource—an event that will never occur according to this allocation policy.

Case 4: Deadlocks in Multiple Device Allocation

Deadlocks aren't restricted to processes that are contending for the same type of device; they can happen when several processes request, and hold on to, several dedicated devices while other processes act in a similar manner, as shown in Figure 5.5.



(Figure 5.5) Case 4. Three processes, shown as circles, are each waiting for a device that has already been allocated to another process, thus creating a deadlock.

Consider the case of an engineering design firm with three programs (P1, P2, and P3) and three dedicated devices: scanner, 3D printer, and plotter. Remember that dedicated resources cannot be shared by multiple processes. The following sequence of events will result in deadlock:

1. Program 1 (P1) requests and gets the only scanner.

2. Program 2 requests and gets the only 3D printer.

3. Program 3 requests and gets the only plotter.

4. Now, Program 1 requests the 3D printer but is put on hold until that resource becomes available.
5. Then, Program 2 requests the plotter but is put on hold until that resource becomes available.

6. Finally, Program 3 requests the scanner but is put on hold until that resource becomes available, thereby, completing the deadlock.

As was the case in the earlier examples, none of these programs can continue because each is waiting for a necessary resource that's not shareable and already being held by another. Still, it's important to note that the remainder of the system can proceed unless, and until, another process needs one of the resources already allocated to a deadlocked process.

Case 5: Deadlocks in Spooling

Although, in the previous example, the 3D printer was a dedicated device, ordinary printers are often shareable devices, and join a category called virtual devices, which uses high-speed storage to transfer data between it and the CPU. The spooler accepts output from several users and acts as a temporary storage area for all output until the printer is ready to accept it. This process is called spooling. However, if the printer needs all of a job's output before it will begin printing, but the spooling system fills the available space with only partially completed output, then a deadlock can occur. It can happen like this.

Let's say it's one hour before the big project is due for an English class. Twenty-five frantic authors submit their final changes and, with only minutes to spare, all issue a print command. The spooler receives the pages one at a time from each of the stu- dents, but the pages are received separately, several page ones, page twos, and so on. The printer is ready to print the first completed document it gets, but as the spooler canvasses its files, it has the first page for many assignments, but the last page for none of them. Alas, the spooler is completely full of partially completed output—no other pages can be accepted—but none of the jobs can be printed (which would release their disk space) because the printer accepts only completed output files. It's an unfortunate state of affairs.

This scenario isn't limited to printers. Any part of the computing system that relies on spooling, such as one that handles incoming jobs or transfers files over a network, is vulnerable to such a deadlock.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 4 page 149

3.5 Necessary Conditions for Deadlock

In each of these seven cases, the deadlock (or livelock) involved the interaction of several processes and resources, but each time, it was preceded by the simultaneous occurrence of four conditions that the operating system (or other systems) could have recognized: mutual exclusion, resource holding, no pre-emption, and circular wait. It's important to remember that each of these four conditions is necessary for the operating system to work smoothly. None of them can be removed easily without causing the system's overall functioning to suffer. Therefore, the system needs to recognize the combination of conditions before they occur.

To illustrate these four conditions, let's revisit the staircase example from the beginning of the chapter to identify the four conditions required for a locked system.

1. When two people meet on the steps between landings, they can't pass because 4 When a deadlock occurs, all four conditions are present, though the opposite is not true—the presence of all four conditions does not always lead to deadlock. the steps can hold only one person at a time. Mutual exclusion, the act of allowing only one person (or process) to have access to a step (or a dedicated resource), is the first condition for deadlock.
2. When two people meet on the stairs and each one holds ground and waits for the other to retreat, this is an example of resource holding (as opposed to resource sharing), which is the second condition for deadlock.
3. In this example, each step is dedicated to the climber (or the descender); it is allocated to the holder for as long as needed. This is called no pre-emption, the lack of temporary reallocation of resources, and is the third condition for deadlock.
4. These three lead to the fourth condition of circular wait, in which each person (or process) involved in the impasse is waiting for another to voluntarily release the step (or resource) so that at least one will be able to continue on and eventually arrive at the destination.

All four conditions are required for the deadlock to occur, and as long as all four conditions are present, the deadlock will continue. However, if one condition is removed, the deadlock will be resolved. In fact, if the four conditions can be prevented from ever occurring at the same time, deadlocks can be prevented. Although this concept may seem obvious, it isn't easy to implement.

3.6 Strategies for Handling Deadlocks

As these examples show, a system's requests and releases can be received in an un-predictable order, which makes it very difficult to design a fool proof preventive policy.

In general, operating systems use some combination of several strategies to deal with deadlocks:

- Prevent one of the four conditions from occurring: prevention.
- Avoid the deadlock if it becomes probable: avoidance.
- Detect the deadlock when it occurs: detection.
- Recover from it gracefully: recovery.

Prevention

To prevent a deadlock, the operating system must eliminate one of the four necessary conditions discussed at the beginning of this chapter, a task complicated by the fact that the same condition can't be eliminated from every resource.

Avoidance

Even if the operating system can't remove one of the conditions for deadlock, it can avoid one if the system knows ahead of time the sequence of requests associated with each of the active processes. As was illustrated in the graphs shown in Figure 5.7 through Figure 5.9, there exists at least one allocation of resource sequences that will allow jobs to continue without becoming deadlocked.

Detection

The directed graphs, presented earlier in this chapter, showed how the existence of a circular wait indicated a deadlock, so it's reasonable to conclude that deadlocks can be detected by building directed resource graphs and looking for cycles. Unlike the avoidance algorithm, which must be performed every time there is a request, the algorithm used to detect circularity can be executed whenever it is appropriate: every hour, once a day, when the operator notices that throughput has deteriorated, or when an angry user complains.

Recovery

Once a deadlock has been detected, it must be untangled and the system returned to normal as quickly as possible. There are several recovery algorithms, but they all have one feature in common: they all require at least one victim, an expendable job, which, when removed from the deadlock, will free the system. Unfortunately for the victim, removal generally requires that the job be restarted from the beginning or from a convenient midpoint.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 5 for more reading

3.7 Introduction to Multi-Core Processors

Multi-core processors have several processors on a single chip. As processors became smaller in size (as predicted by Moore's Law) and faster in processing speed, CPU designers began to use nanometer-sized transistors (one nanometer is one billionth of a meter). Each transistor switches between two positions—0 and 1—as the computer conducts its binary arithmetic at increasingly fast speeds. However, as transistors reach nano-sized dimensions, and the space between transistors become ever closer, the quan- tum physics of electrons get in the way.

In a nutshell, here's the problem. When transistors are placed extremely close together, electrons can spontaneously tunnel, at random, from one transistor to another, causing a tiny but measurable amount of current to leak. The smaller the transistor, the more significant the leak. When an electron does this tunneling, it seems to spontaneously disappear from one transistor and appear in another nearby transistor. (It's as if a Star Trek voyager asked this electron to be "beamed aboard" the second transistor.)

A second problem was the heat generated by the chip. As processors became faster, the heat also climbed and became increasingly difficult to disperse. These heat and tunneling issues threatened to limit the ability of chip designers to make processors any smaller.

One solution was to create a single chip (one piece of silicon) with two "processor cores" in the same amount of space. With this arrangement, two sets of calculations can take place at the same time. The two cores on the chip generate less heat than a single core of the same size, and tunneling is reduced; however, the two cores each run more slowly than the single core chip. Therefore, to get improved performance from a dual-core chip, the software has to be structured to take advantage of the double-calculation capability of the new chip design. Building on their success with two-core chips, designers have created multi-core processors with more than 80 cores on a single chip.

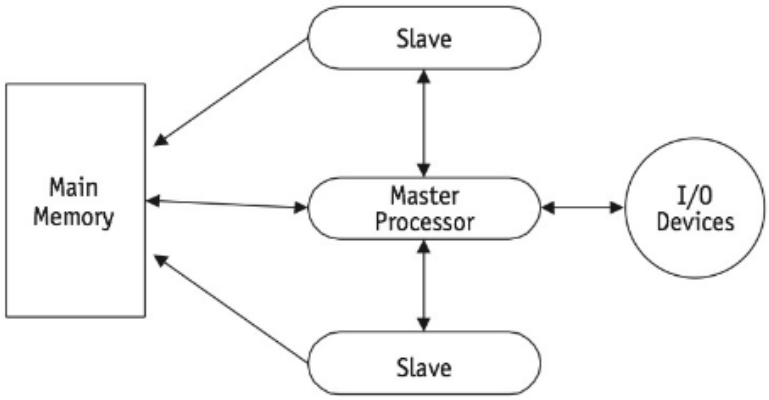
Does this hardware innovation affect the operating system? Yes, because it must manage multiple processors, multiple units of cache and RAM, and the processing of many tasks at once. However, a dual-core chip is not always faster than a single-core chip. It depends on the tasks being performed and whether they're multi-threaded or sequential.

Typical Multiprocessing Configurations

Much depends on how the multiple processors are configured within the system. Three typical configurations are master/slave, loosely coupled, and symmetric.

Master/Slave Configuration

The master/slave configuration is an asymmetric multiprocessing system. Think of it as a single-processor system with additional slave processors, each of which is managed by the primary master processor as shown in Figure 6.1.



(Figure 6.1) In a master/slave mul- tiprocessing configura- tion, slave processors can access main memory directly, but they must send all I/O requests through the master processor.

The master processor is responsible for managing the entire system—all files, devices, memory, and processors. Therefore, it maintains the status of all processes in the system, performs storage management activities, schedules the work for the other processors, and executes all control programs. This configuration is well suited for computing environments in which processing time is divided between front-end and back-end pro- cessors; in these cases, the front-end processor takes care of the interactive users and quick jobs, and the back-end processor takes care of those with long jobs using the batch mode.

The primary advantage of this configuration is its simplicity. However, it has three serious disadvantages:

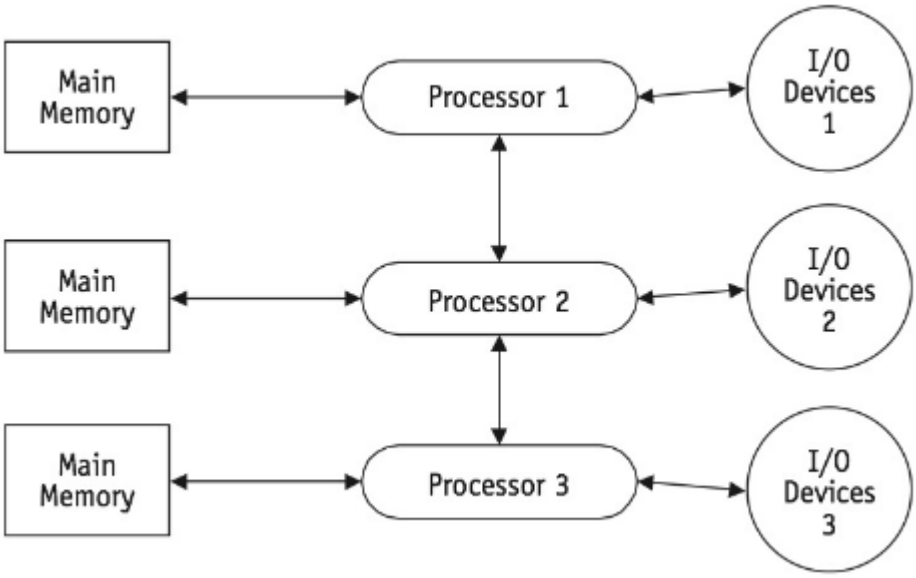
• Its reliability is no higher than it is for a single-processor system because if the mas- ter processor fails, none of the slave processors can take over, and the entire system fails.

• It can lead to poor use of resources because if a slave processor should become free while the master processor is busy, the slave must wait until the master becomes free and can assign more work to it.

• It increases the number of interrupts because all slave processors must interrupt the master processor every time they need operating system intervention, such as for I/O requests. This creates long queues at the master processor level when there are many processors and many interrupts.

Loosely Coupled Configuration

The loosely coupled configuration features several complete computer systems, each with its own memory, I/O devices, CPU, and operating system, as shown in Figure 6.2. This configuration is called loosely coupled because each processor controls its own resources—its own files, access to memory, and its own I/O devices—and this means that each processor maintains its own commands and I/O management tables. The only difference between a loosely coupled multiprocessing system and a collection of independent, single-processing systems is that each processor can communicate and cooperate with the others.



(figure 6.2) In a loosely coupled multi- processing configuration, each processor has its own dedicated resources.

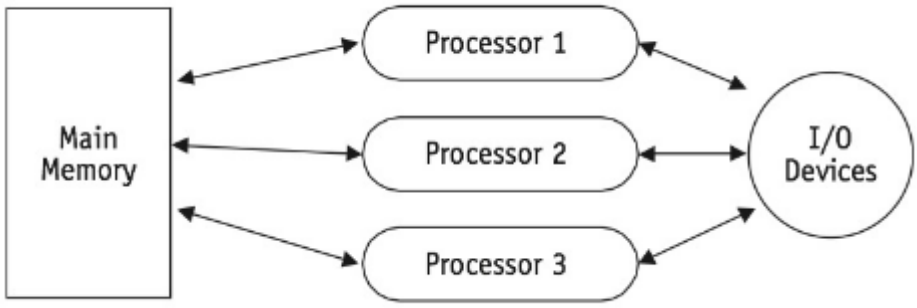
When a job arrives for the first time, it's assigned to one processor. Once allocated, the job remains with the same processor until it's finished. Therefore, each processor must have global tables that indicate where each job has been allocated.

To keep the system well balanced and to ensure the best use of resources, job scheduling is based on several requirements and policies. For example, new jobs might be assigned to the processor with the lightest load, or to the one with the best combination of output devices available.

This system isn't prone to catastrophic system failures because even when a single pro- cessor fails, the others can continue to work independently. However, it can be difficult to detect when a processor has failed.

Symmetric Configuration

A symmetric configuration (as depicted in Figure 6.3) features decentralized processor sched- uling. That is, a single copy of the operating system, and a global table listing each process and its status, is stored in a common area of memory so that every processor has access to it. Each processor uses the same scheduling algorithm to select which process it will run next.



(figure 6.3) In a symmetric, multipro- cessing configuration with homogeneous processors, processes must be care- fully synchronized to avoid deadlocks and starvation.

The symmetric configuration (sometimes called tightly coupled) has four advantages over loosely coupled configuration:

• It's more reliable.

• It uses resources effectively.

• It can balance loads well.

• It can degrade gracefully in the event of a failure.

Whenever a process is interrupted, whether because of an I/O request or another type of interrupt, its processor updates the corresponding entry in the process list and finds another process to run. This means that the processors are kept quite busy. But it also means that any given job or task may be executed by several different processors during its run time. And because each processor has access to all I/O devices, and can reference any storage unit, there are more conflicts as several processors try to access the same resource at the same time.

This presents the obvious need for algorithms to resolve conflicts among processors.

Process Synchronization Software

The success of process synchronization hinges on the capability of the operating system to make a resource unavailable to other processes while it is being used by one of them. These resources can include scanners and other I/O devices, a location in storage, or a data file, to name a few. In essence, the resource that's being used must be locked away from other processes until it is released. Only then is a waiting process allowed to use the resource. This is where synchronization is critical. A mistake could leave a job waiting indefinitely, caus- ing starvation, or, if it's a key resource, cause a deadlock.

It is the same thing that can happen in a crowded ice cream shop. Customers take a number to be served. The number machine on the wall shows the number of the person to be served next, and the clerks push a button to increment the displayed number when they begin attending to a new customer. But what happens when there is no synchro- nization between serving the customers and changing the number? Chaos. This is the case of the missed waiting customer.

Let's say your number is 75. Clerk 1 is waiting on Customer 73 and Clerk 2 is waiting on Customer 74. The sign on the wall says "Now Serving #74," and you're ready with your order. Clerk 2 finishes with Customer 74 and pushes the button so that the sign says "Now Serving #75." But just then, that clerk is called to the telephone and leaves the building due to an emergency, never to return (an interrupt). Meanwhile, Clerk 1 pushes the button and proceeds to wait on Customer 76—and you've missed your turn! If you speak up quickly, you can correct the mistake gracefully; but when it happens in a computer system, the outcome isn't as easily remedied.

Consider the scenario in which Processor 1 and Processor 2 finish with their current jobs at the same time. To run the next job, each processor must:

1. Consult the list of jobs to see which one should be run next.

2. Retrieve the job for execution.

3. Increment the READY list to the next job.

4. Execute it.

Both go to the READY list to select a job. Processor 1 sees that Job 74 is the next job to be run and goes to retrieve it. A moment later, Processor 2 also selects Job 74 and goes to retrieve it. Shortly thereafter, Processor 1, having retrieved Job 74, returns to the READY list and increments it, moving Job 75 to the top. A moment later, Processor 2 returns. It has also retrieved Job 74 and is ready to process it, so it increments the READY list; and now Job 76 is moved to the top and becomes the next job in line to be processed. Job 75 has become the missed waiting customer and will never be processed, while Job 74 is being processed twice—both represent an unacceptable state of affairs.

There are several other places where this problem can occur: memory and page alloca- tion tables, I/O tables, application databases, and any shared resource.

Obviously, this situation calls for synchronization. Several synchronization mechanisms are available to provide cooperation and communication among processes. The common element in all synchronization schemes is to allow a process to finish work on a critical part of the program before other processes have access to it. This is applicable both to multiprocessors and to two or more processes in a single-processor (time-shared) system. It is called a critical region because its execution must be handled as a unit. That is, the processes within a critical region can't be interleaved without threatening the integrity of the operation.

Synchronization is sometimes implemented as a lock-and-key arrangement: Before a process can work on a critical region, it must get the key. And once it has the key, all other processes are locked out until it finishes, unlocks the entry to the critical region, and returns the key so that another process can get the key and begin work.

This sequence consists of two actions: (1) the process must first see if the key is available and (2) if it is available, the process must pick it up and put it in the lock to make it unavailable to all other processes. For this scheme to work, both actions must be per- formed in a single machine cycle; otherwise, it is conceivable that, while the first process is ready to pick up the key, another one would find the key available and prepare to pick up the key first. The result? Each process would block the other from proceeding any further.

Several locking mechanisms have been developed, including test-and-set, WAIT and SIGNAL, and semaphores.

Test-and-Set

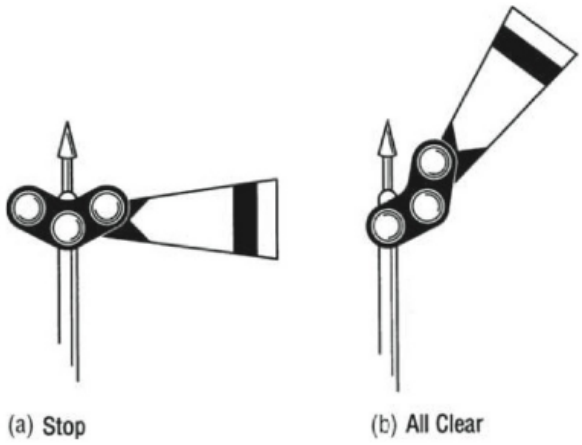
Test-and-set (TS) is a single, indivisible machine instruction that was introduced by IBM for its early multiprocessing computers. In a single machine cycle, it tests to see if the key is available and, if it is, sets it to unavailable.

WAIT and SIGNAL

WAIT and SIGNAL is a modification of test-and-set that's designed to remove busy waiting. Two new operations, WAIT and SIGNAL, are mutually exclusive and become part of the process scheduler's set of operations.

Semaphores

A semaphore is a non-negative integer variable that can be used as a binary signal—a flag. By definition, it can have exactly two positions, no more, no less. One of the most histori- cally significant semaphores was the signaling device, shown in Figure 6.4, used by rail- roads to indicate whether a section of track was clear. When the arm of the semaphore was raised, the track was clear and the train was allowed to proceed. When the arm was lowered, the track was busy and the train had to wait until the arm was raised. It had only two positions: down or up (stop or go). (Today, trains are managed using lights that are also binary, either red or green, reflecting the same two positions: stop or go.)



(Figure 6.4) The semaphore that was once used by railroads indicates whether your train can proceed. When it's lowered (a), another train is approaching and your train must stop to wait for it to pass. If it is raised (b), your train can continue.

In an operating system, a semaphore is set to either zero (off) or one (on) to perform a similar function: It signals if and when a resource is free to be used by a process. Dutch computer scientist, Edsger Dijkstra (1965), introduced two operations to overcome the process synchronization problem we've discussed. Dijkstra called them P and V, and that's how they're known today. The P stands for the Dutch word proberen (to test) and the V stands for verhogen (to increment). The P and V operations do just that: They test a condition and increment it.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 6 page 182 to 186 for more reading

3.8 Concurrent Programming

Until now, we've looked at multiprocessing as several jobs executing at the same time on a single processor (which interacts with I/O processors, for example) or on multiproces- sors. Multiprocessing can also refer to one job using several processors to execute sets of instructions in parallel. The concept isn't new, but it requires a programming language and a computer system that can support this type of construct. This type of system is referred to as a concurrent processing system, and it can generally perform data level parallelism and instruction (or task) level parallelism.

Threads and Concurrent Programming

So far we have considered the cooperation and synchronization of traditional processes, also known as heavyweight processes, which have the following characteristics:

• They pass through several states from their initial entry into the computer system to their completion: ready, running, waiting, delayed, and blocked.

• They require space in main memory where they reside during their execution.

• From time to time they require other resources, such as data.

Two Concurrent Programming Languages

The earliest programming languages did not support the creation of threads or the exis- tence of concurrent processes—work was done in a serial fashion. Typically, they gave programmers the possibility of creating a single process, or thread of control. The Ada programming language was one of the first to do so in the late 1970s.

Ada Language

The U.S. Department of Defense (DoD) first commissioned the creation of the Ada programming language to support concurrent processing and the embedded computer systems found on machines such as jet aircraft, ship controls, and spacecraft, to name just a few. These types of systems typically worked with parallel processors, real-time constraints, fail-safe execution, and nonstandard input and output devices. It took ten years to complete; and Ada was made available to the public in 1980.

Java

Java was introduced as a universal software platform for Internet applications. Java was released in 1995 as the first popular software platform that allowed program- mers to code an application with the capability to run on any computer. This type of universal software platform was an attempt to solve several issues: first, the high cost of developing software applications for each of the many incompatible computer architectures available; second, the needs of distributed client–server environments; and third, the growth of the Internet and the Web, which added more complexity to program development.

Conclusion

Multiprocessing can occur in several configurations: in a single-processor system where interacting processes obtain control of the processor at different times, or in systems with multiple processors, where the work of each processor communicates and cooperates with the others and is synchronized by the Processor Manager. Three multiprocessing hardware systems are described in this chapter: master/slave, loosely coupled, and symmetric.

Chapter Review

To Explore More

To Explore more For additional background on a few of the topics discussed in this chapter, begin a search with these terms.

- Multi-core computers
- Multi-threaded processing
- Detecting starvation
- Deadlock in real-time systems
- Deadlock detection algorithms
- Virtual Memory Principles
- Cache Memory Management

Research Topics

A. Researchtheproblemofstarvationinanetworkedenvironment.Describehow its consequences differ from those of deadlock, and detail a real-life example of the problem that's not mentioned in this chapter. Cite your sources.

B. Research the development of 3D printer technology, and describe, in detail, at least two of the real-life innovative applications for these devices. Then identify the software that is required to integrate such a printer into a secure network, and whether or not these devices can open a security risk to the network. Be sure to clearly identify your opinion versus academic research, and cite your sources.

C. Research the development of 3D printer technology, and describe, in detail, at least two of the real-life innovative applications for these devices. Then identify the software that is required to integrate such a printer into a secure network, and whether or not these devices can open a security risk to the network. Be sure to clearly identify your opinion versus academic research, and cite your sources.

Review Questions

1. Consider a system with 14 dedicated devices of the same type. All jobs currently running on this system require a maximum of five devices to complete their execution, but they each run for long periods of time with just three devices, and request the remaining two only at the very end of the run. Assume that the job stream is endless and that your operating system's device allocation policy is a very conservative one: no job will be started unless all the required drives have been allocated to it for the entire duration of its run.
   a. What is the maximum number of jobs that can be active at once? Explain your answer.
   b. What are the minimum and maximum number of devices that may be idle as a result of this policy? Under what circumstances an additional job would be started?
2. Describe the relationship between a process and a thread in a multi-core system.
3. What is the central goal of most multiprocessing systems?
4. Compare and contrast multiprocessing and concurrent processing. Describe the role of process synchronization for both systems.

Chapter 4: Device Management

LEARNING OUTCOMES

After completing this chapter, you should be able to describe:

- How dedicated, shared, and virtual devices compare
- How blocking and buffering can improve I/O performance
- How seek time, search time, and transfer time are calculated
- How the access times for several types of devices differ
- The strengths and weaknesses of common seek strategies

The Device Manager must manage every peripheral device of the system despite the multitude of input/output (I/O) devices that constantly appear (and disappear) in the marketplace, and the swift rate of change in device technology. To do so, it must main- tain a delicate balance of supply and demand—balancing the system's finite supply of devices with users' almost infinite demand for them.

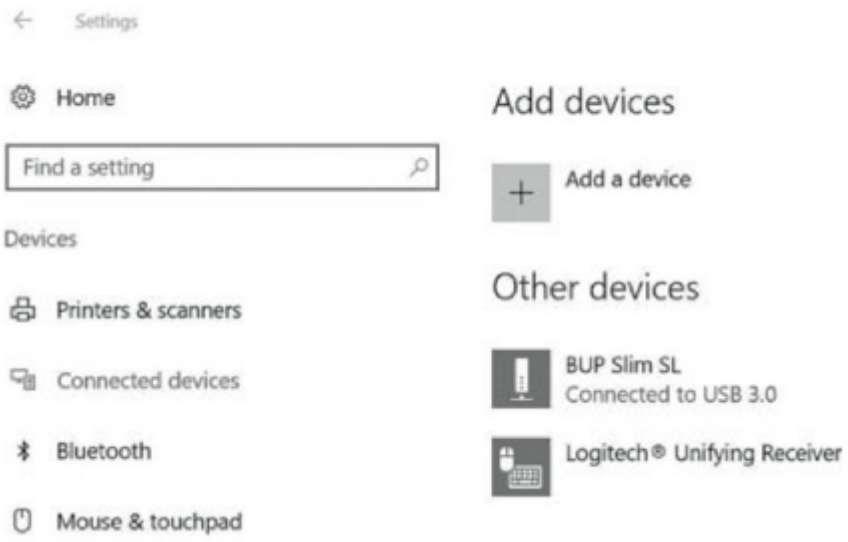
4.1 Device management involves

Looking at four basic functions:

- Monitoring the status of each device, such as storage hardware, monitors, USB tools, and other peripheral devices
- Enforcing preset policies to determine which process will get a device, and for how long
- Allocating each device appropriately
- Deallocating each device at two levels: at the process (or task) level when an I/O command has been executed and has temporarily released its device, and also at the job level when the job is finished and the device is permanently released

Types of Devices

The system's peripheral devices, listed in Figure 7.1, generally fall into one of three categories: dedicated, shared, and virtual. The differences are a function of the charac- teristics of the devices, as well as how they're managed by the Device Manager.



(Figure 7.1) The Windows 10 operating system allows users to manage connected devices through the settings menu.

Dedicated devices are assigned to only one job at a time; they serve that job for the entire time it's active or until it releases them. Some devices, such as monitors, the mouse, and printers, often demand this kind of allocation scheme, because it would be awkward to let several users share them. A shared monitor, for example, might display half of one user's work and half of someone else's game. The disadvantage of having dedicated devices is that they must be allocated to a single user for the duration of a job's execu- tion. This can be quite inefficient, such as with a 3D printer that is exclusively allocated to one person who does not use the device 100 percent of the time.

Shared devices can be allocated to several processes at the same time. For instance, a disk, or any other direct access storage device (DASD), can be shared by several pro- cesses by interleaving their requests, but this interleaving must be carefully controlled by the Device Manager. All conflicts—such as when Process A and Process B need to read from the same disk—must be resolved based on predetermined policies that decide which request is handled first.

Virtual devices are a combination of the first two: They're dedicated devices that have been transformed into shared devices. For example, printers, which are dedicated devices, can be converted into sharable devices by using a spooling program that reroutes all print requests via a storage space on a disk. Only when all of a job's output is complete, and the printer is ready to print out the entire document, is the output sent to the printer for printing. This procedure has to be managed carefully to prevent deadlock, as we explained in Chapter 5. Because disks are sharable devices, this technique can convert one printer into several virtual printers, thus improving both its

performance and use. Spooling is a technique that is often used to speed up slow dedicated I/O devices.

The universal serial bus (USB) controller acts as an interface between the operating system, device drivers, and applications and the devices that are attached via the USB host. One USB host controller (assisted by USB hubs) can accommodate as many as 127 different devices, including flash memory, phones, cameras, scanners, musical key- boards, additional hubs, and so on. Each device is uniquely identified by the USB host controller with an identification number, which allows many devices to exchange data with the computer using the same USB host connection.

The USB controller does this by assigning bandwidth to each device depending on its priority:

• Highest priority is assigned to real-time exchanges where no interruption in the data flow is allowed, such as video or sound data.

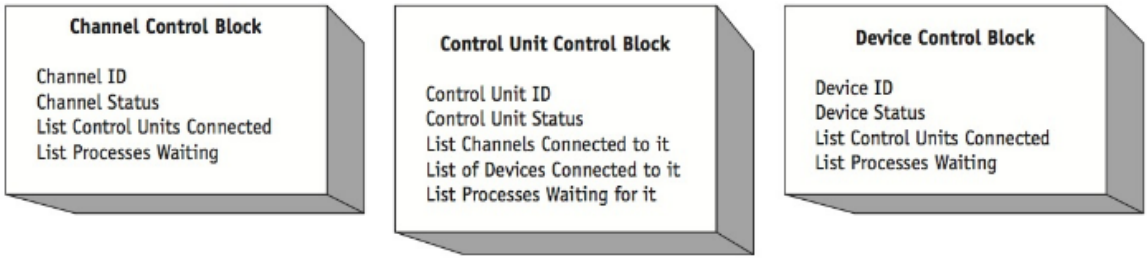
• Medium priority is assigned to devices that can allow occasional interrupts without jeopardizing the use of the device, such as a keyboard or joystick.

• Lowest priority is assigned to bulk transfers or exchanges that can accommodate slower data flow, such as a file update.

4.2 Management of I/O Requests

Although many users may think of an I/O request as an elementary machine action, the Device Manager actually divides the task into three parts with each one handled by a specific software component of the device management subsystem. The I/O traffic controller watches the status of all devices, control units, and channels. The I/O sched- uler implements the policies that determine the allocation of, and access to, the devices, control units, and channels. The I/O device handler performs the actual transfer of data and processes the device interrupts. Let's look at these in more detail.

The I/O traffic controller monitors the status of every device, control unit, and channel. It's a job that becomes more complex as the number of units in the I/O subsystem increases, and as the number of paths between these units increases. The traffic controller has three main tasks to perform for each I/O request: (1) it must determine if there's at least one path to a device of the requested type available; (2) if there's more than one path available, it must determine which to select; and (3) if the paths are all busy, it must determine when one will become available. To do all this, the traffic controller maintains a database containing the status and connections for each unit in the I/O subsystem, grouped into Channel Control Blocks, Control Unit Control Blocks, and Device Control Blocks, as illustrated in Figure 7.2.



(Figure 7.2) Each control block contains the information it needs to manage the channels, control units, and devices in the I/O subsystem.

To choose a free path to satisfy an I/O request, the traffic controller traces backward from the control block of the requested device through the control units to the channels. If a path is not available, a common occurrence under heavy load conditions, the process is linked to the queues kept in the control blocks of the requested device, control unit, and channel. This creates multiple wait queues with one queue per path. Later, when a path becomes available, the traffic controller selects the first PCB from the queue for that path.

The I/O scheduler performs a job analogous to the one performed by the Process Sched- uler described in Chapter 4 on processor management, that is, it allocates the devices, control units, and channels. Under heavy loads, when the number of requests is greater than the number of available paths, the I/O scheduler must decide which request to satisfy first. Many of the criteria and objectives discussed in Chapter 4 also apply here. In many systems, the major difference between I/O scheduling and process scheduling is that I/O requests are not preempted. Once the channel program has started, it's allowed to continue to completion even though I/O requests with higher priorities may have entered the queue. This is feasible because channel programs are relatively short—50 to 100 ms. Other systems subdivide an I/O request into several stages and allow pre- emption of the I/O request at any one of these stages.

Some systems allow the I/O scheduler to give preferential treatment to I/O requests from high-priority programs. In that case, if a process has a high priority, then its I/O requests would also have high priority and would be satisfied before other I/O requests with lower priorities. The I/O scheduler must synchronize its work with the traffic controller to make sure that a path is available to satisfy the selected I/O requests.

The I/O device handler processes the I/O interrupts, handles error conditions, and provides detailed scheduling algorithms, which are extremely device dependent. Each type of I/O device has its own device handler algorithm. Later in this chapter, we explore several algorithms for hard disk drives.

I/O Devices in the Cloud

When a user sends a command from a laptop to access a high-speed hard drive that's many miles away, the laptop's operating system still performs many of the steps outlined in this chapter, and expects an appropriate response from the operating system for the hard drive. Therefore, the concepts explored in this chapter continue to apply even as the cloud allows access to many more devices.

Sequential Access Storage Media

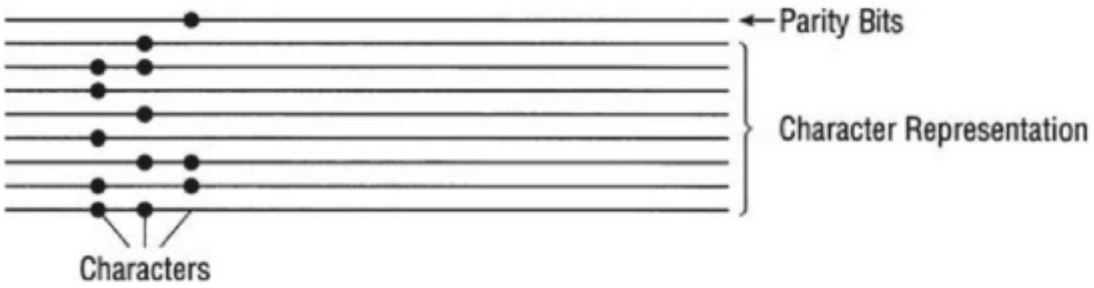
The first secondary storage medium used for electronic digital computers was paper in the form of printouts, punch cards, and paper tape. Magnetic tape followed for routine secondary storage in early computer systems.

A thorough introduction to the details of storage management is to look at the simplest case: a sequential access storage device, which can easily be visualized as an archival magnetic tape that writes and reads records in sequence from the beginning of a reel of tape to the end.

The length of each sequential record is usually determined by the application program, and each record can be identified by its position on the tape. To appreciate just how long this takes, consider a

hypothetical computer system that uses a reel of tape that is 2,400 feet long, see Figure 7.3. Data is recorded on eight of the nine parallel tracks that run the length of the tape. The ninth track, shown at the top of the figure, holds a parity bit that is used for routine error checking.

The number of characters that can be recorded per inch is determined by the density of the tape, such as 1,600 bytes per inch (bpi). For example, if you had records of 160 characters each, and were storing them on a tape with a density of 1,600 bpi, then theoretically you could store 10 records on one inch of tape. However, in actual practice, it would depend on how you decided to store the records: individually or grouped into blocks. If the records were stored individually, each record would need to be separated by a space to indicate its starting and ending places. If the records were stored in blocks, then each block would be preceded by a space and followed by a space, with the individual records stored sequentially within the block.

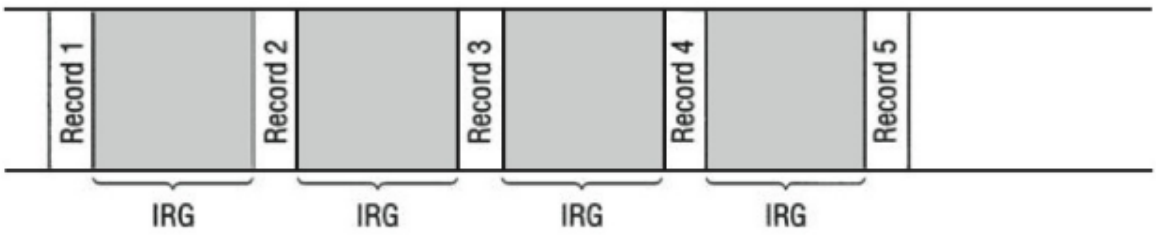


(Figure 7.3) Nine-track magnetic tape with three characters recorded using odd parity. A half-inch wide reel of tape, typically used to back up a mainframe computer, can store thousands of characters, or bytes, per inch.

To appreciate the difference between storing records individually or in blocks, let's look at the mechanics of reading data from a magnetic tape. The tape remains stopped until there's a request to access one or more records. Then the reel of tape starts to rotate and the tape passes under the read/write head. Magnetic tape moves under the read/write head only when there's a need to access a record; at all other times it's standing still.

Records are written in the same way, assuming that there's room on the reel of tape for the data to be written; otherwise a fresh reel of tape would need to be accessed. So the tape moves in jerks as it stops, reads, and moves on at high speed, or stops, writes, and starts again, and so on.

The tape needs time and space to stop, so a gap is inserted between each record. This inter- record gap (IRG) is about 1@2 inch long regardless of the sizes of the records it separates. Therefore, if 10 records are stored individually, there will be nine 1@2-inch IRGs between each record. In Figure 7.4, we assume each record is only 1@10 inch, so 5.5 inches of tape are required to store 1 inch of data—not a very efficient way to use the storage medium.



(Figure 7.4) IRGs on magnetic tape. Each record requires only 1/10 inch of tape. When 10 records are stored individually on magnetic tape, they are separated by IRGs, which adds up to 4.5 inches of tape. This totals 5.5 inches of tape.

An alternative is to group the records into blocks before recording them on tape. This is called

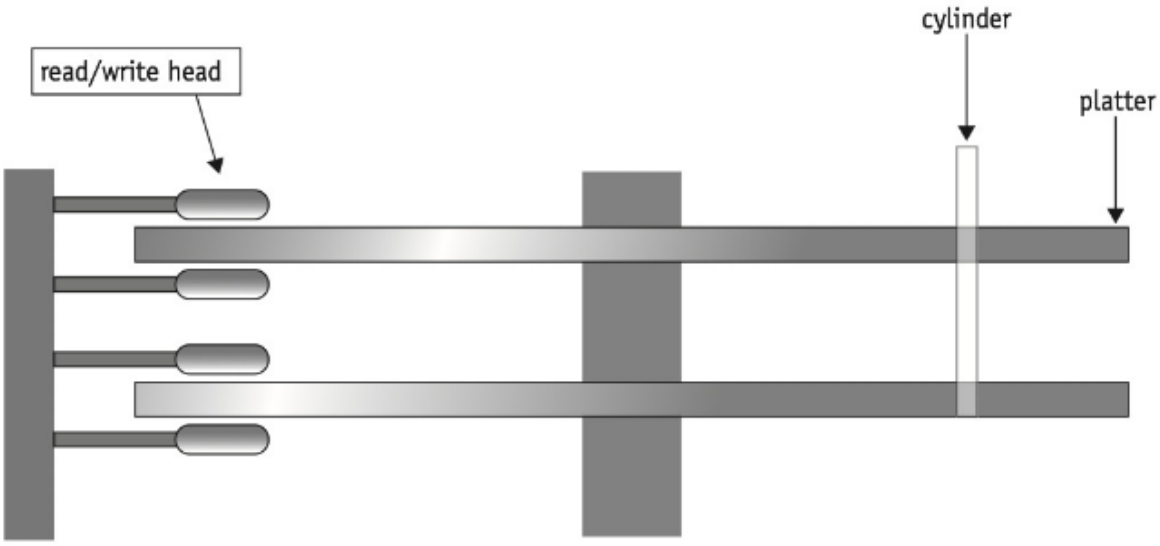
Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 7 page 212 for more reading

blocking, and it's performed when the file is created. Later, when you retrieve them, you must be sure to unblock them accurately.

Magnetic Disk Storage

Magnetic disk drives, such as computer hard drives (and the floppy disk drives of yester- year), usually feature read/write heads that float over each surface of each disk. Disk drives can have a single platter, or a stack of magnetic platters, called a disk pack. Figure 7.6 shows a typical disk pack—several platters stacked on a common central spindle, sepa- rated by enough space to allow the read/write heads to move between each pair of disks.



(Figure 7.6) A disk pack is a stack of magnetic platters. The read/write heads move on each surface, and all of the heads are moved in unison by the arm.

As shown in Figure 7.6, each platter has two surfaces for recording (top and bottom), and each surface is formatted with a specific number of concentric tracks where the data is recorded. The precise number of tracks can vary widely, but typically there are a thousand or more on a high-capacity hard disk. Each track on each surface is numbered with the outermost circle as Track 0 and the highest-numbered track in the center.

The arm, shown in Figure 7.7, moves two read/write heads between each pair of sur- faces: one for the surface above it and one for the surface below, as well as one for the uppermost surface and another for the lowermost surface. The arm moves all of the heads in unison, so if one head is on Track 36, then all of the heads are on Track 36, even if there are many platters—in other words, they're all positioned on the same track but over their respective surfaces, thus, creating a virtual cylinder.

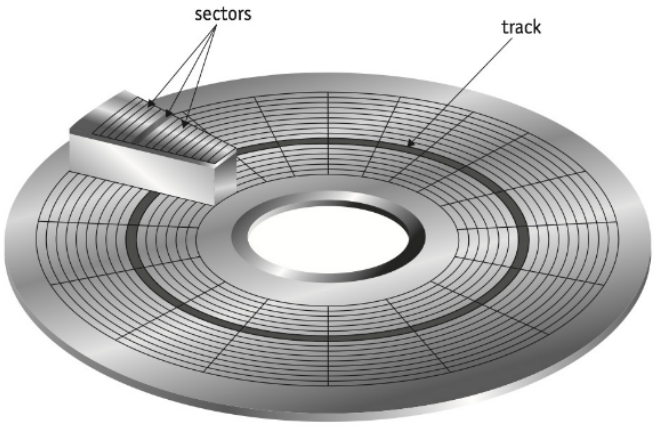
(Figure 7.7) A typical hard drive from a PC showing the arm that floats over the surface of the disk. © Courtesy Seagate Technology

This raises some interesting questions: Is it more efficient to write a series of records on surface one, and then, when that surface is full, to continue writing on surface two, and then on surface three, and so on? Or is it better to fill up every outside track of every surface before moving the heads inward to the next track position to continue writing?

It's slower to fill a disk pack surface-by-surface than it is to fill it up track-by-track—this leads us to a valuable concept. If we fill Track 0 of all of the surfaces, we've got a virtual cylinder of data. There are as many cylinders as there are tracks, and the cylinders are as tall as the disk pack. You could

visualize the cylinders as a series of smaller and smaller soup cans, each nested inside the larger ones.

To access any given record, the system needs three things: its cylinder number, so that the arm can move the read/write heads to it; its surface number, so that the proper read/ write head is activated; and its sector number. A typical disk is shown in Figure 7.8.



(Figure 7.8) On a magnetic disk, the sectors are of different sizes: bigger at the rim and smaller at the cen- ter. The disk spins at a constant angular velocity (CAV) to compensate for this difference.

One clarification: We've used the term surface in this discussion because it makes the concepts easier to understand. However, conventional literature generally uses the term track to identify both the surface and the concentric track. Therefore, our use of surface and track coincides with the term track or head used in some other texts

Access Times

Depending on whether a disk has fixed or movable heads, there can be as many as three factors that contribute to the time required to access a file: seek time, search time, and transfer time.

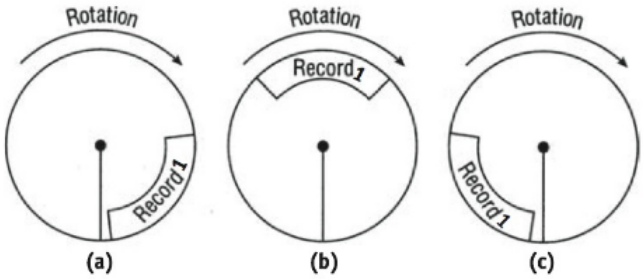
To date, seek time has been the slowest of the three factors. This is the time required to position the read/write head on the proper track. Search time, also known as rotational delay, is the time it takes to rotate the disk until the requested record is moved under the read/write head. Data transfer time is the fastest of the three; that's when the data is actually transferred from secondary storage to main memory (primary storage).

Fixed-Head Magnetic Drives

Fixed-head disk drives are fast because each track has its own read/write head; therefore, seek time doesn't apply. The total amount of time required to access data depends on the rotational speed. Referred to as search time, this varies from device to device, but is constant within each device. The total time also depends on the position of the record relative to the position of the read/write head. Therefore, total access time is the sum of search time plus transfer time.

search time (rotational delay)
+transfer time (data transfer)
access time

Because the disk rotates continuously, there are three basic positions for the requested record in relation to the position of the read/write head. Figure 7.9(a) shows the best possible situation because the record is next to the read/write head when the I/O com- mand is executed; this gives a rotational delay of zero. Figure 7.9(b) shows the average situation because the record is directly opposite the read/write head when the I/O com- mand is executed; this gives a rotational delay of t/2 where t (time) is one full rotation. Figure 7.9(c) shows the worst situation because the record has just rotated past the read/ write head when the I/O command is executed; this gives a rotational delay of t because it will take one full rotation for the record to reposition itself under the read/write head.



(Figure 7.9) As a disk rotates, Record 1 may be near the read/write head, ready to be scanned, as seen in (a); in the farthest position, just past the head, as in (c); or somewhere in between, which is the aver- age case, as in (b).

How long will it take to access a record? If one complete revolution takes 16.8 ms, then the average rotational delay, as shown in Figure 7.9(b), is 8.4 ms. The data-transfer time varies from device to device, but let's say the value is 0.00094 ms per byte—the size of the record dictates this value. These are traditional values. The chapter exercises offer an opportunity to explore current rates.

For example, Table 7.2 shows the resulting access times if it takes 0.094 ms (almost 0.1 ms) to transfer a record with 100 bytes.

Benchmarks	Access Time
Maximum Access Time	16.8 ms + 0.00094 ms/byte
Average Access Time	8.4 ms + 0.00094 ms/byte
Sequential Access Time	Depends on the length of the record (known as the transfer rate)

(Table 7.2)BenchmarksAccess Time16.8 ms 1 0.00094 ms/byte=8.4 ms 1 0.00094 ms/byteDepends on the length of the record (known as the transfer rate)Access times for a fixed- head disk drive at 16.8 ms/ revolution.

Data recorded on fixed head drives may or may not be blocked at the discretion of the application programmer. Blocking isn't used to save space because there are no IRGs between records. Instead, blocking is used to save time.

To illustrate the advantages of blocking the records, let's use the same values shown in Table 7.2 for a record containing 100 bytes and blocks containing 10 records. If we were to read 10 records individually, we would multiply the access time for a single record by 10:

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 7 page 218 up to 226 for more reading

$$\text{access time} = 8.4 + 0.094 = 8.494 \text{ for 1 record}$$
$$\text{total access time} = 10 * (8.4 + 0.094) = 84.940 \text{ for 10 unblocked records}$$

On the other hand, to read one block of 10 records, we would make a single access, so we'd compute the access time only once, multiplying the transfer rate by 10:

$$\text{access time} = 8.4 + (0.094*10)$$
$$\text{total access time} = 8.4 + 0.94$$
$$= 9.34 \text{ for 10 records in 1 block}$$

Once the block is in memory, the software that handles blocking and deblocking takes over. However, the amount of time used in deblocking must be less than what you saved in access time (75.6 ms) for this to be a productive move.

4.3 Communication Among Devices

The Device Manager relies on several auxiliary features to keep running efficiently under the demanding conditions of a busy computer system, and there are three requirements it must fulfil:

- It needs to know which components are busy and which are free.
- It must be able to accommodate the requests that come in during heavy I/O traffic.
- It must accommodate the disparity of speeds between the CPU and the I/O devices.

The first is solved by structuring the interaction between units. The last two problems are handled by queuing requests and buffering records.

As we mentioned previously, each unit in the I/O subsystem can finish its operation independently from the others. For example, after a device has begun writing a record, and before it has completed the task, the connection between the device and its control unit can be cut off so that the control unit can initiate another I/O task with another device. Meanwhile, at the other end of the system, the CPU is free to process data while I/O is being performed, which allows for concurrent processing and I/O.

The success of the operation depends on the system's ability to know when a device has completed an operation. This is done with a hardware flag that must be tested by the CPU. Made up of three bits, the flag resides in the Channel Status Word (CSW), which is in a predefined location in main memory that contains information indicating the status of the channel. Each bit represents one of the components of the I/O subsystem, one each for the channel, control unit, and device. Each bit is changed from 0 to 1 to indicate that the unit has changed from free to busy. Each component has access to the flag, which can be tested before proceeding with the next I/O operation to ensure that the entire path is free. There are two common ways to perform this test—polling and using interrupts.

For example, the CPU periodically tests the channel status bit (in the CSW). If the chan- nel is still busy, the CPU performs some other processing task until the test shows that the channel is free; then the channel performs the I/O operation. The major disadvantage with this scheme is determining how often the flag should be polled. If polling is done too frequently, the CPU wastes time testing the flag just to find out that the channel is still busy. On the other hand, if polling is done too seldom, the channel could sit idle for long periods of time.

The use of interrupts is a more efficient way to test the flag. Instead of the CPU testing the flag, a hardware mechanism does the test as part of every machine instruction exe- cuted by the CPU. If the channel is busy, the flag is set so that execution of the current sequence of instructions is automatically interrupted and control is transferred to the interrupt handler. The interrupt handler is part of the operating system and resides in a predefined location in memory.

The interrupt handler's job is to determine the best course of action based on the cur- rent situation. Because it's not unusual for more than one unit to have caused the I/O interrupt, the interrupt handler must find out which unit sent the signal, analyze its status, restart it when appropriate with the next operation, and, finally, return control to the interrupted process.

Some sophisticated systems are equipped with hardware that can distinguish among several types of interrupts. These interrupts are ordered by priority, and each one can transfer control to a corresponding location in memory. The memory locations are ranked in order, according to the same priorities as the interrupts. Therefore, if the CPU is executing the interrupt-handler routine associated with a given priority, the hardware automatically intercepts all interrupts at the same, or

at lower, priorities. This multiple- priority interrupt system helps improve resource utilization because each interrupt is handled according to its relative importance.

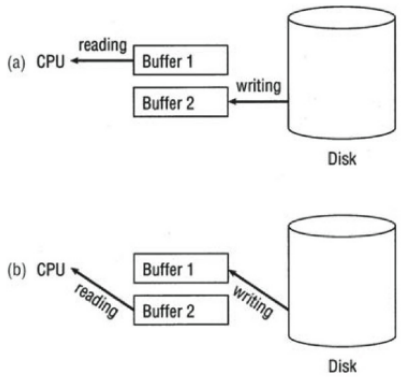
Direct memory access (DMA) is an I/O technique that allows a control unit to directly access main memory. This means that once reading or writing has begun, the remainder of the data can be transferred to and from memory without CPU intervention. However, it is possible that the DMA control unit and the CPU will compete for the system bus if they happen to need it at the same time. To activate this process, the CPU sends enough information—such as the type of operation (read or write), the unit number of the I/O device needed, the location in memory where data is to be read from or written to, and the amount of data (bytes or words) to be transferred—to the DMA control unit to initiate the transfer of data; the CPU then can go on to another task while the control unit completes the transfer independently. The DMA controller sends an interrupt to the CPU to indicate that the operation is completed. This mode of data transfer is used for high-speed devices such as fast secondary storage devices.

Without DMA, the CPU is responsible for the movement of data between main mem- ory and the device—a time-consuming task that results in significant overhead and decreased CPU utilization.

Buffers enable better handling of the movement of data between the relatively slow I/O devices and the very fast CPU. Buffers are temporary storage areas built into convenient locations throughout the system including: main memory, channels, and control units. They're used to store data read from an input device before it's needed by the processor, as well as to store data that will be written to an output device. A typical use of buffers (mentioned earlier in this chapter) occurs when blocked records are either read from or written to an I/O device. In this case, one physical record contains several logical records and must reside in memory while the processing of each individual record takes place.

For example, if a block contains five records, then a physical READ occurs with every six READ commands; all other READ requests are directed to retrieve information from the buffer (this buffer may be set by the application program).

To minimize the idle time for devices and, even more importantly, to maximize their throughput, the technique of double buffering is used, as shown in Figure 7.20. In this system, two buffers are present in the main memory, the channels, and the control units. The objective is to have a record ready to be transferred to or from memory at any time to avoid any possible delay that might be caused by waiting for a buffer to fill up with data. Thus, while one record is being processed by the CPU, another can be read or written by the channel.



(Figure 7.20) Example of double buffer- ing: (a) the CPU is reading from Buffer 1 as Buffer 2 is being filled from the disk; (b) once Buffer 2 is filled, it can be read quickly by the CPU while Buffer 1 is being filled again.

When using blocked records, upon receipt of the command to "READ last logical record," the channel can start reading the next physical record, which results in overlapped I/O and processing. When the first READ command is received, two records are transferred from the device to immediately fill both buffers. Then data transfer begins. Once the data from one buffer has been

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 7 page 238 up to 245 for more reading

processed, the second buffer is already ready. As the second buffer is being read, the first buffer is being filled with data from a third record, and so on.

Conclusion

The Device Manager's job is to manage every system device as effectively as possible despite the unique characteristics of each. The devices can have varying speeds and degrees of shareability; some can handle direct access and some only sequential access. For magnetic media, they can have one or many read/write heads, and the heads can be in a fixed position for optimum speed, or able to move across the surface for opti- mum storage space. For optical media, the Device Manager tracks storage locations and adjusts the disc's speed so that data is recorded and retrieved correctly. For flash memory, the Device Manager tracks every USB device and assures that data is sent and received correctly.

Key Terms

Access times: the total time required to access data in secondary storage.

Blocking: a storage-saving and I/O-saving technique that groups individual records into a block that's stored and retrieved as a unit.

C-LOOK: a scheduling strategy for direct access storage devices that's an optimization of C-SCAN.

C-SCAN: a scheduling strategy for direct access storage devices that's used to optimize seek time. It's an abbreviation for circular-SCAN.

Dedicated devices: a device that can be assigned to only one job at a time; it serves that job for the entire time the job is active.

Chapter Review

To Explore More

For additional background on a few of the topics discussed in this chapter, begin a search with these terms.

- **Hybrid Storage Drive**
- **Circular Buffers**
- **Universal Serial Bus (USB) Technology**

Research Topics

A. To simplify our explanations about magnetic disk access speeds, in this chap- ter, we use rotation speeds of 3,600 rpm and sustained data transfer rates on the order of one megabyte per second. However, current technology has pushed those numbers higher. Research current rotation speeds and data trans- fer rates for magnetic disks. Cite your sources, dates of the published research, and the type of computer (laptop, desktop, and so on) that uses the disk.

Review Questions

1. Name four examples of secondary storage media other than hard disks. 2. Given a typical magnetic hard drive with five platters, answer the following:
   a. How many recording surfaces would you expect it to have?
   b. How many read/write heads would it need to serve all recording surfaces?
   c. How many arms holding read/write heads would it need?
2. Briefly explain the differences between seek time and search time. In your opinion, why do some people confuse the two?
3. Find evidence of the latest technology for optical disc storage and complete the following chart for three optical storage devices. Cite your sources and the dates of their publication.

Chapter 5: File Management

LEARNING OUTCOMES

After completing this chapter, you should be able to describe:

- How files are managed
- How files are named and the role of extensions
- How variable-length record storage differs from fixed-length records
- How several file storage techniques compare
- Comparisons of sequential and direct file access

The File Manager controls every file in the system. In this chapter we learn how files are organized logically, how they're stored physically, how they're accessed, and who is allowed to access them. We'll also study the interaction between the File Manager and the Device Manager.

There are a number of variables that directly affect the efficiency of the File Manager, notably those that are detailed in this chapter including:

- How the system's files are organized (sequential, direct, or indexed sequential);
- How they're stored (contiguously, not contiguously, or indexed);
- How each file's records are structured (fixed-length or variable-length
- How user access to all files is protected (controlled or not controlled).

5.1 The File Manager

The File Manager (the file management system) is the software responsible for creating, deleting, modifying, and controlling access to files, as well as for managing the resources used by the files. The File Manager provides support for libraries of programs and data, doing so in close collaboration with the Device Manager.

The File Manager has a complex job. It's in charge of the system's physical components, its information resources, and the policies used to store and distribute the files. To carry out its responsibilities, it must perform these four tasks:

1. Keep track of where each file is stored.

2. Use a policy that will determine where and how the files will be stored, making sure to efficiently use the available storage space and provide easy access to the files.

3. Allocate each file when a user has been cleared for access to it and then track its use.

4. Deallocate the file when the file is to be returned to storage and communicate its availability to others that may be waiting for it.

For example, the file system is like a library, with the File Manager playing the part of the librarian who performs the same four tasks:

1. A librarian uses the catalog to keep track of each item in the collection; each entry lists the call number, and the details that help patrons find the books they want.

2. The library relies on a policy to store everything in the collection, including oversized books, journals, DVDs, and maps. And they must be physically arranged so people can find what they need.
3. When it's requested, the item is retrieved from its shelf and the borrower's name is noted in the circulation records.

4. When the item is returned, the librarian makes the appropriate notation in the circulation records and puts it back on the shelf.

In a computer system, the File Manager keeps track of its files with directories that contain the filename, its physical location in secondary storage, and important information about it.

The File Manager's policy determines where each file is stored and how the system and its users will be able to access them simply—via commands that are independent from device details. In addition, the policy must determine who will have access to what material, and this involves two factors: flexibility of access to the information and its subsequent protection. The File Manager does this by allowing access to shared files, providing distributed access, and allowing users to browse through public directories. Meanwhile, the operating system must protect its files against system malfunctions and provide security checks via account numbers and passwords to preserve the integrity of the data and safeguard against tampering. These protection techniques are explained later in this chapter.

The computer system allocates a file by activating the appropriate secondary storage device and loading the file into memory while updating its records of who is using what file.

Finally, the File Manager deallocates a file by updating the file tables and rewriting the file (if revised) to the secondary storage device. Any processes waiting to access the file are then notified of its availability.

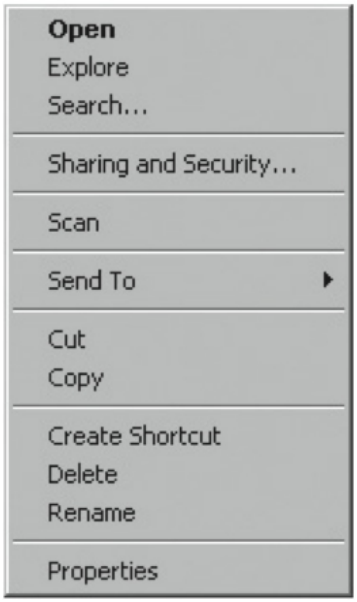
File Management in the Cloud

How does cloud storage of files change the fundamental concepts of file management? The short answer is: it doesn't. Before the cloud existed, the File Manager had to main- tain many files housed on the computer, including those that contained programs, data, and system instructions. Now, with cloud storage, many of these can now be stored on devices located in another part of the world and remain accessible via the Internet. Notable exceptions are the system files, often hidden from the user, that instruct the device on how to power-on and perform boot-up operations.

All of the files stored on the cloud are managed using concepts that are identical to those described in this chapter. That is, our local device's operating system cooperates with the far-off operating system to perform the same services we would expect if the file was stored on our built-in disk drive. In the end, the entire system, with all of its integrated operating systems, must be prepared to store each file in a way that allows all users to readily and accurately store, modify, and retrieve it.

Interacting with the File Manager

The File Manager responds to specific proprietary commands to perform necessary actions. As shown in Figure 8.2, some of the most common commands are OPEN, DELETE, RENAME, and COPY. In addition to these examples, the File Manager extrapolates other system-specific actions from the instructions it receives. For example, although there is no explicit command to create a new file, this is what happens the first time a user gives the command to "save" a new file—one that has not been saved previously. Therefore, this is interpreted as a CREATE command. In some operating.



(Figure 8.2) A typical file menu show- ing the available options for this particular file.

Systems, when a user issues the command NEW (to open a New file), this indicates to the File Manager that a file must be created.

What's involved in creating a new file? That depends on how the writers of the operating system set things up. If the File Manager is required to provide detailed instructions for each system device—how to start it, get it to move to the correct place where the desired record is located, and when to stop—then the program is considered device dependent. More commonly, if this information is already available, usually in a device driver, then the program is considered device independent.

Therefore, to access a file, the user doesn't need to know its exact physical location on the disk pack (such as the cylinder, surface, and sector), the medium in which it's stored (archival tape, magnetic disk, optical disc, or flash storage), or the network specifics. This is fortunate, because file access can be a complex process. Each logical command is broken down into a sequence of signals that trigger the step-by-step actions performed by the device, and supervise the progress of the operation by testing the device's status. For example, when a program issues a command to read a record from a typical hard disk drive, the READ instruction can consist of the following steps:

1. Move the read/write heads to the cylinder or track where the record is to be found.
2. Wait for the rotational delay until the sector containing the desired record passes under the read/write head.
3. Activate the appropriate read/write head, and read the record.
4. Transfer the record into main memory.
5. Set a flag to indicate that the device is now free to satisfy another request. Meanwhile, while all of this is going on, the system must remain vigilant for error conditions that might occur.

Introducing Subdirectories

File Managers create an MFD for each volume that can contain entries for both files and subdirectories. A subdirectory is created when a user opens an account to access the computer system. Although this user directory is treated as a file, its entry in the MFD is flagged to indicate to the File Manager that this file is really a subdirectory and has unique properties—in fact, its records are filenames that point to files.

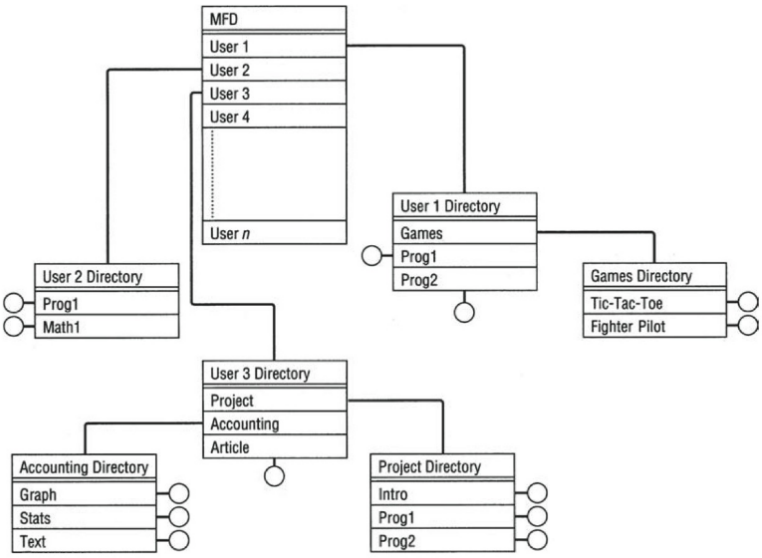
Although this was an improvement from the single directory scheme—now all of the students could name their programs PROGRAM1 so long as it was unique in its subdi- rectory—it didn't solve the problems encountered by prolific users who wanted to group their files in a logical order to improve the accessibility and efficiency of the system.

File Managers now routinely encourage users to create their own subdirectories. Some operating systems call these subdirectories folders or directories. This structure is an extension of the previous two-level directory organization, and it's implemented as an upside-down tree, as shown in Figure 8.4.

Tree structures allow the system to efficiently search individual directories because there are fewer entries in each directory. However, the path to the requested file may lead through several directories. For every file request, the MFD is the point of entry, even though it is usually transparent to the user—it's accessible only by the operating system. When the user wants to access a specific file, the filename is sent to the File Manager. The File Manager first searches the MFD for the user's directory, and it then searches the user's directory and any subdirectories for the requested file and its location. Figure 8.5 shows a typical directory/file structure using a TREE command. By using the CD command (short for change directory) twice, this user pointed to the desired directory, and then used the TREE command to list the structure of the directory. The TREE and CD commands are vestiges of the DOS operating system introduced in 1985, the first operating system released by Microsoft.

Regardless of the complexity of the directory structure, each file entry in every direc- tory contains information describing the file; it's called the file descriptor. Information typically included in a file descriptor includes the following:

• Filename—within a single directory, filenames must be unique; in some operating systems, the filenames are case sensitive
• File type—the organization and usage that are dependent on the system (for exam- ple, files and directories)

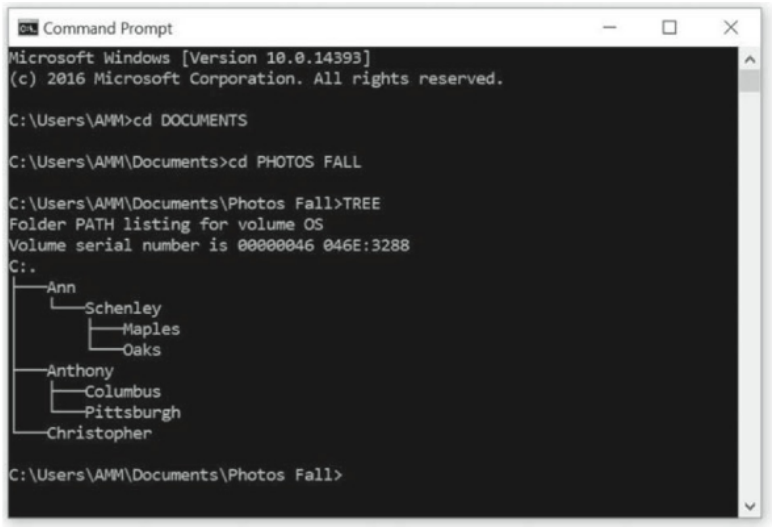


(Figure 8.4) A typical file directory tree structure. The "root" is the MFD shown at the top, each node is a directory file, and each branch is a directory entry pointing to either another directory or to a real file. All program and data files subsequently added to the tree are the leaves, represented by circles.

• File size—although it could be computed from other information, the size is kept here for convenience
• File location—identification of the first physical block (or all blocks) where the file is stored
• Date and time of creation
• Owner
• Protection information—access restrictions based on who is allowed to access the file and what type of access is allowed
• Max record size—its fixed size or its maximum size, depending on the type of record

File-Naming Conventions

A file's name can be much longer than it appears. Depending on the operating system, it can have from two to many components.



(Figure 8.5) By running the command prompt option from a Windows control panel, one can change directo- ries (using the cd com- mand) to the desired folder (here, it is called PHOTOS FALL), and run the TREE command to list all the subfolders and files located in that folder.

To avoid confusion, in the following discussion we'll use the term "complete filename" to identify the file's absolute filename (the long name that includes all path information), and "relative filename" to indicate the name (such as the name assigned by the file's owner) without the path information that appears in directory listings and folders.

The relative filename is the name that differentiates it from other files in the same directory. Examples can include DEPARTMENT ADDRESSES, TAXES_PHOTOG, or AUTOEXEC. Although we often show these in upper case, most operating systems accommodate upper and lower case, and UNIX and Linux file names are most often shown in only lower case characters. Generally, the relative filename can vary in length from one to many characters and can include letters of the alphabet and digits, and some can include special characters. However, every operating system has specific rules that affect the length of the relative name and the types of characters allowed.

For example, the MS-DOS operating system introduced in 1981 and very early versions of Windows required that all file names be a maximum of eight characters followed by a period and a mandatory three-letter extension. Examples of legal names were: BOOK-14.EXE, TIGER.DOC, EXAMPL_3.TXT. This was known as the "eight dot three" (8.3) filename convention and proved to be very limiting for users and admin- istrators alike.

Most operating systems now allow names with dozens of characters, including spaces, exclamation marks, and certain other keyboard characters as well, as shown in Table 8.1. The exact current requirements for your system should be verified.

Some operating systems, such as UNIX, Mac OS X, and Linux, do not require an exten- sion, while other operating systems, including Windows, do require that an extension be appended to the relative filename. This is usually two, three, or four characters in length and is separated from the relative name by a period; its purpose is to identify the type of file required to open it. For example, in a Windows operating system, a typical relative filename for a music file might be BASIA_TUNE.MPG. MPG signifies that this file contains audio data. Similarly, TAKE OUT MENU.RTF and EAT IN MENU.DOC have different file extensions, but both indicate to the File Manager that they can be opened with a word processing application. What happens if an extension is incorrect or unknown? In this case operating systems can ask for guidance from the user to identify the correct application to run the file.

Operating System	Case Sensitive	Special Characters Not Allowed	Extension Required	Maximum Character Length
Android	Yes	* ? $ & [ ] / \	No	126/255*
Linux (Ubuntu)	Yes	* ? $ & [ ] / \	No	256
Mac OS X	Yes	Colon	No	255
MS-DOS	No	Only hyphens and underlines allowed	Yes	8.3
UNIX	Yes	* ? $ & [ ] / \	No	256
Windows	No	Most special characters are not allowed	Yes	255/256

**Officially, Android's maximum length is 255, but developers have noted a 126 limit on some builds of Android, as of this writing.*

(Table 8.1) Typical file name param- eters for several operating systems, listed here in alphabetical order. Note: Specific versions of these operating systems may have different param- eters. See your system for specifics.

Some extensions, such as EXE and DLL, are restricted by certain operating systems because they serve as a signal to the system to use a specific compiler or program to run these files. These operating systems strongly discourage users from choosing these extensions for their files.

Read

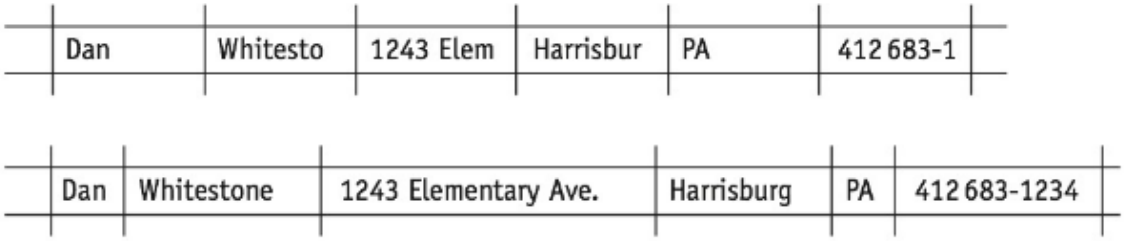
Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 8 page 266 for more reading

File Organization

When we discuss file organization, we are actually talking about the arrangement of records within a file. When a user gives a command to modify the contents of a file, it's actually a command to access records within the file.

Record Format

Files are composed of records. When a user gives a command to modify the contents of a file, it's a command to access records within the file. Within each file, the records are all presumed to have the same format: They can be of fixed length or variable length, as shown in Figure 8.6. These records, regardless of their format, can be accessible individually or grouped into blocks—although data stored in variable-length fields takes longer to access.



(Figure 8.6) Data stored in fixed-length fields (top) that extends beyond the field limit is truncated. Data stored in variable-length fields (bottom) is not truncated.

Fixed-length records are the easiest to access directly. The critical aspect of fixed-length records is the size. If it's too small to hold the entire record, then the data that is leftover is truncated, thereby, corrupting the data. But if the record size is too large—larger than the size of the data to be stored—then storage space is wasted.

Variable-length records don't leave empty storage space and don't truncate any char- acters, thus, eliminating the two disadvantages of fixed-length records. But, while they can easily be read one after the other sequentially, they're not as easy to access directly because it's not easy to calculate exactly where each record is located. This is why they're used most frequently in files that are likely to be accessed sequentially. The record format, how it's blocked, and other related information are kept in the file descriptor.

The amount of space that's actually used to store the supplementary information varies from system to system and conforms to the physical limitations of the storage medium, as we see later in this chapter.

5.2 Physical File Organization

The physical organization of a file has to do with the way records are arranged and the characteristics of the medium used to store it.

On magnetic disks (hard drives), files can be organized in one of several ways including sequential, direct, or indexed sequential. To select the best of these file organizations, the system designer considers these practical characteristics:

• Volatility of the data—the frequency with which additions and deletions are made
• Activity of the file—the percentage of records accessed during a given run
• Size of the file
• Response time—the amount of time the user is willing to wait before the requested operation is completed, which is especially crucial when doing time-sensitive searches

Sequential record organization is by far the easiest to implement because the variable length records are stored and retrieved serially, one after the other. To find a specific record, the File Manager can search the file from its beginning until the requested record is found.

To speed up the process, some optimization features may be built into the system. One is to select a key field from the record and then sort the records by that field before storing them. Later, when a user requests a specific record, the system searches only the key field of each record in the file. The search is ended when either an exact match is found, or the key field for the requested record is smaller than the value of the record last compared. If this happens, the message "record not found" is sent to the user and the search is terminated.

Although this technique aids the search process, it complicates file maintenance because the original order must be preserved every time records are added or deleted. And to preserve the physical order, the file must be completely rewritten or maintained in a sorted fashion every time it's updated.

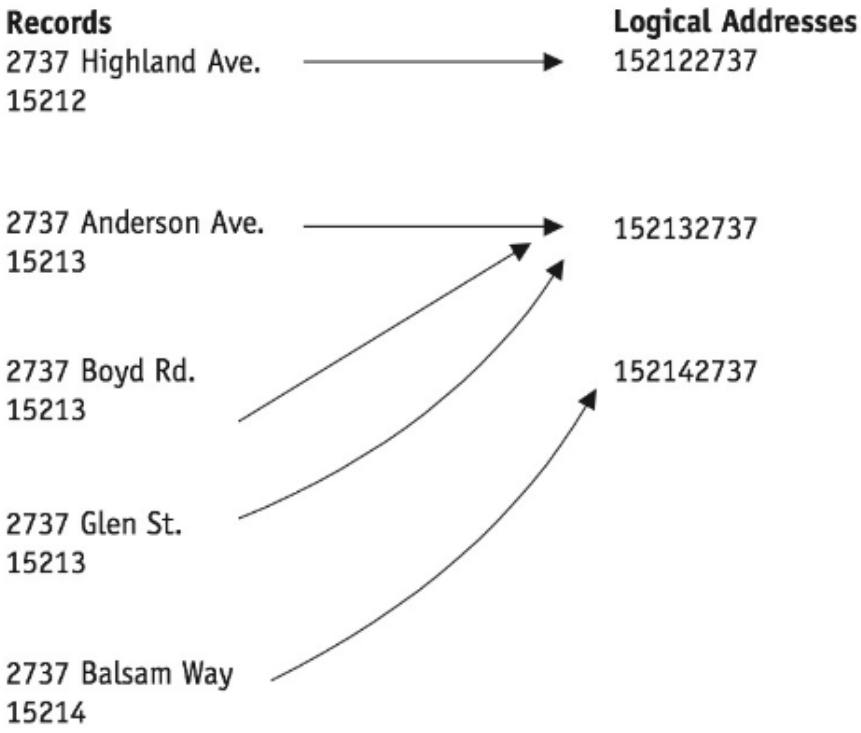
A direct record organization uses direct access files, which, of course, can be implemented only on DASDs—direct access storage devices. These files give users the flexibility of accessing any record in any order without having to begin a search from the beginning of the file to do so. It's also known as random organization, and its files are called random access files.

Records are identified by their relative addresses—their addresses relative to the beginning of the file. These logical addresses are computed when the records are stored, and then again when the records are retrieved.

The method used is quite straightforward. A field (or combination of fields) in the record format is designated as the key field—it's called that because it uniquely identifies each record. The program used to store the data follows a set of instructions, called a **hashing algorithm**, which transforms each key into a number: the record's logical address. This is given to the File Manager, which takes the necessary steps to translate the logical address into a physical address (cylinder, surface, and record numbers), preserving the file organization. The same procedure is used to retrieve a record.

A direct access file can also be accessed sequentially by starting at the first relative address, and going to each record down the line.

The problem with hashing algorithms is that several records with unique keys, such as customer numbers, may generate the same logical address, and then there's a collision, as shown in Figure 8.7. When that happens, the program must generate another logical address before presenting it to the File Manager for storage. Records that collide are stored in an overflow area that was set aside when the file was created. Although the program does all the work of linking the records from the overflow area to their corresponding logical address, the File Manager must handle the physical allocation of space.



(figure 8.7) The hashing algorithm causes a collision. Using a combination of a street address and a postal code, the hashing algorithm generates the same logical address (152132737) for three different records.

The maximum size of the file is established when it's created. If either the file fills to its maximum capacity, or the number of records in the overflow area becomes too large, then retrieval efficiency is lost.

At that time, the file must be reorganized and rewritten, which requires intervention by the File Manager.

Read

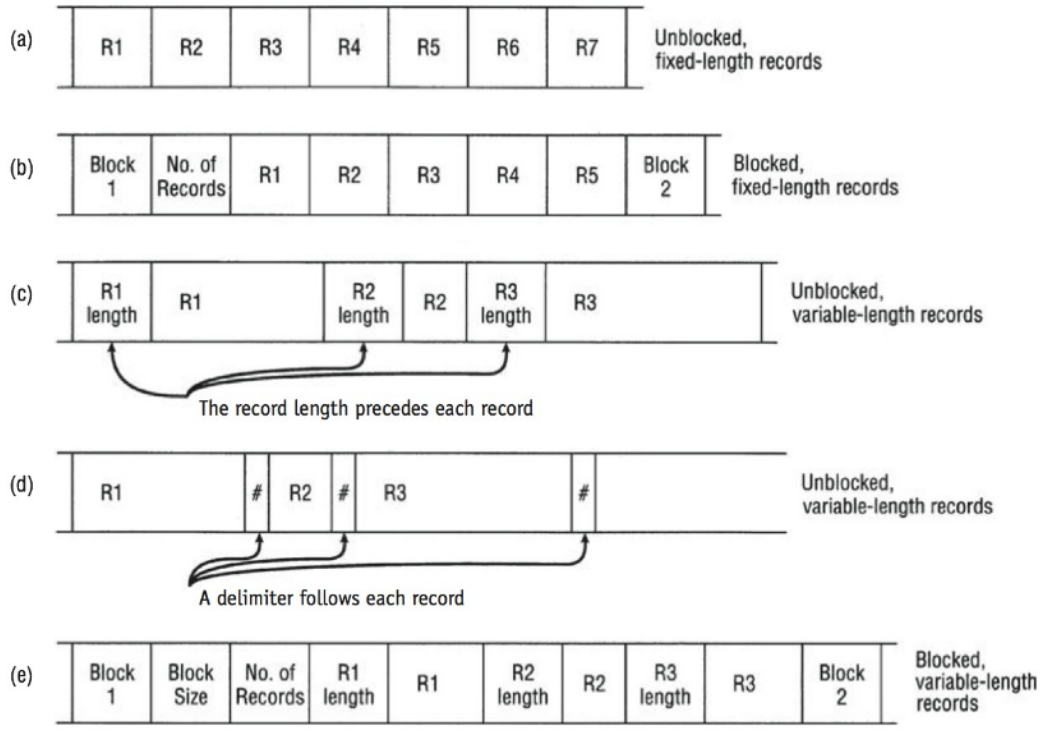
Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 8 page 266-268 for more reading

Physical Storage Allocation

The File Manager must work with files not just as whole units but also as logical units or records. By putting individual records into blocks, some efficiency can be gained. Records within a file must have the same format, but they can vary in length, as shown in Figure 8.8.

In turn, records are subdivided into fields. In most cases, their structure is managed by application programs and not by the operating system. An exception is made for those systems that are heavily oriented toward database applications, where the File Manager handles field structure.

So when we talk about file storage, we're actually referring to record storage. How are the records within a file stored? Looked at this way, the File Manager and Device Manager have to cooperate to ensure successful storage and retrieval of records.



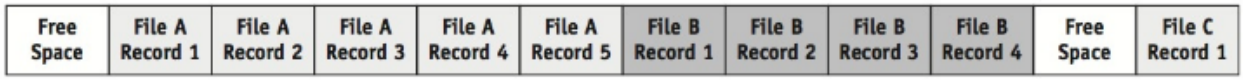
(Figure 8.8) Every record in a file must have the same format but can be of different sizes, as shown in these five examples of the most common record formats. The supplementary information in (b), (c), (d), and (e) is provided by the File Manager, when the record is stored.

5.3 Contiguous Storage

When records use contiguous storage, they're stored one after the other. This was the scheme used in early operating systems. It's very simple to implement and manage. Any record can be found and read, once its starting address and size are known. This makes the directory very streamlined. Its second advantage is ease of direct access because every part of the file is stored in the same compact area.

The primary disadvantage is that a file can't grow, it cannot be expanded, unless there's empty space available immediately following it, as shown in Figure 8.9. Therefore, room for expansion must be provided when the file is first created. If the file runs out of room, the entire file must be recopied to a larger section of the disk every time records are added. The second disadvantage is fragmentation (slivers of unused storage space), which can be overcome by compacting and rearranging files. And, of course, the files can't be accessed while compaction is taking place.

The File Manager keeps track of the empty storage areas by treating them as files— they're entered in the directory but are flagged to differentiate them from real files. Usually the directory is kept in order by sector number, so adjacent empty areas can be combined into one large free space.



(Figure 8.9) With contiguous file storage, File A cannot be expanded without being rewritten in its entirety to a larger storage area. File B can be expanded by only one record, if the free space preceding File C is used.

Non-contiguous Storage

Non-contiguous storage allocation allows files to use any storage space available on the disk. A file's records are stored in a contiguous manner, only if there's enough empty space. Any remaining records, and all other additions to the file, are stored in other sections of the disk. In some systems, these are called the extents of the file and are linked together with pointers. The physical size of each extent is determined by the operating system and is usually 256 bytes, some other power of two bytes.

File extents are usually linked in one of two ways. Linking can take place at the storage level, where each extent points to the next one in the sequence, as shown in Figure 8.10. The directory entry consists of the filename, the storage location of the first extent, the location of the last extent, and the total number of extents not counting the first. Then, by following the pointer from the first extent, the system can locate each of the linked extents until it reaches the last one. Each storage block holds its address, a pointer to the next file block, and the file data.

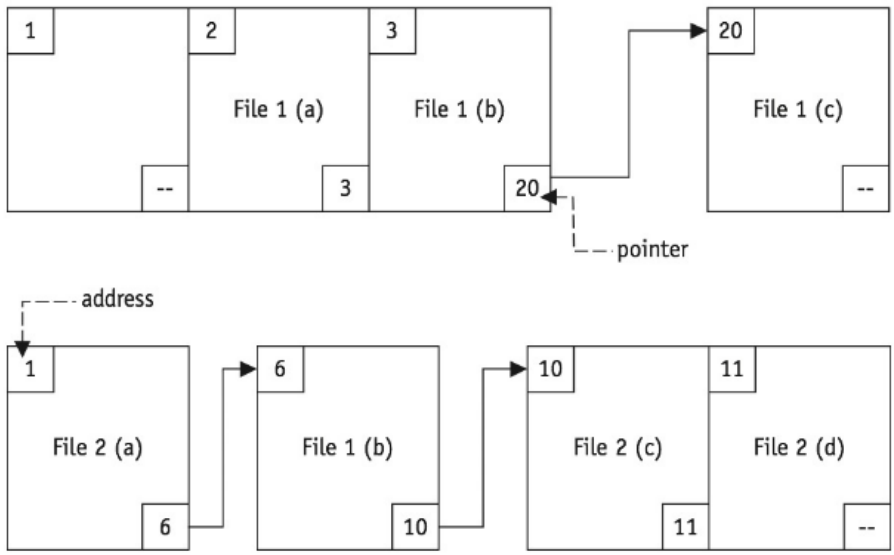
In Figure 8.10, the beginning of File 1 is stored in extent 2, which points to extent 3, which points to extent 20 where the end of the file can be found. Then, File 2 is stored in Extent 1, 6, 10, and 11.

An alternative to file-level linking is to link the file extents at the directory level, as shown in Figure 8.11. Each extent is listed with its physical address, size, and a pointer to the next extent. A null pointer, shown in the last column as a double hyphen (--), indicates that it's the last extent.

Although both non-contiguous allocation schemes eliminate external storage fragmentation and the need for compaction, they don't support direct access because there's no easy way to determine the exact location of a specific record.

Directory

File Number	Start	End	No. Extents
1	2	20	3
2	1	11	4



(Figure 8.10) Directory File Number Start End No. Extents 1 2 20 3 2 1 11 4 non-contiguous file storage with linking taking place at the storage level. The directory lists the file's starting address, ending address, and the number of extents it uses

Files are usually declared to be either sequential or direct when they're created, so that the File Manager can select the most efficient method of storage allocation: contiguous for direct files and non-contiguous for sequential. Operating systems must have the capability to support both storage allocation schemes.

Files can then be converted from one type to another by creating a file of the desired type and copying the contents of the old file into the new file, using a program designed for this specific purpose.

Read

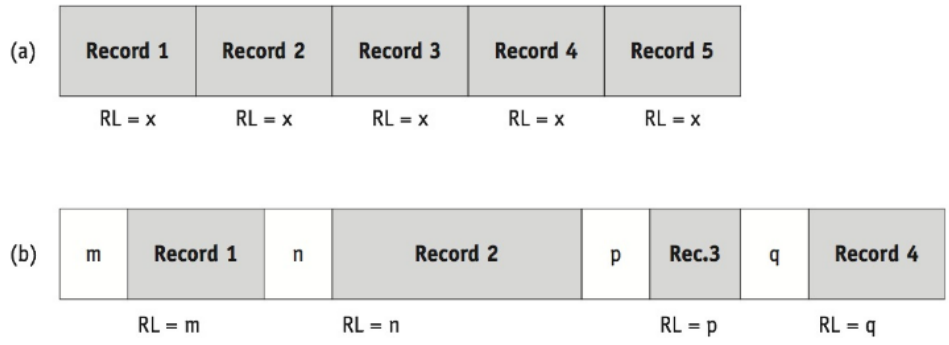
Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 8 page 266-275 for more reading

2. 4 Access Methods

Access methods are dictated by a file's organization; the most flexibility is allowed with indexed sequential files and the least with sequential. This seems to imply that there are few uses for sequential access, but that's not exactly true because it is very well suited to environments that primarily use sequential records.

A file that has been organized in sequential fashion can support sequential access to its records, and these records can be of either fixed or variable length. The File Manager uses the address of the last byte read to access the next sequential record. Therefore, the current byte address (CBA) must be updated every time a record is accessed, such as when the READ command is executed.

Figure 8.13 shows the difference between storage of fixed-length and of variable-length records.



(Figure 8.13) Fixed- versus variable- length records. Fixed- length records (a) each have the same number of bytes, so the record length (RL) variable is constant (x). With variable-length records (b), RL isn't a constant, so record length is recorded on the sequential Access media with each record.

Sequential Access

For sequential access of fixed-length records, the CBA is updated simply by incrementing it by the record length (RL), which is a constant:

Current_Byte_Address = Current_Byte_Address +Record_Length

CBA = CBA + RL

For sequential access of variable-length records, the File Manager adds the length of the record (RL) plus the number of bytes used to hold the record length (N, which holds the value shown in Figure 8.13 as m, n, p, or q) to the CBA.

CBA = CBA + N + RL

Direct Access

If a file is organized in a direct fashion, and if the records are of a fixed length, it can be accessed in either direct or sequential order. In the case of direct access with fixed-length records, the CBA can be computed directly from the record length and the desired record number, RN (information provided through the READ command) minus 1:

$$CBA = (RN - 1)*RL$$

For example, if we're looking for the beginning of the eleventh record, and the fixed record length is 25 bytes, the CBA would be:

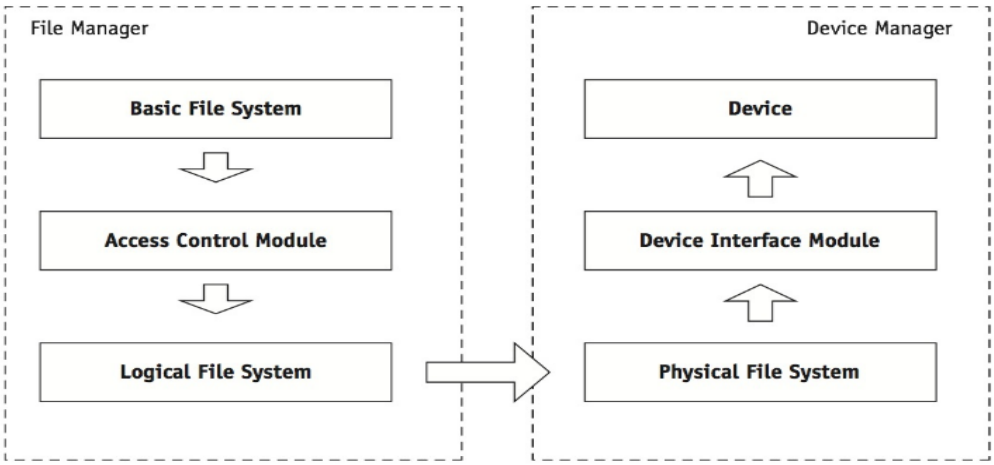
$$(11-2)*25=250$$

However, if the file is organized for direct access with variable-length records, it's virtually impossible to access a record directly because the address of the desired record can't be easily computed. Therefore, to access a record, the File Manager must do a sequential search through the records. In fact, it becomes a half-sequential read through the file because the File Manager could save the address of the last record accessed, and, when the next request arrives, it could search forward from the CBA—if the address of the desired record was between the CBA and the end of the file. Otherwise, the search would start from the beginning of the file. It could be said that this semi-sequential search is only semi-adequate.

Levels in a File Management System

The efficient management of files can't be separated from the efficient management of the devices that house them. This chapter and the previous one on device management present the wide range of functions that have to be organized for an I/O system to perform efficiently. In this section, we outline one of the many hierarchies used to perform these functions.

The highest-level module is called the basic file system. It passes information through the access control verification module to the logical file system, which, in turn, notifies the physical file system, which works with the Device Manager. Figure 8.14 shows the hierarchy.



(Figure 8.14) Typical modules of a file management system showing how communication is sent from the File Manager to the Device Manager.

Each level of the file management system is implemented using structured and modular programming techniques that also set up a hierarchy; that is, the higher positioned modules pass information to the lower modules, so that they, in turn, can perform the required service and continue the communication down the chain to the lowest module. The lowest module communicates with the physical device and interacts with the Device Manager. Only then is the record made available to the user's program.

Each of the modules can be further subdivided into more specific tasks, as we can see when we follow this I/O instruction through the file management system:

READ RECORD NUMBER 7 FROM FILE CLASSES INTO STUDENT

CLASSES is the name of a direct access file previously opened for input, and STUDENT is a data record previously defined within the program, and occupying specific memory locations.

Because the file has already been opened, the file directory has already been searched to verify the existence of CLASSES, and pertinent information about the file has been brought into the operating system's active file table. This information includes its record size, the address of its first physical record, its protection, and access control information, as shown in the UNIX directory listing in Table 8.2.

This information is used by the basic file system, which activates the access control verification module to verify that this user is permitted to perform this operation with this file. If access is allowed, information and control are passed along to the logical file system. If not, a message saying "access denied" is sent to the user.

Using the information passed down by the basic file system, the logical file system trans- forms the record number to its byte address using the familiar formula:

$$CBA = (RN- 1)*RL$$

Access Control	No. of Links	Group	Owner	No. of Bytes	Date	Time	Filename
drwxrwxr-x	2	journal	comp	12820	Jan 10	19:32	ArtWarehouse
drwxrwxr-x	2	journal	comp	12844	Dec 15	09:59	bus_transport
-rwxr-xr-x	1	journal	comp	2705221	Mar 6	11:38	CLASSES
-rwxr--r--	1	journal	comp	12556	Feb 20	18:08	PAYroll
-rwx------	1	journal	comp	8721	Jan 17	07:32	supplier

(table 8.2) A typical list of files on a UNIX system stored in the directory called journal. Notice that the file names are case sensitive and can be differentiated by using upper case, lower case, or a combination of both.

This result, together with the address of the first physical record (and, in the case where records are blocked, the physical block size), is passed down to the physical file system, which computes the location where the desired record physically resides. If there's more than one record in the block, it computes the record's offset within that block using these formulas:

$$\text{block number} = \text{integers}\left[\frac{\text{byte address}}{\text{physical block size}}\right] + \text{address of the first physical record}$$

$$\text{offset} = \text{remainder}\left[\frac{\text{byte address}}{\text{physical block size}}\right]$$

This information is passed on to the device interface module, which, in turn, transforms the block number to the actual cylinder/surface/record combination needed to retrieve the information from the secondary storage device. Once retrieved, the device-scheduling algorithms come into play: The information is placed in a buffer and control returns to the physical file system, which copies the information into the desired memory location. Finally, when the operation is complete, the "all clear" message is passed on to all other modules.

Although we used a READ command for our example, a WRITE command is handled in the same way until the process reaches the device handler. At that point, the allocation module, the portion of the device interface module that handles allocation of free space, is called into play because it's responsible for keeping track of unused areas in each storage device.

We need to note here that verification, the process of making sure that a request is valid, occurs at every level of the file management system. The first verification occurs at the directory level when the file system checks to see if the requested file exists. The second occurs when the access control verification module determines whether access is allowed. The third occurs when the logical file system checks to see if the requested byte address is within the file's limits. Finally, the device interface module checks to see whether the storage device exists.

Therefore, the correct operation of this simple user command requires the coordinated effort of every part of the file management system.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 8 page 276-282 for more reading

5.5 Capability Lists

A capability list shows the access control information from a different perspective. It lists every user and the files to which each has access, as shown in Table 8.6. When users are added or deleted from the system, capability lists are easier to maintain than access control lists.

User	Access
User 1	File 1 (RWED), File 4 (R-E-)
User 2	File 1 (R-E-), File 2 (R-E-), File 3 (RWED)
User 3	File 2 (R-E-)
User 4	File 1 (RWE-), File 2 (--E-), File 3 (--E-)
User 5	File 1 (--E-), File 4 (RWED), File 5 (RWED)

(Table 8.6) A capability list shows files for each user and requires less storage space than an access control matrix.

Of the three schemes described so far, the one most commonly used is the access control list. However, capability lists are gaining in popularity because UNIX-like operating systems (including Linux and Android) control devices by treating them as files.

Although both methods seem to be the same, there are some subtle differences, which is best explained with an analogy. A capability list may be equated to specific concert tickets that are only made available to individuals whose names appear on the list. On the other hand, an access control list can be equated to a reservation list for a restaurant that has limited seating, with each seat assigned to a certain individual.

Data Compression

Data compression algorithms consist of two types: lossless algorithms, which are typi- cally used for text or arithmetic files, and retain all the data in the file throughout the compression-decompression process; and lossy algorithms, which are typically used for image and sound files, that remove data permanently without (we expect) compromis- ing the quality of the file. At first glance, one would think that a loss of data would not be tolerable, but when the deleted data is unwanted noise, tones beyond a human's ability to hear, or parts of the light spectrum that we can't see, deleting this data can be undetectable and perhaps even desirable.

Image and Sound Compression

Lossy compression allows a loss of data from the original image file to allow significant compression. This means the compression process is irreversible, as the original file can- not be reconstructed. The specifics of the compression algorithm are highly dependent on the type of file being compressed. JPEG is a popular option for still images, and MPEG for video images. For video and music files, the International Organization for Standardization (ISO) has issued MPEG standards that "are international standards dealing with the compression, decompression, processing, and coded representation of moving pictures, audio, and their combination."

ISO is the world's leading developer of international standards. For more information about current compression standards and other industry standards, we encourage you to visit www.iso.org.

Chapter Review

The File Manager controls every file in the system and processes user commands (read, write, modify, create, delete, and so on) to interact with available files on the system. It also manages the access control procedures to maintain the integrity and security of files under its control.

To achieve its goals, the File Manager must be able to accommodate a variety of file organizations, physical storage allocation schemes, record types, and access methods. And, as we've seen, this often requires complex file management software.

THINK POINT/CASE STUDY

Research Topics

A. Consult current literature to research file-naming conventions for four different operating systems, not including UNIX, Windows, Linux, or Android, which are discussed in later chapters of this text. Give the acceptable range of characters, maximum length, case sensitivity, and other details. Give examples of both acceptable and unacceptable filenames. For extra credit, explain how the File Managers for those operating systems shorten long filenames (if they do so) in their internal lists to make them easier to manipulate. Cite your sources.

Review Questions

1. Using a computing device, identify four files – each stored in a different directory. For each file, list both the relative filename and its complete filename, showing all paths to that file from the root directory. Identify the device and the software or tool you used to discover the complete filename.
2. Using your own computer, give the name of the operating system that runs it and give examples of five legal file names. Then create examples of five illegal file names, attempt to use them on your system to save a file, and describe what error message you received. Finally, explain how you could fix the file- name to work on your system.
3. As described in this chapter, files can be formatted with fixed-length fields or variable-length fields. In your opinion, would it be feasible to combine both formats for storage on a single magnetic disk? Explain the reasons for your answer.
4. In your opinion, is file deallocation important? Explain your answer and describe how long you believe your own system would perform adequately if this service were no longer available. Discuss the consequences and any alter- natives that users might have.

LEARNING OUTCOMES

After completing this chapter, you should be able to describe:

- **Several network topologies and how they connect hosts**
- **Several types of networks: LAN, MAN, WAN, and wireless LAN**
- **How circuit switching and packet switching compare**
- **How a NOS performs memory, process, device, and file management**
- **How a DO/S performs memory, process, device, and file management**
- **Important features that differentiate between networks and distributed operating systems**
- **How the levels of security that can be implemented**
- **How system threats change as technologies evolve**
- **How computer viruses, worms, and blended threats differ**

When computer resources are connected together by data communication components, they form a network to support the many functions of the organization. Networks provide an essential infrastructure for the members of an information-based society to process, manipulate, and distribute data and information to each other. This chapter introduces the terminology and basic concepts of networks.

6.1 Network Topologies

Sites in any networked system can be physically or logically connected to one another in a certain topology, the geometric arrangement of connections (cables, wireless, or both) that link the nodes. The most common geometric arrangements are star, ring, bus, tree, and hybrid. In each topology, there are trade-offs among the need for fast communications among all sites, the tolerance of failure at a site or communication link, the cost of long communication lines, and the difficulty of connecting one site to a large number of other sites. It's important to note that the physical topology of a network may not reflect its logical topology. For example, a network that is wired in a star configurations can be logically arranged to operate as if it is a ring. That is, it can be made to manipulate a token in a ring-like fashion even though its cables are arranged in a star topology. To keep our explanations in this chapter as simple as possible, whenever we discuss topologies, we assume that the logical structure of the network is identical to the physical structure.

For the network designer, there are many alternatives available, all of which will prob- ably solve the customer's requirements. When deciding which configuration to use, designers should keep in mind four criteria:

• Basic cost—the expense required to link the various sites in the system

• Communications cost—the time required to send a message from one site to another

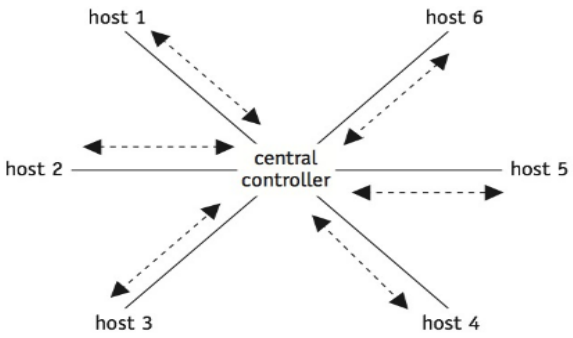
• Reliability—the assurance that many sites can still communicate with each other even if a system link or site fails

• User environment—the critical parameters that the network must meet to be a suc- cessful business investment

It's quite possible that there are several possible solutions for each customer. The best choice would need to consider all four criteria. For example, an international data retrieval company might consider the fastest communications and the most flexible hardware configuration to be a cost-effective investment. But a neighbourhood charity might put the most emphasis on having a low-cost networking solution. Over-engineering the neighbourhood system could be just as big a mistake as under-engineering the international customer's network. The key to choosing the best design is to understand the available technology, as well as the customer's business requirements and budget.

Star

A star topology, sometimes called a hub or centralized topology, is a traditional approach to interconnecting devices in which all transmitted data must pass through a central controller when going from a sender to a receiver. For example, as shown in Figure 9.3, host 1 can communicate with host 5 directly via the central controller.

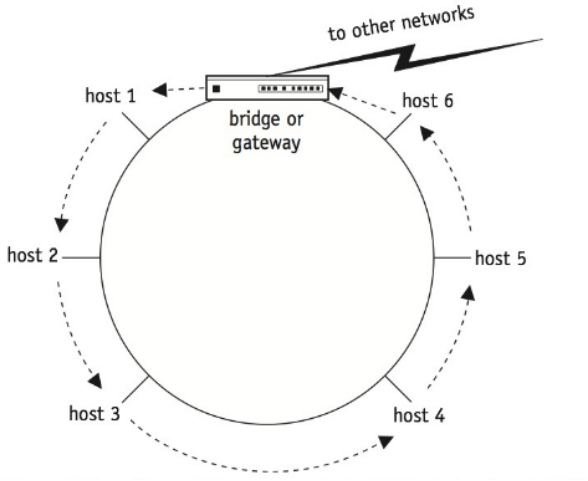


(Figure 9.3) Star topology. Hosts are connected to each other through a central control- ler, which assumes all responsibility for routing messages to the appropriate host. Data flow between the hosts and the central controller is represented by dotted lines. Direct host-to-host communication isn't permitted.

Star topology permits easy routing because the central station knows the path to all other sites. Also, because there is a central control point, access to the network can be controlled easily, and priority status can be given to selected sites. However, this centralization of control requires that the central site be extremely reliable and able to handle all network traffic, no matter how heavy.

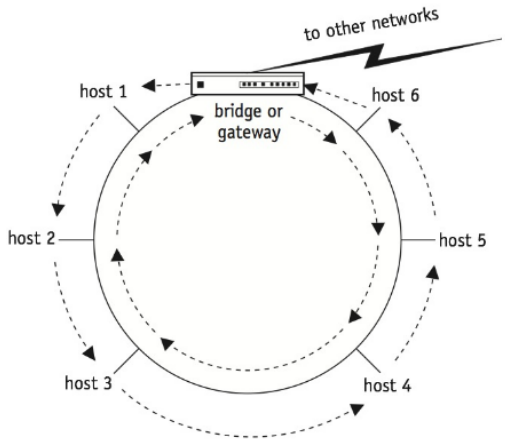
Ring

In the ring topology, all sites are connected to each other in a closed loop, with the first site connected to the last. As shown in Figure 9.4, for example, Host 1 can communicate with Host 5 by way of Hosts 2, 3, and 4. A ring network can connect to other networks via the bridge or gateway, depending on the protocol used by each network. The protocol is the specific set of rules used to control the flow of messages through the network. If the other network has the same protocol, then a bridge is used to connect the networks. If the other network has a different protocol, a gateway is used.



(Figure 9.4) Ring topology. Hosts are connected to each other in a circular fashion with data flowing in one direction only, shown here as dotted lines. The network can be connected to other networks via a bridge or gateway.

Data is transmitted in packets that also contain source and destination address fields. Each packet is passed from node to node in one direction only, and the destination station copies the data into a local buffer. Typically, the packet continues to circulate until it returns to the source station, where it's removed from the ring. There are some variations to this basic topology that provide more flexibility, but at a cost. See the double loop network, shown in Figure 9.5, and a set of multiple rings bridged together, shown in Figure 9.6.



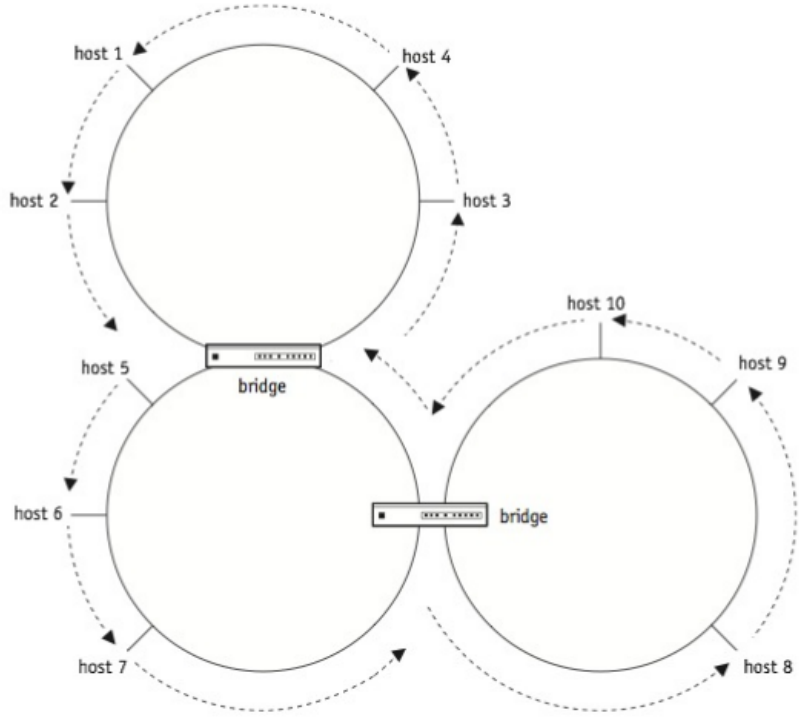
(Figure 9.5) A double loop computer network using a ring topology. Packets of data flow in both directions.

Although ring topologies share the disadvantage that every node must be functional in order for the network to perform properly, rings can be designed that allow failed nodes to be bypassed—a critical consideration for network stability.

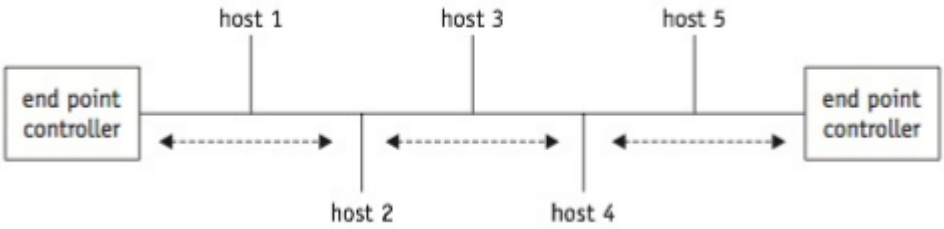
Bus

In the bus topology all sites are connected to a single communication line running the length of the network, as shown in Figure 9.7.

Devices are physically connected by means of cables that run between them, but the cables don't pass through a centralized controller mechanism. Messages from any site circulate in both directions through the entire communication line and can be received by all other sites.



(Figure 9.6) Multiple rings bridged together. Three rings con- nected to each other by two bridges. This variation of ring topology allows several networks with the same protocol to be linked together.

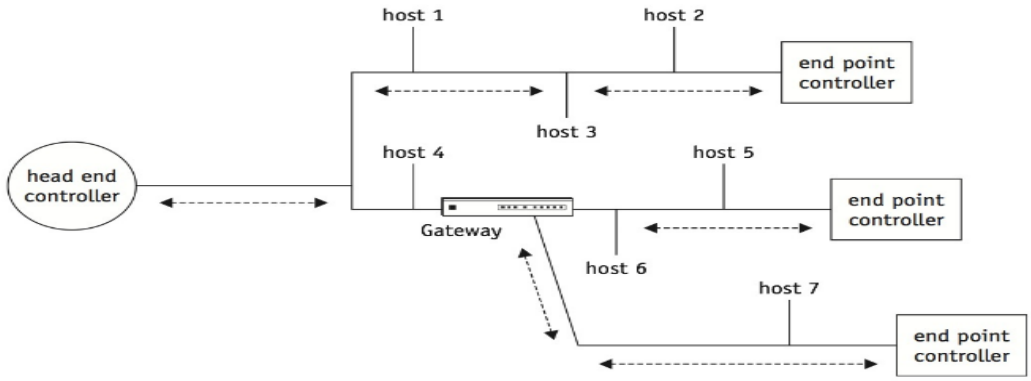


(Figure 9.7)Bus topology. Hosts are connected to one another in a linear fashion. Data flows in both directions from host to host, and is turned around when it reaches an end point controller.

Because all sites share a common communication line, only one of them can successfully send messages at any one time. Therefore, a control mechanism is needed to prevent col- lisions. In this environment, data may pass directly from one device to another, or it may be routed to an end point controller at the end of the line. In a bus, if the data reaches an end point controller without being accepted by a host, the end point controller turns it around and sends it back so that the message can be accepted by the appropriate node as it moves along its way to the other end point controller. With some buses, each mes- sage must always go to the end of the line before going back down the communication line to the node to which it's addressed, but other bus networks allow messages to be sent directly to the target node without reaching an end point controller.

Tree

The tree topology is a collection of buses. The communication line is a branching cable with no closed loops, as shown in Figure 9.8.

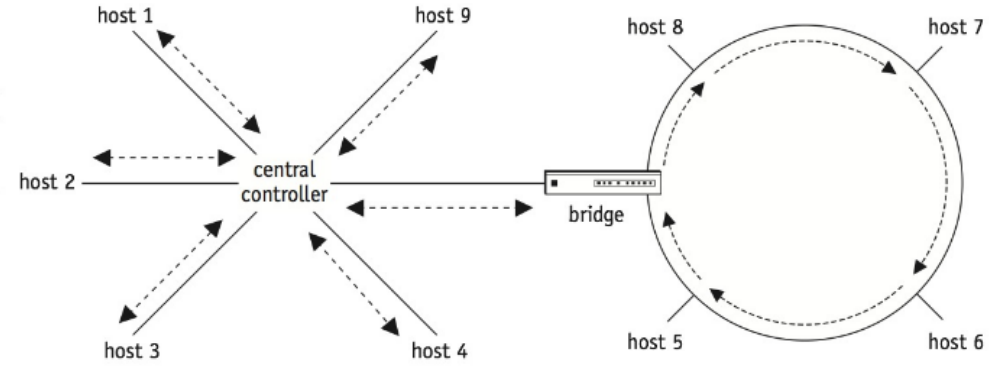


(Figure 9.8) Tree topology. Data flows up and down the branches of the trees and is absorbed by controllers at the end points. Gateways help minimize the differences between the protocol used on one part of the network and a different protocol used on the branch with host 7.

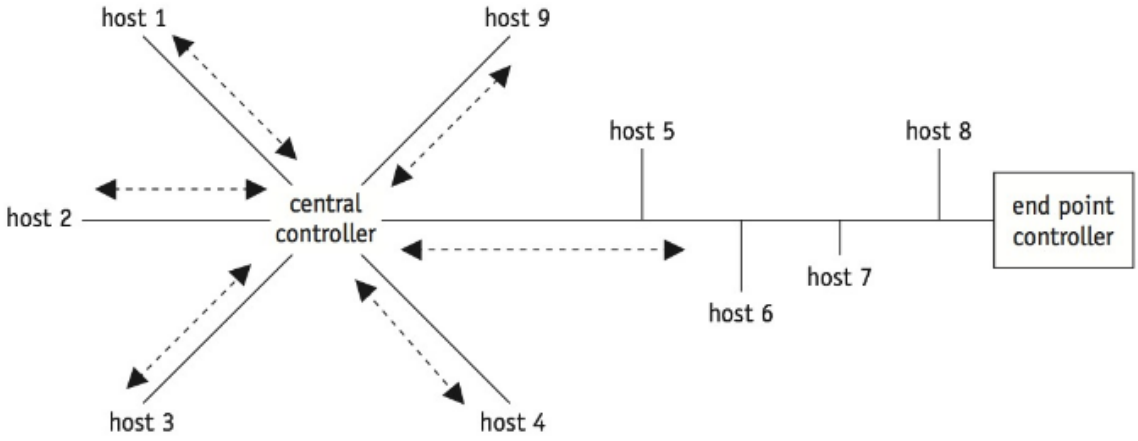
The tree layout begins at the head end—where one or more cables start. Each cable may have branches that may, in turn, have additional branches, potentially resulting in quite complex arrangements. Using bridges as special fitters between buses of the same protocol and as translators to those with different protocols allows designers to create networks that can operate at speeds highly responsive to the hosts in the network. In a tree configuration, a message from any site circulates through the communication line and can be received by all other sites until it reaches the end points. If a message reaches an end point controller without being accepted by a host, the end point controller absorbs it—it isn't turned around as it is when using a bus topology. One advantage of bus and tree topologies is that even if a single node fails, message traffic can still flow through the network.

Hybrid

A hybrid topology is a combination of any of the topologies discussed here. For example, a hybrid can be made by replacing a single host in a star configuration with a ring, as shown in Figure 9.9. Alternatively, a star configuration could have a bus topology as one of the communication lines feeding its hub, as shown in Figure 9.10.



(Figure 9.9) Hybrid topology, version 1. This network combines a star and a ring, connected by a bridge. Hosts 5, 6, 7, and 8 are located on the ring.



(Figure 9.10) Hybrid topology, version 2. This network combines star and bus topologies. Hosts5, 6, 7, and 8 are located on the bus.

The objective of a hybrid configuration is to select among the strong points of each topology, and combine them to meet that system's communication requirements most effectively.

6.2 Network Types

It's often useful to group networks according to the physical distances they cover. Networks are generally divided into local area networks, metropolitan area networks, and wide area networks. However, as communications technology advances, the characteristics that define each group are blurring. In recent years, the wireless local area network has become ubiquitous.

Personal Area Network

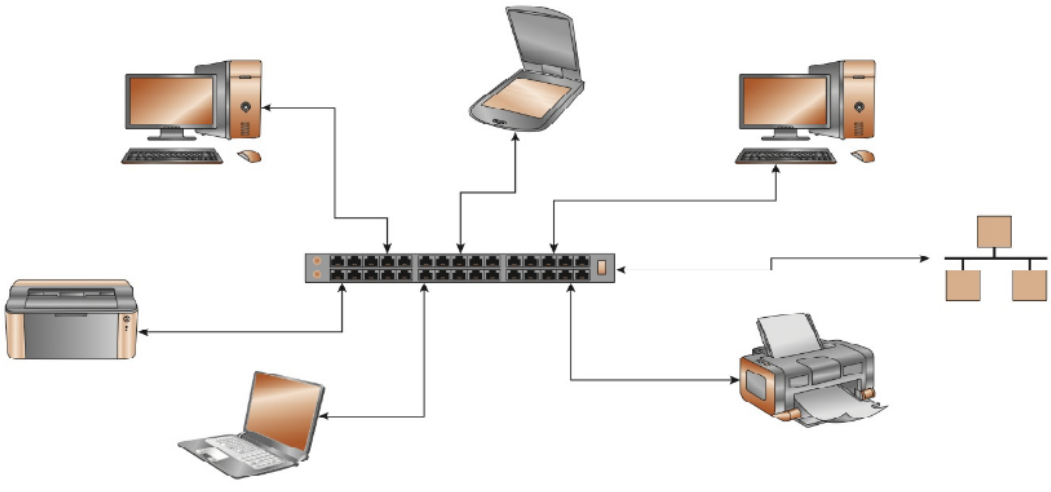
A personal area network (PAN) includes information technology that operates within a radius of approximately 10 meters of an individual, and is centered around that one person. Also called body area networks (BANs), PANs include networks for wearable technology (gloves, caps, monitors, and so on) that use the natural connectivity of the human body to communicate. These small networks can usually connect to a bigger network.

Local Area Network

A local area network (LAN) defines a configuration that includes multiple users found within a single office building, warehouse, campus, or similar computing environment. Such a network is generally owned, used, and operated by a single organization, and allows computers to communicate directly through a common communication line. Typically, it's a cluster of personal computers or workstations located in the same general area. Although a LAN is usually confined to a well-defined local area, its communications aren't limited to that area because the LAN can be a component of a larger communication network and can allow users to have easy access to other networks through a bridge or a gateway.

A bridge is a device that is software operated that connects two or more local area networks that use the same protocols. For example, a simple bridge could be used to connect two local area networks using Ethernet networking technology.

A gateway, on the other hand, is a more complex device and software that connects two or more local area networks or systems using different protocols. A gateway translates one network's protocol into another, resolving hardware and software incompatibilities. For example, the systems network architecture (commonly called SNA) gateway can connect a microcomputer network to a mainframe host.



(Figure 9.11) The primary LAN uses a bridge to link several computers, printers, a scanner, and another LAN together.

High-speed LANs have a data rate that varies from 100 megabits per second to more than 40 gigabits per second. Because the sites are close to each other, bandwidths are available to support very high-speed transmissions for fully animated, full-color graphics and video; digital voice transmissions; and other high data-rate signals. The previously described topologies—star, ring, bus, tree, and hybrid—are normally used to construct local area networks. The transmission medium used may vary from one topology to another. Factors to be considered when selecting a transmission medium are cost, data rate, reliability, number of devices that can be supported, distance between units, and technical limitations.

Metropolitan Area Network

A metropolitan area network (MAN) defines a configuration spanning an area larger than a LAN, ranging from several blocks of buildings to an entire city, but generally not exceeding a circumference of 100 kilometers. In some instances, MANs are owned and operated as public utilities, providing the means for internetworking several LANs.

A MAN is a high-speed network that can be configured as a logical ring. Depending on the protocol used, messages are either transmitted in one direction using only one ring, as illustrated in Figure 9.4, or in both directions using two counter-rotating rings, as illustrated in Figure 9.5. One ring always carries messages in one direction and the other always carries messages in the opposite direction.

Wide Area Network

A wide area network (WAN) defines a configuration that interconnects communication facilities in different parts of the world, or that's operated as part of a public utility. WANs use the communication lines of common carriers, which are government-regulated private companies, such as telephone companies, that already provide homes and offices with communication facilities. WANs use a broad range of communication media. In some cases, the speed of transmission is limited by the capabilities of the communication line. WANs are generally slower than LANs.

The first WAN, called ARPANET, was developed in 1969 by the U.S. Department of Defense group called the Advanced Research Projects Agency (ARPA). ARPANET's successor, the Internet, is the most widely recognized WAN.

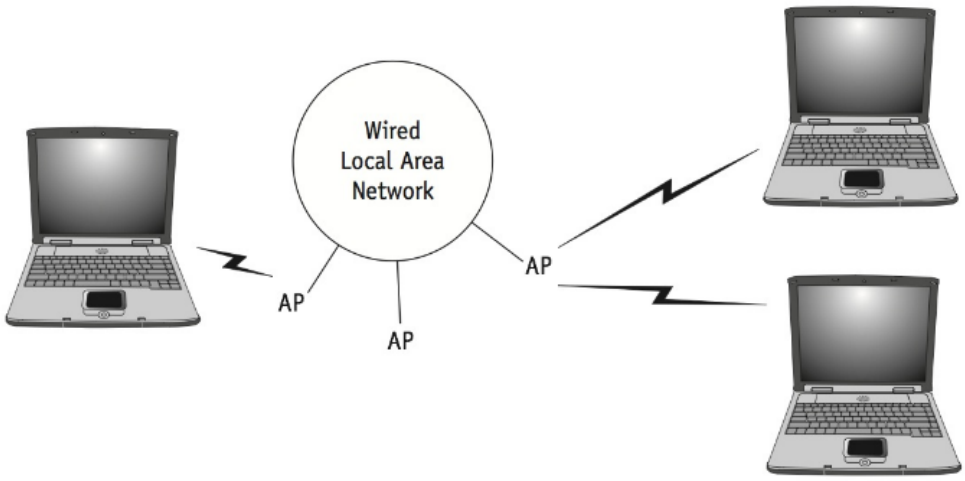
Wireless Local Area Network

A wireless local area network (WLAN) is a local area network that uses wireless technology to connect computers or workstations located within the range of the network. As shown in Table 9.1, the Institute of Electrical and Electronics Engineers (IEEE) has specified several standards for wireless networking, each with different ranges.

IEEE Standard	Net Bit Rate	Frequency	Compatibility
802.11a	54 Mbps	5 GHz	First IEEE wireless standard
802.11b	11 Mbps	2.4 GHz	Not compatible with 802.11g
802.11g	54 Mbps	2.4 GHz	Fully compatible with 802.11b Not compatible with later standards
802.11n	600 Mbps	5 GHz	Fully compatible with 802.11g Not compatible with later standards
802.11ac	1.3+ Gbps	5 GHz	Fully compatible with 802.11n

(Table 9.1) Comparison of IEEE standards for wireless networks (IEEE, 2014).

For wireless nodes (workstations, laptops, and so on), a WLAN can provide easy access to a larger network or the Internet, as shown in Figure 9.12. Keep in mind that a WLAN typically poses security vulnerabilities because of its open architecture, and the constant difficulty of keeping out unauthorized intruders.



(Figure 9.12) In a WLAN, wireless- enabled nodes connect to the cabled LAN via access points (APs), if they are located within the range of the device sending the signal.

The mobile WiMAX standard (802.16), approved by the Institute of Electrical and Electronics Engineers, delivers high-bandwidth data over long distances (IEEE, 2014). This is a fast-changing subject, so we encourage you to research current literature for the latest developments.

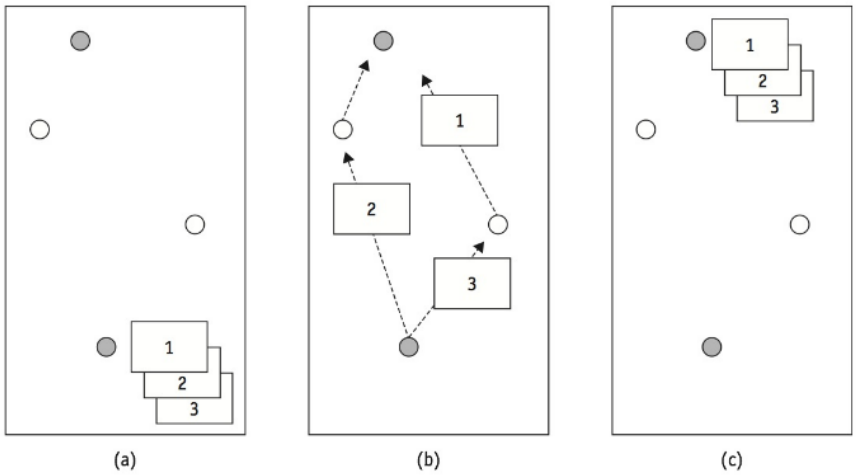
6.3 Circuit Switching

Circuit switching is a communication model in which a dedicated communication path is established between two hosts. The path is a connected sequence of links—the connection between the two points exists until one of them is disconnected. The connection path must be set up before data transmission begins; therefore, if the entire path becomes unavailable, messages can't be transmitted because the circuit would not be complete. The telephone system is a good example of a circuit-switched network.

In terms of performance, there is a delay before signal transfer begins while the connection is set up. However, once the circuit is completed, the network is transparent to users, and information is transmitted at a fixed rate of speed with insignificant delays at intermediate nodes.

Packet Switching

Packet switching is basically a store-and-forward technique in which a message is divided into multiple, equal-sized units called packets, the packets are duplicated and then sent individually through the networks toward their destination, often along different paths. After they arrive, the packets are reassembled into their original order, duplicates are discarded, and the entire message is delivered intact, as illustrated in Figure 9.13.



(Figure 9.13) A packet switching network does not require a dedicated connection. It sends packets using a three- step procedure: (a) divide the data into addressed packets; (b) send each packet toward its destination; (c) and, at the destination, confirm receipt of all packets, place them in order, reassemble the data, and deliver it to the recipient.

Packet switching is an effective technology for long-distance data transmission. Because it permits data transmission between devices that receive or transmit data at different rates, it provides more flexibility than circuit switching. However, there is no guaran- tee that after a message has been divided into packets, the packets will all travel along the same path to their destination, or that they will arrive in their physical sequential order. In addition, packets from one message may be interspersed with those from other messages as they travel toward their destinations. To prevent these problems, a header containing pertinent information about the packet is attached to each packet before it's transmitted. The information contained in the packet header varies according to the routing method used by the network.

The idea is similar to sending a series of 30 reference books through a package delivery system. Six boxes contain five volumes each, and each box is labeled with its sequence number (e.g., box 2 of 6), as well as its ultimate destination. As space on passing delivery trucks becomes available, each box is forwarded to a central switching center, where it's stored until space becomes available to send it to the next switching center, getting closer to its destination each time. Eventually, when all six boxes arrive, they're put in their original order, the 30 volumes are unpacked, and the original sequence is restored.

As shown in Table 9.2, packet switching is fundamentally different from circuit switching.

Circuit Switching	Packet Switching
• Transmits in real time	• Transmits in batches
• Preferred in low-volume networks	• Preferred in high-volume networks
• Reduced line efficiency	• High line efficiency
• Dedicated to a single transmission	• Shared by many transmissions
• Preferred for voice communications	• Not good for voice communications
• Easily overloaded	• Accommodates varying priorities among packets

(Table 9.2) Comparison of circuit and packet switching.

Packet switching provides greater line efficiency because a single node-to-node cir- cuit can be shared by several packets and does not sit idle over long periods of time. Although delivery may be delayed as traffic increases, packets can still be accepted and transmitted.

This is also in contrast to circuit switching networks, which, when they become over- loaded, refuse to accept new connections until the load decreases. If you have ever tried to buy tickets for a popular concert when they first go on sale, then you have experienced an issue similar to a circuit switching network's overload response.

Packet switching allows users to allocate priorities to their messages so that a router with several packets queued for transmission can send the higher priority packets first. In addition, packet switching networks are more reliable than other types because most nodes are connected by more than one link, so that if one circuit should fail, a completely different path may be established between nodes.

6.4 NOS Development

A NOS typically runs on a computer called a server, and the NOS performs services for network computers, called clients. Although computers can assume the role of clients most or all of the time, any given computer can assume the role of server (or client), depending on the requirements of the network. Client and server are not hardware- specific terms but are, instead, role-specific terms.

Many network operating systems are true operating systems that include the four management functions: memory management, process scheduling, file management, and device management (including disk and I/O operations). In addition, they have a net- work management function with a responsibility for network communications, pro- tocols, and so on. In a NOS, the network management functions come into play only when the system needs to use the network. At all other times, the Network Manager is dormant, and the operating system operates as if it's a stand-alone system.

Although a NOS can run applications as well as other operating systems, its focus is on sharing resources instead of running programs. For example, a single-user operating system, such as early versions of Windows, focuses on the user's ability to run applica- tions. On the other hand, network operating systems focus on the user's ability to share resources available on a server, including applications and data, as well as expensive shared resources.

In the following pages we describe some of the features commonly found in a network operating system, without focusing on any one in particular. The best NOS choice depends on many factors, including the applications to be run on the server, the techni- cal support required, the user's level of training, and the compatibility of the hardware with other networking systems.

Important NOS Features

Most network operating systems provide for standard local area network technolo- gies and client desktop operating systems. Most networks are heterogeneous; that is, they support computers running a wide variety of operating systems. A single network might include devices running Windows, the Macintosh operating system (UNIX), and Linux. For a NOS to serve as the networking glue, it must provide strong support for every operating system in the larger information network, sustaining as many current standards as necessary. Therefore, it must have a robust architecture that adapts easily to new technologies.

At a minimum, a NOS should preserve the user's expectations for a desktop system. That means that the network's resources should appear as simple extensions of that user's existing system. For example, on a Windows computer, a network drive should appear as just another hard disk but with a different volume name and a different drive letter. On a Macintosh computer, the network drive should appear as an icon for a volume on the desktop. And on a Linux or UNIX system, the drive should appear as a mountable file system.

A NOS is designed to operate a wide range of third-party software applications and hardware devices, including hard disk drives, optical disc drives, USB devices, and net- work interface cards. A NOS also supports software for multiuser network applications, such as electronic messaging, as well as networking expansions such as new protocol stacks.

Finally, the NOS must blend efficiency with security. Its constant goal is to provide network clients with quick and easy access to the network's data and resources without compromising network security.

Major NOS Functions

An important NOS function is to let users transfer files from one computer to another. In this case, each system controls and manages its own file system. For example, the Internet allows the use of the file transfer protocol (FTP). With FTP, students in a UNIX programming class can copy a data file from a campus computer to their laptops. To do this, each student begins by issuing something like the following command to create the FTP connection:

```
ftp unixs.cis.pitt.edu
```

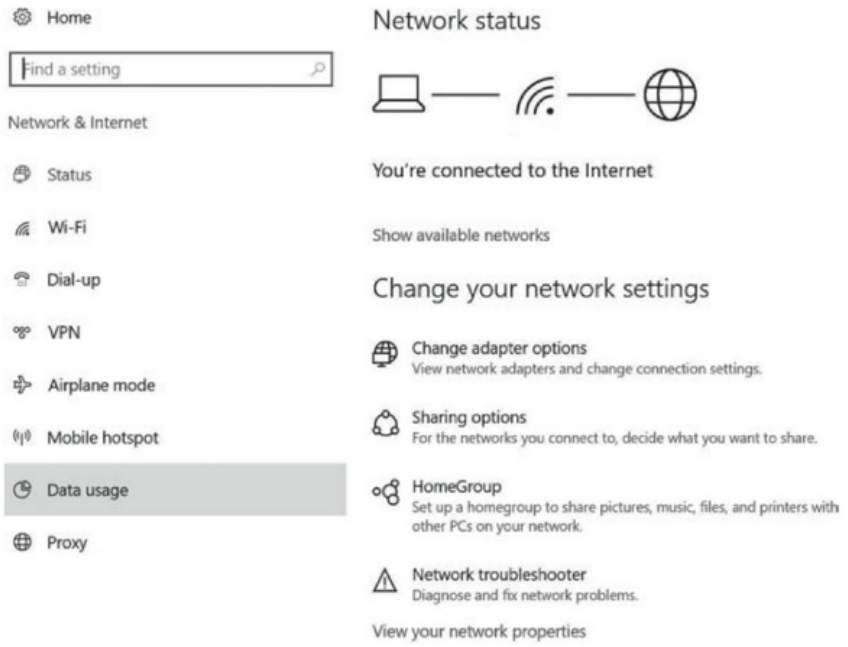
This opens the FTP program, which then asks the student for a login name and pass- word. Once this information has been verified by the UNIX operating system, each student is granted permission to copy the file from the host computer.

```
get filename.ext
```

In this example, filename.ext is the absolute filename and extension of the required data file. That means the user must know exactly where the file is located—in which direc- tory and subdirectory the file is stored. This is because the file location isn't necessarily transparent to the user.

DO/S Development

Because a DO/S manages the entire group of resources within the network in a global fashion, resources are allocated based on negotiation and compromise among equally important peer sites in the distributed system. One advantage of this type of system is its ability to support file copying, electronic mail, and remote printing, without requir- ing the user to install special server software on local machines. See Figure 10.3 for a Windows menu that allows the user to identify valid network connections.



(figure 10.3) Using a Windows 10 oper- ating system, the user can personalize available net- work communications by using the Settings menu.

6.5 Memory Management

For each node, the Memory Manager uses a kernel with a paging algorithm to track the amount of memory that's available. The algorithm is based on the goals of the local system, but the policies and mechanisms that are used at the local sites are driven by the requirements of the global system. To accommodate both local and global needs, mem- ory allocation and deallocation depend on scheduling and resource-sharing schemes that optimize the resources of the entire network.

The Memory Manager for a network works the same way as it does for a stand-alone operating system, but it's extended to accept requests for memory from both local and global sources. On a local level, the Memory Manager allocates pages based on the local policy. On a global level, it receives requests from the Process Manager to provide memory to new, or expanding, client or server processes. The Memory Manager also uses local resources to perform garbage collection in memory, perform compaction, decide which are the most and least active processes, and determine which processes to preempt to provide space for others.

To control the demand, the Memory Manager handles requests from the Process Manager to allocate and deallocate space based on the network's usage patterns. In a distributed environment, the combined memory for the entire network is made up of several subpools (one for each processor), and the Memory Manager has a subcompo- nent that exists on each processor.

When an application tries to access a page that's not in memory, a page fault occurs, and the Memory Manager automatically brings that page into memory. If the page is changed while in memory, the Memory Manager writes the changed page back to the file when it's time to swap the page out of memory.

Before allocating space, the Memory Manager examines the total free memory table. If the request can be filled, the memory is allocated and the table is modified to show the location of the allocated space.

The Memory Manager also manages virtual memory. Specifically, it allocates and deal- locates virtual memory, reads and writes to virtual memory, swaps virtual pages to disk, gets information about virtual pages, locks virtual pages in memory, and protects the pages that need to be protected.

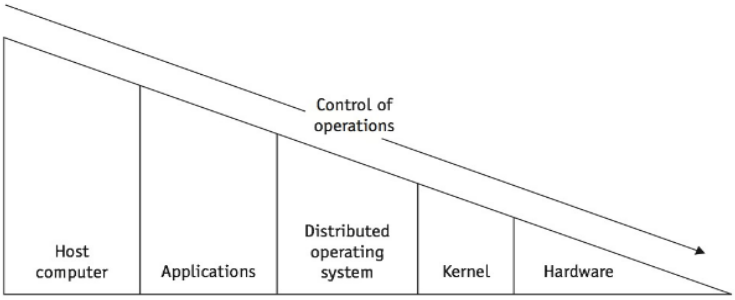
Pages are protected using hardware or low-level memory management software in each site's kernel. This protection is summoned as pages are loaded into memory. Several typical protection checks are performed on the pages, as shown in Table 10.2.

Type of Access Allowed	Levels of Protection Granted
Read and write	Allows users to have full access to the page's contents, giving them the ability to read and write.
Read-only	Allows users to read the page, but they're not allowed to modify it.
Execute-only	Allows users to use the page, but they're not allowed to read or modify it. This means that although a user's process can't read or write to the page, it can jump to an address within the page and start execut- ing. This is appropriate for shared application software, editors, and compilers.
No access	Prevents users from gaining access to the page. This is typically used by debugging or virus protection software to prevent a process from reading from or writing to a certain page.

(Table 10.2) Typical protection checks are performed on pages, as they're loaded into memory. The last three controls shown in this table are needed to make sure processes don't write to pages that should be read-only.

Process Management

In a network, the Processor Manager provides the policies and mechanisms to create, delete, abort, name, rename, find, schedule, block, run, and synchronize processes; it also provides real-time priority execution if required. The Processor Manager also manages the states of execution: READY, RUNNING, and WAIT. To do this, each CPU in the network is required to have its own run-time kernel that manages the hardware—the lowest-level operation on the physical device, as shown in Figure 10.4



(Figure 10.4) The kernel controls each piece of the host's hard- ware, including the CPU. Each kernel is operated by the DO/S, which, in turn, is directed by the application software running on the host computer. In this way, the most cumber- some functions are hidden from the user.

The kernel is the entity that controls and operates the CPU and manages the queues used for states of execution, although upper-level system policies direct how process control blocks (PCBs) are stored in the queues and how they're selected to be run. Therefore, each kernel assumes the role of helping the system reach its operational goals.

The kernel's states are dependent on the global system's process scheduler and dis- patcher, which organize the queues within the local CPU and choose the running policy, 4 which is used to execute the processes on those queues. Typically, the system's scheduling function has three parts: a decision mode, a priority function, and an arbitration rule.

• The decision mode determines which policies are used when scheduling a resource. Options could include pre-emptive, non-pre-emptive, round robin, and so on.

• The priority function gives the scheduling algorithm the policy that's used to assign an order to processes in the execution cycle. This priority is often determined using a calculation that's based on system characteristics, such as occurrence of events, task recurrence, system loading levels, or program run-time characteristics, such as most time remaining, least time remaining, and so on.

• The arbitration rule is a policy that's used to resolve conflicts between jobs of equal priority. That is, it typically dictates the order in which jobs of the same priority are to be executed. Two examples of arbitration rules are last-in first-out (LIFO), and first-in first-out (FIFO).

Most advances in job scheduling rely on one of three theories: queuing theory, statistical decision theory, or estimation theory. (These queuing and statistical decision theories are the same as those discussed in statistics courses.) An example of estimation theory is a scheduler based on process priorities and durations. It maximizes the system's throughput by using durations to compute and schedule the optimal way to interleave process chunks. Distributed scheduling is better achieved when migration of the scheduling function and policies considers all aspects of the system, including I/O, devices, processes, and communications.

Process-Based DO/S

A process-based DO/S provides process management through the use of client/server processes that are synchronized and linked together through messages and ports (the ports are also known as channels or pipes). The major emphasis is on processes and messages, and how they provide the basic features essential to process management, such as process creation, scheduling, pausing, communication, and identification, to name a few.

The issue of how to provide these features can be addressed in several ways. The pro- cesses can be managed from a single copy of the operating system, from multiple cooperating peers, or from some combination of the two. Operating systems for distributed computers are typically configured as a kernel on each site. All other services that are dependent on particular devices are typically found on the sites where the devices are located. As users enter the system, the scheduling manager gives them a unique process identifier and then assigns them to a site for processing.

In a distributed system, there is a high level of cooperation and sharing among the sites when determining which process should be loaded and where it should be run. This is done by exchanging messages among site operating systems. Once a process is scheduled for service, it must be initiated at the assigned site by a dispatcher. The dispatcher takes directions from the operating system's scheduler, allocates the device to the process, and initiates its execution. This procedure may necessitate one of the following: moving a process from memory in one site to memory at another site; reorganizing a site's memory allocation; reorganizing a site's READY, RUNNING, and WAIT queues; and initiating the scheduled process. The Processor Manager recognizes only processes and their demands for service. It responds to them based on the established scheduling policy, which determines what must be done to manage the processes. As mentioned in earlier chapters, policies for scheduling must consider issues such as load balancing, overhead minimization, memory loading minimization, first-come first-served, and least-time-remaining.

Object-Based DO/S

An object-based DO/S has a different way of looking at the computer system than a process-based DO/S. Instead of viewing the system as a collection of individual resources and processes, the system is viewed as a collection of objects. An object can represent a piece of hardware (such as a CPU or memory), software (such as files, programs, semaphores, and data), or a combination of the two (printers, scanners, and USB ports — each bundled with the software required to operate it). Each object in the system has a unique identifier to differentiate it from all other objects in the system.

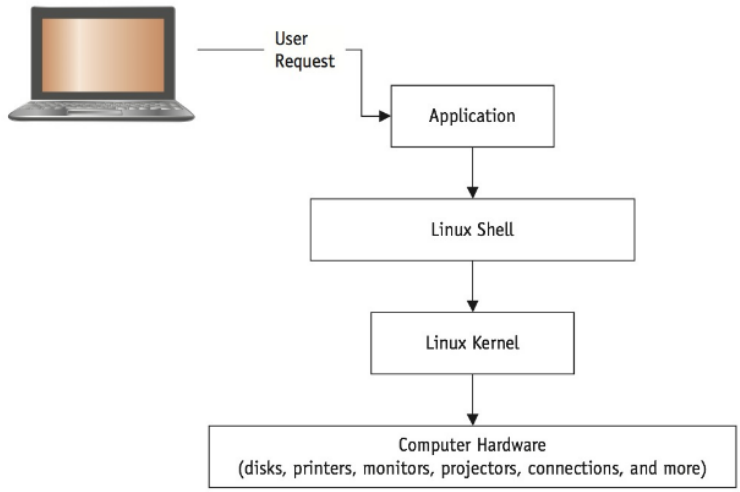
Objects are viewed by the operating system as abstract entities—data types that can go through a change of state, act according to set patterns, be manipulated, or exist in relation to other objects in a manner appropriate to the object's semantics in the system. This means that objects have a set of unchanging properties that defines them and their behavior within the context of their defined parameters. For example, a writ- able CD (CD-R) drive has unchanging properties that include the following: Data can be written to a disc, data can be read from a disc, reading and writing can't take place concurrently, and the data's beginning and ending points can't be compromised. If we use these simple rules to construct a simulation of a CD-R drive, we have created an accurate representation of this object.

Kernel Level

The kernel level, illustrated in Figure 10.5, provides the basic mechanisms for build- ing the operating system by creating, managing, scheduling, synchronizing, and deleting objects; and it does so dynamically. For example, when an object is created, it's assigned all the resources needed for its operation and is given control until the task is completed. Then the object returns control to the kernel, which selects the next object to be executed.

The kernel also has ultimate responsibility for the network's capability lists. Each site has both a capability manager that maintains the capability list for its objects, and a directory, which lists the location for all capabilities in the system. This directory guides local requests for capabilities to the sites on which they're located.

For example, if a process requests access to a region in memory, the capability manager first determines whether the requesting process has been previously granted rights.



(figure 10.5) In a Linux operating system, the kernel level is logically positioned between the computer hardware and the shell. User requests for hard- ware interaction move from the application to the shell, to the kernel, and then to the hardware control level.

If so, then it can proceed. If not, it processes the request for access rights. When the requester has access rights, the capability manager grants the requester access to the named object—in this case, the region in memory. If the named object is at a remote site, the local capability manager directs the requester, using a new address computation and message, to the remote capability manager.

The kernel is also responsible for process synchronization mechanisms and commu- nication support. Typically, synchronization is implemented as some form of shared variable, such as the WAIT and SIGNAL codes discussed in Chapter 6. Communication among distributed objects can be in the form of shared data objects, message objects, or control interactions. Most systems provide different communication primitives to their objects, which are either synchronous (the sender and receiver are linked and ready to send and receive;) or asynchronous (there is some shareable area such as a mailbox, queue, or stack to which the communicated information is sent). In some cases, the receiver periodically checks to see if anyone has sent anything. In other cases, the

communicated information arrives at the receiver's workstation without any effort on the part of the receiver; it just waits. There can also be a combination of these. An example of this communication model might have a mechanism that signals the receiver whenever a communication arrives at the sharable area, so that the information can be fetched whenever it's convenient. The advantage of this system is that it eliminates unnecessary checking when no messages are waiting.

6.6 File Management

Distributed file management gives users the illusion that the network is a single logical file system that's implemented on an assortment of devices and computers. Therefore, the main function of a DO/S File Manager is to provide transparent mechanisms to find, open, read, write, close, create, and delete files no matter where they're located in the network, as shown in Table 10.3.

Desired File Function	File Manager's Action
Find and Open	Uses a master directory with information about all files stored anywhere on the system and sets up a channel to the file.
Read	Establishes a channel to the file and attempts to read it using simple file access schemes. However, a read operation won't be immediately fulfilled if the file is currently being created or modified.
Write	Sets up a channel to the file and attempts to write to it using simple file access schemes. To write to a file, the requesting process must have exclusive access to it. This can be accomplished by locking the file, a technique frequently used in database systems. While a file is locked, all other requesting processes must wait until the file is unlocked before they can write to or read the file.
Close	Sends a command to the remote server to unlock a certain file. This is typically accomplished by changing the information in the directory at the file's storage site.
Create	Places a unique file identifier in the network's master directory and assigns space for it on a storage device.
Delete	Erases the unique file identifier in the master directory and deallocates the space that it occupies on the storage device. Notice that the file is not quickly erased; its space is just marked as available for writing another file.

(Table 10.3) Typical file management functions and the necessary actions of the File Manager

File management systems are a subset of database managers, which provide more capa- bilities to user processes than file systems, and are implemented as distributed database management systems as part of local area network systems.

Therefore, the tasks required by a DO/S include those typically found in a distributed database environment. These involve a host of controls and mechanisms necessary to provide consistent, synchronized, and reliable management of system and user information assets, including the following:

- Concurrency control
- Data redundancy
- Location transparency and distributed directory
- Deadlock resolution or recovery
- Query processing

Concurrency Control

Concurrency control techniques give the system the ability to perform concurrent reads and writes, as long as the results of these actions don't jeopardize the contents of the database. This means that the results of all concurrent transactions are the same as if the transactions had been executed one at a time, in some arbitrary serial order, thereby providing the serial execution view on a database. The concurrency control mechanism keeps the database in a consistent state as the transactions are processed.

Data Redundancy

Data redundancy also has beneficial aspects from a disaster recovery standpoint because, if one site fails, operations can be restarted at another site with the same resources. Later, the failed site can be reinstated by copying all the files that were updated since the failure. The disadvantage of redundant data is the task of keeping multiple copies of the same file up-to-date at all times. Every time a file is updated, every other copy of the file must be updated in an identical way; and the update must be performed in strict adherence to the system's reliability standards.

Location Transparency and Distributed Directory

Location transparency means that users don't need to be concerned with the physical location of their files. This is the essence of cloud computing. Instead, users deal with the network as a single system. Location transparency is provided by mechanisms and directories, which map logical data items to physical locations. The mechanisms usu- ally use information about data, which are stored at all sites in the form of directories.

Deadlock Resolution or Recovery

Deadlock detection, prevention, avoidance, and recovery are all strategies used by a distributed system.

• To detect circular waits, the system uses directed resource graphs and looks for cycles.

• To prevent circular waits, the system tries to delay the start of a transaction until it has all the resources it will request during its execution.

• To avoid circular waits, the system tries to allow execution only when it knows that the transaction can run to completion.

• To recover from a deadlock caused by circular waits, the system selects the best victim—one that can be restarted without much difficulty, and one that, when terminated, will free enough resources so that the others can finish. Then the system kills the victim, forces that process to restart from the beginning, and reallocates its resources to other waiting processes.

Query Processing

Query processing is the function of processing requests for information. Query process- ing techniques try to increase the effectiveness of global query execution sequences, local site processing sequences, and device processing sequences. All of these relate directly to the network's global process scheduling problem. Therefore, to ensure consistency of the entire system's scheduling scheme, the query processing strategy must be an integral part of the processing scheduling strategy.

6.7 Role of the Operating System in Security

Because it has access to every part of the system, the operating system plays a key role in computer system security. Any vulnerability at the operating system level opens the entire system to attack. The more complex and powerful the operating system, the more likely it is to have vulnerabilities to attack. As a result, system administrators must remain on guard in order to arm their operating systems with all the available defences against attack and possible failure.

System Survivability

The definition of system survivability is "the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents" (Linger, 2002).

• The term system refers to any system. It's used here in the broadest possible sense from laptop to distributed system to supercomputer.

• A mission is a very high-level set of requirements or goals.

• In a timely manner refers to system response time, a critical factor for most systems.

• The terms attack, failure, and accident refer to any potentially damaging incident, regardless of the cause, whether intentional or not.

Before a system can be considered survivable, it must meet all of these requirements, espe- cially with respect to services that are considered essential to the organization in the face of adverse challenges. The four key properties of survivable systems are resistance to attacks, recognition of attacks and resulting damage, recovery of essential services after an attack, and adaptation and evolution of system defense mechanisms to mitigate future attacks.

Backup and Recovery

Standard procedures for most computing systems include having sufficient backup and 4 recovery policies in place, and performing other archiving techniques. Many system manag- ers use a layered backup schedule. That is, they back up the entire system once a week, and only back up, daily, the files that were changed that day. As an extra measure of safety, cop- ies of complete system backups are stored for three to six months in a safe off-site location.

Backups are essential when the system becomes unavailable because of a natural disaster or when a computer virus infects the system. If the problem is discovered early, the admin- istrator can run eradication software and reload damaged files from your backup copies. Of course, any changes made since the files were backed up will need to be regenerated.

Off-site backups are also crucial to disaster recovery. The disaster could come from anywhere. Here are just a few of the threats:

• Water from firehoses being used to fight a fire upstairs

• Fire from an electrical connection

• Malfunctioning server

• Corrupted archival media

• Intrusion from unauthorized users

Security Breaches

A gap in system security can be malicious or not. For instance, some intrusions are the result of an uneducated user gaining unauthorized access to system resources. But others stem from the purposeful disruption of the system's operation. Still others are purely accidental, such as hardware malfunctions, undetected errors in the operating system or applications, or natural disasters. Malicious or not, a breach of security severely dam- ages the system's credibility. Following are some types of security breaks that can occur.

Unintentional Data Modifications

An unintentional attack is any breach of security or modification of data that was not the result of a planned intrusion.

Intentional System Attacks

Intentional unauthorized access includes denial of service attacks, browsing, wiretap- ping, repeated trials, trapdoors, and trash collection. These attacks are fundamentally different from viruses, worms, and the like (covered shortly), which inflict widespread damage to numerous companies and organizations—without specifying certain ones as targets.

Viruses

A virus is defined as a small program written to alter the way a computer operates and is run without the permission or knowledge of the user. A virus meets both of the fol- lowing criteria:

• It must be self-executing. Often, this means placing its own code in the path of another program.

• It must be self-replicating. Usually, this is accomplished by copying itself from infected files to clean files. Viruses can infect desktop computers and network serv- ers alike, and spread each time the host file is executed, as shown in Figure 11.2.

Latest Risks

Type	Name	Discovered
Adware	Adware.Browext	10/20/2016
Adware	Adware.GoSave	10/13/2016
Adware	Adware.CornerSunshine	10/12/2016
Potentially Unwanted App	PUA.GoPCPro	10/12/2016
Potentially Unwanted App	PUA.BildDownloader	10/12/2016
Potentially Unwanted App	PUA.MediaCodec	10/12/2016
Potentially Unwanted App	SONAR.BC.PUA!g21	10/06/2016
Potentially Unwanted App	SONAR.BC.PUA!g20	10/06/2016
Potentially Unwanted App	SONAR.BC.PUA!g19	10/06/2016
Potentially Unwanted App	SONAR.BC.PUA!g18	10/06/2016
Potentially Unwanted App	SONAR.BC.PUA!g17	10/06/2016
Potentially Unwanted App	SONAR.BC.PUA!g16	10/06/2016

(figure 11.2) A partial list of risks identified by Symantec. com as of October 2016 (Symantec, 2016a) Current information can be found at https://www.symantec. com/security_response/ landing/risks/.

Viruses spread via a wide variety of applications and have even been found in legitimate shrink-wrapped software. In one case, a virus was inadvertently picked up at a trade show by a developer who unknowingly allowed it to infect the finished code of a com- pleted commercial software package just before it was marketed.

Worms

A worm is a memory-resident program that copies itself from one system to the next without requiring the aid of an infected program file. The immediate result of a worm is slower processing time of legitimate work because the worm siphons off processing time and memory space. Worms are especially destructive on networks, where they hoard critical system resources, such as main memory and processor time.

Adware and Spyware

No standard industry-wide definitions exist for adware and spyware, although the concepts are broadly recognized as intrusive software covertly loaded on someone's computing device with malicious intent. Adware is software that gains access to a com- puter after the user visits a corrupted website, opens an infected email, loads infected shareware, or performs a similar innocent action. Adware examines the user's browser history, cookies, and other installed apps with the intent of facilitating the delivery of user-personalized advertising. And all of this is done without the user's permission or knowledge.

Trojans

A Trojan (originally called a Trojan Horse after the ancient Greek story) is a destructive program that's disguised as a legitimate or harmless program that carries within itself the means to allow the program's creator to secretly access the user's system. Intruders have been known to capture user passwords by using a Trojan to replace the standard login program on the computer with an identical fake login that captures keystrokes.

- The user sees a login prompt and types in the user ID.
- The user sees a password prompt and types in the password.
- The rogue program records both the user ID and password and sends a typical login failure message to the user.
- Then the Trojan program stops running and returns control to the legitimate program.
- Now, the user sees the legitimate login prompt and retypes the user ID.
- The user sees the legitimate password prompt and retypes the password.
- Finally, the user gains access to the system, unaware that the rogue program has used the first attempt to record the user ID and password.

Blended Threats

A blended threat combines into one program the characteristics of other attacks, including a virus, a worm, a Trojan, adware, spyware, key loggers, and other malicious code. That is, this single program uses a variety of tools to attack systems and spread to others. Because the threat from these programs is so diverse, the only defence against them is a comprehensive security plan that includes multiple layers of protection. A blended threat shows the following characteristics:

• Harms the affected system. For example, it might launch a denial of service attack at a target IP address, deface a website, or plant Trojans for later execution.

• Spreads to other systems using multiple methods. For example, it might scan for vulnerabilities to compromise a system, such as embedding code in Web page files on a server, infecting the systems of visitors who visit a compromised website, or sending unauthorized email from compromised servers with a worm attachment.

• Attacks other systems from multiple points. For example, it might inject malicious code into the .exe files on a system, raise the privilege level of an intruder's guest account, create world-readable and world-writeable access to private files, and add bogus script code to Web page files.

• Propagates without human intervention. For example, it might continuously scan the Internet for vulnerable servers that are unpatched and open to attack.

• Exploits vulnerabilities of target systems. For example, it might take advantage of oper- ating system problems (such as buffer overflows), Web page validation vulnerabilities on input screens, and commonly known default passwords to gain unauthorized access.

System Protection

Threats can come from outsiders (those outside the organization) as well as from insiders (employees or others with access to the system) and can include theft of intellectual prop- erty, or other confidential or sensitive information, fraud, and acts of system sabotage.

System protection is multifaceted. Four protection methods are discussed here: installing antivirus software (and running it regularly), using firewalls (and keeping them up-to- date), ensuring that only authorized individuals access the system, and taking advantage of encryption technology when the overhead required to implement it is mandated by the risk.

Antivirus Software

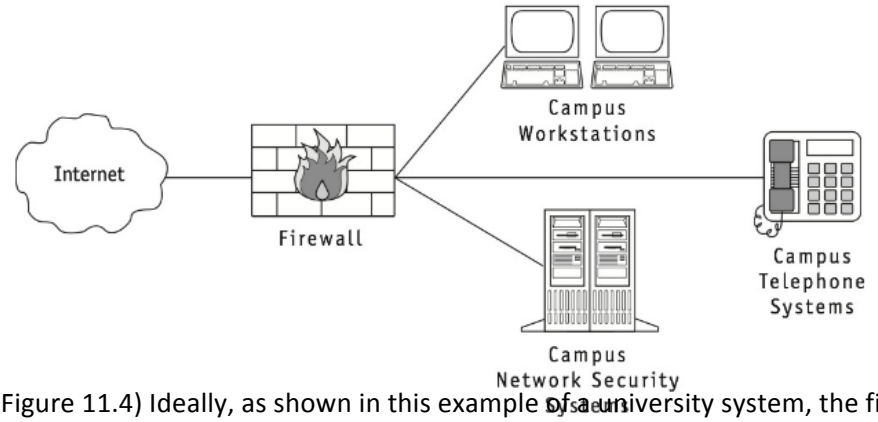
Antivirus software protects systems from attack by malicious software. The level of protection is usually in proportion to the importance of its data. Medical data should be highly protected. Photos and music files might not deserve the same level of security.

Software to combat viruses can be preventive or diagnostic, or both. Preventive programs may calculate a checksum for each production program, putting the values in a master file. Later, before a program is executed, its checksum is compared with the master. Generally, diagnostic software compares file sizes (checking for added code when none is expected), looks for replicating instructions, and searches for unusual file activity. Some software may look for certain specific instructions and monitor the way the programs execute. But remember: soon after these packages are marketed, system intruders start looking for ways to thwart them. Hence, only the most current software can be expected to uncover the latest viruses. In other words, old software will only find old viruses.

Firewalls

Network assaults can include compromised Web servers, email servers, circumvented firewalls, and FTP sites accessed by unauthorized users.

A firewall is hardware and/or software designed to protect a system by disguising its IP address from outsiders who don't have authorization to access it or ask for information about it. A firewall sits between the Internet and the network, as shown in Figure 11.4, and blocks curious inquiries and potentially dangerous intrusions from outside the system.



(Figure 11.4) Ideally, as shown in this example of a university system, the firewall sits between the campus networks and the Internet, filtering traffic that moves to and from campus Firewall Campus Telephone systems.

The typical tasks of the firewall are to:

• log activities that access the Internet,

• maintain access control based on the senders' or receivers' IP addresses, Systems

• maintain access control based on the services that are requested,

• hide the internal network from unauthorized users requesting network information,

• verify that virus protection is installed and being enforced,

• perform authentication based on the source of a request from the Internet.

Authentication Protocols

Network authentication software performs verification that the individual who is trying to access a protected system is authorized to do so. One popular authentication tool was Kerberos, a network authentication protocol developed as part of the Athena Project at Massachusetts Institute of Technology (MIT) to answer the need for password encryp- tion to improve network security. Kerberos was designed to provide strong authentica- tion for client/server applications, and was named after the three-headed dog of Greek mythology that guarded the gates of Hades.

Encryption

The most extreme protection for sensitive data is encryption—putting it into a secret code. Total network encryption, also called communications encryption, is the most extreme form—all communications with the system are encrypted. The system then decrypts them for processing. To communicate with another system, the data is encrypted, transmitted, decrypted, and processed.

How Encryption Works

A good way to learn about cryptography is to first understand the role of a public key and a private key. The private key is a pair of two prime numbers (usually with 75 or more digits each) chosen by the person who wants to receive a private message. The two prime numbers are multiplied together, forming a third number with 150 or more digits. The person who creates this private key is the only one who knows which two prime numbers were used to create this number, which is known as the public key.

Sniffers and Spoofing

If sensitive data is sent over a network or the Internet in cleartext, without encryption, it becomes vulnerable at numerous sites across the network. Packet sniffers, also called sniffers, are programs that reside on computers attached to the network. They peruse data packets as they pass by, examine each one for specific information, and log copies of interesting packets for more detailed examination. Sniffing is particularly problem- atic in wireless networks. Anyone with a wireless device can detect a wireless network that's within range. If the network is passing cleartext packets, it's quite easy to inter- cept, read, modify, and resend them. The information sought ranges from passwords, Social Security numbers, and credit card numbers to industrial secrets and marketing information. This vulnerability is a fundamental shortcoming of cleartext transmission over the Internet.

Wireless security is a special area of system security that is too vast to cover here. Read- ers are encouraged to pursue the subject in the current literature for the latest research, tools, and best practices.

Spoofing is a security threat that relies on cleartext transmission, whereby the assail- ant falsifies the IP addresses of an Internet server by changing the address recorded in packets it sends over the Internet. This technique is useful when unauthorized users want to disguise themselves as friendly sites. For example, to guard confidential information on an internal network (intranet), some Web servers allow access only to users from certain sites; however, by spoofing the IP address, the server could inadvertently admit unauthorized users.

Password Management

The most basic techniques used to protect hardware and software investments are good passwords and careful user training, but this is not as simple as it might appear. Many users forget their passwords, rarely change them, often share them, and consider them bothersome.

Password Construction

If they're constructed correctly, passwords can be one of the easiest and most effective protection schemes to implement. A good password is unusual, memorable to the user, not obvious, and changed often. Ideally, the password should be a combination of characters and numbers, something that's easy for the user to remember but difficult for someone else to guess. Ideally, the password is committed to memory, not written down, and not included in a script file to log on to a network.

There are several reliable techniques for generating a good password:

• Use a minimum of eight characters, including numbers and symbols.

• Create a misspelled word, or join bits of phrases into a word that's easy to remember.

• Following a certain pattern on the keyboard, generate new passwords easily by starting your sequence with a different letter each time.

THINK POINT/CASE STUDY

Conclusion

Although operating systems for networks necessarily include the functions of the four managers discussed so far in this textbook—the Memory Manager, Processor Manager, Device Manager, and File Manager—they also need to coordinate all those functions among the network's many varied pieces of hardware and software, no matter where they're physically located.

Every networked system, whether a NOS or DO/S, has specific requirements. Each must be secure from unauthorized access yet accessible to authorized users. Each must monitor its available system resources, including memory, processors, devices, and files, as well as its communications links. In addition, because it's a networking operating system, it must perform the required networking tasks.

The system is only as secure as the integrity of the data that's stored on it. A single breach of security—whether catastrophic or not, whether accidental or not—damages the system's integrity. Damaged integrity threatens the viability of even the best-designed system, as well as its managers, its designers, and its users. Therefore, vigilant security precautions are essential.

• Create acronyms from memorable sentences, such as MDWB4 YOIA, which stands for: "My Dog Will Be 4 Years Old In April."

• If the operating system differentiates between uppercase and lowercase characters, as UNIX, Linux, and Android do, users should take advantage of this feature by using both in the password: MDwb4YOia.

• Avoid words that appear in any dictionary.

Typical Password Attacks

A dictionary attack is the term used to describe a method of breaking encrypted pass- words. Its requirements are simple: a copy of the encrypted password file and the algorithm used to encrypt the passwords. With these two tools, the intruder runs a software program that takes every word in the dictionary, runs it through the password encryption algorithm, and compares the encrypted result to the encrypted passwords contained in the file. If both encrypted versions match, then the intruder knows which dictionary word was used as a legitimate password.

Chapter Review

Research Topics

A. In current literature, research the use of biometrics to replace typed pass- words. Describe the historical milestones of the technology and list its significant advantages and disadvantages. Find at least two real world examples where this technology is used, citing your sources for each and the dates of their publication.
B. Identify four early operating systems for networks (those used before 2010) and explain which network operating systems were and which distributed operating systems were. State the reasons for your answers and cite the sources of your research as well as their publication dates.
C. A. In current literature, research the use of biometrics to replace typed pass- words. Describe the historical milestones of the technology and list its significant advantages and disadvantages. Find at least two real world examples where this technology is used, citing your sources for each and the dates of their publication.

Review Questions

1. Describe the primary functions of a network.
2. Describe, in your own words, the differences between PANs, LANs, MANs, and WANs, and give an example of each.
3. If you are setting up a simple network at home to connect your newest devices, which of the network topologies discussed in this chapter would you choose? Explain why.
6. If your NOS had four nodes, how many operating systems would it have? Explain your reasons for your answer.
7. If your DO/S had four nodes, how many operating systems would it have? Explain your answer.
8. Explain, in your own words, the steps a DO/S File Manager uses to open a file, read data from it, update that data, and close the file. Do these steps change if the data is not changed? Describe the reasons for your answer.
9. Give three examples of excellent passwords and explain why each would be a good choice to protect a system from unauthorized users.
10. Password management software/apps promise to keep passwords safe, while allowing fast and easy access for authorized users when they need to retrieve them. For at least one of these software packages or apps:
a. Describe in detail how secure you believe it is.
b. List the vulnerabilities that you believe it faces in day-to-day use.
c. Compare your software's most significant benefits to at least two other competitors.

Chapter 7: System Management

LEARNING OUTCOMES

After completing this chapter, you should be able to describe:

- **How to evaluate the trade-offs that should be considered when improving system performance**
- **How positive and negative feedback loops pertain to system performance**
- **How the two system-monitoring techniques compare**
- **How patch management protects systems**
- **How sound accounting practices affect system administrators**

7.1 Evaluating an Operating System

To evaluate an operating system, we need to understand its design goals, how it communicates with its users, how its resources are managed, and what trade-offs were made to achieve its goals. In other words, an operating system's strengths and weaknesses need to be weighed in relation to who will be using the operating system, on what hardware, and for what purpose.

On the other hand, a Linux operating system is the choice for the management of small and large networks because it is easy for professionals to use and customize, enabling them to accomplish a variety of tasks with a myriad of hardware configurations. As of this writing, novice users have not taken to Linux with the same passion.

Cooperation Among Components

The performance of any one resource depends on the performance of the other resources in the system. For example, memory management is intrinsically linked with device management when memory is used to buffer data between a very fast processor and slower secondary storage devices. Many other examples of resource interdependence have been shown throughout the preceding chapters.

If you managed an organization's computer system and were allocated money to upgrade it, where would you put the investment to best use? Your choices could include:

• More memory,

• A faster CPU,

• Additional processors,

• More disk drives,

• A RAID system,

• New file management software.

Or, if you bought a new system, what characteristics would you look for to make it more efficient than the old one?

Any system improvement can only be made after extensive analysis of the needs of the system's resources, requirements, managers, and users. But whenever changes are made to a system, administrators are often trading one set of problems for another. The key is to consider the performance of the entire system and not just the individual components.

Role of Memory Management

Memory management when you consider increasing memory or changing to another memory allocation scheme, you must con- sider the operating environment in which the system will reside. There's a trade-off between memory use and CPU overhead.

For example, if the system will be running student programs exclusively, and the aver- age job is only three pages long, your decision to increase the size of virtual memory wouldn't speed up throughput because most jobs are too small to use virtual memory effectively. Remember, as the memory management algorithms grow more complex, the CPU overhead increases and overall performance can suffer. On the other hand, some operating systems perform remarkably better with additional memory. We explore this issue further in the exercises at the end of this chapter.

Role of Processor Management

Processor management. Let's say you decide to implement a multiprogramming system to increase your processor's utilization. If so, you'd have to remember that multiprogramming requires a great deal of synchronization between the Memory Manager, the Processor Manager, and the I/O devices. The trade-off: better use of the CPU versus increased overhead, slower response time, and decreased throughput.

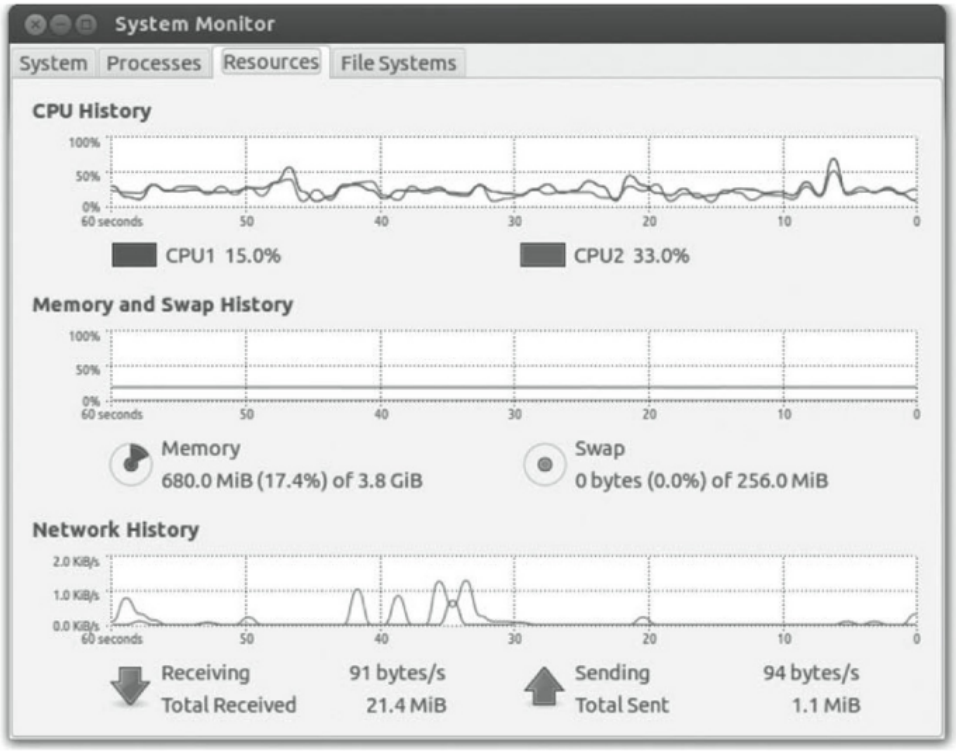
Among the several problems to watch for are:

• A system can reach a saturation point if the CPU is fully utilized, but is allowed to accept additional jobs—this results in higher overhead and less time to run programs.

• Under heavy loads, the CPU time required to manage I/O queues, which under nor- mal circumstances don't require a great deal of time, can dramatically increase the time required to run jobs.

• With an I/O-heavy mix, long queues can form at the channels, control units, and I/O devices, leaving the CPU idle as it waits for processes to finish their tasks.

Likewise, increasing the number of processors necessarily increases the overhead required to manage multiple jobs among multiple processors. But under certain circumstances, the payoff can be faster turnaround time. Many operating systems offer a system monitor to display current system usage, as shown in Figure 12.1.



(Figure 12.1) Ubuntu Linux System Monitor showing (top) the percent usage of the two CPUs for this laptop, (middle) memory usage, and (bottom) network connection history.

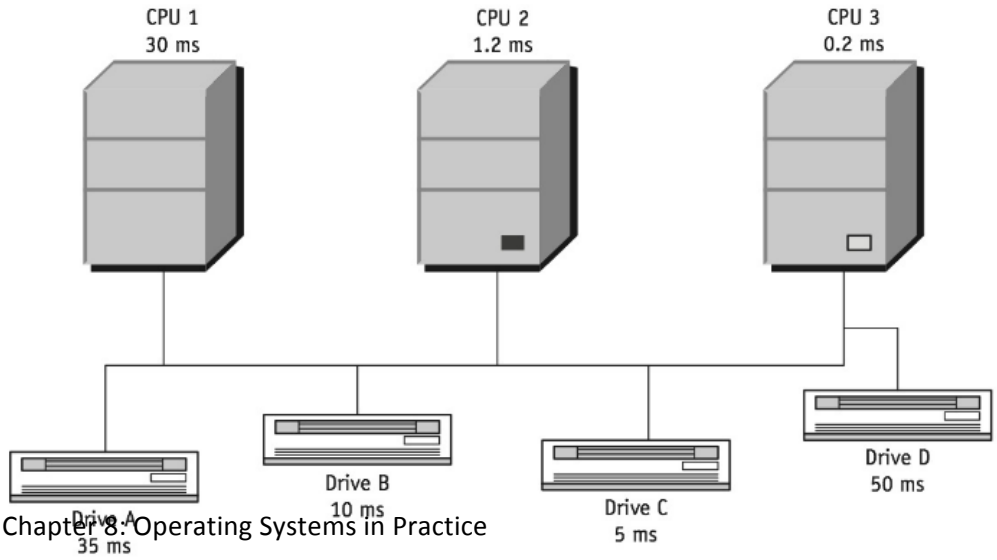
Role of Device Management

Device management, contains several ways to improve I/O device utilization including buffering, blocking, and rescheduling I/O requests to optimize access times. But there are trade-offs: Each of these options also increases CPU overhead and uses additional memory space.

Blocking reduces the number of physical I/O requests, and this is good. But it's the CPU's responsibility to block and later unblock the records, and that's overhead.

Buffering helps the CPU match the slower speed of I/O devices, and vice versa, but it requires memory space for the buffers—either dedicated space or a temporarily allocated section of main memory. This, in turn, reduces the level of processing that can take place. For example, if each buffer requires 64K of memory and the system requires two sets of double buffers, we've dedicated 256K of memory to the buffers. The trade-off is reduced multiprogramming versus better use of I/O devices.

Rescheduling requests is a technique that can help optimize I/O times; it's a queue reor- dering technique. But it's also an overhead function, so the speed of both the CPU and the I/O device must be weighed against the time it would take to execute the reordering algorithm. The following example illustrates this point. Figure 12.2 shows three differ- ent sample CPUs, the average time with which each can execute 1,000 instructions, and four disk drives with their average data access speeds.



(Figure 12.2) Drive B 10 ms Drive C 5 ms Three CPUs and the average speed with which each can perform 1,000 instructions. All CPUs have access to four drives that have different data access speeds.

Using the data provided in Figure 12.2, and assuming that a typical reordering module consists of 1,000 instructions, which combinations of one CPU and one disk drive war- rant a reordering module? To learn the answer, we need to compare disk access speeds before and after reordering.

For example, let's assume the following:

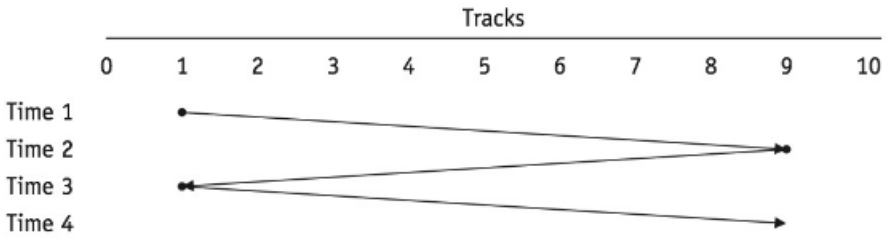
• This very simple system consists of CPU 1 and Disk Drive A.

• CPU 1 requires 30 ms to reorder the waiting requests.

• Drive A requires approximately 35 ms for each move, regardless of its location on the disk.

• The system is instructed to access Track 1, Track 9, Track 1, and then Track 9.

• The arm is already located at Track 1, as shown in Figure 12.3.



(Figure 12.3) Using the combination of CPU 1 and Drive A from Figure 12.2, data access requires 105 ms without reordering.

Without reordering, the time required for this system to access all four tracks on the disk takes 105 ms (35 + 35 + 35 = 1052)

Role of File Management

The discussion of file management looks at how secondary storage allocation schemes help the user organize and access the files on the system. Almost every factor discussed in that chapter can affect overall system performance.

For example, file organization is an important consideration. If a file is stored non- contiguously, and has several sections residing in widely separated cylinders of a disk pack, sequentially accessing all of its records could be time consuming. Such a case would suggest that the files should be compacted (also called defragmented), so that each section of the file resides near the others. However, defragmentation takes CPU time and leaves the files unavailable to users while it's being done.

Another file management issue that could affect retrieval time is the location of a volume's directory. For instance, some systems read the directory into main memory and hold it there until the user terminates the session. If we return to our example in Figure 12.2, with the four disk drives of varying speeds, the first retrieval would take 35 ms when the system retrieves the directory for Drive A and loads it into memory. But every subsequent access would be performed at the CPU's much faster speed without the need to access the disk. Similar results would be achieved with each of the other disk drives, as shown in Table 12.1.

Disk Drive	Access Speed for First Retrieval	Subsequent Retrievals
A	35 ms	1.2 ms
B	10 ms	1.2 ms
C	5 ms	1.2 ms
D	50 ms	1.2 ms

(table 12.1) A sample system with four disk drives of different speeds and a CPU speed of 1.2 ms. If the file's directory is loaded into C memory, access speed affects only the initial retrieval and none of the subsequent retrievals.

This poses a problem if the system abruptly loses power before any modifications have been recorded permanently in secondary storage. In such a case, the I/O time that was saved by not having to access secondary storage every time the user requested to see the directory would be negated by not having current information in the user's directory.

Similarly, the location of a volume's directory on the disk might make a significant difference in the time it takes to access it. For example, if directories are stored on the outermost or innermost track, then, on average, the disk drive arm has to travel farther to access each file than it would if the directories were kept in the center tracks.

Overall, file management is closely related to the device on which the files are stored; designers must consider both issues at the same time when evaluating or modifying computer systems. Different schemes offer different flexibility, but the trade-off for increased file flexibility is increased CPU overhead.

7.2 Role of Network Management

The discussion of network management examines the impact of adding networking capability to the operating system, and its overall effect on system performance. The Network Manager routinely synchronizes the load among remote processors, determines message priorities, and tries to select the most efficient communication paths over multiple data communication lines.

Measuring System Performance

Total system performance can be defined as the efficiency with which a computer system meets its goals, that is, how well it serves its users. However, system efficiency is not easily measured because it's affected by three major components: user programs, operating system programs, and hardware. The main problem, however, is that system performance can be very subjective and difficult to quantify: How, for instance, can anyone objectively gauge ease of use? While some aspects of ease of use can be quantified—for example, the time it takes to log on—the overall concept is difficult to quantify.

Even when performance is quantifiable, such as the number of disk accesses per minute, it isn't an absolute measure but a relative one, based on the interactions among the three components and the workload being handled by the system.

Measurement Tools

Most designers and analysts rely on certain measures of system performance: throughput, capacity, response time, turnaround time, resource utilization, availability, and reliability.

Throughput is a composite measure that indicates the productivity of the system as a whole; the term is often used by system managers. Throughput is usually measured under steady-state conditions and reflects quantities, such as the number of jobs pro- cessed per day, or the number of online transactions handled per hour. Throughput can also be a measure of the volume of work handled by one unit of the computer system, an isolation that's useful when analysts are looking for bottlenecks in the system.

Bottlenecks tend to develop when resources reach their capacity, or maximum through- put level; the resource becomes saturated and the processes in the system aren't being passed along. For example, thrashing is a result of a saturated disk. In this case, a bottle- neck occurs when main memory has been overcommitted and the level of multiprogramming has reached a peak point. When this occurs, the working sets for the active jobs can't be kept in main memory, so the Memory Manager is continuously swapping pages between main memory and secondary storage. The CPU processes the jobs at a snail's pace because it's very busy flipping pages, or it's idle while pages are being swapped.

Essentially, bottlenecks develop when a critical resource reaches its capacity. For exam- ple, if the disk drive is the bottleneck because it is running at full capacity, then the bottleneck can be relieved by alleviating the need for it. One solution might be restricting the flow of requests to that drive or adding more disk drives.

Bottlenecks can be detected by monitoring the queues forming at each resource: When a queue starts to grow rapidly, this is an indication that the arrival rate is greater than, or close to, the service rate and the resource will be saturated soon. These are called feedback loops; we discuss them later in this chapter. Once a bottleneck is detected, the appropriate action can be taken to resolve the problem.

To online interactive users, response time is an important measure of system performance. Response time is the interval required to process a user's request; it's measured from when the user presses the key to send the message until the system indicates receipt of the message. For batch jobs, this is known as turnaround time: the time it takes from the submission of a job to the return of the output to the user. Whether in an online or batch context, this measure depends on both the workload being handled by the system at the time of the request, and the type of job or request being submitted. Some requests, for instance, might be handled faster than others because they require fewer resources.

To be an accurate measure of the predictability of the system, measurement data show- ing response time and turnaround time should include not just their average values, but also their variance.

Resource utilization is a measure of how much each unit is contributing to the overall operation. It's usually given as a percentage of time that a resource is actually in use. For example: Is the CPU busy 60 percent of the time? Is the printer busy 90 percent of the time? How about each of the terminals? Or the seek mechanism on a disk? This data helps the analyst determine whether there is balance among the units of a system, or whether a system is I/O-bound or CPU-bound.

Availability indicates the likelihood that a resource will be ready when a user needs it. For online users, it may mean the probability that a port is free when they attempt to log on. For those already on the system, it may mean the probability that one or several specific resources, such as a plotter or a group of dedicated devices, will be ready when their programs need them. Availability in its

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 12 page 392 for more reading

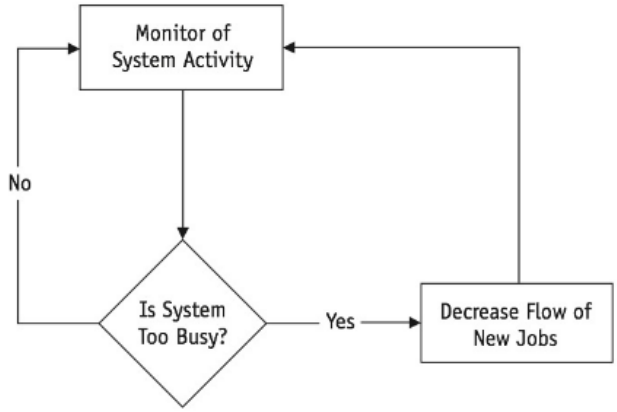
simplest form means that a unit will be operational and not out of service when a user needs it.

Feedback Loops

To prevent the processor from spending more time doing overhead than executing jobs, the operating system must continuously monitor the system and feed this information to the Job Scheduler. Then the Scheduler can either allow more jobs to enter the system, or prevent new jobs from entering until some of the congestion has been relieved. This is called a feedback loop and it can be either negative or positive.

One example of a feedback loop was instituted by a California school district, which installed dynamic “Your Speed” roadside displays to give drivers instant feedback concerning their actual speed as they approached the pedestrians and bicycles in school zones. It’s important to note that the displays did not offer drivers any information that they did not already have, but it did bring it to the drivers’ conscious attention. The result? Most drivers slowed an average of 14 percent; and, at three participating schools, average speeds were below the posted speed limit (Goetz, 2011).

In an operating system, a negative feedback loop mechanism monitors the system and, when it becomes too congested, signals the Job Scheduler to slow down the arrival rate of the processes, as shown in Figure 12.5.

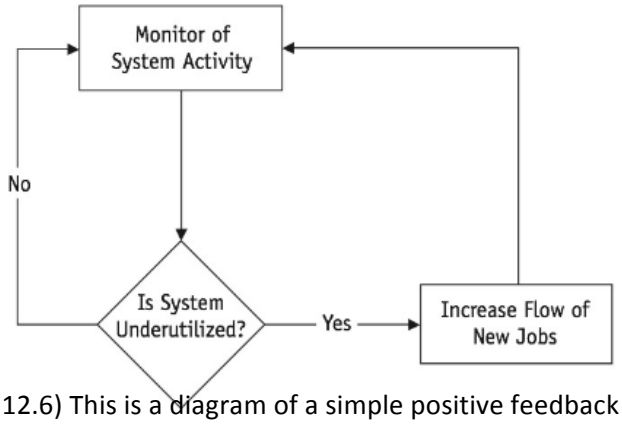


(Figure 12.5) This is a simple negative feedback loop. Feedback loops monitor system activity and go into action only when the system is too busy.

In a computer system, a negative feedback loop monitoring I/O devices, for example, would inform the Device Manager that Printer 1 has too many jobs in its queue, caus- ing the Device Manager to direct all newly arriving jobs to Printer 2, which isn’t as busy. The negative feedback helps stabilize the system and keeps queue lengths close to expected mean values.

People on vacation use negative feedback loops all the time. For example, if you’re look- ing for a gas station and the first one you find has too many cars waiting in line, you collect the data and you react negatively. Therefore, your processor suggests that you drive on to another station, assuming, of course, that you haven’t procrastinated too long and are about to run out of gas. As a result of using this feedback loop, long lines at some gas stations are reduced when drivers choose to look elsewhere.

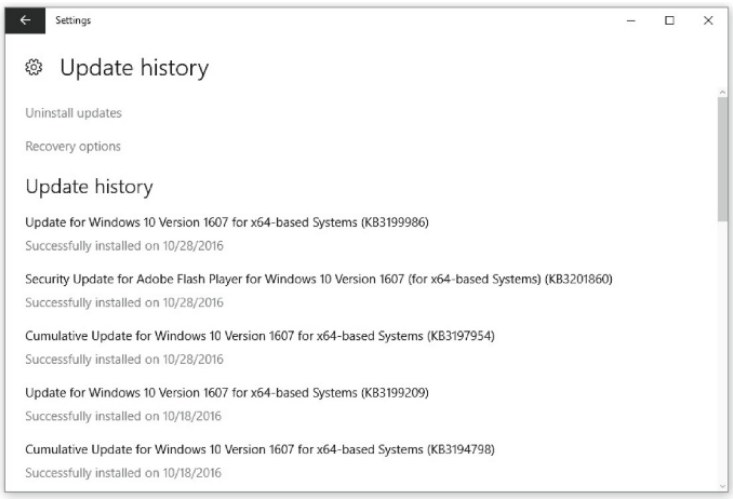
A positive feedback loop mechanism works in the opposite way. It frequently monitors the system, and, when the system becomes underutilized, the positive feedback loop causes the arrival rate to increase, as shown in Figure 12.6. Positive feedback loops are used in paged virtual memory systems, but they must be used cautiously because they're more difficult to implement than negative loops.



(Figure 12.6) This is a diagram of a simple positive feedback loop. This type of loop constantly monitors system activity and goes into action only when the system is not busy enough. System activity monitoring is critical here because the system can become unstable.

Patch Management

Patch management is the systematic updating of the operating system, as well as other system software. Typically, a patch is a piece of programming code that replaces or changes some of the code that makes up the software. There are three primary reasons for the emphasis on software patches for sound system administration: the need for vigilant security precautions against constantly changing system threats; the need to assure system compliance with government regulations and other policies regarding privacy and financial accountability; and the need to keep systems running at peak efficiency. An example of system patching, also called software updating, is shown in Figure 12.7.



(Figure 12.7) The update history for this computer running Windows 10 lists a few of the most recent updates made to the operating system and other vulnerable software. Source: Microsoft

Patches aren't new—they've been written and distributed since programming code was first created. However, the task of keeping computing systems patched correctly has become a challenge because of the complexity of each piece of software, as well as the numerous interconnections among them. Another factor is the speed with which soft- ware vulnerabilities are exploited by worms, viruses, and other system assaults.

Patching Fundamentals

While the installation of the patch is the most public event, there are several essential steps that take place before this happens:

1. Identify the required patch.

2. Verify the patch's source and integrity.

3. Test the patch in a safe testing environment.

4. Deploy the patch throughout the system.

5. Audit the system to gauge the success of the patch deployment.

Although this discussion is limited to managing operating system patches, all changes to the operating system or other critical system software must be undertaken in a test environment that mimics a production environment.

7.3 Software to Manage Deployment

Patches can be installed manually, one at a time, or via software that's written to perform the task automatically. Organizations that choose software can decide to build their own deployment software or buy ready-made software to perform the task for them. Deployment software falls into two groups: those programs that require an agent running on the target system (called agent-based software), and those that do not (agentless software).

If the deployment software uses an agent, which is software that assists in patch installation, then the agent must be installed on every target computer system before patches can be deployed. On a very large or dynamic system, this can be a daunting task. Therefore, for administrators of large, complex networks, agentless software may offer some time-saving efficiencies.

Timing the Patch Cycle

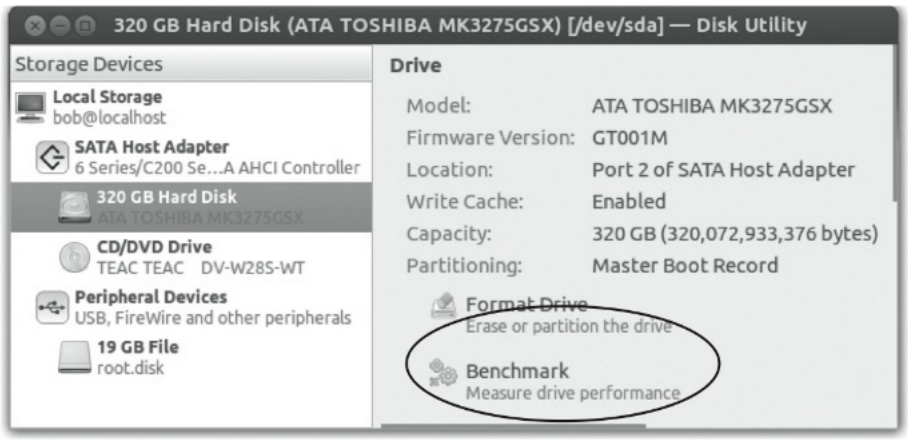
While critical system patches must be applied immediately, less critical patches can be scheduled at the convenience of the systems group. These patch cycles can be based on calendar events or vendor events. For example, routine patches can be applied monthly or quarterly, or they can be timed to coincide with a vendor's service pack release. Service pack is a term used by certain vendors for patches to the operating system and other software. The advantage of having routine patch cycles is that they allow for thorough review of the patch and testing cycles before deployment.

System Monitoring

As computer systems have evolved, several techniques for measuring performance have been developed, which can be implemented using either hardware or software components. Of the two, hardware monitors are more expensive, but, because they're outside of the system and are attached electronically, they have the advantage of having a minimum impact on the system. They include hard-wired counters, clocks, and comparative elements.

Software monitors are relatively inexpensive, but, because they become part of the computing system, they can distort the results of the analysis. After all, the monitoring software must use the resources it's trying to monitor. In addition, software tools must be developed for each specific system, so it's difficult to move them from one system to another.

Today, system measurements must include the other hardware units as well as the operating system, compilers, and other system software. An example is shown in Figure 12.10.



(Figure 12.10) In Ubuntu Linux, the Disk Utility has a benchmark button for measuring the performance of the highlighted hard drive, shown here at the bottom of the screen.

Measurements are made in a variety of ways. Some are made using real programs, usually production programs that are used extensively by the users of the system, which are run with different configurations of CPUs, operating systems, and other components. The results are called benchmarks and are useful when comparing systems that have gone through extensive changes.

THINK POINT/CASE STUDY

Conclusion

The operating system is more than the sum of its parts—it's the orchestrated cooperation of every piece of hardware and every piece of software. As we've shown, when one part of the system is favored, it's often at the expense of the others. If a trade-off must be made, the system's managers must make sure they're using the appropriate measurement tools and techniques to verify the effectiveness of the system before and after modification, and then evaluate the degree of improvement.

Benchmarks are often used by vendors to demonstrate to prospective clients the specific advantages of a new CPU, operating system, compiler, or piece of hardware.

Chapter Review

Research Topics

A. Research the current literature to find instances of fake antivirus software posing as legitimate software. Identify the estimated number of computers affected by the false software. Cite your sources.

	Review Questions 1. Give your own example of a positive feedback loop from a real-life situation, and explain why a positive loop is preferable.

	2. Describe how you would use a negative feedback loop to manage your bank balance. Describe how you would do so with a positive feedback loop. Explain which you would prefer and why. 3. Let's say your own computer has failed unexpectedly and catastrophically twice over the past 12 months. a. Identify the worst possible time for failure and the best possible time. b. Compare the time and cost it would have required for you to recover from those two catastrophic failures.

Chapter 8: Operating Systems in Practice

LEARNING OUTCOMES

After completing this chapter, you should be able to describe:

- **Goals of UNIX designers**
- **Significance of using files to manipulate devices**
- **Strengths and weaknesses of UNIX**
- **How the Linux operating system is designed**
- **How the Linux community keeps the software up-to-date**
- **How the strengths and weaknesses of Linux compare**
- **The goals for designers of Windows operating systems**
- **How the Memory Manager and Virtual Memory Manager work**
- **The Windows user interfaces**
- **The design goals for the Android™ operating system**
- **The role of open source software**
- **How Android works with its Linux foundation**

The UNIX operating system, authored by Ken Thompson, has three major advantages: It is portable from large systems to small systems, it has very powerful utilities, and it provides device independence to application programs. Its portability is attributed to the fact that most of it is written in a high-level language called C (authored by Dennis Ritchie), instead of assembly language, which is a low-level language that's specific to the architecture of a particular computer. UNIX utilities are brief, single-operation commands that can often be combined into a single command line to achieve almost any desired result—a feature that many programmers find endearing. And it can be con- figured to operate virtually any type of device. UNIX commands are case sensitive and are strongly oriented toward lowercase characters, which are faster and easier to type.

8.1 The Evolution of UNIX

The first official UNIX version by Thompson and Ritchie was designed to “do one thing well” and to run on a popular minicomputer. Before long, UNIX became so widely adopted that it had become a formidable competitor in the market. For Version 3, Ritchie took the innovative step of developing a new programming language, which he called C; and he wrote a compiler, making it easier and faster for system designers to write code.

As UNIX grew in fame and popularity, AT&T found itself in a difficult situation. At the time, the company was a legal monopoly and was, therefore, forbid- den by U.S. federal government antitrust regulations to sell software. But it could, for a nominal fee, make the operating system available to universities, and it did. Between 1973 and 1975, several improved versions were developed, and the most popular was written at the University of California at Berkeley, which became known as BSD, which is short for Berkeley Software Distribution. Over time, its popularity with academic programmers created a demand for UNIX in business and industry.

Design Goals

From the beginning, Thompson and Ritchie envisioned UNIX as an operating system created by programmers, for programmers. It was built with an efficient command-line interface to be fast, flexible, and easy for experts to use.

The immediate goals were twofold: to develop an operating system that would sup- port software development, and to keep its algorithms as simple as possible, without becoming rudimentary. To achieve their first goal, they included utilities in the operating system that programmers, at the time, needed to write customized code. Each utility was designed for simplicity—to perform only one function but to perform it very well. These utilities were designed to be used in combination with each other so that programmers could select and combine the appropriate utilities needed to carry out specific jobs. This concept of using small manageable sections of code was a perfect fit with the second goal: to keep the operating system simple. To do this, Thompson and Ritchie selected algorithms based on simplicity instead of speed or sophistication. As a result, even early UNIX systems could be mastered by experienced programmers in a matter of weeks.

The designers’ long-term goal was to make the operating system, and any application software developed for it, portable from one computer to another. The obvious advantage of portability is that it reduces conversion costs, and doesn’t cause application packages to become obsolete with every change in hardware. This goal was finally achieved when device independent became a standard feature of UNIX.

Numerous versions of UNIX conform to the specifications for Portable Operating System Interface for Computer Environments (POSIX)—a registered trademark of the IEEE. POSIX is a family of IEEE standards that define a portable operating system inter- face that enhances the portability of programs from one operating system to another.

Process Management

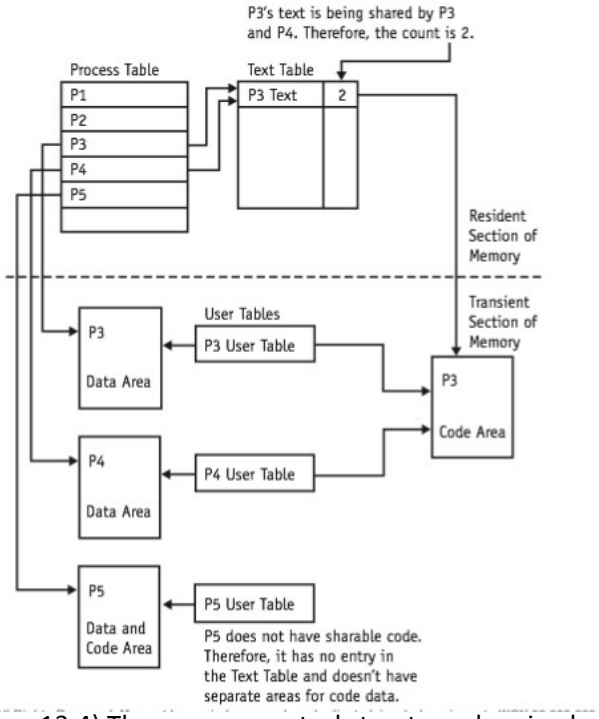
The Processor Manager of a UNIX system kernel handles the allocation of the CPU, process scheduling, and the satisfaction of process requests. To perform these tasks, the kernel maintains several important tables to coordinate the execution of processes and the allocation of devices.

The process scheduling algorithm picks the process with the highest priority to run first. Because one of the values used to compute the priority is accumulated CPU time, any processes that have used a lot of CPU time will get a lower priority than those that have not. The system updates the compute-to-total-time ratio for each job every second. This ratio divides the amount of CPU time that a process has used up by the total time the same process has spent in the system. A result close to one would indicate that the process is CPU-bound. If several processes have the same computed priority, they're handled round robin—low-priority processes are pre-empted by high-priority processes. Interactive processes typically have a low compute-to-total-time ratio, so interactive response is maintained without any special policies.

The overall effect of this negative feedback is that the system balances I/O-bound jobs with CPU-bound jobs to keep the processor busy and to minimize the overhead for waiting pro

Process Table versus User Table

UNIX uses several tables to keep the system running smoothly, as shown in Figure 13.4. Information on simple processes, those without sharable code, is stored in two sets of tables: the process table, which always resides in memory; and the user



(Figure 13.4) The process control structure showing how Process Table the process table and text table interact for processes with sharable code, as well as for those without sharable code.

Table, which resides in memory only while the process is active. In Figure 13.4, Process 3 and Process 4 show the same program code indicated in the text table by the number 2. Their data areas and user tables are kept separate while the code area is being shared. Process 5 is not sharing its code with another process; therefore, it's not recorded in the text table, and its data and code area are kept together as a unit.

Each entry in the process table contains the following information: process identifica- tion number, user identification number, address of the process in memory or second- ary storage, size of the process, and scheduling information. This table is set up when the process is created and is deleted when the process terminates.

For processes with sharable code, the process table maintains a sub-table, called the text table, which contains the memory address or secondary storage address of the text segment (sharable code) as well as a counter to keep track of the number of processes using this code. Every time a process starts using this code, the count is increased by one; and every time a process stops using this code, the count is decreased by one. When the count is equal to zero, the code is no longer needed and the table entry is released, together with any memory locations that had been allocated to the code segment.

A user table is allocated to each active process. As long as the process is active, this table is kept in the transient area of memory, and contains information that must be accessible when the process is running. This information includes: the user and group identification numbers to determine file access privileges, pointers to the system's file table for every file being used by the process, a pointer to the current directory, and a list of responses for various interrupts. This table, together with the process data segment and its code segment, can be swapped into or out of main memory as needed.

8.2 Process Synchronization

A profound strength of UNIX is its true multitasking capabilities. It achieves process synchronization by requiring processes to wait for certain events. For example, if a process needs more memory, it's required to wait for an event associated with memory allocation. Later, when memory becomes available, the event is signaled and the process can continue. Each event is represented by integers that, by convention, are equal to the address of the table associated with the event.

However, a race may occur if an event happens during the process's transition between deciding to wait for the event and entering the WAIT state. In this case, the process is waiting for an event that has already occurred and may not recur.

Fork

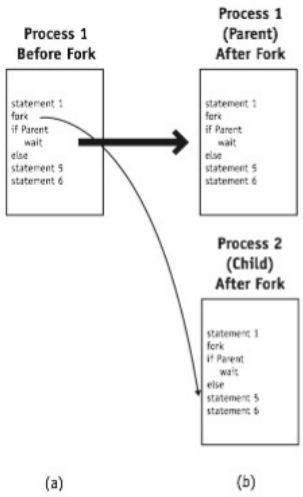
An unusual feature of UNIX is that it gives the programmer the capability of executing one program from another program using the fork command. While the fork system call gives the second program all the attributes of the first program (such as any open files) the first program is saved in its original form.

Here's how it works. The fork splits a program into two copies, which are both running from the statement after the fork command. When fork is executed, a "process id" (called pid for short) is generated for the new process. This is done in a way that ensures that each process has its own unique ID number. Figure 13.5 shows what happens after the fork. The original process (Process 1) is called the parent process, and the result- ing process (Process 2) is the child process. A child inherits

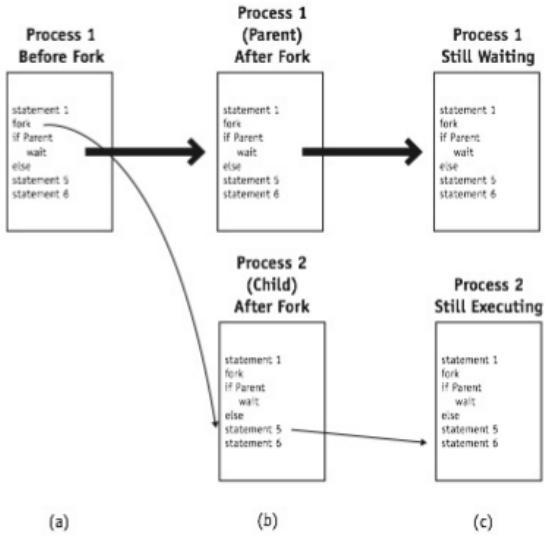
the parent's open files and runs asynchronously with it unless the parent is instructed to wait for the termination of the child process.

Wait

A related command, wait, allows the programmer to synchronize process execution by suspending the parent until the child is finished, as shown in Figure 13.6.



(Figure 13.5) When the fork command is received, the parent process shown in (a) begets the child process shown in (b) and Statement 5 is executed twice.



(Figure 13.6) The wait command used in conjunction with the fork command will synchronize the parent and child processes. In these three snapshots of time, (a) the parent process is shown before the fork, (b) shows the parent and child after the fork, and (c) shows the parent and child during the wait.

In Figure 13.6, the IF-THEN-ELSE structure is controlled by the value assigned to the pid, which is returned by the fork system call. A pid greater than zero indicates a parent process; a pid equal to zero indicates a child process; and a negative pid indicates an error in the fork call.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 13 page 422-423 for more reading

Device Management

An innovative feature of UNIX is the treatment of devices—the operating system is designed to provide device independence to the applications running under it. This is achieved by treating each I/O device as a special type of file. Every device that's installed in a UNIX system is assigned a name that's similar to the name given to any other file, which is given descriptors called i-nodes. These descriptors identify the devices, contain information about them, and are stored in the device directory. The subroutines that work with the operating system to supervise the transmission of data between main memory and a peripheral unit are called the device drivers.

If the computer system uses devices that are not supplied with the operating system, their device drivers must be written by an experienced programmer, or obtained from a reliable source, and installed on the operating system.

The actual incorporation of a device driver into the kernel is done during the system configuration. UNIX has a program called config that automatically creates a conf. c file for any given hardware configuration. This conf.c file contains the parameters that control resources, such as the number of internal buffers for the kernel and the size of the swap space. In addition, the conf.c file contains two tables, bdevsw (short for block device switch) and cdevsw (short for character device switch). These allow the UNIX system kernel to adapt easily to different hardware configurations by installing different driver modules.

Device Classifications

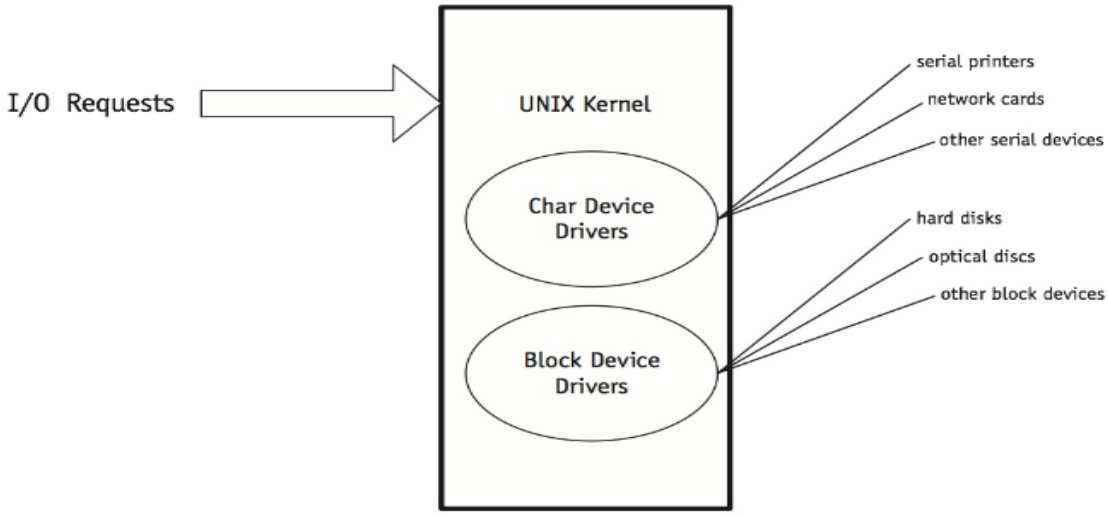
The I/O system is divided into the block I/O system (sometimes called the structured I/O system) and the character I/O system (sometimes called the unstructured I/O system).

Each physical device is identified by a minor device number, a major device number, and a class—either block or character. See Figure 13.8.

Each class has a Configuration Table that contains an array of entry points into the device drivers. This table is the only connection between the system code and the device drivers, and it's an important feature of the operating system. This table allows system programmers to create new device drivers quickly in order to accommodate a differently configured system. The major device number is used as an index to the appropriate array so that it can access the correct code for a specific device driver.

The minor device number is passed to the device driver as an argument, and is used to access one of several identical or similar physical devices controlled by the driver.

As its name implies, the block I/O system is used for devices that can be addressed as a sequence of identically sized blocks. This allows the Device Manager to use buffering



(Figure 13.8) When a process sends an I/O request, it goes to the UNIX kernel where the char and block device drivers reside.

to reduce the physical disk I/O. UNIX has 10 to 70 buffers for I/O; and information related to these buffers is kept on a list.

Every time a read command is issued, the I/O buffer list is searched. If the requested data is already in a buffer, then it's made available to the requesting process. If not, then it's physically moved from secondary storage to a buffer. If a buffer is available, the move is made. If all buffers are busy, then one must be emptied out to make room for the new block. This is done by using an LRU policy: The contents of frequently used buffers are left intact, which, in turn, should reduce physical disk I/O.

Devices in the character class are handled by device drivers that implement character lists. Here's how it operates: A subroutine puts a character on the list, or queue, and another subroutine retrieves the character from the list.

A terminal is a typical character device that has two input queues and one output queue. The two input queues are labeled as the raw queue and the canonical queue. It works like this: As the user types in each character, it's collected in the raw input queue. When the line is completed and the enter key is pressed, the line is copied from the raw input queue to the canonical input queue, and the CPU interprets the line. Similarly, the section of the device driver that handles characters going to the output module of a terminal stores them in the output queue until it holds the maximum number of characters.

The I/O procedure is synchronized through hardware completion interrupts. Each time there's a completion interrupt, the device driver gets the next character from the queue and sends it to the hardware. This process continues until the queue is empty. Some devices can actually belong to both the block and character classes. For instance, disk drives and tape drives can be accessed in block mode using buffers, or the system can bypass the buffers when accessing the devices in character mode.

8.3 File Management

UNIX has three types of files: directories, ordinary files, and special files. Each enjoys certain privileges.

• Directories are files used by the system to maintain the hierarchical structure of the file system. Users are allowed to read information in directory files, but only the system is allowed to modify directory files.

• Ordinary files are those in which users store information. Their protection is based on a user's requests and is related to the read, write, execute, and delete functions that can be performed on a file.

• Special files are the device drivers that provide the interface for I/O hardware. Special files appear as entries in directories. They're part of the file system, and most of them reside in the /dev directory. The name of each special file indicates the type of device with which it's associated. Most users don't need to know much about special files, but system programmers should know where they are and how to use them.

UNIX stores files as sequences of bytes and doesn't impose any structure on them. Therefore, the structure of files is controlled by the programs that use them, not by the system. For example, text files (those written using an editor) are strings of char- acters with lines delimited by the line feed, or new line, character. On the other hand, binary files (those containing executable code generated by a compiler or assembler) are sequences of binary digits grouped into words as they appear in memory during the execution of the program.

The UNIX file management system organizes the disk into blocks, and divides the disk into four basic regions:

• The first region (starting at address 0) is reserved for booting.

• The second region, called a superblock, contains information about the disk as a whole, such as its size and the boundaries of the other regions.

• The third region includes a list of file definitions, called the i-list, which is a list of file descriptors—one for each file. The descriptors are called i-nodes. The position of an i-node on the list is called an i-number, and it is this i-number that uniquely identifies a file.

• The fourth region holds the free blocks available for file storage. The free blocks are kept in a linked list where each block points to the next available empty block. Then, as files grow, non-contiguous blocks are linked to the already existing chain.

Whenever possible, files are stored in contiguous empty blocks. And because all disk allocation is based on fixed-size blocks, allocation is very simple and there's no need to compact the files.

Each entry in the i-list is called an i-node (also spelled inode) and contains 13 disk addresses. The first 10 addresses point to the first 10 blocks of a file. However, if a file is larger than 10 blocks, the eleventh address points to a block that contains the addresses of the next 128 blocks of the file. For larger files, the twelfth address points to another set of 128 blocks, each one pointing to 128 blocks. For files that require even more space, there is a thirteenth address.

Each i-node contains information on a specific file, such as the owner's identification, protection bits, its physical address, file size, time of creation, last use, last update, and the number of links, as well as whether the entity is a directory, an ordinary file, or a special file.

8.4 User Interfaces

UNIX typically supports both menu-driven systems and the traditional command-driven interfaces. When using menu-driven interfaces, there are a number of keyboard shortcuts that can be used to facilitate common actions, as shown in Table 13.3. Notice that Com- mand-G performs the grep command (a famous UNIX command) listed in Table 13.4. The Command key on a Macintosh computer keyboard looks like a series of loops.

Shortcut	Stands For	Description
Command-X	Cut	Remove the selected item and copy it to the Clipboard.
Command-C	Copy	Copy the selected item to the Clipboard. This also works for files in the Finder.
Command-V	Paste	Paste the contents of the Clipboard into the current document or app. This also works for files in the Finder.
Command-Z	Undo	Undo the previous command.
Command-Shift-Z	Redo	Reverse the previous undo command.
Command-A	All	Select all items in the target space.
Command-F	Find	Find: Open a Find window, or find items in a document.
Command-G	grep	Find again: Find the next occurrence of the item previously found.
Command-Shift-G		To find the previous occurrence of the item previously found.
Command-H	Hide	Hide the windows of the front app. To view the front app but hide all other apps, press Command-Option-H.
Command-M	Minimize	Minimize the front window to the Dock.
Command-N	New	Open a new document or window.
Command-O	Open	Open the selected item, or open a dialog to select a file to open.
Command-P	Print	Print the current document.

(Table 13.3) This list of Macintosh keyboard shortcuts shows how a combination of keystrokes can accomplish common tasks, including cut and paste.

Command	Stands For	Action to be Performed
(filename)	Run File	Run or execute the file with that name
cat	Concatenate	Concatenate files
cd	Change Directory	Change working directory
cp	Copy	Copy a file into another file or directory
date	Date	Show date and time
grep	Global Regular Expression/Print	Find a specified string in a file
lpr	Print	Submit a print request
ls	List Directory	Show a listing of the filenames in the directory
ls -l	Long Directory List	Show a comprehensive directory list
man	Manual	Show the online manual
mkdir	Make Directory	Make a new directory
more	Show More	Type the file's contents to the screen
mv	Move	Move or rename a file or directory
rm	Remove File	Remove/delete a file or directory

(Table 13.4) Small sample of UNIX user commands, which are entered in all lower case. Linux systems use almost identical commands check your technical documentation for proper spelling and syntax.

While many users know this operating system today as menu-driven, UNIX was first designed only as a command-driven system, and its user commands (some shown in Table 13.4) were very short: either one character (usually the first letter of the command), or a group of characters (an acronym of the words that make up the command). In command mode, the system prompt is very economical, often only one character, such as a dollar sign ($) or percent sign (%).

In command mode, many commands can be combined on a single line for additional power and flexibility, however the commands themselves must not be abbreviated or expanded, and must be in the correct case. Remember that in most circumstances, UNIX commands are entered only in lowercase letters.

The general syntax of commands is this:

Command arguments file_name

The command is any legal operating system command. Arguments are required for some commands and optional for others. The filename can be a relative or absolute path name. Commands are interpreted and executed by the shell, one of the two most widely used programs. The shell is technically known as the command line interpreter because that's its function. But it isn't only an interactive command interpreter; it's also the key to the coordination and combination of system programs. In fact, it's a sophisticated programming language in itself.

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 13 page 434-441 for more reading

Design Goals

Many Windows operating systems feature built-in support for networks. Most use an object model to manage operating system resources and to allocate them to users in a consistent manner. They also use symmetric multiprocessing (often abbreviated as SMP) to achieve maximum performance from multiprocessor computers.

To accommodate its user community and to optimize resources, the Windows team has identified five overarching design goals: extensibility, portability, reliability, compat- ibility, and performance.

Extensibility

Knowing that operating systems must adapt to support new hardware devices or new software technologies, the design team decided that the operating system had to be easily enhanced. This feature is called extensibility. In a concerted effort to ensure the integrity of the Windows code, the designers separated operating system functions into two groups: a privileged executive process, and a set of non-privileged processes called protected subsystems. The privileged processor mode is called kernel mode and the non- privileged processor mode is called user mode. The term privileged refers to a proces- sor's mode of operation. Most processors have a privileged mode, in which all machine instructions are allowed and system memory is accessible; and a non-privileged mode, in which certain instructions are not allowed and system memory is not accessible.

Usually, operating systems execute only in kernel mode, and application programs execute only in user mode, except when they call operating system services. In Windows, the protected subsystems execute in user mode as if they were applications, which allows protected subsystems to be modified or added without affecting the integrity of the executive process.

In addition to protected subsystems, Windows designers included key features to address extensibility issues, including: a modular structure allowing new components to be added easily to the executive process; a group of abstract data types called objects that are manipulated by a special set of services, allowing system resources to be managed uniformly; and, a remote procedure call that allows an application to call remote ser- vices regardless of their location on the network.

Portability

Portability refers to the operating system's ability to operate on different platforms that use different processors or configurations with a minimum amount of recoding. To address this goal, Windows system developers used a four-prong approach. First, they wrote their new operating system in a standardized, high-level programming language. Second, the system was built to accommodate the hardware to which it was to be ported (32-bit, 64-bit, and so on). Third, all code that interacted directly with the hardware was minimized to reduce incompatibility errors. Fourth, all hardware-dependent code was isolated into modules that could be modified or replaced more easily whenever the operating system was ported.

Reliability

The term reliability refers to the robustness of a system—that is, its predictability in responding to error conditions, even those caused by hardware failures. It also refers to the operating system's ability to protect itself and its users from accidental or deliber- ate damage.

Structured exception handling is one way to capture error conditions and respond to them uniformly. Whenever such an event occurs, either the operating system or the processor issues an exception call, which automatically invokes the exception handling code that's appropriate to handle the condition, ensuring that no harm is done to either user programs or the system. In addition, the following features strengthen the system:

• A modular design that divides the executive process into individual system compo- nents that interact with each other through specified programming interfaces. For example, if it becomes necessary to replace the Memory Manager with a new one, the new one will use the same interfaces.

• A file system called New Technology File System (NTFS) can recover from all types of errors, including those that occur in critical disk sectors. To ensure recoverability, NTFS uses redundant storage and a transaction-based scheme for storing data.

• A security architecture that provides a variety of security mechanisms, such as user logon, resource quotas, and object protection.

• A virtual memory strategy that provides every program with a large set of memory addresses and prevents one user from reading or modifying memory that's occupied by another user unless the two are explicitly sharing memory.

Compatibility

Compatibility usually refers to an operating system's ability to execute programs writ- ten for other operating systems or for earlier versions of the same system. However, for Windows, compatibility is a more complicated topic.

By using protected subsystems, Windows provides execution environments for applica- tions that are different from its primary programming interface—the Win32 applica- tion programming interface (API). When running on Intel processors, the protected subsystems supply binary compatibility with existing Microsoft applications. Windows also provides source-level compatibility with the Portable Operating System Interface, or POSIX, applications that adhere to the POSIX operating system interfaces defined by the Institute of Electrical and Electronics Engineers (IEEE). POSIX is an operating system API that defines how a service is invoked through a software package. POSIX was developed by the IEEE to increase the portability of application software.

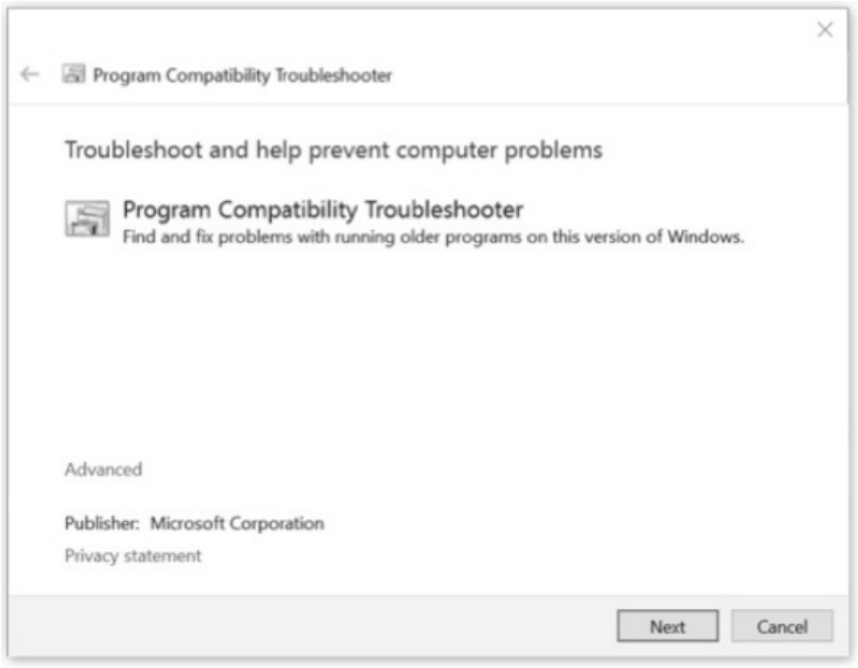
Performance

The operating system should respond quickly to CPU-bound applications. To do so, Windows is built with the following features:

• System calls, page faults, and other crucial processes are designed to respond in a timely manner.

• A mechanism called the local procedure call (LPC) is incorporated into the operat- ing system so that communication among the protected subsystems doesn't restrain performance.

• Critical elements of Windows' networking software are built into the privileged por-tion of the operating system to improve performance. In addition, these components can be loaded and unloaded from the system dynamically, if necessary.



(Figure 14.2) To enhance compatibility, this trouble-shooter identifies installed pro- grams that are designed to run on older Windows operating systems.

In the early days of Windows operating systems, they were found to slow down with use as more and more programs or applications were installed, and the computer was accessed on a regular basis. Even when users tried to uninstall applications that were no longer needed, performance could remain slow and did not return to the benchmarks attained when the computer and its operating system were new. The solution suggested by experts at that time was to completely reformat the hard drive and reload the current set of necessary programs, which restored the system to fast response times, but this solution was often painful and time consuming. Current versions of Windows have improved in this regard and do not exhibit the same lethargy as a computer is used over the years.

8.5 Virtual Memory Implementation

The Virtual Memory Manager relies on address space management and paging techniques.

Address Space Management

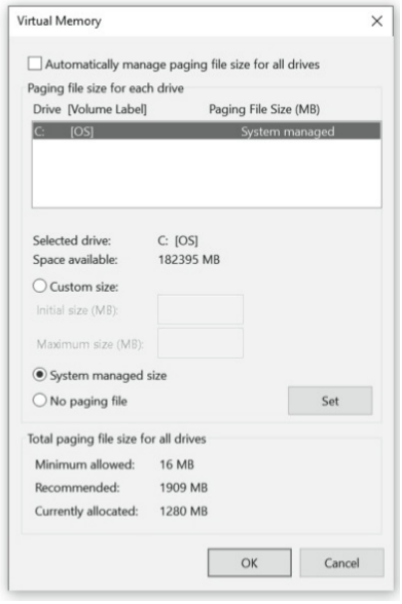
As shown earlier in Figure 14.3, the upper half of the virtual address space is accessible only to kernel-mode processes. Code in the lower part of this section, kernel code and data, is never paged out of memory. In addition, the addresses in this range are translated by the hardware, providing exceedingly fast data access. Therefore, the lower part of the resident operating system code is used for sections of the kernel that require maximum performance, such as the code that dispatches units of execution (called threads of execution) in a processor.

When users create a new process, they can specify that the VM Manager initialize their virtual address space by duplicating the virtual address space of another process. This allows environment subsystems to present their client processes with views of memory that don't correspond to the virtual address space of a native process.

Paging

The pager is the part of the VM Manager that transfers pages between page frames in memory and disk storage, shown in Figure 14.4. As such, it's a complex combination of both software policies and hardware mechanisms. Software policies determine when to bring a page into memory and where to put it. Hardware mechanisms determine the exact manner in which the VM Manager translates virtual addresses into physical addresses.

Because the hardware features of each system directly affect the success of the VM Manager, the precise implementation of virtual memory varies from one processor to another. Therefore, this portion of the operating system is not portable and must be modified for each new hardware platform. To make the transition easier, Windows keeps this VM code small and isolated. The processor chip handles address translation, and exception handling looks at each address generated by a program and translates it into a physical address



(Figure 14.4) Using this screen, the user can fine-tune the system's virtual memory and paging file settings.

If the page containing the address is not already in memory, then the hardware generates a page fault and issues a call to the pager. The translation look-aside buffer (TLB) is a hardware array of associative memory used by the processor to speed memory access. As pages are brought into memory by the VM Manager, the TLB creates entries for them.

Read

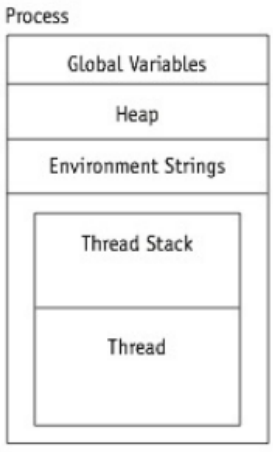
Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 14 page 454-455 for more reading

Processor Management

In general, a process is the combination of an executable program, a private memory area, and system resources allocated by the operating system as the program executes. However, a process requires a fourth component before it can do any work: at least one thread of execution. A thread is the entity within a process that the kernel schedules for execution; it could be roughly equated to a task. Using multiple threads (also called multi-threading) allows a programmer to break up a single process into several executable segments and also to take advantage of the extra CPU power available in computers with multiple processors.

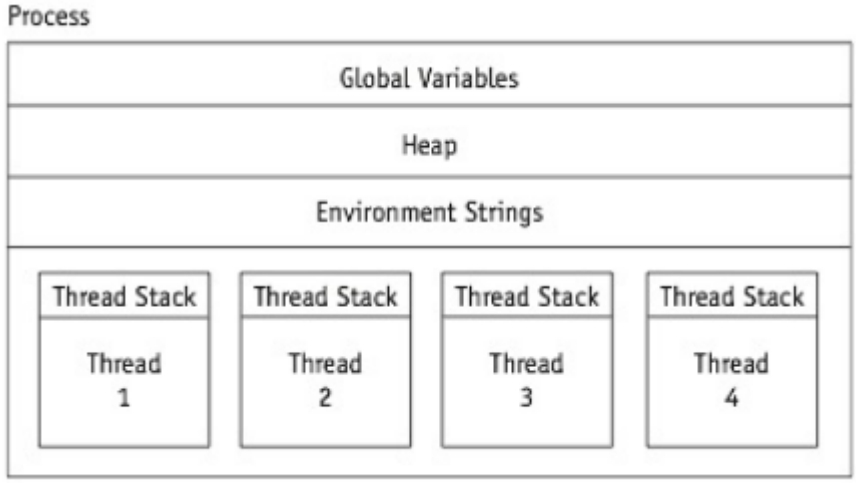
Windows is a preemptive, multitasking, multi-threaded operating system. By default, a process contains one thread, which is composed of a unique identifier, the contents of a volatile set of registers, indicating the processor's state; two stacks used during the thread's execution; and a private storage area that can be used by subsystems and dynamic link libraries.

These components are called the thread's context; the actual data forming this context varies from one processor to another. The kernel schedules threads for execution on a processor. For example, when you use the mouse to double-click an application's icon, the operating system creates a process, and that process has one thread that runs the code. The process is like a container for the global variables, the environment strings, the heap owned by the application, and the thread. Figure 14.5 shows a diagram of a process with a single thread.



(Figure 14.5) Uni-tasking in Windows. Here's how a process with a single thread is scheduled for execution on a system with a single processor.

For systems with multiple processors, a process can have as many threads as there are CPUs available. The overhead incurred by a thread is minimal. In some cases, it's actually advantageous to split a single application into multiple threads because the entire program is then much easier to understand. The creation of threads isn't as complicated as it may seem. Although each thread has its own stack, as shown in Figure 14.6, all threads belonging to one process share its global variables, heap, and environment strings.



(Figure 14.6) Multitasking using multi- threading. Here's how a process with four threads can be scheduled for execution on a system with four processors.

Multiple threads can present problems because it's possible for several different threads to modify the same global variables independently of each other. To prevent this, Windows operating systems include synchronization mechanisms to give exclusive access to global variables as these multi-threaded processes are executed.

For example, let's say the user is modifying a database application. When the user enters a series of records into the database, the cursor changes into a hold symbol that indicates a thread is writing the last record to the disk while another thread is accepting new data. Therefore, even as processing is going on, the user can perform other tasks. In this way, the concept of overlapped I/O now occurs on the user's end as well as on the computer's end.

Multi-threading improves database searches because data is retrieved faster when the system has several threads of execution that search an array simultaneously, especially if each thread has its own CPU. Programs written to take advantage of these features must be designed very carefully to minimize contention, such as when two CPUs attempt to access the same memory location at the same time, or when two threads compete for single shared resources, such as a hard disk.

Client–server applications tend to be CPU-intensive for the server because, although queries on the database are received as requests from a client computer, the actual query is managed by the server's processor. A Windows multiprocessing environment can satisfy these requests by allocating additional CPU resources.

Network Management

In a Windows operating system, networking is an integral part of the operating system executive by providing services such as user accounts, and resource security. It also pro- vides mechanisms to implement communication among networked computers, such as with named pipes and mailslots. Named pipes provide a high-level interface for passing data between two processes regardless of their locations. Mailslots provide one-to-many and many-to-one communication mechanisms, useful for broadcasting messages to any number of processes.

The Active Directory database stores many types of information and serves as a general- purpose directory service for a heterogeneous network.

Microsoft built the Active Directory entirely around the Domain Name Service or Domain Name System (DNS) and Lightweight Directory Access Protocol (LDAP). DNS is the hierarchical replicated naming service on which the Internet is built. However, although DNS is the backbone directory protocol for one of the largest data networks, it doesn't provide enough flexibility to act as an enterprise directory by itself. That is, DNS is primarily a service for mapping machine names to Internet protocol (IP) addresses, which is not enough for a full directory service. A full directory service must be able to map names of arbitrary objects, such as machines and applications, to any kind of information about those objects.

The Active Directory groups machines into administrative units called domains, each of which gets a DNS domain name, such as pitt.edu. Each domain must have at least one domain controller, that is, a machine running the Active Directory server.

For improved fault tolerance and performance, a domain can have more than one domain controller, with each holding a complete copy of that domain's directory database.

Security Management

Windows operating systems provide an object-based security model. That is, a security object can represent any resource in the system: a file, device, process, program, or user. This allows system administrators to give precise security access to specific objects in the system while allowing them to monitor and record how objects are used.

One of the most important tasks for those managing Windows operating systems is the need for aggressive patch management to combat the many viruses and worms that target these systems.

Security Concerns

Because Windows is so widely used, it is a favorite target for system intruders. Therefore, Windows operating systems include several important security features. The first is a secure logon facility that requires users to identify themselves by entering a unique logon identifier and a password before they're allowed access to the system. In an attempt to move access control beyond the keyboard, biometric identification is available on some systems, as well as built-in options for creating and using graphic passwords.

Security Terminology

The built-in security for recent Windows network operating systems is a necessary ele- ment for managers of Web servers and networks. Its directory service lets users find what they need, and a communications protocol lets users interact with it. However, because not everyone should be able to find or interact with everything in the network, controlling such access is the job of distributed security services.

Effective distributed security requires an authentication mechanism that allows a client to prove its identity to a server. Then the client needs to supply authorization information, which the server then uses to determine which specific access rights have been given to this client. Finally, it needs to provide data integrity using a variety of methods ranging from a cryptographic checksum for all

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 14 page 466-474 for more reading

transmitted data to completely encrypting all transmitted data.

8.6 The design goals for the Android™ operating system

Android is designed to run mobile devices. It was originally only designed to run smart- phones and tablets, but now it also runs watches, wearable devices, desktops, televisions, and more. It is built on a Linux foundation and relies on Linux to perform some of the most fundamental tasks, including management of main memory, processors, devices, files, and network access.

The most customizable part of Android is its user interface, which can be arranged by each user to include almost any configuration of applications (often shortened to apps). Apps are programmed in Java using a software developer kit (SDK), which is download- able for free, so that anyone with an interest in programming can install the necessary software, learn the basics of Java, and begin creating apps for distribution to anyone using a compatible device.

Like Linux, Android is an open-source operating system, publishing key elements of its source code. While not as open as Linux, it is much more so than the operating system that runs Apple's mobile products (Noyes, 2011).

Design Goals

The goals of the Android system focus on the user experience in a mobile environment, including the use of touch screens, and connecting to networks either through tele- phony or Wi-Fi. The following design guidelines are taken directly from the Android website for app developers at https://developer.android.com/design/get-started/ creative-vision.html.

"Android design is shaped by three overarching goals for users that apply to apps as well as the system at large. As you work with Android, keep these goals in mind.

Enchant Me

Beauty is more than skin deep. Android apps are sleek and aesthetically pleas- ing on multiple levels. Transitions are fast and clear; layout and typography are crisp and meaningful. App icons are works of art in their own right. Just like a well-made tool, your app should strive to combine beauty, simplicity, and purpose to create a magical experience that is effortless and powerful.

Simplify My Life

Android apps make life easier and are easy to understand. When people use your app for the first time, they should intuitively grasp the most important features. The design work doesn't stop at the first use, though. Android apps remove ongoing chores like file management and syncing. Simple tasks never require complex procedures, and complex tasks are tailored to the human hand and mind. People of all ages and cultures feel firmly in control, and are never overwhelmed by too many choices or irrelevant flash.

Make Me Amazing

It's not enough to make an app that is easy to use. Android apps empower people to try new things and to use apps in inventive new ways. Android lets people combine applications into new workflows through multitasking, notifications, and sharing across apps. At the same time, your app should feel personal, giving people access to superb technology with clarity and grace." (Android Open Source Project, 2016)

Device Management

Because Android runs on a wide variety of mobile devices, apps must be designed to accommodate numerous devices, often without user help.

Screen Requirements

Developers creating apps for Android must account for a wide range of screen sizes and rotations, from wristwatches to large tablets with many variations in between. For example, two devices that claim to have a large size screen might have slightly different physical characteristics, including aspect ratios and actual pixel densities, which could complicate the job of app developers. To ease this burden, Android takes over the precise mechanics of matching apps with each device. Developers only need to know the general size they are targeting, without having to include specific instructions to accommodate each size variation on each physical device.

Android accommodates the screen sizes, densities (in dots per inch, dpi), and resolutions shown in Table 16.2.

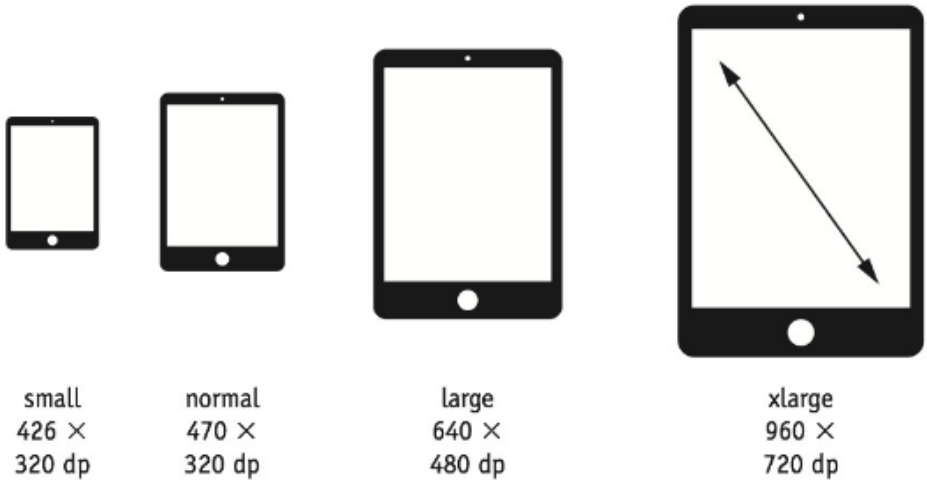
Design Factor	Sample Values	What It Means
Screen size	Phone: 3 in (5.1 cm) Tablet: 10 in (25 cm)	The screen's physical size as measured from one corner diagonally to the other.
Screen density	Phone: 164 dpi Tablet: 284 dpi	The number of pixels located within a certain area on the screen, such as a one-inch line.
Orientation	Portrait (vertical) Landscape (horizontal)	Two potential screen orientations as viewed by the user.
Resolution	Phone: 240×320 Tablet: 1280×800	The number of physical pixels.

(Table 16.2) Four device display variables that directly impact the design of user interface screens.

To aid designers, Android has introduced a fifth factor, called a density-independent pixel (dp). This is a virtual pixel, which is the equivalent to one physical pixel on a 160 dpi screen.

When designers create an app, they use the dp unit and let Android take care of making any small adjustments that might be required to run the app on a device that does not exactly fit the standard size. For example, if designers were to create a game using precise dpi units, they would have to develop code for every possible screen combination. But by writing the app using dp units, designers need to only create one game for an entire category of devices (small, normal, large, etc.) and that game can be expected to work successfully on every screen size within that category. Note that using dp units does not absolve the designer from stating in the app's manifest file the types of screens that are supported.

Given the wide variation in screen configurations, designers may choose to create four different user interfaces to accommodate Android's four general categories of screen sizes, which are shown in Figure 16.8.



(figure 16.8) A comparison of four general physical screen sizes supported by Android, not drawn to scale. The physical size is measured as the diagonal distance of the screen from corner to corner. Each screen is measured by a virtual measurement called a density-independent pixel (dp).

The ultimate goal of app designers is to give every user the impression that the app was designed specifically for that user's device, and not merely stretched or shrunk to accommodate various screens. Screen requirements and how to develop apps for them is a subject that's sure to change

Read

Understanding Operating Systems 8th Edition by Ann Mc-Hoes chapter 16 page 520-529 for more reading

frequently. For current details about user interface screen support, see Android's Best Practices at https://developer.android. com/guide/practices/index.html.

THINK POINT/CASE STUDY

Research Topics

A. The Macintosh OS X shown in this chapter is only one implementation of UNIX. Explore at least one other widespread operating system based on UNIX. Give details about the platform that your operating system runs, its networking capabilities, and its scalability to larger or newer hardware configurations. Cite your sources and the dates of their publications.

B. Research current literature to discover the current state of IEEE POSIX Standards and find out if the version of Windows on the computer that you use is currently 100 percent POSIX-compliant. Explain the significance of this compliance and why you think some popular operating systems choose not to be compliant.

C. Many distributions of Linux are available free of charge to individual users. First, using your own words, explain how you believe the operating system continues to flourish in spite of the fact that it generates little income for distributors. Second, research the answer and explain if your research affirms your speculation or not and give your explanation as to why this is the case. Cite your sources.

Review Questions

1. Assume that a file listing showed the following permissions: prwxr-xr-x. Answer the following questions:
   a. Is this a file or a directory?
   b. What permissions does the WORLD have?
   c. What permissions have been denied to the GROUP?

2. Without using jargon, explain the key advantages of a UNIX operating system as if you were explaining it to a high school student who is unfamiliar with computer-specific terminology.

3. Describe the importance of hardware independence to systems administrators. Explain the possible consequences if hardware was not device independent.

4. In your opinion, is Windows implemented more easily using a touch screen (such as on a tablet or phone), or on a desktop with no touchscreen. Explain your answer and include examples to prove your point.

5. What are the advantages of operating a Windows system using the menu- driven interface instead of typing commands? Give at least one example where having menus is an advantage.

6. Windows operating systems continue to allow the use of a command-driven interface similar to that introduced with MS-DOS long ago. In your opinion, when would a systems administrator find it helpful to use typed commands instead of menus? Give at least one example and explain your answer with- out using computer jargon.

7. In Linux, devices are identified by a major device number and a minor device number.
   a. Describe, in your own words, the primary differences between the two categories.
   b. List at least three types of devices that fall into each category.

8. Google owns Android as well as the Chrome operating system. Compare the similarities and differences between the two systems.

9. Compare and contrast the Android and Windows operating systems.

10. Explain in your own words why a software developer might want to request from the user only a minimum number of permissions for an app. Explain why another developer might request the maximum number of permissions. Give your opinion as to which is better for the user and which for the developers.