SENGAMALA THAYAAR EDUCATIONAL TRUST WOMEN'S COLLEGE
(Affiliated to Bharathidasan University, Tiruchirappalli)
Recognized by UGC under section 2(f) & 12(B)
(Accredited with 'A' Grade {CGPA: 3.45/4.00} by NAAC)
(An ISO 9001: 2015 Certified Institution)
SILVER JUBILEE YEAR (2019-2020)
CPG Certification Partner Global ISO 9001: 2015

COURSE MATERIAL

COURSE : B.Sc[CS]

TITLE OF THE PAPER : OPERATING SYSTEMS

PAPER CODE : 16SCCCS8

Prepared By,

Mrs.K.SAVIMA

STET WOMEN'S COLLEGE

STET WOMEN'S COLLEGE

Objective:

To provide the Fundamental Concepts in an Operating System.

Unit I Introducing Operating Systems

Introduction - What Is an Operating System-Operating System Software -A Brief History of Machine Hardware -Types of Operating Systems -Brief History of Operating System Development-Object-Oriented Design

Unit II Memory Management

Early Systems: Single-User Contiguous Scheme -Fixed Partitions-Dynamic Partitions- Best-Fit versus First-Fit Allocation -Deallocation - Relocatable Dynamic Partitions. Virtual Memory: Paged Memory Allocation-Demand Paging- Page Replacement Policies and Concepts -Segmented Memory Allocation-Segmented/Demand Paged Memory Allocation - Virtual Memory-Cache Memory

Unit III Processor Management

Overview-About Multi-Core Technologies-Job Scheduling Versus Process Scheduling- Process Scheduler-Process Scheduling Policies-Process Scheduling Algorithms -A Word About Interrupts-Deadlock-Seven Cases of Deadlock -Conditions for Deadlock- Modeling Deadlock-Strategies for Handling Deadlocks – Starvation-Concurrent Processes: What Is Parallel Processing-Evolution of Multiprocessors- Introduction to Multi-Core Processors-Typical Multiprocessing Configurations--Process Synchronization Software

Unit IV Device Management

Types of Devices-Sequential Access Storage Media-Direct Access Storage Devices-Magnetic Disk Drive Access Times- Components of the I/O Subsystem-Communication among Devices-Management of I/O Requests

Unit: V File Management

The File Manager -Interacting with the File Manager -File Organization - Physical Storage Allocation -Access Methods-Levels in a File Management System - Access Control Verification Module

Text Book:

1. Understanding Operating Systems, Ann McIver McHoes and Ida M. Flynn, Course Technology, Cengage Learning, 2011.

UNIT - I

Introducing Operating Systems

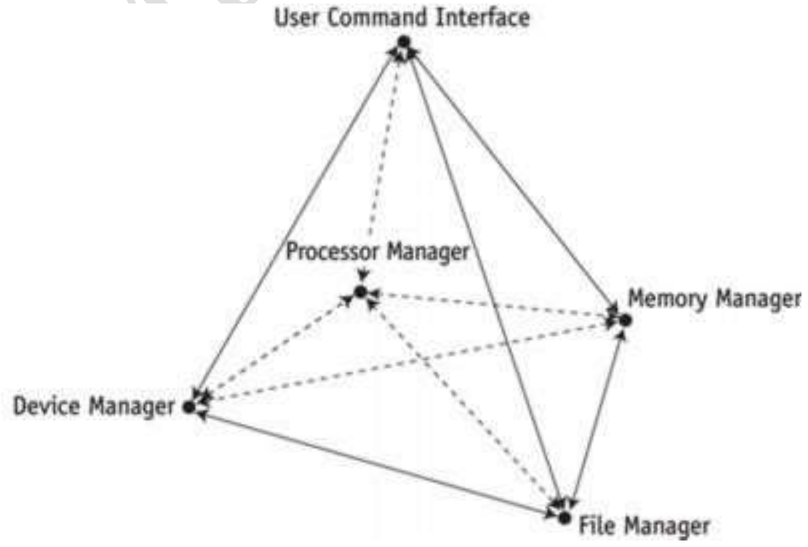
To understand an operating system is to understand the workings of an entire computer system, because the operating system manages each and every piece of hardware and software. This text explores what operating systems are, how they work, what they do, and why.

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

The pyramid shown in Figure 1.1 is an abstract representation of an operating system and demonstrates how its major components work together. At the base of the pyramid are the four essential managers of every operating system: the Memory Manager, the Processor Manager, the Device Manager, and the File Manager. In fact, these managers are the basis of all operating systems and each is discussed in detail throughout the first part of this book. Each manager works closely with the other managers and performs its unique role regardless of which specific operating system is being discussed. At the top of the pyramid is the User Interface, from which users issue commands to the operating system. This is the component that's unique to each operating system—sometimes even between different versions of the same operating system.

(figure 1.1)

This model of a non-networked operating system shows four subsystem managers supporting the User Command Interface.



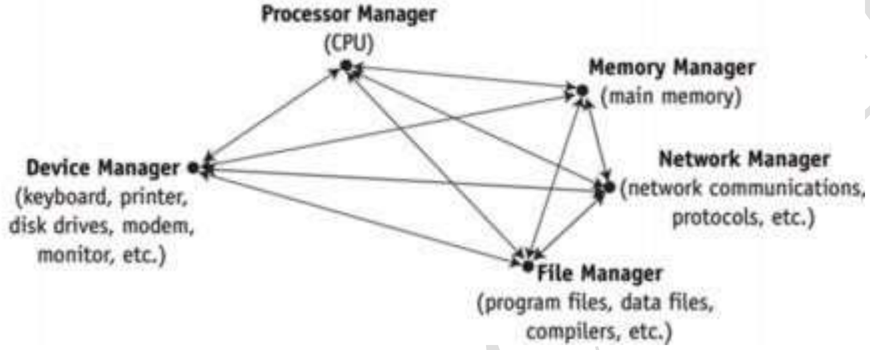
A network was not always an integral part of operating systems; early systems were self-contained with all network capability added on top of existing operating systems. Now most operating systems routinely incorporate a Network Manager. The base of a pyramid for a networked operating system is shown in Figure 1.2. Regardless of the size or configuration of the system, each of the subsystem managers, shown in Figure 1.3, must perform the following tasks:

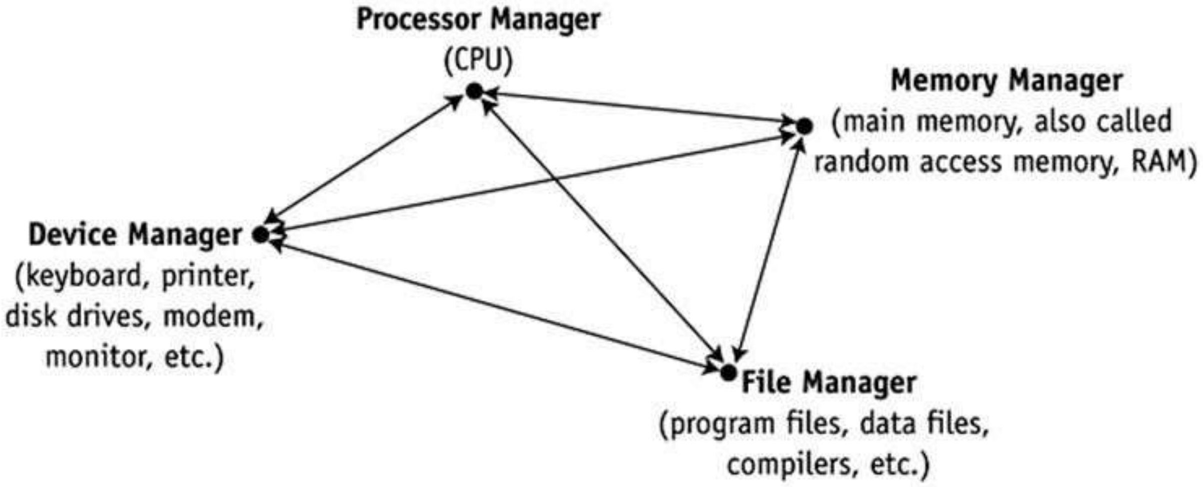
S MENS COLLEGE

- Monitor its resources continuously
- Enforce the policies that determine who gets what, when, and how much
- Allocate the resource when appropriate
- Deallocate the resource when appropriate

(figure 1.2)

Networked systems have a Network Manager that assumes responsibility for networking tasks while working harmoniously with every other manager.


Processor Manager
(CPU)
Memory Manager
(main memory)
Device Manager
(keyboard, printer, disk drives, modem, monitor, etc.)
Network Manager
(network communications, protocols, etc.)
File Manager
(program files, data files, compilers, etc.)


Processor Manager
(CPU)
Memory Manager
(main memory, also called random access memory, RAM)
Device Manager
(keyboard, printer, disk drives, modem, monitor, etc.)
File Manager
(program files, data files, compilers, etc.)

Main Memory Management

The Memory Manager (the subject of Chapters 2–3) is in charge of main memory, also 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, it allocates a portion of memory that isn't already in use. In a multiuser environment, the Memory Manager sets up a table to keep track of who is using which section of memory. Finally, when the time comes to reclaim the memory, the Memory Manager deallocates memory. A primary responsibility of the Memory Manager is to protect the space in main memory occupied by the operating system itself—it can't allow any part of it to be accidentally or intentionally altered.

Processor Management

The Processor Manager (the subject of Chapters 4–6) 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 process. A process is defined here as an instance of execution of a program. The Processor Manager monitors whether the CPU is executing a process or waiting for a READ or WRITE command to finish execution. Because it handles the processes' transitions from one state of execution to another, it can be compared to a traffic controller. Once the Processor Manager allocates the processor, it sets up the necessary registers and tables and, when the job is finished or the maximum amount of time has expired, it reclaims the processor.

Think of it this way: The Processor Manager has two levels of responsibility. One is to handle jobs as they enter the system and the other is to manage each process within those jobs. The first part is handled by the Job Scheduler, the high-level portion of the Processor Manager, which accepts or rejects the incoming jobs. The second part is handled by the Process Scheduler, the low-level portion of the Processor Manager, which is responsible for deciding which process gets the CPU and for how long.

Device Management

The Device Manager monitors every device, channel, and control unit. Its job is to choose the most efficient way to allocate all of the system's devices, printers, ports, disk drives, and so forth, based on a scheduling policy chosen by the system's designers. The Device Manager does this by allocating each resource, starting its operation, and, finally, deallocating the device, making it available to the next process or job.

File Management

The File Manager (the subject of Chapter 8) keeps track of every file in the system, including data files, program files, compilers, and applications. By using predetermined access policies, it enforces restrictions on who has access to which files. The File Manager also controls what users are allowed to do with files once they access them. For example, a user might have read-only access, read-and-write access, or the authority to create and delete files. Managing access control is a key part of file management. Finally, the File Manager allocates the necessary resources and later deallocates them.

Network Management

Operating systems with Internet or networking capability have a fifth essential manager called the Network Manager (the subject of Chapters 9–10) that provides a convenient way for users to share resources while controlling users' access to them. These resources include hardware (such as CPUs, memory areas, printers, tape drives, modems, and disk drives) and software (such as compilers, application programs, and data files). User Interface The user interface is the portion of the operating system that users interact with directly.

Cooperation Issues

However, it is not enough for each manager to perform its individual tasks. It must also be able to work harmoniously with every other manager. Here is a simplified example. Let's say someone chooses an option from a menu to execute a program. The following major steps must occur in sequence:

1. The Device Manager must receive the electrical impulses from the mouse or keyboard, form the command, and send the command to the User Interface, where the Processor Manager validates the command.
2. The Processor Manager then sends an acknowledgment message to be displayed on the monitor so the user realizes the command has been sent.
3. When the Processor Manager receives the command, it determines whether the program must be retrieved from storage or is already in memory, and then notifies the appropriate manager.
4. If the program is in storage, the File Manager must calculate its exact location on the disk and pass this information to the Device Manager, which retrieves the program and sends it to the Memory Manager.
5. The Memory Manager then finds space for it and records its exact location in memory. Once the program is in memory, the Memory Manager must track its location in memory (even if it's moved) as well as its progress as it's executed by the Processor Manager.
6. When the program has finished executing, it must send a finished message to the Processor Manager so that the processor can be assigned to the next program waiting in line.
7. Finally, the Processor Manager must forward the finished message to the Device Manager, so that it can notify the user and refresh the screen.

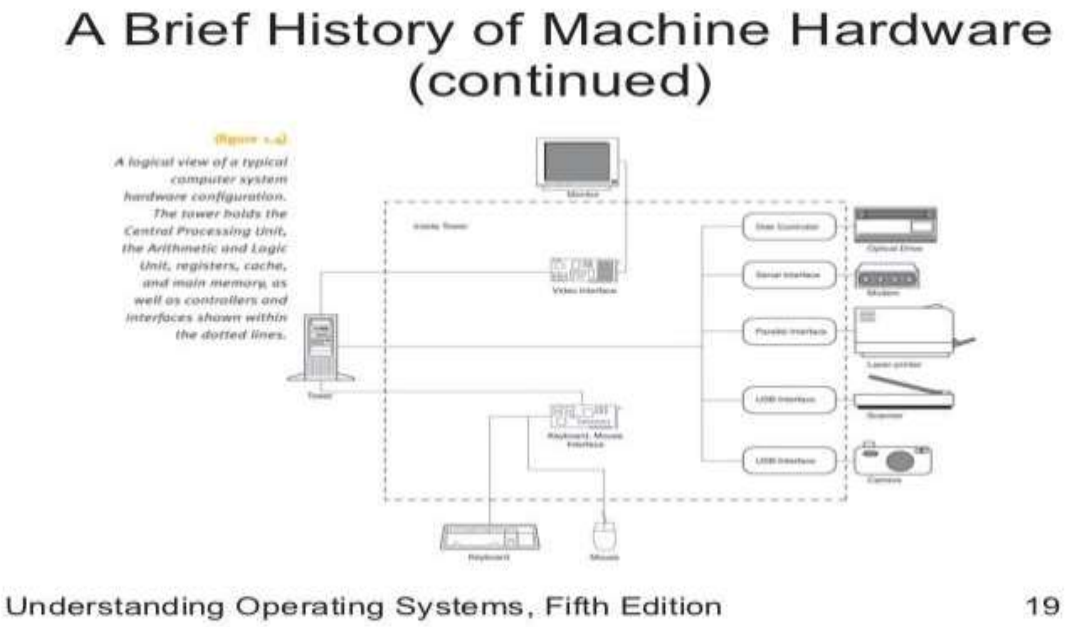
STET WOMEN'S COLLEGE

A Brief History of Machine Hardware

To appreciate the role of the operating system (which is software), we need to discuss the essential aspects of the computer system’s hardware, the physical machine and its electronic components, including memory chips, input/output devices, storage devices, and the central processing unit (CPU).

- Main memory (random access memory, RAM) is where the data and instructions must reside to be processed.
- I/O devices, short for input/output devices, include every peripheral unit in the system such as printers, disk drives, CD/DVD drives, flash memory, keyboards, and so on.
- The central processing unit (CPU) is the brains with the circuitry (sometimes called the chip) to control the interpretation and execution of instructions. In essence, it controls the operation of the entire computer system, as illustrated in Figure 1.5. All storage references, data manipulations, and I/O operations are initiated or performed by the CPU. Until the mid-1970s, computers were classified by capacity and price. A mainframe was a large machine— in size and in internal memory capacity. The IBM 360, introduced in 1964, is a classic example of an early mainframe. The IBM 360 model 30 required an air-conditioned room about 18 feet square to house the CPU, the operator’s console, a printer, a card reader, and a keypunch machine.

Figure: A logical view of a typical computer system hardware configuration. The tower holds the central processing unit, the arithmetic and logic unit, registers, cache, and main memory, as well as controllers and interfaces shown within the dotted lines.



A Brief History of Machine Hardware (continued)

A logical view of a typical computer system hardware configuration. The tower holds the Central Processing Unit, the Arithmetic and Logic Unit, registers, cache, and main memory, as well as controllers and interfaces shown within the dotted lines.

Understanding Operating Systems, Fifth Edition 19

The minicomputer was developed to meet the needs of smaller institutions, those with only a few dozen users. One of the early minicomputers was marketed by Digital Equipment Corporation to satisfy the needs of large schools and small colleges that began offering computer science courses in the early 1970s. (The price of its PDP-8 was less than $18,000.) Minicomputers are smaller in size and memory capacity and cheaper than mainframes. Today, computers that fall between microcomputers and mainframes in capacity are often called midrange computers.

The supercomputer was developed primarily for government applications needing massive and fast number-crunching ability to carry out military operations and weather forecasting. Business and industry became interested in the technology when the massive computers became faster and less expensive. A Cray supercomputer is a typical example with six to thousands of processors performing up to 2.4 trillion floating point operations per second. Supercomputers are used for a wide range of tasks from scientific research to customer support and product development. They're often used to perform the intricate calculations required to create animated motion pictures. And they help oil companies in their search for oil by analyzing massive amounts of data (Stair, 1999).

The microcomputer was developed to offer inexpensive computation capability to individual users in the late 1970s. Early models featured a revolutionary amount of memory: 64K. Their physical size was smaller than the minicomputers of that time, though larger than the microcomputers of today. Eventually, microcomputers grew to accommodate software with larger capacity and greater speed. The distinguishing characteristic of the first microcomputer was its single-user status.

Powerful microcomputers developed for use by commercial, educational, and government enterprises are called workstations. Typically, workstations are networked together and are used to support engineering and technical users who perform massive mathematical computations or computer-aided design (CAD), or use other applications requiring very powerful CPUs, large amounts of main memory, and extremely high-resolution graphic displays to meet their needs.

Servers are powerful computers that provide specialized services to other computers on client/server networks. Examples can include print servers, Internet servers, e-mail servers, etc. Each performs critical network tasks. For instance, a file server, usually a powerful computer with substantial file storage capacity (such as a large collection of hard drives), manages file storage and retrieval for other computers, called clients, on the network.

Types of Operating Systems

Operating systems for computers large and small fall into five categories distinguished by response time and how data is entered into the system: batch, interactive, real-time, hybrid, and embedded systems.

Batch systems date from the earliest computers, when they relied on stacks of punched cards or reels of magnetic tape for input. Jobs were entered by assembling the cards into a deck and running the entire deck of cards through a card reader as a group—a batch. The efficiency of a batch system is measured in throughput—the number of jobs completed in a given amount of time (for example, 550 jobs per hour).

Interactive systems give a faster turnaround than batch systems but are slower than the real-time systems we talk about next. They were introduced to satisfy the demands of users who needed fast turnaround when debugging their programs. The operating system required the development of time-sharing software, which would allow each user to interact directly with the computer system via commands entered from a typewriter-like terminal. The operating system provides immediate feedback to the user and response time can be measured in fractions of a second. Real-time systems are used in time-critical environments where (though it often is), but system response time must meet the deadline or risk significant consequences.

There are two types of real-time systems depending on the consequences of missing the deadline:

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

STET WOMENS COLLEGE

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

Hybrid systems are a combination of batch and interactive. They appear to be interactive because individual users can access the system and get fast responses, but such a system actually accepts and runs 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. Many large computer systems are hybrids.

Embedded systems are computers placed inside other products to add features and capabilities. For example, you find embedded computers in household appliances, automobiles, digital music players, elevators, and pacemakers. In the case of automobiles, embedded computers can help with engine performance, braking, and navigation. For example, several projects are under way to implement “smart roads,” which would alert drivers in cars equipped with embedded computers to choose alternate routes when traffic becomes congested.

Brief History of Operating System Development

The evolution of operating system software parallels the evolution of the computer hardware it was designed to control. Here’s a very brief overview of this evolution.

1940s

The first generation of computers (1940–1955) was a time of vacuum tube technology and computers the size of classrooms. Each computer was unique in structure and purpose. There was little need for standard operating system software because each computer’s use was restricted to a few professionals working on mathematical, scientific, or military applications, all of whom were familiar with the idiosyncrasies of their hardware. A typical program would include every instruction needed by the computer to perform the tasks requested.

To run programs, the programmers would have to reserve the machine for the length of time they estimated it would take the computer to execute the program. As a result, the machine was poorly utilized. The CPU processed data and made calculations for only a fraction of the available time and, in fact, the entire system sat idle between reservations.

1950s

Second-generation computers (1955–1965) were developed to meet the needs of new markets—government and business researchers. The business environment placed much more importance on the cost effectiveness of the system. Computers were still very expensive, especially when compared to other office equipment (the IBM 7094 was priced at $200,000). Therefore, throughput had to be maximized to make such an investment worthwhile for business use, which meant dramatically increasing the usage of the system.

Two improvements were widely adopted: Computer operators were hired to facilitate each machine’s operation, and job scheduling was instituted. Job scheduling is a productivity improvement scheme that groups together programs with similar requirements.

Job scheduling introduced the need for control cards, which defined the exact nature of each program and its requirements. This was one of the first uses of a job control language, which helped the operating system

STET WOMEN'S COLLEGE

coordinate and manage the system resources by identifying the users and their jobs and specifying the resources required to execute each job.

Eventually, several factors helped improve the performance of the CPU:

- First, the speeds of I/O devices such as drums, tape drives, and disks gradually increased.
- Second, to use more of the available storage area in these devices, records were grouped into blocks before they were retrieved or stored. (This is called blocking, meaning that several logical records are grouped within one physical record, and is discussed in detail in Chapter 7.)
- Third, to reduce the discrepancy in speed between the I/O and the CPU, an interface called the control unit was placed between them to act as a buffer. A buffer is an interim storage area that works as a temporary holding place. As the slow input device reads one record, the control unit places each character of the record into the buffer. When the buffer is full, the entire record is quickly transmitted to the CPU. The process is just the opposite for output devices: The CPU places the entire record into the buffer, which is then passed on by the control unit at the slower rate required by the output device.

The buffers of this generation were conceptually similar to those now used routinely by Internet browsers to make video and audio playback smoother.

During the second generation, programs were still assigned to the processor one at a time. The next step toward better use of the system's resources was the move to shared processing.

1960s

Third-generation computers date from the mid-1960s. They were designed with faster CPUs, but their speed still caused problems when they interacted with printers and other I/O devices that ran at slower speeds. The solution was multiprogramming, which introduced the concept of loading many programs at one time and sharing the attention of a single CPU.

The first multiprogramming systems allowed each program to be serviced in turn, one after another. The most common mechanism for implementing multiprogramming was the introduction of the concept of the interrupt, whereby the CPU was notified of events needing operating system services. For example, when a program issued a print command (called an input/output command or an I/O command), it generated an interrupt requesting the services of the I/O processor and the CPU was released to begin execution of the next job. This was called passive multiprogramming.

Program scheduling, which was begun with second-generation systems, continued at this time but was complicated by the fact that main memory was occupied by many jobs. To solve this problem, the jobs were sorted into groups and then loaded into memory according to a preset rotation formula. The sorting was often determined by priority or memory requirements—whichever was found to be the most efficient use of the available resources. In addition to scheduling jobs, handling interrupts, and allocating memory, the operating systems also had to resolve conflicts whenever two jobs requested the same device at the same time.

1970s

STET WOMEN'S COLLEGE

After the third generation, during the late 1970s, computers had faster CPUs, creating an even greater disparity between their rapid processing speed and slower I/O access time. The first Cray supercomputer was released in

1976. Multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive.

A solution to this physical limitation was the development of virtual memory, which took advantage of the fact that the CPU could process only one instruction at a time. With virtual memory, the entire program didn't need to reside in memory before execution could begin. A system with virtual memory would divide the programs into parts and keep them in secondary storage, bringing each part into memory only as it was needed. (Programmers of second-generation computers had used this concept with the roll in/roll out programming method, also called overlay, to execute programs that exceeded the physical memory of those computers.) At this time there was also growing attention to the need for data resource conservation. Database management software became a popular tool because it organized data in an integrated manner, minimized redundancy, and simplified updating and access of data.

A number of query systems were introduced that allowed even the novice user to retrieve specific pieces of the database. These queries were usually made via a terminal, which in turn mandated a growth in terminal support and data communication software.

1980s

Development in the 1980s dramatically improved the cost/performance ratio of computer components. Hardware was more flexible, with logical functions built on easily replaceable circuit boards. And because it was less costly to create these circuit boards, more operating system functions were made part of the hardware itself, giving rise to a new concept—firmware, a word used to indicate that a program is permanently held in read-only memory (ROM), as opposed to being held in secondary storage. The job of the programmer, as it had been defined in previous years, changed dramatically because many programming functions were being carried out by the system's software, hence making the programmer's task simpler and less hardware dependent.

Eventually the industry moved to multiprocessing (having more than one processor), and more complex languages were designed to coordinate the activities of the multiple processors servicing a single job. As a result, it became possible to execute programs in parallel, and eventually operating systems for computers of every size were routinely expected to accommodate multiprocessing.

With network operating systems, users generally became aware of the existence of many networked resources, could log in to remote locations, and could manipulate files on networked computers distributed over a wide geographical area. Network operating systems were similar to single-processor operating systems in that each machine ran its own local operating system and had its own users. The difference was in the addition of a network interface controller with low-level software to drive the local operating system, as well as programs to allow remote login and remote file access. Still, even with these additions, the basic structure of the network operating system was quite close to that of a standalone system.

1990s

STET WOMEN'S COLLEGE

Web accessibility and e-mail became standard features of almost every operating system. However, increased networking also sparked increased demand for tighter security to protect hardware and software.

The decade also introduced a proliferation of multimedia applications demanding additional power, flexibility, and device compatibility for most operating systems. A typical multimedia computer houses devices to perform audio, video, and graphic creation and editing. Those functions can require many specialized devices such as a microphone, digital piano, Musical Instrument Digital Interface (MIDI), digital camera, digital video disc (DVD) drive, optical disc (CD) drives, speakers, additional monitors, projection devices, color printers, and high-speed Internet connections. These computers also require specialized hardware (such as controllers, cards, busses) and software to make them work together properly.

Multimedia applications need large amounts of storage capability that must be managed gracefully by the operating system. For example, each second of a 30-frame-perminute full-screen video requires 27MB of storage unless the data is compressed in some way. To meet the demand for compressed video, special-purpose chips and video boards have been developed by hardware companies. What's the effect of these technological advances on the operating system? Each advance requires a parallel advance in the software's management capabilities.

2000s

The new century emphasized the need for operating systems to offer improved flexibility, reliability, and speed. To meet the need for computers that could accommodate multiple operating systems running at the same time and sharing resources, the concept of virtual machines, shown in Figure 1.13, was developed and became commercially viable. Virtualization is the creation of partitions on a single server, with each partition supporting a different operating system. In other words, it turns a single physical server into multiple virtual servers, often with multiple operating systems.

Virtualization requires the operating system to have an intermediate manager to oversee each operating system's access to the server's physical resources. For example, with virtualization, a single processor can run 64 independent operating systems on workstations using a processor capable of allowing 64 separate threads (instruction sequences) to run at the same time.

Until recent years, the silicon wafer that forms the base of the computer chip circuitry held only a single CPU. However, with the introduction of dual-core processors, a single chip can hold multiple processor cores. Thus, a dual-core chip allows two sets of calculations to run at the same time, which sometimes leads to faster completion of the job. It's as if the user has two separate computers, and two processors, cooperating on a single task. As of this writing, designers have created chips that can hold 80 simple cores.

Threads

Multi-core technology helps the operating system handle threads, multiple actions that can be executed at the same time. First, an explanation: The Processor Manager is responsible for processing each job submitted by a user. Jobs are made up of processes (sometimes called tasks in other textbooks), and processes consist of multiple threads. A process has two characteristics:

STET WOMEN'S COLLEGE

- It requires space in main memory where it resides during its execution; although, from time to time, it requires other resources such as data files or I/O devices.

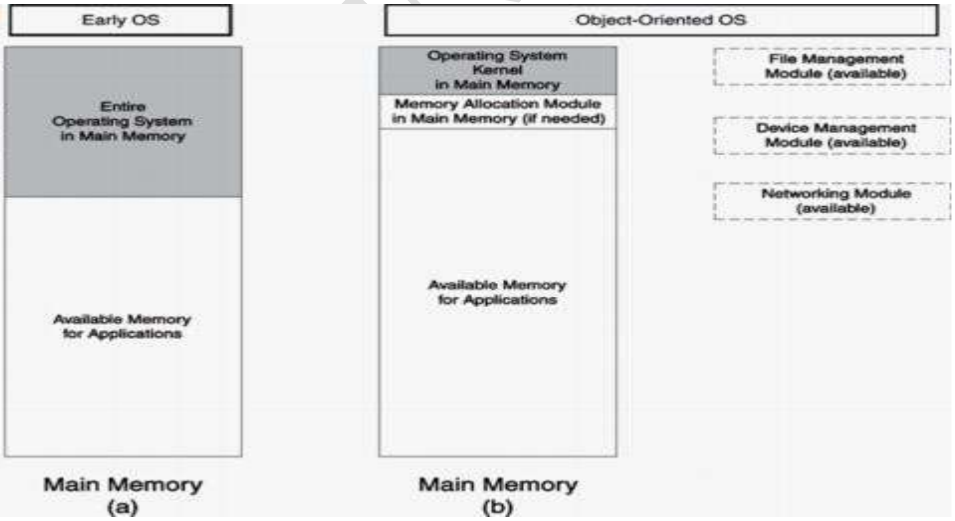
- It passes through several states (such as running, waiting, ready) from its initial arrival into the computer system to its completion. Multiprogramming and virtual memory dictate that processes be swapped between main memory and secondary storage during their execution. With conventional processes (also known as heavyweight processes), this swapping results in a lot of overhead. That's because each time a swap takes place, all process information must be saved to preserve the process's integrity.

A thread (or lightweight process) can be defined as a unit smaller than a process, which can be scheduled and executed. Using this technique, the heavyweight process, which owns the resources, becomes a more passive element, while a thread becomes the element that uses the CPU and is scheduled for execution.

Manipulating threads is less time consuming than manipulating processes, which are more complex. Some operating systems support multiple processes with a single thread, while others support multiple processes with multiple threads. Multithreading allows applications to manage a separate process with several threads of control. Web browsers use multithreading routinely. For instance, one thread can retrieve images while another sends and retrieves e-mail. Multithreading is also used to increase responsiveness in a time-sharing system to increase resource sharing and decrease overhead

Object-Oriented Design

An important area of research that resulted in substantial efficiencies was that of the system architecture of operating systems—the way their components are programmed and organized.


Early OS
Entire Operating System in Main Memory
Available Memory for Applications
Main Memory (a)
Object-Oriented OS
Operating System Kernel in Main Memory
Memory Allocation Module in Main Memory (if needed)
Available Memory for Applications
File Management Module (available)
Device Management Module (available)
Networking Module (available)
Main Memory (b)

(figure 1.13)

Early operating systems (a) loaded in their entirety into main memory. Modern object-oriented systems (b) load only the critical elements into main memory and call other objects as needed.

specifically the use of object-oriented design and the reorganization of the operating system's nucleus, the kernel. The kernel is the part of the operating system that resides in memory at all times, performs the most essential operating system tasks, and is protected by hardware from user tampering.

The first operating systems were designed as a comprehensive single unit. They stored all required elements of the operating system in memory such as memory allocation, process scheduling, device allocation, and file management. This type of architecture made it cumbersome and time consuming for programmers to add new components to the operating system, or to modify existing ones.

STE N'S COLLEGE

Most recently, the part of the operating system that resides in memory has been limited to a few essential functions, such as process scheduling and memory allocation, while all other functions, such as device allocation, are provided by special modules, which are treated as regular applications.

Object-oriented design was the driving force behind this new organization. Objects are self-contained modules (units of software) that provide models of the real world and can be reused in different applications. By working on objects, programmers can modify and customize pieces of an operating system without disrupting the integrity of the remainder of the system. In addition, using a modular, object-oriented approach can make software development groups more productive than was possible with procedural structured programming.

STET WOMEN'S COLLEGE

UNIT-II

Memory management

In operating systems, **memory management** is the function responsible for managing the computer's primary memory. The memory management function keeps track of the status of each memory location, either *allocated* or *free*. It determines how memory is allocated among competing processes, deciding which gets memory, when they receive it, and how much they are allowed. When memory is allocated it determines which memory locations will be assigned. It tracks when memory is freed or *unallocated* and updates the status.

The management of main memory is critical. In fact, from a historical perspective, the performance of the entire system has been directly dependent on two things: How much memory is available and how it is optimized while jobs are being processed. Pictured in Figure 2.1 is a main memory circuit from 1961. Since then, the physical size of memory units has become increasingly small and they are now available on small boards.

These early memory management schemes are seldom used by today's operating systems, but they are important to study because each one introduced fundamental concepts that helped memory management.

Four types of memory allocation schemes: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions.

Single-User Contiguous Scheme

The first memory allocation scheme worked like this: Each program to be processed was loaded in its entirety into memory and allocated as much contiguous space in memory as it needed, as shown in Figure 2.2. The key words here are entirety and contiguous. If the program was too large and didn't fit the available memory space, it couldn't be executed. And, although early computers were physically large, they had very little memory

This demonstrates a significant limiting factor of all computers—they have only a finite amount of memory and if a program doesn't fit, then either the size of the main memory must be increased or the program must be modified. It's usually modified by making it smaller or by using methods that allow program segments (partitions made to the program) to be overlaid. (To overlay is to transfer segments of a program from secondary storage into main memory for execution, so that two or more segments take turns occupying the same memory locations.)

Single-user systems in a non networked environment work the same way. Each user is given access to all available main memory for each job, and jobs are processed sequentially, one after the other. To allocate memory, the operating system uses a simple algorithm (step-by-step procedure to solve a problem):

Algorithm to Load a Job in a Single-User System

1 Store first memory location of program into base register (for memory protection)

2 Set program counter (it keeps track of memory space used by the program) equal to address of first memory location

3 Read first instruction of program

4 Increment program counter by number of bytes in instruction

5 Has the last instruction been reached? if yes, then stop loading program if no, then continue with step 6 6 Is program counter greater than memory size? if yes, then stop loading program if no, then continue with step 7

STET WOMEN'S COLLEGE

7 Load instruction in memory

8 Read next instruction of program

9 Go to step 4

Notice that the amount of work done by the operating system's Memory Manager is minimal, the code to perform the functions is straightforward, and the logic is quite simple. Only two hardware items are needed: a register to store the base address and an accumulator to keep track of the size of the program as it's being read into memory. Once the program is entirely loaded into memory, it remains there until execution is complete, either through normal termination or by intervention of the operating system.

Fixed Partitions

The first attempt to allow for multiprogramming used fixed partitions (also called static partitions) within the main memory—one partition for each job. Because the size of each partition was designated when the system was powered on, each partition could only be reconfigured when the computer system was shut down, reconfigured, and restarted. Thus, once the system was in operation the partition sizes remained static.

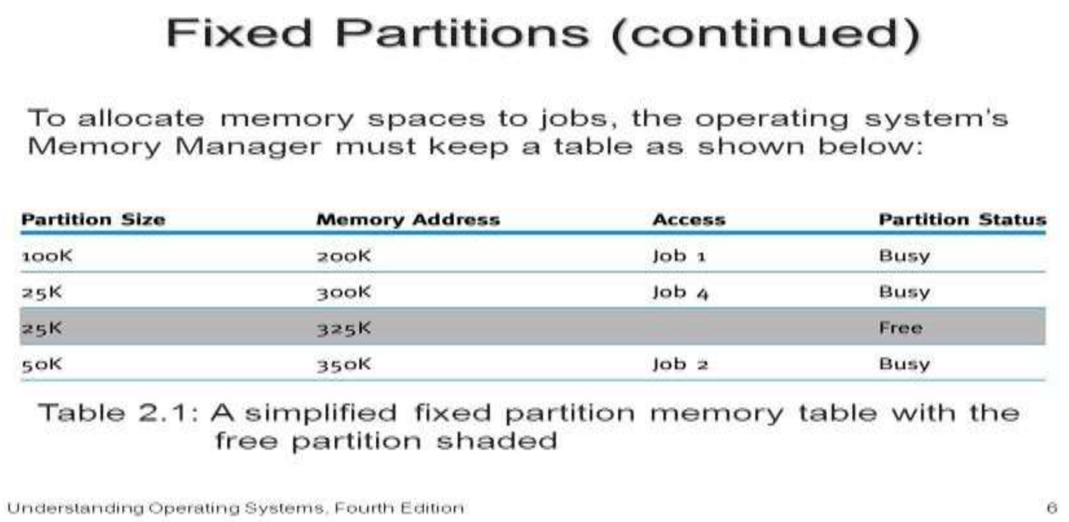
The algorithm used to store jobs in memory requires a few more steps than the one used for a single-user system because the size of the job must be matched with the size of the partition to make sure it fits completely. Then, when a block of sufficient size is located, the status of the partition must be checked to see if it's available.

Algorithm to Load a Job in a Fixed Partition

```
1 Determine job's requested memory size
2 If job_size > size of largest partition Then reject the job print appropriate message to operator go to step 1 to handle next job in line Else continue with step 3
3 Set counter to 1
4 Do while counter <= number of partitions in memory If job_size > memory_partition_size(counter) Then counter = counter + 1 Else If memory_partition_size(counter) = "free" Then load job into memory_partition(counter) change memory_partition_status(counter) to "busy" go to step 1 to handle next job in line Else counter = counter + 1 End do
5 No partition available at this time, put job in waiting queue
6 Go to step 1 to handle next job in line
```

This partition scheme is more flexible than the single-user scheme because it allows several programs to be in memory at the same time. However, it still requires that the entire program be stored contiguously and in memory from the beginning to the end of its execution. In order to allocate memory spaces to jobs, the operating system's Memory Manager must keep a table, such as Table 2.1, which shows each memory partition size, its address, its access restrictions, and its current status (free or busy) for the system illustrated in Figure 2.3.
As each job terminates, the status of its memory partition is changed from busy to free so an incoming job can be assigned to that partition.

STET WOMENS COLLEGE



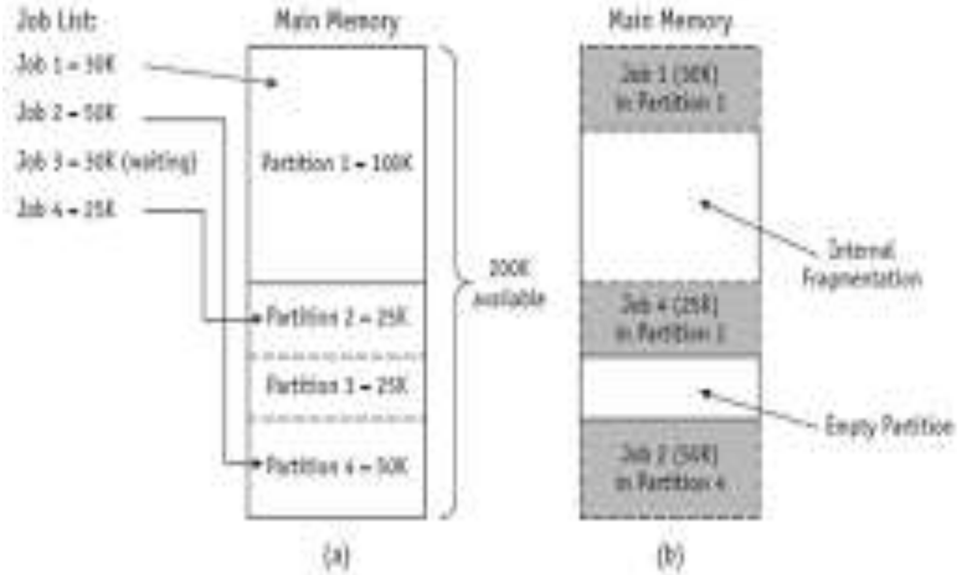
Fixed Partitions (continued)

To allocate memory spaces to jobs, the operating system's Memory Manager must keep a table as shown below:

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy

Table 2.1: A simplified fixed partition memory table with the free partition shaded

Understanding Operating Systems, Fourth Edition 6



(figure 2.1)

Main memory use during fixed partition allocation of Table 2.1. Job 3 must wait even though 70K of free space is available in Partition 1, where Job 1 only occupies 30K of the 100K available. The jobs are allocated space on the basis of "first available partition of required size."

The fixed partition scheme works well if all of the jobs run on the system are of the same size or if the sizes are known ahead of time and don't vary between reconfigurations. Ideally, that would require accurate advance knowledge of all the jobs to be run on the system in the coming hours, days, or weeks. However, unless the operator can accurately predict the future, the sizes of the partitions are determined in an arbitrary fashion and they might be too small or too large for the jobs coming in.

There are significant consequences if the partition sizes are too small; larger jobs will be rejected if they're too big to fit into the largest partitions or will wait if the large partitions are busy. As a result, large jobs may have a longer turnaround time as they wait for free partitions of sufficient size or may never run.

On the other hand, if the partition sizes are too big, memory is wasted. If a job does not occupy the entire partition, the unused memory in the partition will remain idle; it can't be given to another job because each partition is allocated to only one job at a time. It's an indivisible unit. Figure 2.3 demonstrates one such circumstance.

This phenomenon of partial usage of fixed partitions and the coinciding creation of unused spaces within the partition is called internal fragmentation, and is a major drawback to the fixed partition memory allocation scheme.

Dynamic Partitions

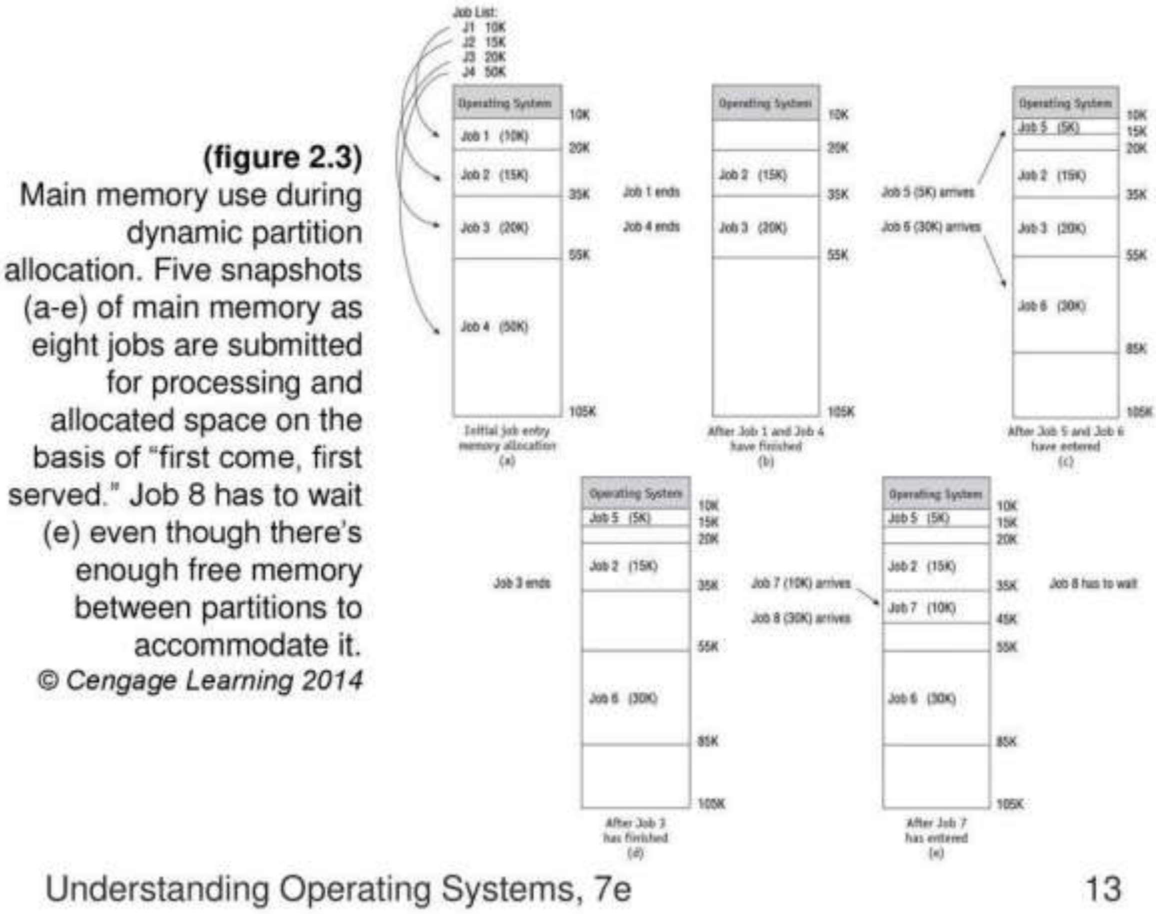
With dynamic partitions, available memory is still kept in contiguous blocks but jobs are given only as much memory as they request when they are loaded for processing. Although this is a significant improvement over fixed partitions because memory isn't wasted within the partition, it doesn't entirely eliminate the problem.

As shown in Figure 2.4, a dynamic partition scheme fully utilizes memory when the first jobs are loaded. But as new jobs enter the system that are not the same size as those that just vacated memory, they are fit into the available spaces on a priority basis.

Figure 2.4 demonstrates first-come, first-served priority. Therefore, the subsequent allocation of memory creates fragments of free memory between blocks of allocated memory. This problem is called external fragmentation and, like internal fragmentation, lets memory go to waste.

In the last snapshot, (e) in Figure 2.4, there are three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which only requires 30K. However they are not contiguous and, because the jobs are loaded in a contiguous manner, this scheme forces Job 8 to wait.

STET WOMEN'S COLLEGE



(figure 2.3)
Main memory use during dynamic partition allocation. Five snapshots (a-e) of main memory as eight jobs are submitted for processing and allocated space on the basis of "first come, first served." Job 8 has to wait (e) even though there's enough free memory between partitions to accommodate it.
© Cengage Learning 2014

Understanding Operating Systems, 7e 13

Best-Fit Versus 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.

These partitions may be allocated on the basis of first-fit memory allocation (first partition fitting the requirements) or best-fit memory allocation (least wasted space, the smallest partition fitting the requirements). For both schemes, the Memory Manager organizes the memory lists of the free and used partitions (free/busy) either by size or by location. The best-fit allocation method keeps the free/busy lists in order by size, smallest to largest.

The first-fit method keeps the free/busy lists organized by memory locations, low-order memory to high-order memory. Each has advantages depending on the needs of the particular allocation scheme— best-fit usually makes the best use of memory space; first-fit is faster in making the allocation.

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. When they're organized by location, the spaces closest to your lending desk are at the top of the list and the areas farthest away are at the bottom. Which option is best? It depends on what you want to optimize: space or speed of allocation. 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.

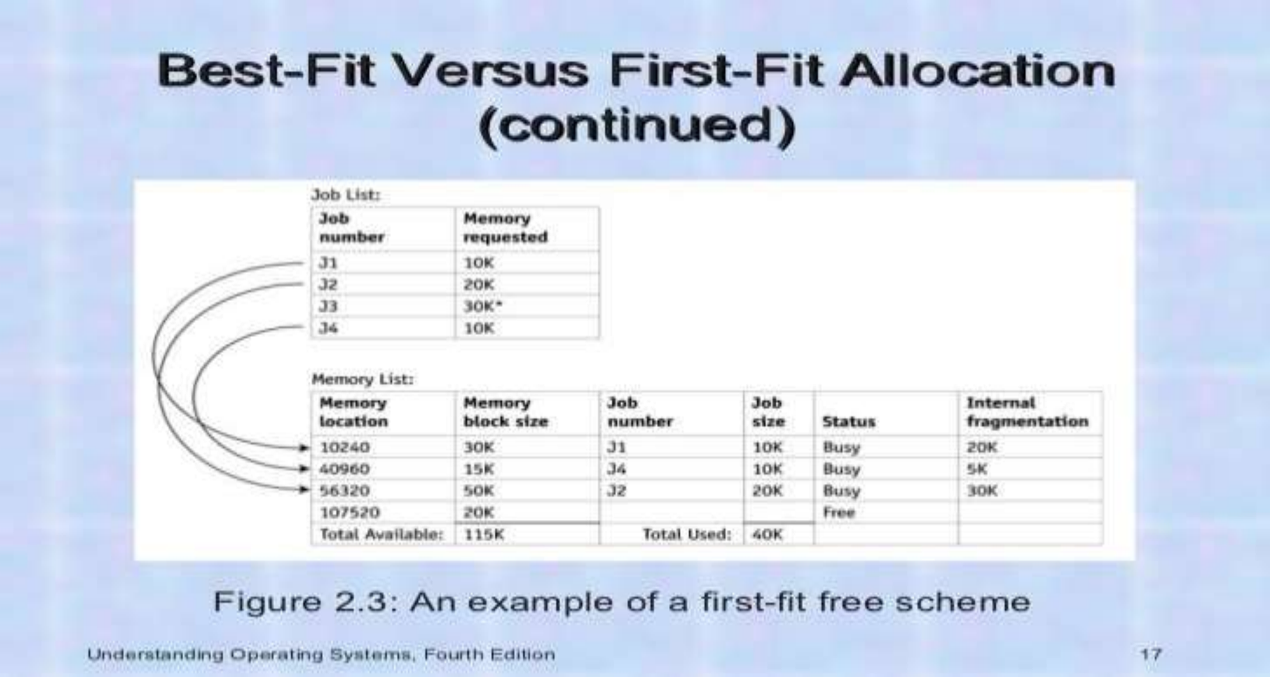
If a paperback is returned, you'll place it on a shelf with the other paperbacks or at least with other small books. Similarly, oversized books will be shelved with other large books. Your lists make it easy to find the smallest available empty space where the book can fit. The disadvantage of this system 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 on the shelves. This is a first-fit scheme. This system ignores the size of the book that you're trying to put away.

Figure 2.5 shows how a large job can have problems with a first-fit memory allocation list. Jobs 1, 2, and 4 are able to enter the system and begin execution; Job 3 has to wait even though, if all of the fragments of memory were added together, there would be more than enough room to accommodate it. 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 2.6. In this particular case, a best-fit scheme would yield better memory utilization.

Figure 2.5



Best-Fit Versus First-Fit Allocation (continued)

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 2.3: An example of a first-fit free scheme

Understanding Operating Systems, Fourth Edition 17

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 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 Memory Manager moves out of the loop to fetch 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.

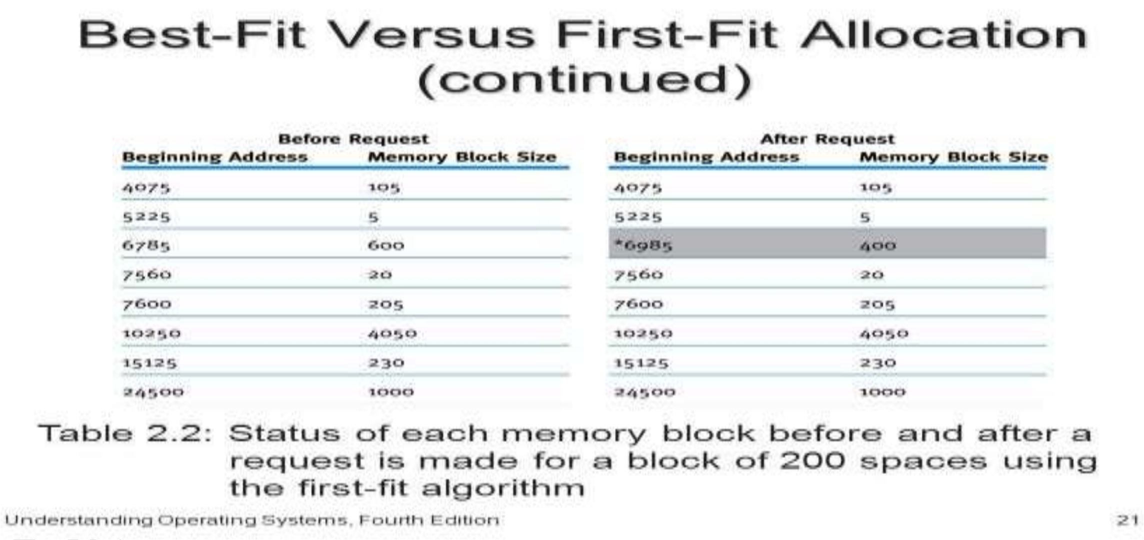
COLLEGE

The algorithms for best-fit and first-fit are very different. Here's how first-fit is implemented: First-Fit Algorithm

```
 1 Set counter to 1
2 Do while counter <= number of blocks in memory
If job_size > memory_size(counter)
     Then counter = counter + 1
   Else
      load job into memory_size(counter)
      adjust free/busy memory lists
      go to step 4
     End do
3 Put job in waiting queue
 4 Go fetch next job
```

In Table 2.2, 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 algorithm and starting from the top of the list, the Memory Manager locates the first block of memory large enough to accommodate the job, which is at location 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).

Table 2.2 These two snapshots of memory show the status of each memory block before and after a request is made using the first-fit algorithm. (Note: All values are in decimal notation unless otherwise indicated.)



Best-Fit Versus First-Fit Allocation (continued)

Before Request		After Request	
Beginning Address	Memory Block Size	Beginning Address	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 2.2: Status of each memory block before and after a request is made for a block of 200 spaces using the first-fit algorithm

Understanding Operating Systems, Fourth Edition 21

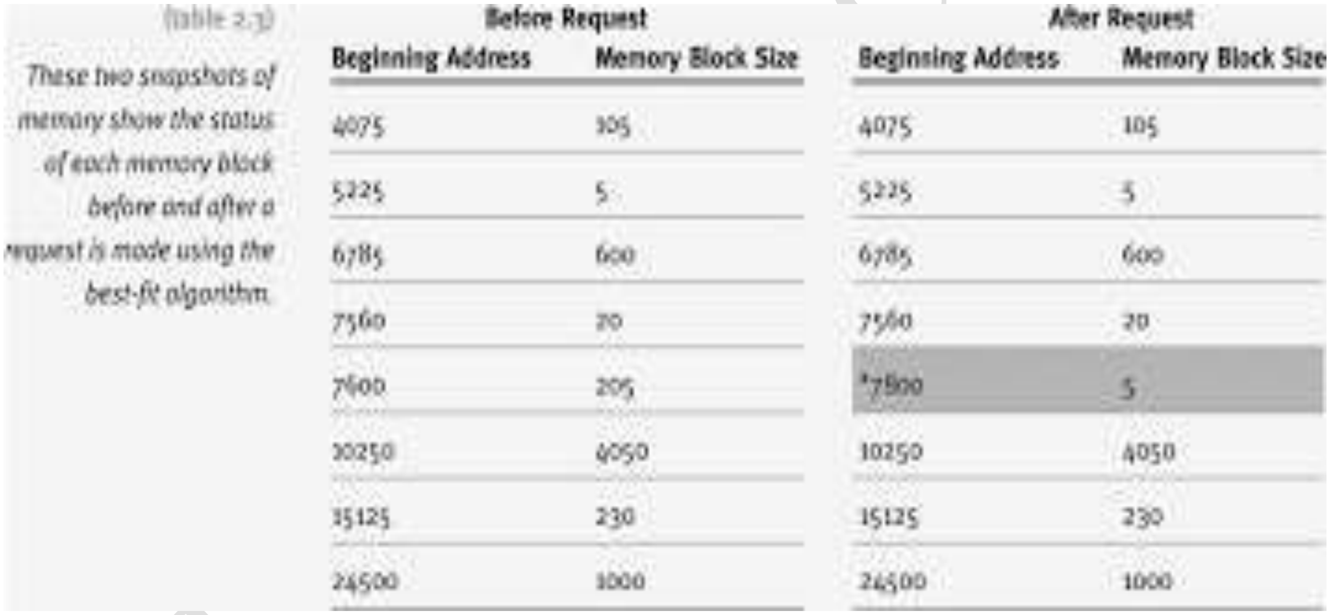
The algorithm for best-fit is slightly more complex because the goal is to find the smallest memory block into which the job will fit:

Best-Fit Algorithm

```
 1 Initialize memory_block(0) = 99999
2 Compute initial_memory_waste = memory_block(0) – job_size
3 Initialize subscript = 0
 4 Set counter to 1
5 Do while counter <= number of blocks in memory
If job_size > memory_size(counter)
```

```
Then counter = counter + 1
 Else
memory_waste = memory_size(counter) – job_size
If initial_memory_waste > memory_waste
Then subscript = counter
initial_memory_waste = memory_waste
counter = counter + 1
End do
6 If subscript = 0
Then put job in waiting queue
Else
load job into memory_size(subscript)
 adjust free/busy memory lists
7 Go fetch next job
```

The best-fit algorithm is illustrated showing only the list of free memory blocks. Table 2.3 shows the free list before and after the best-fit block has been allocated to the same request presented in Table 2.2.



(table 2.3)

These two snapshots of memory show the status of each memory block before and after a request is made using the best-fit algorithm.

Before Request		After Request	
Beginning Address	Memory Block Size	Beginning Address	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

In Table 2.3, 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 Memory Manager searches the entire list and locates a block of memory starting at location 7600, which is the smallest block that's 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).

In recent years, access times have become so fast that the scheme that saves the more valuable resource, memory space, may be the best in some cases. Research continues to focus on finding the optimum allocation scheme. This

STET COLLEGE

includes optimum page size— a fixed allocation scheme we will cover in the next chapter, which is the key to improving the performance of the best-fit allocation scheme.

Deallocation

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager resets the status of the memory block where the job was stored to "free." Any code—for example, binary values with 0 indicating free and 1 indicating busy—may be used so the mechanical task of deallocating a block of memory is relatively simple.

A dynamic partition system uses a more complex algorithm because the algorithm tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations:

- Case 1. When the block to be deallocated is adjacent to another free block
- Case 2. When the block to be deallocated is between two free blocks
- Case 3. When the block to be deallocated is isolated from other free blocks

The deallocation algorithm must be prepared for all three eventualities with a set of nested conditionals. The following algorithm is based on the fact that memory locations are listed using a lowest-to-highest address scheme. The algorithm would have to be modified to accommodate a different organization of memory locations. In this algorithm, job_size is the amount of memory being released by the terminating job, and beginning_address is the location of the first instruction for the job.

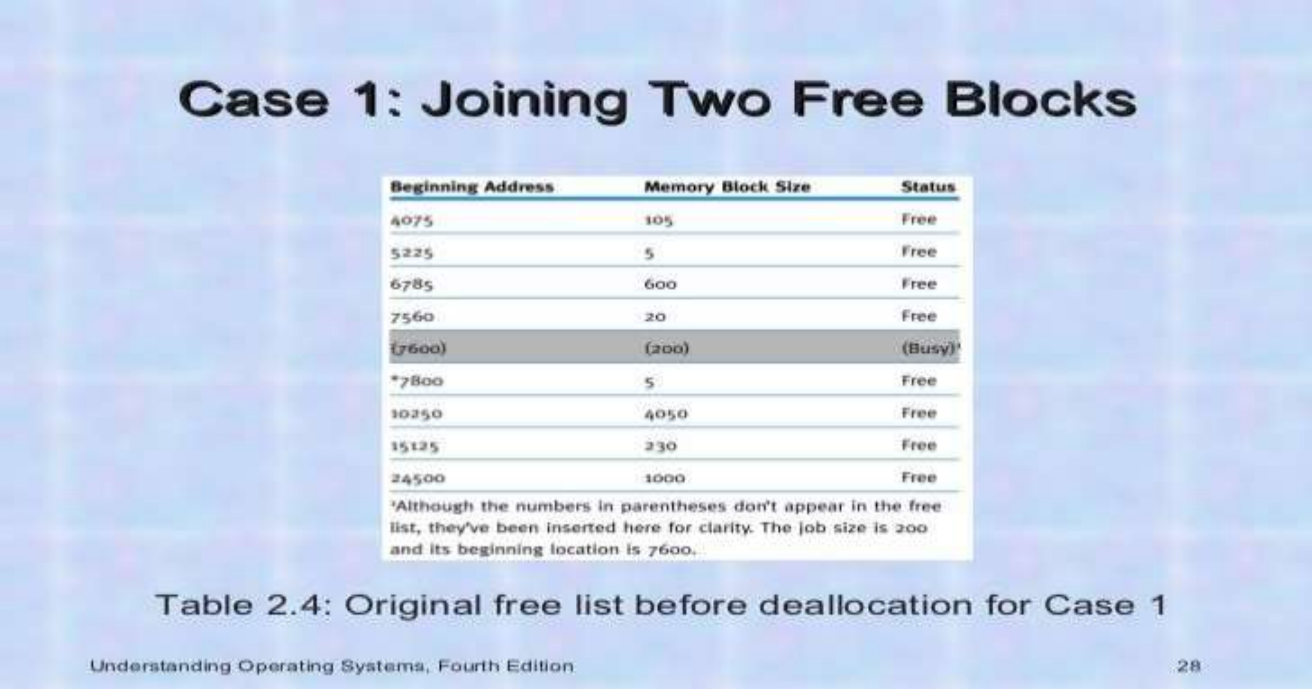
Algorithm to Deallocate Memory Blocks

```
If job_location is adjacent to one or more free blocks
Then
 If job_location is between two free blocks
Then merge all three blocks into one block
    memory_size(counter-1) = memory_size(counter-1) + job_size + memory_size(counter+1)
set status of memory_size(counter+1) to null entry
   Else
merge both blocks into one
memory_size(counter-1) = memory_size(counter-1) + job_size
   Else
search for null entry in free memory list
 enter job_size and beginning_address in the entry slot
 set its status to "free"
```

Case 1: Joining Two Free Blocks

Table 2.4 shows how deallocation occurs in a dynamic memory allocation system when the job to be deallocated is next to one free memory block.

SIET WOMENS COLLEGE



Case 1: Joining Two Free Blocks

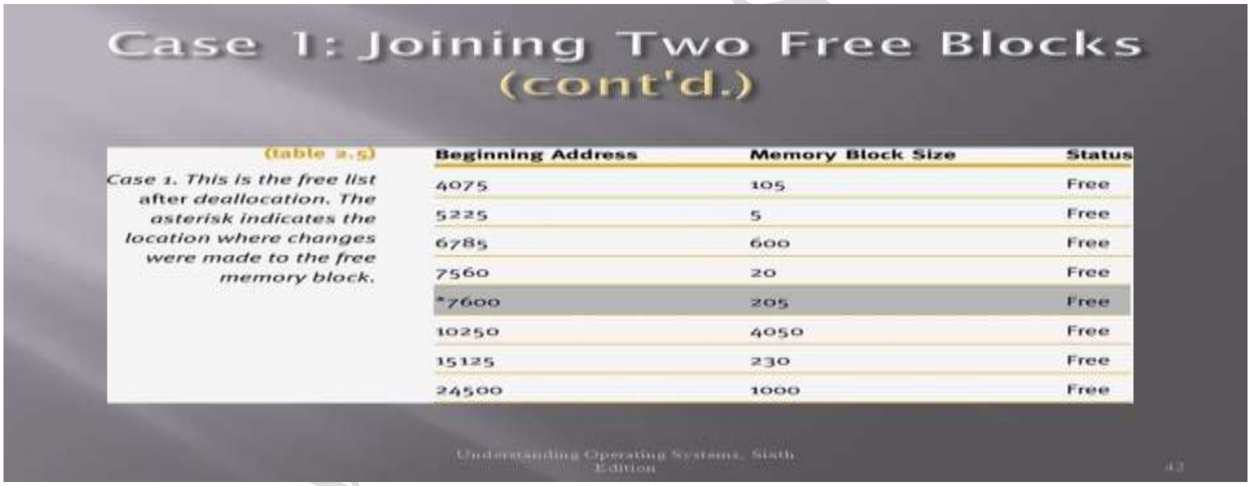
Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
(7600)	(200)	(Busy)¹
*7800	5	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

¹Although the numbers in parentheses don't appear in the free list, they've been inserted here for clarity. The job size is 200 and its beginning location is 7600.

Table 2.4: Original free list before deallocation for Case 1

Understanding Operating Systems, Fourth Edition 28

After deallocation the free list looks like the one shown in Table 2.5.



Case 1: Joining Two Free Blocks (cont'd.)

(table 2.5)

Case 1. This is the free list after deallocation. The asterisk indicates the location where changes were made to the free memory block.

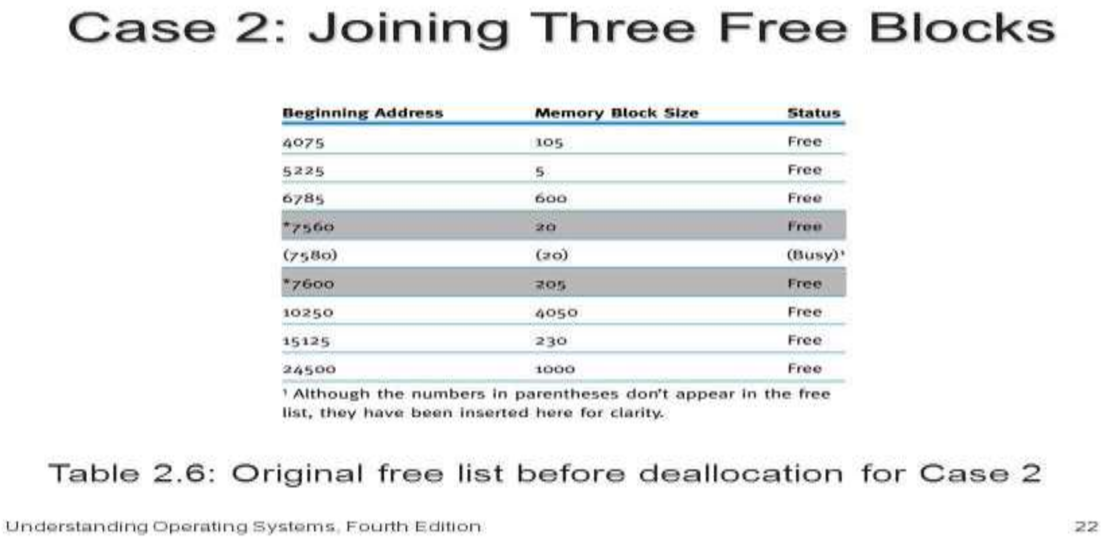
Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

Understanding Operating Systems, Sixth Edition 42

Using the deallocation algorithm, the system sees that the memory to be released is next to a free memory block, which starts at location 7800. Therefore, the list must be changed to reflect the starting address of the new free block, 7600, which was the address of the first instruction of the job that just released this block. In addition, the memory block size for this new free space must be changed to show its new size, which is the combined total of the two free partitions (200 + 5).

Case 2: Joining Three Free Blocks : When the deallocated memory space is between two free memory blocks, the process is similar, as shown in Table 2.6.

Using the deallocation algorithm, the system learns that the memory to be deallocated is between two free blocks of memory. Therefore, the sizes of the three free partitions (20 + 20 + 205) must be combined and the total stored with the smallest beginning address, 7560.



Case 2: Joining Three Free Blocks

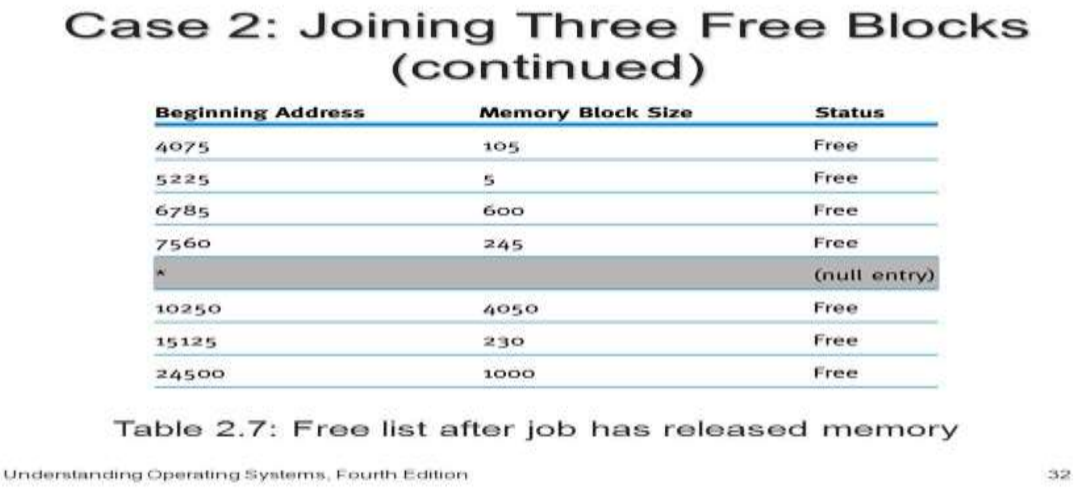
Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
*7560	20	Free
(7580)	(20)	(Busy)[1]
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

[1] Although the numbers in parentheses don't appear in the free list, they have been inserted here for clarity.

Table 2.6: Original free list before deallocation for Case 2

Understanding Operating Systems, Fourth Edition 22

Because the entry at location 7600 has been combined with the previous entry, we must empty out this entry. We do that by changing the status to null entry, with no beginning address and no memory block size as indicated by an asterisk in Table 2.7. This negates the need to rearrange the list at the expense of memory .



Case 2: Joining Three Free Blocks (continued)

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

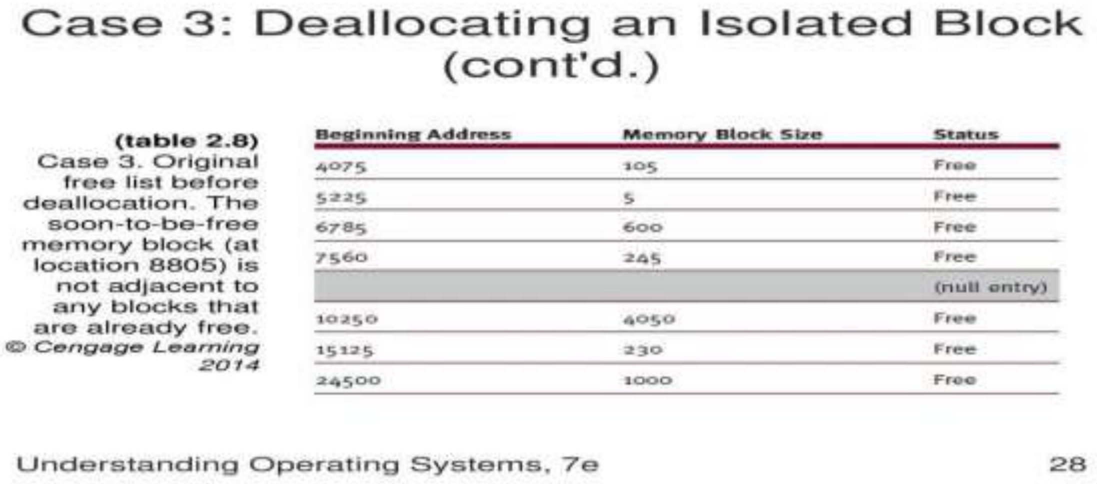
Table 2.7: Free list after job has released memory

Understanding Operating Systems, Fourth Edition 32

Case 3: Deallocating an Isolated Block :

The third alternative is when the space to be deallocated is isolated from all other free areas.

For this example, we need to know more about how the busy memory list is configured. To simplify matters, let's look at the busy list for the memory area between locations 7560 and 10250. Remember that, starting at 7560, there's a free memory block of 245, so the busy memory area includes everything from location 7805 (7560 + 245) to 10250, which is the address of the next free block. The free list and busy list are shown in Table 2.8 and Table2.9.

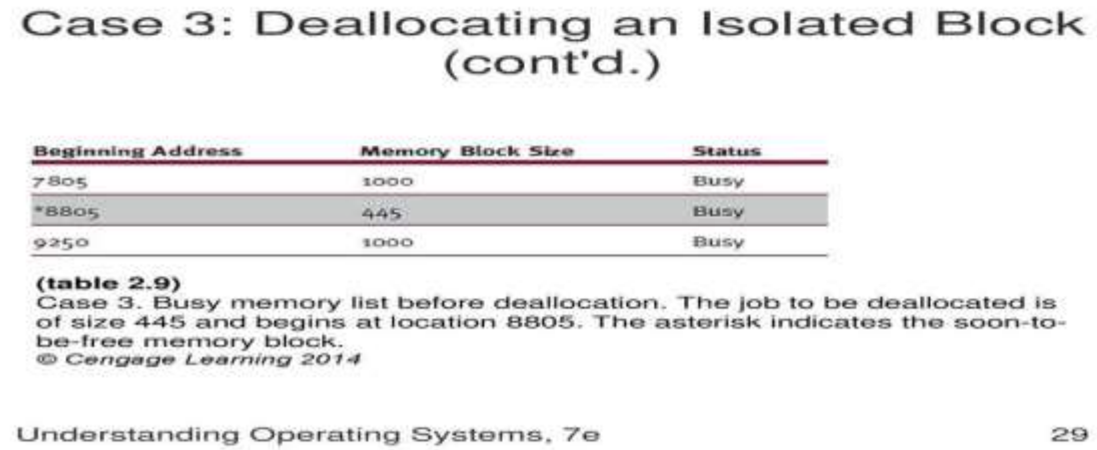


Case 3: Deallocating an Isolated Block (cont'd.)

(table 2.8) Case 3. Original free list before deallocation. The soon-to-be-free memory block (at location 8805) is not adjacent to any blocks that are already free.
© Cengage Learning 2014

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

Understanding Operating Systems, 7e 28



Case 3: Deallocating an Isolated Block (cont'd.)

Beginning Address	Memory Block Size	Status
7805	1000	Busy
*8805	445	Busy
9250	1000	Busy

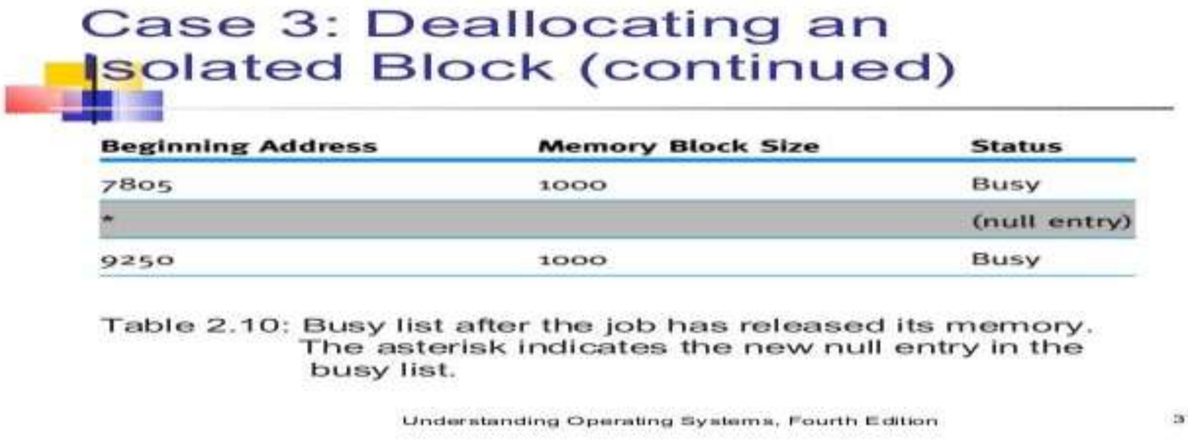
(table 2.9)
Case 3. Busy memory list before deallocation. The job to be deallocated is of size 445 and begins at location 8805. The asterisk indicates the soon-to-be-free memory block.
© Cengage Learning 2014

Understanding Operating Systems, 7e 29

sing the deallocation algorithm, the system learns that the memory block to be released is not adjacent to any free blocks of memory; instead it is between two other busy areas. Therefore, the system must search the table for a null entry.

The scheme presented in this example creates null entries in both the busy and the free lists during the process of allocation or deallocation of memory. An example of a null entry occurring as a result of deallocation was presented in Case 2. A null entry in the busy list occurs when a memory block between two other busy memory blocks is returned to the free list, as shown in Table 2.10. This mechanism ensures that all blocks are entered in the lists according to the beginning address of their memory location from smallest to largest.

Table 2.10 : Case 3. This is the busy list after the job has released its memory. The asterisk indicates the new null entry in the busy list.



Case 3: Deallocating an Isolated Block (continued)

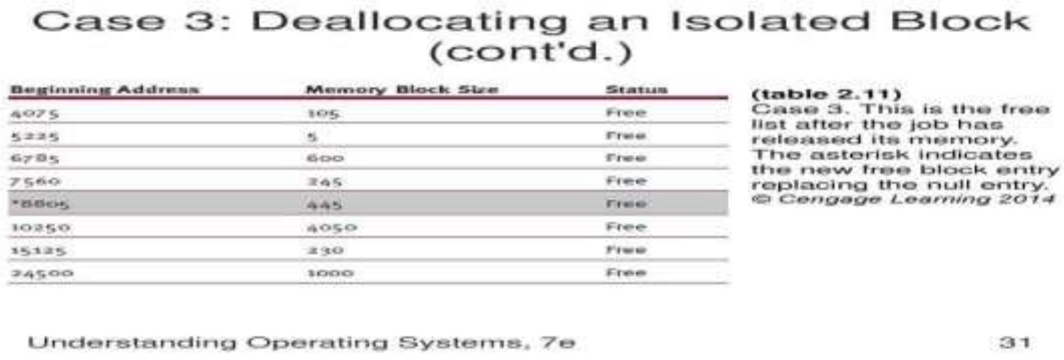
Beginning Address	Memory Block Size	Status
7805	1000	Busy
*		(null entry)
9250	1000	Busy

Table 2.10: Busy list after the job has released its memory. The asterisk indicates the new null entry in the busy list.

Understanding Operating Systems, Fourth Edition 37

When the null entry is found, the beginning memory location of the terminating job is entered in the beginning address column, the job size is entered under the memory block size column, and the status is changed from a null entry to free to indicate that a new block of memory is available, as shown in Table 2.11.

Table 2.11 : Case 3. This is the free list after the job has released its memory. The asterisk indicates the new free block entry replacing the null entry



Case 3: Deallocating an Isolated Block (cont'd.)

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*8805	445	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

(table 2.11) Case 3. This is the free list after the job has released its memory. The asterisk indicates the new free block entry replacing the null entry. © Cengage Learning 2014

Understanding Operating Systems, 7e 31

Relocatable Dynamic Partitions

Both of the fixed and dynamic memory allocation schemes described thus far shared some unacceptable fragmentation characteristics that had to be resolved before the number of jobs waiting to be accepted became unwieldy. In addition, there was a growing need to use all the slivers of memory often left over.

The solution to both problems was the development of relocatable dynamic partitions. With this memory allocation scheme, the Memory Manager relocates programs to gather together all of the empty blocks and compact them to make one block of memory large enough to accommodate some or all of the jobs waiting to getin.

The compaction of memory, sometimes referred to as garbage collection or defragmentation, is performed by the operating system to reclaim fragmented sections of the memory space. Remember our earlier example of the makeshift lending library? If you stopped lending books for a few moments and rearranged the books in the most effective order, you would be compacting your collection. But this demonstrates its disad vantage—it’s an overhead process, so that while compaction is being done everything else must wait. Compaction isn’t an easy task. First, every program in memory must be relocated so they’re contiguous, and then every address, and every reference to an address, within each program must be adjusted to account for the program’s new location inmemory.

However, all other values within the program (such as data values) must be left alone. In other words, the operating system must distinguish between addresses and data values, and the distinctions are not obvious once the program has been loaded into memory. To appreciate the complexity of relocation, let's look at a typical program. Remember, all numbers are stored in memory as binary values, and in any given program instruction it's not uncommon to find addresses as well as data values. For example, an assembly language program might include the instruction to add the integer 1 to I. The source code instruction looks like this:

```
ADDI I, 1
```

However, after it has been translated into actual code it could look like this (for readability purposes the values are represented here in octal code, not binary code): 000007 271 01 0 00 000001

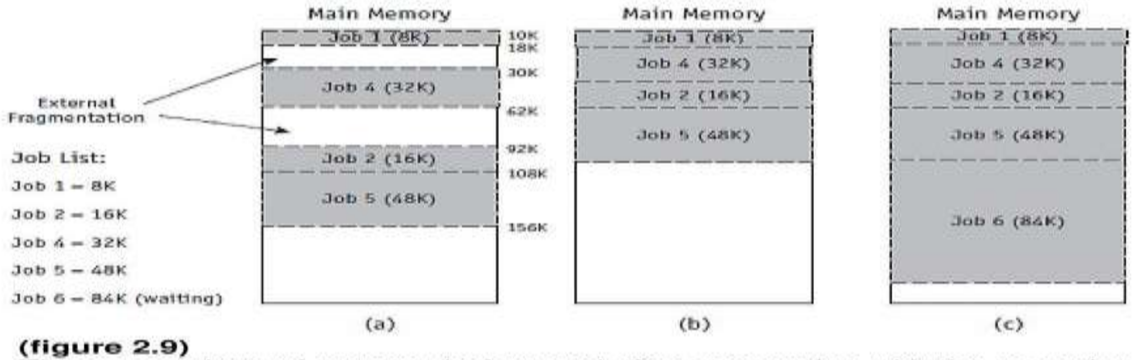
It's not immediately obvious which elements are addresses and which are instruction codes or data values. In fact, the address is the number on the left (000007). The instruction code is next (271), and the data value is on the right (000001).

The operating system can tell the function of each group of digits by its location in the line and the operation code. However, if the program is to be moved to another place in memory, each address must be identified, or flagged. So later the amount of memory locations by which the program has been displaced must be added to (or subtracted from) all of the original addresses in the program.

Internally, the addresses are marked with a special symbol (indicated in Figure 2.8 by apostrophes) so the Memory Manager will know to adjust them by the value stored in the relocation register. All of the other values (data values) are not marked and won't be changed after relocation. Other numbers in the program, those indicating instructions, registers, or constants used in the instruction, are also left alone.

Figure 2.9 illustrates what happens to a program in memory during compaction and relocation.

Figure 2.9: Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).



(figure 2.9)
Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).
© *Cengage Learning 2014*

Understanding Operating Systems, 7e 34

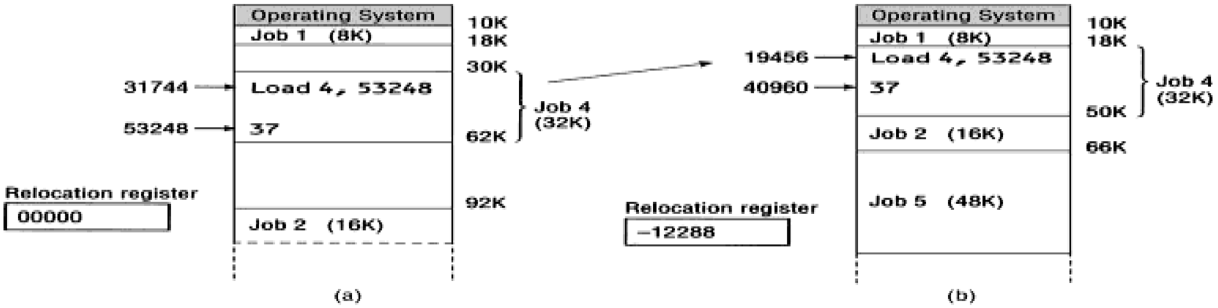
This discussion of compaction raises three questions: 1. What goes on behind the scenes when relocation and compaction take place? 2. What keeps track of how far each job has moved from its original storage area? 3. What lists have to be updated?

Special-purpose registers are used to help with the relocation. In some computers, two special registers are set aside for this purpose: the bounds register and the relocation register.

The bounds register is used to store the highest (or lowest, depending on the specific system) location in memory accessible by each program. This ensures that during execution, a program won't try to access memory locations that don't belong to it—that is, those that are out of bounds. The relocation register contains the value that must be added to each address referenced in the program so that the system will be able to access the correct memory addresses after relocation. If the program isn't relocated, the value stored in the program's relocation register is zero.

Figure 2.10 illustrates what happens during relocation by using the relocation register (all values are shown in decimal form).

Figure 2.10: Contents of relocation register and close-up of Job 4 memory area (a) before relocation and (b) after relocation and compaction.



Originally, Job 4 was loaded into memory starting at memory location 30K. (1K equals 1,024 bytes. Therefore, the exact starting address is: 30 * 1024 = 30,720.) It required a block of memory of 32K (or 32 * 1024 = 32,768) addressable locations. Therefore, when it was originally loaded, the job occupied the space from memory location 30720 to memory location 63488-1. Now, suppose that within the program, at memory location 31744, there's an instruction that looks like this:

LOAD 4, ANSWER

This assembly language command asks that the data value known as ANSWER be loaded into Register 4 for later computation. ANSWER, the value 37, is stored at memory location 53248. (In this example, Register 4 is a working/computation register, which is distinct from either the relocation or the bounds register.)

After relocation, Job 4 has been moved to a new starting memory address of 18K (actually 18 * 1024 = 18,432). Of course, the job still has its 32K addressable locations, so it now occupies memory from location 18432 to location 51200-1 and, thanks to the relocation register, all of the addresses will be adjusted accordingly. What does the relocation register contain? In this example, it contains the value –12288. As calculated previously, 12288 is the size of the free block that has been moved forward toward the high addressable end of memory. The sign is negative because Job 4 has been moved back, closer to the low addressable end of memory, as shown at the top of Figure 2.10(b).

However, the program instruction (LOAD 4, ANSWER) has not been changed. The original address 53248 where ANSWER had been stored remains the same in the program no matter how many times it is relocated. Before the instruction is executed, however, the true address must be computed by adding the value stored in the relocation register to the address found at that instruction. If the addresses are not adjusted by the value stored in the relocation register, then even though memory location 31744 is still part of the job's accessible set of memory locations, it would not contain the LOAD command. Not only that, but location 53248 is now out of bounds. The instruction that was originally at 31744 has been moved to location 19456. That's because all of the instructions in this program have been moved back by 12K (12 * 1024 = 12,288), which is the size of the free block. Therefore, location 53248 has been displaced by –12288 and ANSWER, the data value 37, is now located at address 40960.

One approach is to do it when a certain percentage of memory becomes busy, say 75 percent. The disadvantage of this approach is that the system would incur unnecessary overhead if no jobs were waiting to use the remaining 25 percent.

A second approach is to compact memory only when there are jobs waiting to get in. This would entail constant checking of the entry queue, which might result in unnecessary overhead and slow down the processing of jobs already in the system.

A third approach is to do it after a prescribed amount of time has elapsed. If the amount of time chosen is too small, however, then the system will spend more time on compaction than on processing. If it's too large, too many jobs will congregate in the waiting queue and the advantages of compaction are lost.

Paged Memory Allocation

Before a job is loaded into memory, it is divided into parts called pages that will be loaded into memory locations called page frames. Paged memory allocation is based on the concept of dividing each incoming job into pages of equal size. Some operating systems choose a page size that is the same as the memory block size and that is also the same size as the sections of the disk on which the job is stored. The sections of a disk are called sectors (or sometimes blocks), and the sections of main memory are called page frames.

The 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 and fit into one page frame of memory.

Before executing a program, the 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 them

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'll refer to the program's instructions as bytes or words.

In fact, each page can be stored in any available page frame anywhere in main memory.

The primary advantage of storing programs in noncontiguous locations 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 (and no internal fragmentation in most pages).

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 that means enlarging the size and complexity of the operating system software, which increases overhead.

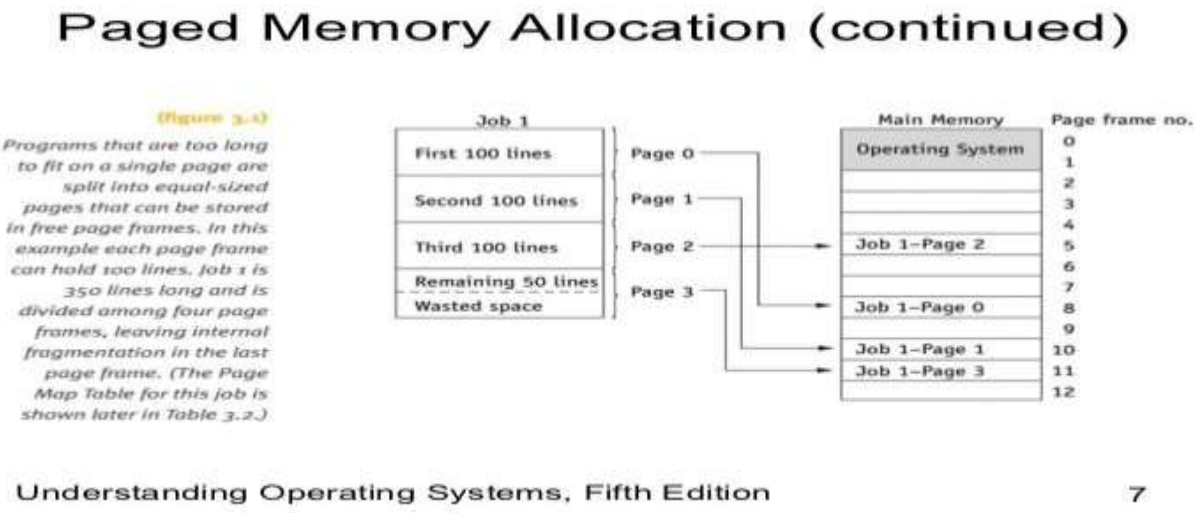
The simplified example in Figure 3.1 shows how the Memory Manager keeps track of a program that is four pages long. To simplify the arithmetic, we've arbitrarily set the page size at 100 bytes. Job 1 is 350 bytes long and is being readied for execution.

Notice in Figure 3.1 that the last page (Page 3) is not fully utilized because the job is less than 400 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 last page of a job).

In Figure 3.1 (with seven free page frames), the operating system can accommodate jobs that vary in size from 1 to 700 bytes because they can be stored in the seven empty page frames. But a job that is larger than 700 bytes can't be accommodated until Job 1 ends its execution and releases the four page frames it occupies. And a job that is larger than 1100 bytes will never fit into the memory of this tiny system. Therefore, although paged memory allocation offers the advantage of noncontiguous storage, it still requires that the entire job be stored in memory during its execution.

Figure 3.1: Programs that are too long to fit on a single page are split into equal-sized pages that can be stored in free page frames. In this example, each page frame can hold 100 bytes. Job 1 is 350 bytes long and is divided among four page frames, leaving internal fragmentation in the last page frame. (The Page Map Table for this job is shown later in Table 3.2.)

STET WOMEN'S COLLEGE



Paged Memory Allocation (continued)

(figure 3.1)

Programs that are too long to fit on a single page are split into equal-sized pages that can be stored in free page frames. In this example each page frame can hold 100 lines. Job 1 is 350 lines long and is divided among four page frames, leaving internal fragmentation in the last page frame. (The Page Map Table for this job is shown later in Table 3.2.)

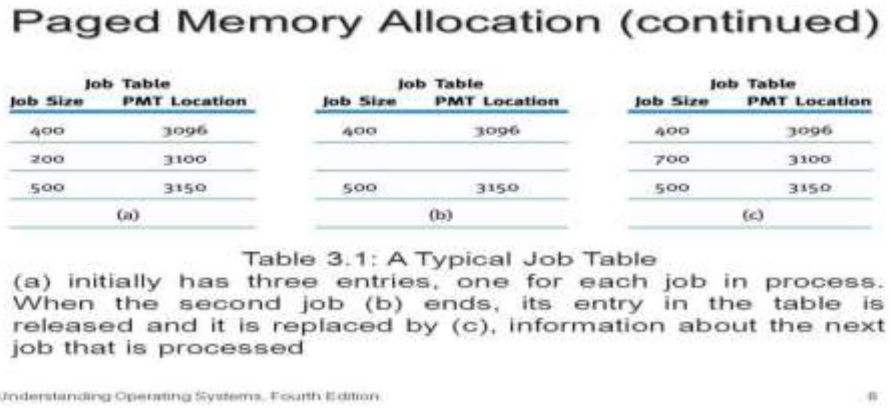
Understanding Operating Systems, Fifth Edition 7

Figure 3.1 uses arrows and lines to show how a job's pages fit into page frames in memory, but the Memory Manager uses tables to keep track of them. There are essentially three tables that perform this function: the 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 job size of 400 located at 3096 in 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 later completed .

Job Table		**Job Table**		**Job Table**	
Job Size	**PMT Location**	**Job Size**	**PMT Location**	**Job Size**	**PMT Location**
400	3096	400	3096	400	3096
200	3100			700	3100
500	3150	500	3150	500	3150

Table 3.1: This section of the Job Table (a) initially has three entries, one for each job in progress. When the second job ends (b), its entry in the table is released and it is replaced (c) by information about the next job that is to be processed.



Paged Memory Allocation (continued)

Job Table		Job Table		Job Table	
Job Size	PMT Location	Job Size	PMT Location	Job Size	PMT Location
400	3096	400	3096	400	3096
200	3100			700	3100
500	3150	500	3150	500	3150
(a)		(b)		(c)	

Table 3.1: A Typical Job Table
(a) initially has three entries, one for each job in process. When the second job (b) ends, its entry in the table is released and it is replaced by (c), information about the next job that is processed

Understanding Operating Systems, Fourth Edition 8

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

The Memory Map Table (MMT) has one entry for each page frame listing its location and free/busy status. At compilation time, every job is divided into pages. Using Job 1 from Figure 3.1, we can see how this works:

- Page 0 contains the first hundred bytes.
- Page 1 contains the second hundred bytes.
- Page 2 contains the third hundred bytes.
- Page 3 contains the last 50 bytes.

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 349.

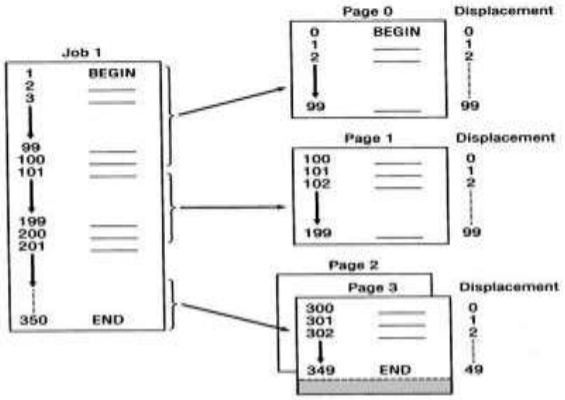
The displacement, or offset, of a byte (that is, how far away a byte is from the beginning of its page) is the factor used to locate that byte within its page frame. It is a relative factor. In the simplified example shown in Figure 3.2, bytes 0, 100, 200, and 300 are the first bytes for pages 0, 1, 2, and 3, respectively, so each has a displacement of zero. Likewise, if the operating system needs to access byte 214, it can first go to page 2 and then go to byte 14 (the fifteenth line).

The first byte of each page has a displacement of zero, and the last byte, has a displacement of 99. So once the operating system finds the right page, it can access the correct bytes using its relative position within its page.

In this example, it is easy for us to see intuitively that all numbers less than 100 will be on Page 0, all numbers greater than or equal to 100 but less than 200 will be on Page 1, and so on. (That is the advantage of choosing a fixed page size, such as 100 bytes.) The operating system uses an algorithm to calculate the page and displacement; it is a simple arithmetic calculation. To find the address of a given program instruction, the byte number is divided by the page size, keeping the remainder as an integer. The resulting quotient is the page number, and the remainder is the displacement within that page. When it is set up as a long division problem, it looks like this:

Figure 3.2: Job 1 is 350 bytes long and is divided into four pages of 100 lines each.

Paged Memory Allocation (continued)



Job 1 is 350 lines long and is divided into four pages of 100 lines each.

Figure 3.2: Paged Memory Allocation Scheme

Understanding Operating Systems, Fourth Edition 9

To find the address of a given program instruction, the byte number is divided by the page size, keeping the remainder as an integer. The resulting quotient is the page number, and the remainder is the displacement within that page. When it is set up as a long division problem, it looks like this:

For example, if we use 100 bytes as the page size, the page number and the displacement (the location within that page) of byte 214 can be calculated using long division like this:

The quotient (2) is the page number, and the remainder (14) is the displacement. So the byte is located on Page 2, 15 lines (Line 14) from the top of the page.

Let's try another example with a more common page size of 256 bytes. Say we are seeking the location of byte 384. When we divide 384 by 256, the result is 1.5. Therefore, the byte is located at the midpoint on the second page (Page 1).

To find the line's exact location, multiply the page size (256) by the decimal (0.5) to discover that the line we're seeking is located on Line 129 of Page 1.

Using the concepts just presented, and using the same parameters from the first example, answer these questions:

1. Could the operating system (or the hardware) get a page number that is greater than 3 if the program was searching for byte 214?
2. If it did, what should the operating system do?
3. Could the operating system get a remainder of more than 99?
4. What is the smallest remainder possible?

Here are the answers:

1. No, not if the application program was written correctly.
2. Send an error message and stop processing the program (because the page is out of bounds).
3. No, not if it divides correctly.
4. Zero

This procedure gives the location of an instruction with respect to the job's pages. However, these pages are only relative; each page is actually stored in a page frame that can be located anywhere in available main memory. Therefore, the 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 Page Map Table. For example, if we look at the PMT for Job 1 from Figure 3.1, we see that it looks like the data in Table 3.2.

Table 3.2 : Page Map Table for Job 1 in Figure 3.1.

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

In the first division example, we were looking for an instruction with a displacement of 14 on Page 2. To find its exact location in memory, the operating system (or the hardware) has to perform the following four steps. (In actuality, the operating system identifies the lines, or data values and instructions, as addresses [bytes or words].

STEP 1 Do the arithmetic computation just described to determine the page number and displacement of the requested byte.

- Page number = the integer quotient from the division of the job space address by the page size
- Displacement = the remainder from the page number division In this example, the computation shows that the page number is 2 and the displacement is 14.

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).

STET WOMEN'S COLLEGE

ADDR_PAGE_FRAME = PAGE_FRAME_NUM * PAGE_SIZE
ADDR_PAGE_FRAME = 5(100)

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

INSTR_ADDR_IN_MEM = ADDR_PAGE_FRAME + DISPL
INSTR_ADDR_IN_MEM = 500 + 14

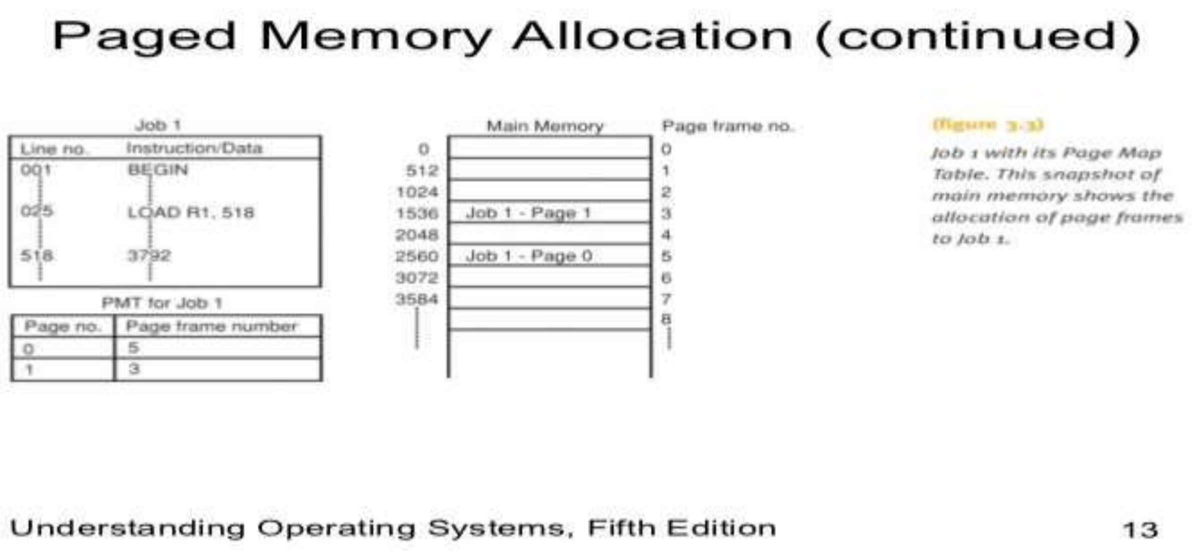
The result of this maneuver tells us exactly where byte 14 is located in main memory. Figure 3.3 shows another example and follows the hardware (and the operating system) as it runs an assembly language program that instructs the system to load into Register 1 the value found at byte 518.

In Figure 3.3, the page frame sizes in main memory are set at 512 bytes each and the page size is 512 bytes for this system. From the PMT we can see that this job has been divided into two pages. To find the exact location of byte 518 (where the system will find the value to load into Register 1), the system will do the following:

1. Compute the page number and displacement—the page number is 1, and the displacement is 6.
2. Go to the Page Map Table and retrieve the appropriate page frame number for Page 1. It is Page Frame 3.
3. Compute the starting address of the page frame by multiplying the page frame number by the page frame size: (3 * 512 = 1536).
4. Calculate the exact address of the instruction in main memory by adding the displacement to the starting address: (1536 + 6 = 1542). Therefore, memory address 1542 holds the value that should be loaded into Register 1.

Every time an instruction is executed, or a data value is used, the operating system (or the hardware) must translate the job space address, which is relative, into its physical address, which is absolute. This is called resolving the address, also called address resolution, or address translation. Of course, all of this processing is overhead, which takes processing capability away from the jobs waiting to be completed. However, in most systems the hardware does the paging, although the operating system is involved in dynamic paging, which will be covered later.

Figure 3.3 : Job 1 with its Page Map Table. This snapshot of main memory shows the allocation of page frames to Job 1.



Paged Memory Allocation (continued)

Job 1

Line no.	Instruction/Data
001	BEGIN
025	LOAD R1, 518
518	3792

PMT for Job 1

Page no.	Page frame number
0	5
1	3

	Main Memory	Page frame no.
0		0
512		1
1024		2
1536	Job 1 - Page 1	3
2048		4
2560	Job 1 - Page 0	5
3072		6
3584		7
		8

(figure 3.3)
Job 1 with its Page Map Table. This snapshot of main memory shows the allocation of page frames to Job 1.

Understanding Operating Systems, Fifth Edition 13

The advantage of a paging scheme is that it allows jobs to be allocated in noncontiguous memory locations so that memory is used more efficiently and more jobs can fit in the main memory (which is synonymous). However, there are disadvantages— overhead is increased and internal fragmentation is still a problem, although only in the last page of each job. The key to the success of this scheme is the size of the page. A page size that is too small will generate very long PMTs while a page size that is too large will result in excessive internal fragmentation.

Determining the best page size is an important policy decision—there are no hard and fast rules that will guarantee optimal use of resources—and it is a problem we'll see again as we examine other paging alternatives. The best size depends on the actual job environment, the nature of the jobs being processed, and the constraints placed on the system.

Demand Paging

Demand paging introduced the concept of loading only a part of the program into memory for processing. It was the first widely used scheme that removed the restriction of having 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 that initially reside in secondary storage. When the job begins to run, its pages are brought into memory only as they are needed.

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

- User-written error handling modules are processed only when a specific error is detected during execution. (For instance, they can be used to indicate to the operator 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 modules are mutually exclusive. For example, if the input module is active (such as while a worksheet is being loaded) then the processing module is inactive. Similarly, if the processing module is active then the output module (such as printing) is idle.
- Certain program options are either mutually exclusive or not always accessible. This is easiest to visualize in menu-driven programs. For example, an application program may give the user several menu choices as shown in Figure 3.4. The system allows the operator to make only one selection at a time. If the user selects the first option then the module with the program instructions to move records to the file is the only one that is being used, so that is the only module that needs to be in memory at this time. The other modules all remain in secondary storage until they are called from the menu.
- Many tables are assigned a large fixed amount of address space even though only a fraction of the table is actually used. For example, a symbol table for an assembler might be prepared to handle 100 symbols. If only 10 symbols are used then 90 percent of the table remains unused.

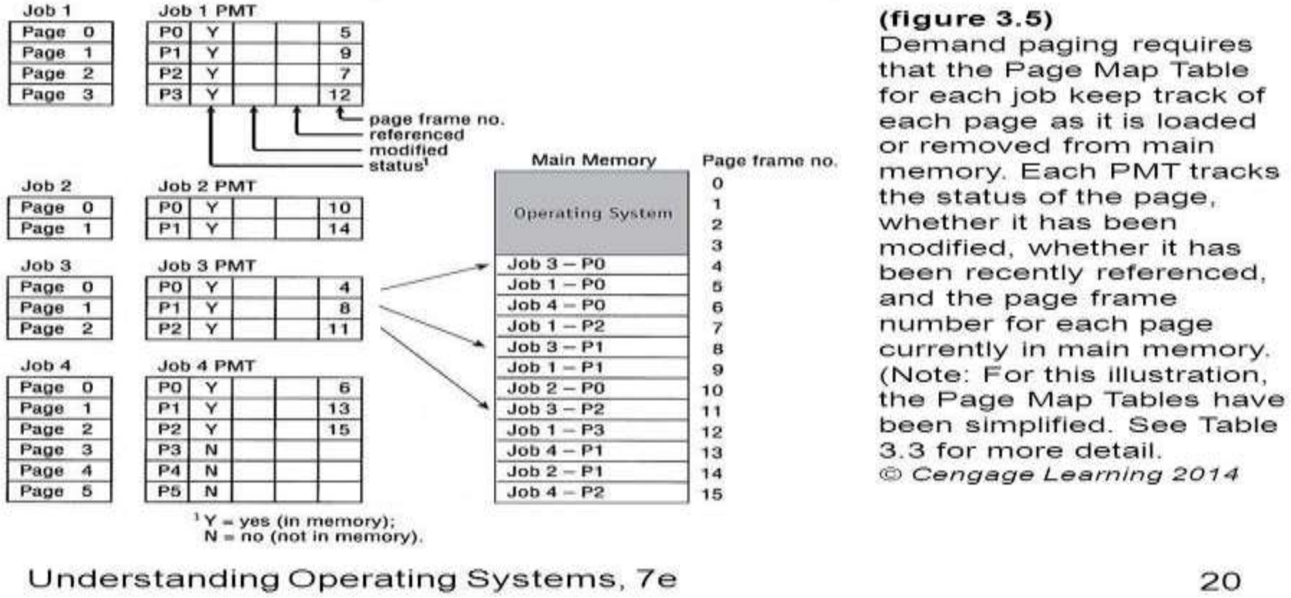
Figure 3.4 : When you choose one option from the menu of an application program such as this one, the other modules that aren't currently required (such as Help) don't need to be moved into memory immediately

One of the most important innovations of demand paging was that it made virtual memory feasible. (Virtual memory will be discussed later in this chapter.) The demand paging scheme allows the user to run jobs with less main memory than is required if the operating system is using the paged memory allocation scheme described earlier. In fact, a demand paging scheme can give the appearance of an almost-infinite or nonfinite amount of physical memory when, in reality, physical memory is significantly less than infinite. The key to the successful implementation of this scheme is the use of a high-speed direct access storage device (such as hard drives or flash memory) that can work directly with the CPU. That is vital because pages must be passed quickly from secondary storage to main memory and back again.

How and when the pages are passed (also called swapped) depends 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 Table, and the Memory Map Table) to implement the algorithm. These tables are basically the same as for paged memory allocation but with the addition of three new fields for each page in the PMT: one to determine if the page being requested is already in memory; a second to determine if the page contents have been modified; and a third to determine if the page has been referenced recently, as shown at the top of Figure 3.5.

STET WOMEN'S COLLEGE

Figure 3.5: Demand paging requires that the Page Map Table



Job 1 PMT

P0	Y			5
P1	Y			9
P2	Y			7
P3	Y			12

Job 2 PMT

P0	Y			10
P1	Y			14

Job 3 PMT

P0	Y			4
P1	Y			8
P2	Y			11

Job 4 PMT

P0	Y			6
P1	Y			13
P2	Y			15
P3	N			
P4	N			
P5	N			

Main Memory	Page frame no.
Operating System	0
	1
	2
	3
Job 3 – P0	4
Job 1 – P0	5
Job 4 – P0	6
Job 1 – P2	7
Job 3 – P1	8
Job 1 – P1	9
Job 2 – P0	10
Job 3 – P2	11
Job 1 – P3	12
Job 4 – P1	13
Job 2 – P1	14
Job 4 – P2	15

[1] Y = yes (in memory);
N = no (not in memory).

(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, whether it has been 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.
© Cengage Learning 2014

Understanding Operating Systems, 7e 20

The first field tells the system where to find each page. If it is already in memory, the system will be spared the time required to bring it from secondary storage. It is faster for the operating system to scan a table located in main memory than it is to retrieve a page from a disk.

The second field, noting if the page has been modified, 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 rewritten to secondary storage. The original, already there, is correct.

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

For example, in Figure 3.5 the number of total job pages is 15, and the number of total available page frames is 12. Assuming the processing status illustrated in Figure 3.5, what happens when Job 4 requests that Page 3 be brought into memory if there are no empty page frames available? To move in a new page, a resident page must be swapped back into secondary storage. Specifically, that includes copying the resident page to the disk (if it was modified), and writing the new page into the empty page frame.

The hardware components generate the address of the required page, find the page number, and determine whether it is already in memory. The following algorithm makes up the hardware instruction processing cycle.

Hardware Instruction Processing Algorithm

```
1 Start processing instruction
2 Generate data address
3 Compute page number
 4 If page is in memory
    Then
       get data and finish instruction
      advance to next instruction
      return to step 1
 Else
```

STET WOMEN'S C

```
generate page interrupt

      call page fault handler
End if
```

The same process is followed when fetching an instruction. When the test fails (meaning that the page is in secondary storage but not in memory), the operating system software takes over. The section of the operating system that resolves these problems is called the page fault handler. It determines whether there are empty page frames in memory so the requested page can be immediately copied from secondary storage. If all page frames are busy, the page fault handler must decide which page will be swapped out. (This decision is directly dependent on the predefined policy for page removal.) Then the swap is made.

Page Fault Handler Algorithm

```
1 If there is no free page frame
     Then
        select page to be swapped out using page removal algorithm
        update job's Page Map Table
     If content of page had been changed then
      write page to disk End if End if
2 Use page number from step 3 from the Hardware Instruction Processing Algorithm to get disk address where the
requested page is stored
3 Read page into memory
4 Update job's Page Map Table
5 Update Memory Map Table
6 Restart interrupted instruction
```

Before continuing, three tables must be updated: the Page Map Tables for both jobs (the PMT with the page that was swapped out and the PMT with the page that was swapped in) and the Memory Map Table. Finally, the instruction that was interrupted is resumed and processing continues.

Although demand paging is a solution to inefficient memory utilization, it is not free of problems. When there is an excessive amount of page swapping between main memory and secondary storage, the operation becomes inefficient. This phenomenon is called thrashing. It uses a great deal of the computer's energy but accomplishes very little, and it is caused when a page is removed from memory but is called back shortly thereafter. Thrashing can occur across jobs, when a large number of jobs are vying for a relatively low number of free pages or it can happen within a job.

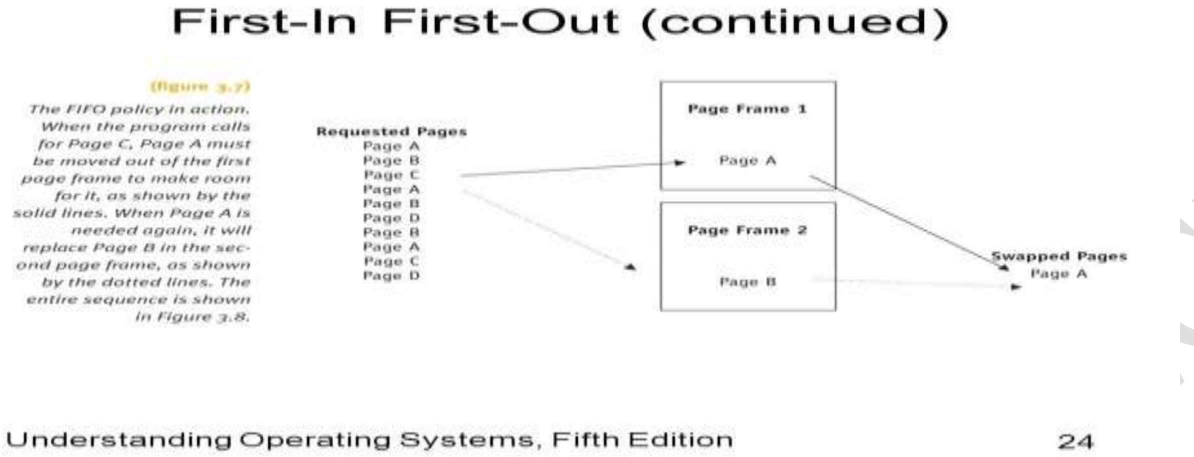
Page Replacement Policies and Concepts

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 are first-in first-out and least recently used. The first-in first-out (FIFO) policy is based on the theory 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 .

First-In First-Out

The first-in first-out (FIFO) page replacement policy will remove the pages that have been in memory the longest.

SFET WOMEN'S COLLEGE



First-In First-Out (continued)

(figure 3.7)

The FIFO policy in action. When the program calls for Page C, Page A must be moved out of the first page frame to make room for it, as shown by the solid lines. When Page A is needed again, it will replace Page B in the second page frame, as shown by the dotted lines. The entire sequence is shown in Figure 3.8.

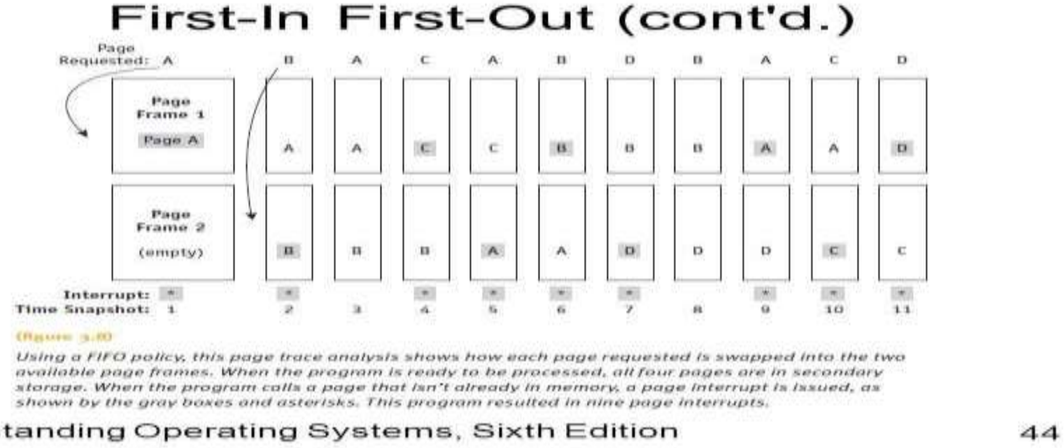
Understanding Operating Systems, Fifth Edition 24

Figure 3.8 shows how the FIFO algorithm works by following a job with four pages (A, B, C, D) as it is processed by a system with only two available page frames. Figure 3.8 displays how each page is swapped into and out of memory and marks each interrupt with an asterisk. We then count the number of page interrupts and compute the failure rate and the success rate. The job to be processed needs its pages in the following order: A, B, A, C, A, B, D, B, A, C, D.

When both page frames are occupied, each new page brought into memory will cause an existing one to be swapped out to secondary storage. A page interrupt, which we identify with an asterisk (*), is generated when a new page needs to be loaded into memory, whether a page is swapped out or not.

The efficiency of this configuration is dismal—there are 9 page interrupts out of 11 page requests due to the limited number of page frames available and the need for many new pages. To calculate the failure rate, we divide the number of interrupts by the number of page requests. 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 this high is usuallyunacceptable.

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 storage. 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. This program resulted in nine page interrupts.



First-In First-Out (cont'd.)

Page Requested:	A	B	A	C	A	B	D	B	A	C	D
Page Frame 1	Page A	A	A	C	C	B	B	B	A	A	D
Page Frame 2	(empty)	B	B	B	A	A	D	D	D	C	C
Interrupt:	*	*		*	*	*	*		*	*	*
Time Snapshot:	1	2	3	4	5	6	7	8	9	10	11

(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 storage. 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. This program resulted in nine page interrupts.

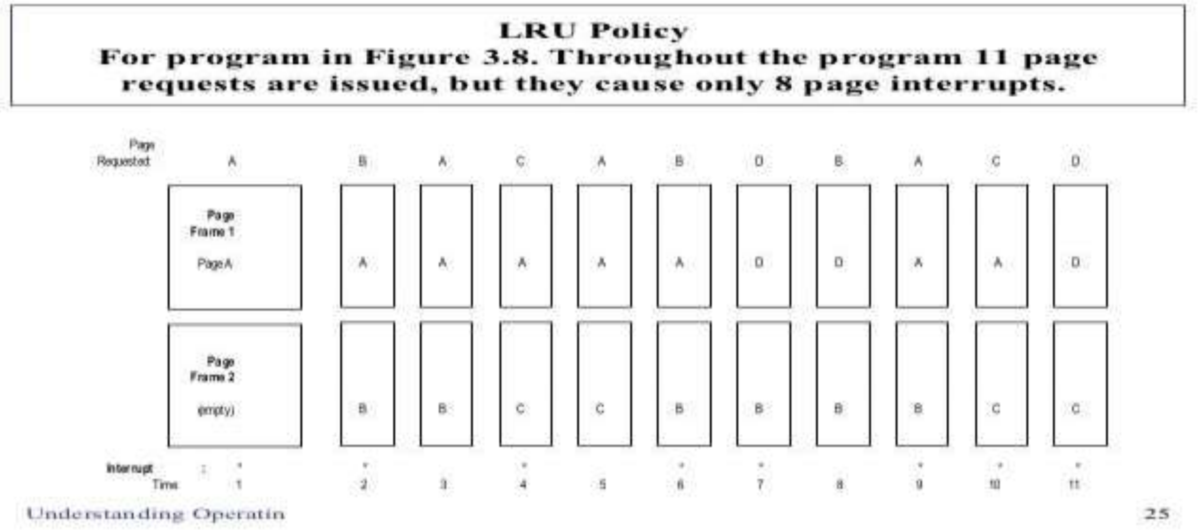
Understanding Operating Systems, Sixth Edition 44

There is no guarantee that buying more memory will always result in better performance; this is known as the FIFO anomaly.

Least Recently Used

The least recently used (LRU) page replacement policy swaps out the pages that show the least amount of recent activity, figuring that these pages are the least likely to be used again in the immediate future. Conversely, if a page is used, it is likely to be used again soon; this is based on the theory of locality, which will be explained later in this chapter. To see how it works, let us follow the same job in Figure 3.8 but using the LRU policy. The results are shown in Figure 3.9. To implement this policy, a queue of the requests is kept in FIFO order, a time stamp of when the job entered the system is saved, or a mark in the job's PMT is made periodically.

Figure 3.9: Memory management using an LRU page removal policy for the program shown in Figure 3.8. Throughout the program, 11 page requests are issued, but they cause only 8 page interrupts.



LRU Policy
For program in Figure 3.8. Throughout the program 11 page requests are issued, but they cause only 8 page interrupts.

Page Requested	A	B	A	C	A	B	D	B	A	C	D
Page Frame 1	Page A	A	A	A	A	A	D	D	A	A	D
Page Frame 2	(empty)	B	B	C	C	B	B	B	B	C	C
Interrupt	*	*		*		*	*		*	*	*
Time	1	2	3	4	5	6	7	8	9	10	11

Understanding Operatin 25

The efficiency of this configuration is only slightly better than with FIFO. Here, there are 8 page interrupts out of 11 page requests, so the failure rate is 8/11, or 73 percent. In this example, an increase in main memory by one page frame would increase the success rate of both FIFO and LRU. However, we can't conclude on the basis of only one example that one policy is better than the others. In fact, LRU is a stack algorithm removal policy, which means that an increase in memory will never cause an increase in the number of page interrupts.

On the other hand, it has been shown that under certain circumstances adding more memory can, in rare cases, actually cause an increase in page interrupts when using a FIFO policy. As noted before, it is called the FIFO anomaly. But although it is an unusual occurrence, the fact that it exists coupled with the fact that pages are removed regardless of their activity (as was the case in Figure 3.8) has removed FIFO from the most favored policy position it held in some cases.

A variation of the LRU page replacement algorithm is known as the clock page replacement policy because it is implemented with a circular queue and uses a pointer to step through the reference bits of the active pages, simulating a clockwise motion. The algorithm is paced according to the computer's clock cycle, which is the time span between two ticks in its system clock. The algorithm checks the reference bit for each page. If the bit is one (indicating that it was recently referenced), the bit is reset to zero and the bit for the next page is checked. However, if the reference bit is zero (indicating that the page has not recently been referenced), that page is targeted for removal. If all the reference bits are set to one, then the pointer must cycle through the entire circular queue again giving each page a second and perhaps a third or fourth chance. Figure 3.10 shows a circular queue containing the reference bits for eight pages currently in memory. The pointer indicates the page that would be considered next for removal. Figure 3.10 shows what happens to the reference bits of the pages that have been given a second chance.

When a new page, 146, has to be allocated to a page frame, it is assigned to the space that has a reference bit of zero, the space previously occupied by page 210.

A second variation of LRU uses an 8-bit reference byte and a bit-shifting technique to track the usage of each page currently in memory. When the page is first copied into memory, the leftmost bit of its reference byte is set to 1; and all bits to the right of the one are set to zero, as shown in Figure 3.11. At specific time intervals of the clock cycle, the Memory Manager shifts every page's reference bytes to the right by one bit, dropping their rightmost bit. Meanwhile, each time a page is referenced, the leftmost bit of its reference byte is set to 1.

This process of shifting bits to the right and resetting the leftmost bit to 1 when a page is referenced gives a history of each page's usage. For example, a page that has not been used for the last eight time ticks would have a reference byte of 00000000, while one that has been referenced once every time tick will have a reference byte of 11111111.

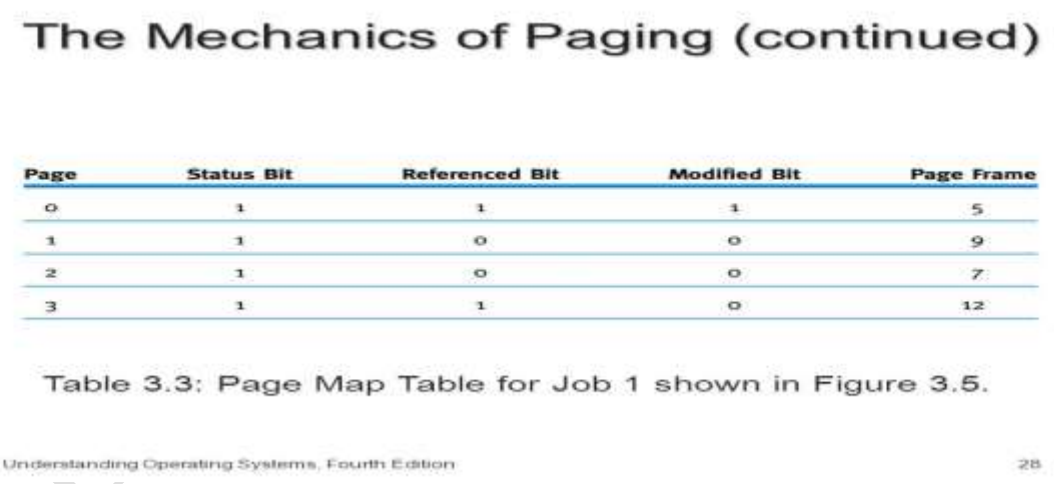
When a page fault occurs, the LRU policy selects the page with the smallest value in its reference byte because that would be the one least recently used. Figure 3.11 shows how the reference bytes for six active pages change during four snapshots of usage. In (a), the six pages have been initialized; this indicates that all of them have been referenced once. In (b), pages 1, 3, 5, and 6 have been referenced again (marked with 1), but pages 2 and 4 have not (now marked with 0 in the leftmost position). In (c), pages 1, 2, and 4 have been referenced. In (d), pages 1, 2, 4, and 6 have been referenced. In (e), pages 1 and 4 have been referenced.

The Mechanics of Paging

Before the Memory Manager can determine which pages will be swapped out, it needs specific information about each page in memory—information included in the Page Map Tables.

For example, in Figure 3.5, the Page Map Table for Job 1 included three bits: the status bit, the referenced bit, and the modified bit (these were the three middle columns: the two empty columns and the Y/N column representing "in memory"). But the representation of the table shown in Figure 3.5 was simplified for illustration purposes. It actually looks something like the one shown in Table 3.3.

Table 3.3: Page Map Table for Job 1 shown in Figure 3.5.



The Mechanics of Paging (continued)

Page	Status Bit	Referenced Bit	Modified Bit	Page Frame
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	1	0	12

Table 3.3: Page Map Table for Job 1 shown in Figure 3.5.

Understanding Operating Systems, Fourth Edition 28

The status bit indicates whether the page is currently in memory. The referenced bit indicates whether the page has been called (referenced) recently. This bit is important because it is used by the LRU algorithm to determine which pages should be swapped out. The modified bit indicates whether the contents of the page have been altered and, if so, the page must be rewritten to secondary storage when it is swapped out before its page frame is released. (A page frame with contents that have not been modified can be overwritten directly, thereby saving a step.)

That is because when a page is swapped into memory it isn't removed from secondary storage. The page is merely copied—the original remains intact in secondary storage. Therefore, if the page isn't altered while it is in main memory (in which case the modified bit remains unchanged, zero), the page needn't be copied back to secondary storage when it is swapped out of memory—the page that is already there is correct. However, if modifications were made to the page, the new version of the page must be written over the older version—and that takes time. Each bit can be either 0 or 1 as shown in Table 3.4.

The status bit for all pages in memory is 1. A page must be in memory before it can be swapped out so all of the candidates for swapping have a 1 in this column. The other two bits can be either 0 or 1, so there are four possible combinations of the referenced and modified bits as shown in Table 3.5.

The FIFO algorithm uses only the modified and status bits when swapping pages, but the LRU looks at all three before deciding which pages to swap out. Which page would the LRU policy choose first to swap? Of the four cases described in Table 3.5, it would choose pages in Case 1 as the ideal candidates for removal because they've been neither modified nor referenced. That means they wouldn't need to be rewritten to secondary storage, and they haven't been referenced recently. So the pages with zeros for these two bits would be the first to be swapped out. What is the next most likely candidate? The LRU policy would choose Case 3 next because the other two, Case 2 and Case 4, were recently referenced. The bad news is that Case 3 pages have been modified, so it will take more time to swap them out. By process of elimination, then we can say that Case 2 is the third choice and Case 4 would be the pages least likely to be removed.

You may have noticed that Case 3 presents an interesting situation: apparently these pages have been modified without being referenced. How is that possible? The key lies in how the referenced bit is manipulated by the operating system. When the pages are brought into memory, they are all usually referenced at least once and that means that all of the pages soon have a referenced bit of 1. Of course the LRU algorithm would be defeated if every page indicated that it had been referenced. Therefore, to make sure the referenced bit actually indicates recently referenced, the operating system periodically resets it to 0. Then, as the pages are referenced during processing, the bit is changed from 0 to 1 and the LRU policy is able to identify which pages actually are frequently referenced. As you can imagine, there is one brief instant, just after the bits are reset, in which all of the pages (even the active pages) have reference bits of 0 and are vulnerable. But as processing continues, the most-referenced pages soon have their bits reset to 1, so the risk is minimized.

The Working Set

One innovation that improved the performance of demand paging schemes was the concept of the working set. A job's working set is the set of pages residing in memory that can be accessed directly without incurring a page fault.

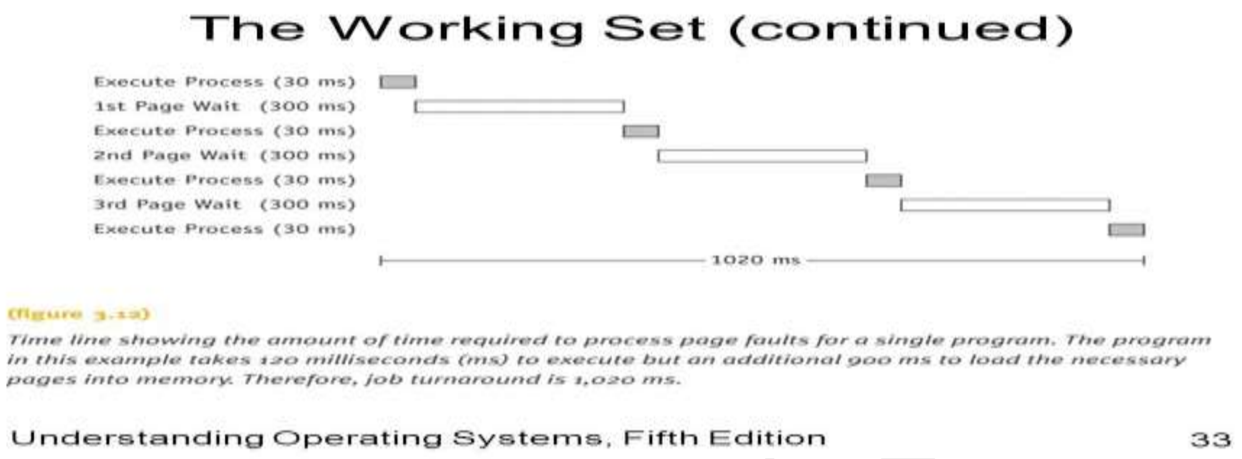
When a user requests execution of a program, the first page is loaded into memory and execution continues as more pages are loaded: those containing variable declarations, others containing instructions, others containing data, and so on. After a while, most programs reach a fairly stable state and processing continues smoothly with very few additional page faults. At this point the job's working set 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. Of course, it is possible that a poorly structured program could require that every one of its pages be in memory before processing can begin.

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. For example, if a job is executing a loop then the instructions within the loop are referenced extensively while those outside the loop aren't used at all until the loop is completed—that is locality of reference. The same 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.

We have looked at several examples of demand paging memory allocation schemes. Demand paging had two advantages. It was the first scheme in which a job was no

STET WOMEN'S COLLEGE

(figure 3.12) Time line showing the amount of time required to process page faults for a single program. The program in this example takes 120 milliseconds (ms) to execute but an additional 900 ms to load the necessary pages into memory. Therefore, job turnaround is 1020 ms.



The Working Set (continued)

(figure 3.12)

Time line showing the amount of time required to process page faults for a single program. The program in this example takes 120 milliseconds (ms) to execute but an additional 900 ms to load the necessary pages into memory. Therefore, job turnaround is 1,020 ms.

Understanding Operating Systems, Fifth Edition 33

longer constrained by the size of physical memory and it introduced the concept of virtual memory. The second advantage was that it utilized memory more efficiently than the previous schemes because the sections of a job that were used seldom or not at all (such as error routines) weren't loaded into memory unless they were specifically requested. Its disadvantage was the increased overhead caused by the tables and the page interrupts. The next allocation scheme built on the advantages of both paging and dynamic partitions.

Segmented Memory Allocation

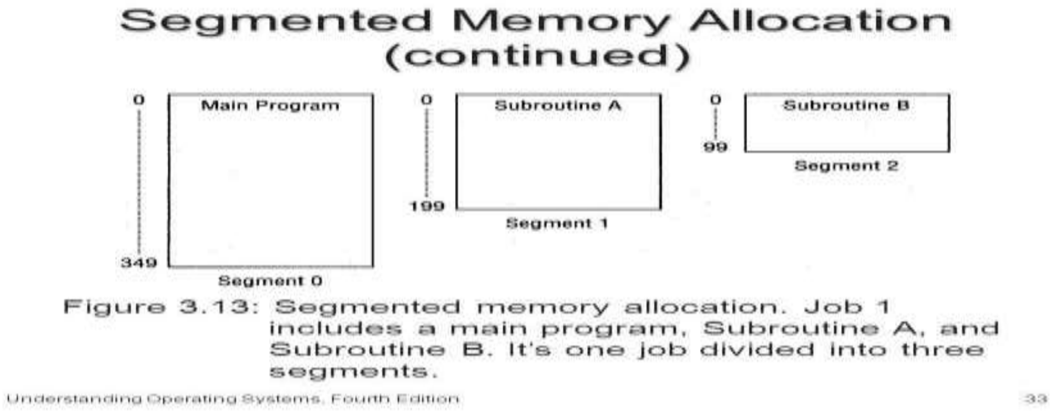
The concept of segmentation is based on the common practice by programmers of structuring their programs in modules—logical groupings of code. With segmented memory allocation, each job is divided into several segments of different sizes, one for each module that contains pieces that perform related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

A second important difference is that main memory is no longer divided into page frames because the size of each segment is different—some are large and some are small.

When a program is compiled or assembled, the segments are set up according to the program'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 3.13 and 3.14 show the same job, Job 1, composed of a main program and two subroutines, together with its Segment Map Table and actual main memory allocation.

Figure 3.13: Segmented memory allocation. Job 1 includes a main program, Subroutine A, and Subroutine B. It is one job divided into three segments.

STET WOMENS C



Segmented Memory Allocation (continued)

Figure 3.13: Segmented memory allocation. Job 1 includes a main program, Subroutine A, and Subroutine B. It's one job divided into three segments.

Understanding Operating Systems, Fourth Edition 33

he 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 only need to know where each segment is stored. The contents of the segments themselves are contiguous in this scheme.

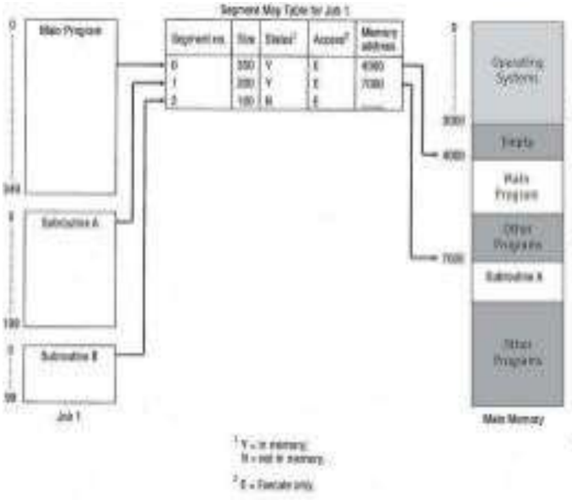
To access a specific location within a segment, we can perform an operation similar to the one used for paged memory management. The only difference is that we work with segments instead of pages. The addressing scheme requires the segment number and the displacement within that segment; and because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

In Figure 3.15, Segment 1 includes all of Subroutine A so the system finds the beginning address of Segment 1, address 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). Its code could look like this:

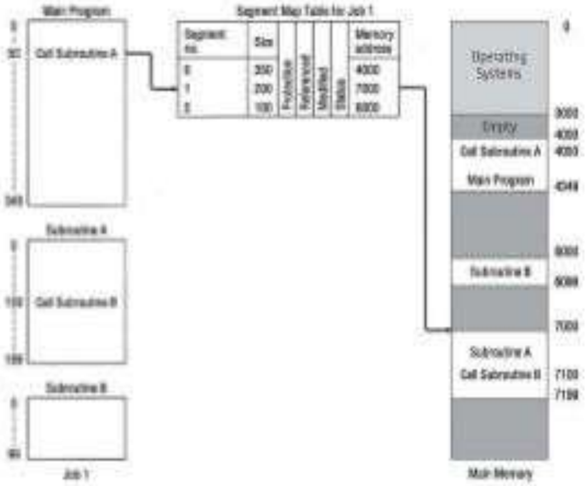
```
ACTUAL_MEM_LOC = BEGIN_MEM_LOC + DISPLACEMENT
```

figure 3.15 - During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.

STET WOMEN'S COLLEGE



(figure 3.15) The Segment Map Table tracks each segment for this job. Notice that Subroutine B has not yet been loaded into memory.
© Cengage Learning 2014



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

Can the displacement be larger than the size of the segment? No, not if the program is coded correctly; however, accidents 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.

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

SEGMENT_NUMBER & DISPLACEMENT

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

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.

Segmented/Demand Paged Memory Allocation

The segmented/demand paged memory allocation scheme evolved from the two we have just discussed. It is a combination of segmentation and demand paging, and it offers the logical benefits of segmentation, as well as the physical benefits of paging. The logic isn't new. The algorithms used by the demand paging and segmented memory management schemes are applied here with only minor modifications.

This allocation scheme doesn't keep each segment as a single contiguous unit but subdivides it into pages of equal size, smaller than most segments, and more easily manipulated than whole segments. Therefore, many of the problems of segmentation (compaction, external fragmentation, and secondary storage handling) are removed because the pages are of fixed length. This scheme, illustrated in Figure 3.16, requires four tables:

- The Job Table lists every job in process (one for the whole system).

STET WOMEN

- The Segment Map Table lists details about each segment (one for each job).
- The Page Map Table lists details about every page (one for each segment).
- The Memory Map Table monitors the allocation of the page frames in main memory (one for the whole system).

Note that the tables in Figure 3.16 have been simplified. The SMT actually includes additional information regarding protection (such as the authority to read, write, execute, and delete parts of the file), as well as which users have access to that segment (user only, group only, or everyone—some systems call these access categories owner, group, and world, respectively).

In addition, the PMT includes the status, modified, and referenced bits. To access a location in memory, the system must locate the address, which is composed of three entries: segment number, page number within that segment, and displacement within that page. It is a three-dimensional addressing scheme:

SEGMENT_NUMBER & PAGE_NUMBER & DISPLACEMENT

The major disadvantages of this memory allocation scheme are the overhead required for the extra tables and the time required to reference the segment table and the page table. To minimize the number of references, many systems use associative memory to speed up the process

Associative memory is a name given to several registers that are allocated to each job that is active. Their task is to associate several segment and page numbers belonging to the job being processed with their main memory addresses. These associative registers reside in main memory, and the exact number of registers varies from system to system.

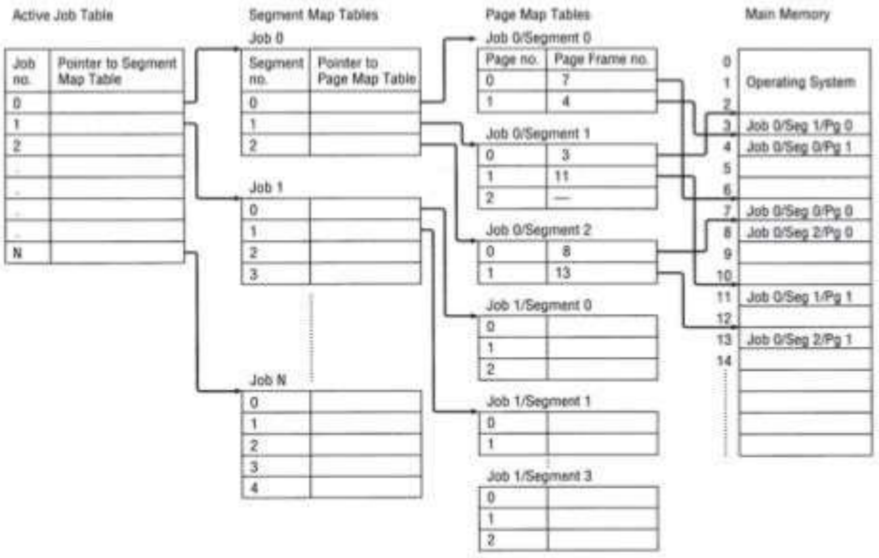
Here is a typical procedure: when a page is first requested, the job's SMT is searched to locate its PMT; then the PMT is loaded and searched to determine the page's location in memory. If the page isn't in memory, then a page interrupt is issued, the page is brought into memory, and the table is updated. (As the example indicates, loading the PMT can cause a page interrupt, or fault, as well.) This process is just as tedious as it sounds, but it gets easier. Since this segment's PMT (or part of it) now resides in memory, any other requests for pages within this segment can be quickly accommodated because there is no need to bring the PMT into memory. However, accessing these tables (SMT and PMT) is time-consuming.

That is the problem addressed by associative memory, which stores the information related to the most- recently-used pages. Then when a page request is issued, two searches begin—one through the segment and page tables and one through the contents of the associative registers. If the search of the associative registers is successful, then the search through the tables is stopped (or eliminated) and the address translation is performed using the information in the associative registers. However, if the search of associative memory fails, no time is lost because the search through the SMTs and PMTs had already begun (in this schema). When this search is successful and the main memory address from the PMT has been determined, the address is used to continue execution of the program and the reference is also stored in one of the associative registers. If all of the associative registers are full, then an LRU (or other) algorithm is used and the least-recently-referenced associative register is used to hold the information on this requested page.

Figure 3.16 : How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

STET WOMEN'S COLLEGE

Segmented/Demand Paged Memory Allocation (continued)



(figure 3.16)

How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

Understanding Operating Systems, Fifth Edition 43

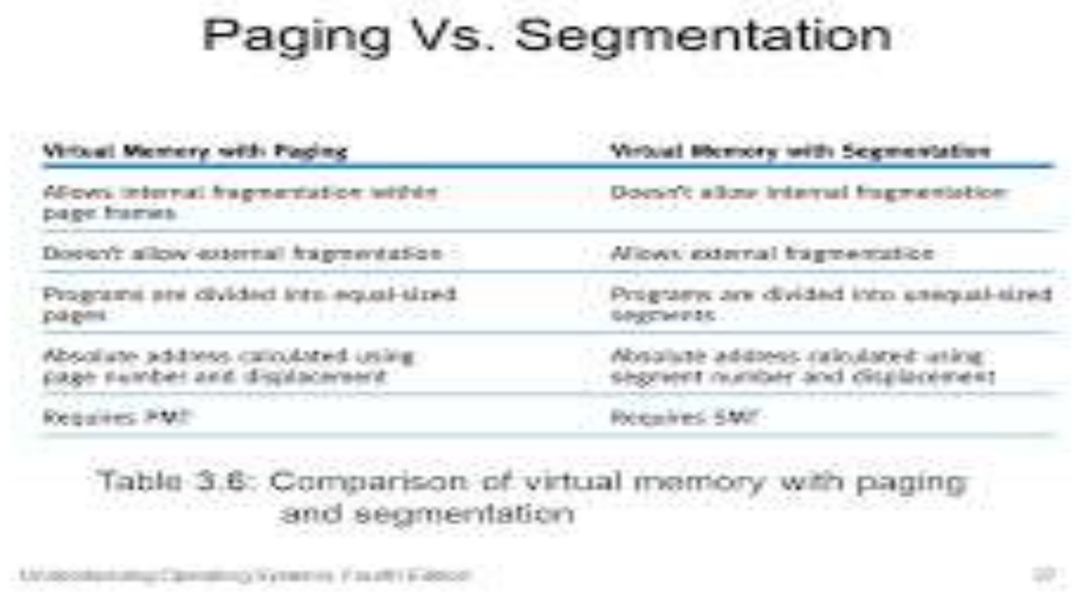
Virtual Memory

Demand paging made it possible for a program to execute even though only a part of a program was loaded into main memory. In effect, virtual memory removed the restriction imposed on maximum program size. This capability of moving pages at will between main memory and secondary storage gave way to a new concept appropriately named virtual memory. Even though only a portion of each program is stored in memory, it gives users the appearance that their programs are being completely loaded in main memory during their entire processing time—a feat that would require an incredible amount of main memory.

During the second generation, programmers started dividing their programs into sections that resembled working sets, really segments, originally called roll in/roll out and now called overlays. The program could begin with only the first overlay loaded into memory. As the first section neared completion, it would instruct the system to lay the second section of code over the first section already in memory. Then the second section would be processed. As that section finished, it would call in the third section to be overlaid, and so on until the program was finished. Some programs had multiple overlays in main memory at once.

Although the swapping of overlays between main memory and secondary storage was done by the system, the tedious task of dividing the program into sections was done by the programmer. It was the concept of overlays that suggested paging and segmentation and led to virtual memory, which was then implemented through demand paging and segmentation schemes. These schemes are compared in Table 3.6.

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



Paging Vs. Segmentation

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
Absolute address calculated using page number and displacement	Absolute address calculated using segment number and displacement
Requires PMT	Requires SMT

Table 3.6: Comparison of virtual memory with paging and segmentation

Understanding Operating Systems, Fourth Edition 20

Segmentation allowed for sharing program code among users. This means that 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 schema users share the code, which cannot be modified, and can modify the information stored in their own data areas as needed without affecting the data stored in other users' data areas.

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 is needed that is not already in memory, 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 programs spend a lot of time waiting—they wait for I/O to be performed; they wait for 93Virtual Memory (table 3.6) Comparison of the advantages and disadvantages of virtual memory with paging and segmentation. C7047_03_Ch03.qxd 1/12/10 4:13 PM Page 93 pages to be swapped in or out; and in a time-sharing environment, they wait when their time slice is up (their turn to use the processor is expired). In a multiprogramming environment, the waiting time isn't lost, and the CPU simply moves to another job.

Virtual memory management has several advantages:

- A job's size is no longer restricted to the size of main memory (or the free space 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 many users in a time-sharing environment.
- It eliminates external fragmentation and minimizes internal fragmentation by combining segmentation and paging (internal fragmentation occurs in the program).
- It allows the sharing of code and data.
- It facilitates dynamic linking of program segments. The advantages far outweigh these disadvantages:
- Increased processor hardware costs.
- Increased overhead for handling paging interrupts.
- Increased software complexity to prevent thrashing.

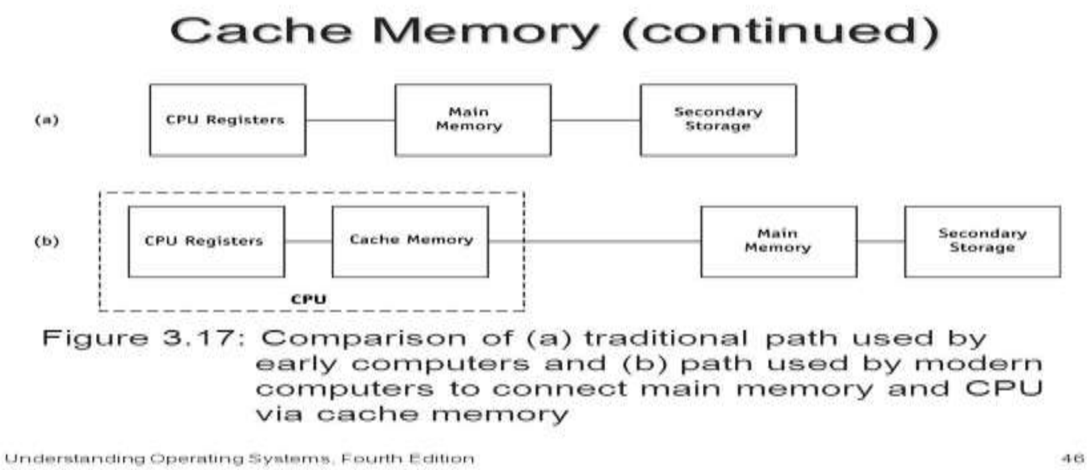
STET WOMEN'S COLLEGE

Cache Memory

Caching is based on the idea that the system can use a small amount of expensive highspeed memory to make a large amount of slower, less-expensive memory work faster than main memory.

Because the cache is small in size (compared to main memory), it can use faster, more expensive memory chips and can be five to ten times faster than main memory and match the speed of the CPU. Therefore, when frequently used data or instructions are stored in cache memory, memory access time can be cut down significantly and the CPU can execute instructions faster, thus raising the overall performance of the computer system.

Figure 3.17 : Comparison of (a) the traditional path used by early computers between main memory and the CPU and (b) the path used by modern computers to connect the main memory and the CPU via cache memory



Cache Memory (continued)

Figure 3.17: Comparison of (a) traditional path used by early computers and (b) path used by modern computers to connect main memory and CPU via cache memory

Understanding Operating Systems, Fourth Edition 46

As shown in Figure 3.17(a), the original architecture of a computer was such that data and instructions were transferred from secondary storage to main memory and then to special-purpose registers for processing, increasing the amount of time needed to complete a program. However, because the same instructions are used repeatedly in most programs, computer system designers thought it would be more efficient if the system would not use a complete memory cycle every time an instruction or data value is required. Designers found that this could be done if they placed repeatedly used data in general-purpose registers instead of in main memory, but they found that this technique required extra work for the programmer. Moreover, from the point of view of the system, this use of general-purpose registers was not an optimal solution because those registers are often needed to store temporary results from other calculations, and because the amount of instructions used repeatedly often exceeds the capacity of the general-purpose registers.

To solve this problem, computer systems automatically store 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 3.17(b).

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 bookmark file. If main memory is the Web and cache memory is your private bookmark file where you collect your most frequently used Web addresses, then your bookmark file is small and may contain only 0.00001 percent of all the addresses in 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 recently accessed addresses so you can access them quickly, just as the purpose of cache memory is to keep handy the most recently accessed data and instructions so that the CPU can access them repeatedly without wasting time.

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 and 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 implement, as follows:

Main Memory Transfer Algorithm

```
1 CPU puts the address of a memory location in the Memory Address
  Register and requests data or an instruction to be retrieved from that address
2 A test is performed to determine if the block containing this address is already in a cache slot:
   If YES, transfer the information to the CPU register – DONE
   If NO: Access main memory for the block containing the requested
          address
          Allocate a free cache slot to the block
          Perform these in parallel: Transfer the information to CPU Load the block into slot
DONE
```

This algorithm becomes more complicated if there aren't any free slots, which can occur because the size of cache memory is smaller than that of main memory, which means that individual slots cannot be permanently allocated to blocks. To address this contingency, the system needs a policy for block replacement, which could be one similar to those used in page replacement.

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 substantially improve the performance of the computer system.
- Block size. Because of the principle of locality of reference, as block size increases, the ratio of number of references found in the cache to the total number of references will be high.
- Block replacement algorithm. When all the slots are busy and a new block has to be brought into the cache, a block that is least likely to be used in the near future should be selected for replacement. However, as we saw in paging, this is nearly impossible to predict. 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 replacement, which requires a hardware mechanism to specify 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. On the one hand, it could be done every time that a change occurs, which would increase the number of memory writes, increasing overhead. On the other hand, it could be done 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 in the cache, making for a very efficient memory system. This measure of efficiency, called the cache hit ratio (h), is used to determine the performance of cache memory and represents the percentage of total memory requests that are found in the cache:

HitRatio = number of requests found in the cache/ 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.

HitRatio = (6 / 10) * 100 = 60%

STET WOMEN'S COLLEGE

On the other hand, 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.

HitRatio = (9 / 100) * 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 using the following formula:

AvgMemAccessTime = AvgCacheAccessTime + (1 – h) * AvgMainMemAccTime

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:

AvgMemAccessTime = 200 + (1 - 0.60) * 1000 = 600 nsec

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

AvgMemAccessTime = 200 + (1 - 0.09) * 1000 = 1110 nsec

STET WOMEN'S COLLEGE

Unit-III

Processor management

Process management is an integral part of any modern-day operating system (OS). The OS must allocate resources to processes, enable processes to share and exchange information, protect the resources of each process from other processes and enable synchronization among processes. To meet these requirements, the OS must maintain a data structure for each process, which describes the state and resource ownership of that process, and which enables the OS to exert control over each process.

Overview

In a simple system, one with a single user and one processor, the process is busy only when it is executing the user's jobs. However, when there are many users, such as in a multiprogramming environment, or when there are multiple processes competing to be run by a single CPU, the processor must be allocated to each job in a fair and efficient manner. This can be a complex task as we'll see in this chapter, which is devoted to single processor systems.

A **program** is an inactive unit, such as a file stored on a disk. A program is not a process. To an operating system, a program or job is a unit of work that has been submitted by the user. On the other hand, a process is an active entity that requires a set of resources, including a processor and special registers, to perform its function. A process, also called a task, is a single instance of a program in execution .

A **thread** is a portion of a process that can run independently.

The processor, also known as the CPU (for central processing unit), is the part of the machine that performs the calculations and executes the programs. Multiprogramming requires that the processor be allocated to each job or to each process for a period of time and deallocated at an appropriate moment. If the processor is deallocated during a program's execution, it must be done in such a way that it can be restarted later as easily as possible. It's a delicate procedure. To demonstrate, let's look at an everyday example.

In operating system terminology, you played the part of the CPU or processor. There were two programs, or jobs—one was the mission to assemble the toy and the second was to bandage the injury. When you were assembling the toy (Job A), each step you performed was a process. The call for help was an interrupt; and when you left the toy to treat your wounded friend, you left for a higher priority program. When you were interrupted, you performed a context switch when you marked Step 3 as the last completed instruction and put down your tools. Attending to the neighbor's injury became Job B. While you were executing the first-aid instructions, each of the steps you executed was again a process. And, of course, when each job was completed it was finished or terminated.

The Processor Manager would identify the series of events as follows:

get the input for Job A	(find the instructions in the box)
identify resources	(collect the necessary tools)
interrupt	(neighbor calls)
context switch to Job B	(mark your place in the instructions)
get the input for Job B	(find your first-aid book)
identify resources	(collect the medical supplies)
execute the process	(follow each first-aid step)

About Multi-Core Technologies

A dual-core, quad-core, or other multi-core CPU has more than one processor (also called a core) on the computer chip. Multi-core engineering was driven by the problems caused by nano-sized transistors and their ultra-close placement on a computer chip. Although chips with millions of transistors that were very close together helped increase system performance dramatically, the close proximity of these transistors also increased current leakage and the amount of heat generated by the chip.

One solution was to create a single chip (one piece of silicon) with two or more processor cores. In other words, they replaced a single large processor with two halfsized processors, or four quarter-sized processors. This design allowed the same sized chip to produce less heat and offered the opportunity to permit multiple calculations to take place at the same time.

STET WOMEN'S COLLEGE

Job Scheduling Versus Process Scheduling

The Processor Manager is a composite of two submanagers: 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.

The scheduling of the two jobs, to assemble the toy and to bandage the injury, was on a first-come, first-served and priority basis. Each job is initiated by the Job Scheduler based on certain criteria. Once a job is selected for execution, the Process Scheduler determines when each step, or set of steps, is executed—a decision that's also based on certain criteria. When you started assembling the toy, each step in the assembly instructions would have been selected for execution by the Process Scheduler.

Each job (or program) 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 only concerned with selecting jobs from a queue of incoming jobs and placing them in the process queue, whether batch or interactive, based on each job's characteristics. The Job Scheduler's goal is to put the jobs in a sequence that will use all of the system's resources as fully as possible.

This is an important function. For example, if the Job Scheduler selected several jobs to run consecutively and each had a lot of I/O, then the I/O devices would be kept very busy. The CPU might be busy handling the I/O (if an I/O controller were not used) so little computation might get done. On the other hand, if the Job Scheduler selected several consecutive jobs with a great deal of computation, then the CPU would be very busy doing that. The I/O devices would be idle waiting for I/O requests. Therefore, the Job Scheduler strives for a balanced mix of jobs that require large amounts of I/O interaction and jobs that require large amounts of computation. Its goal is to keep most components of the computer system busy most of the time.

Process Scheduler

Most of this chapter is dedicated to the Process Scheduler because after a job has been placed on the READY queue by the Job Scheduler, the Process Scheduler takes over. It determines which jobs will get the CPU, when, and for how long. It also decides when processing should be interrupted, determines which queues the job should be moved to during its execution, and recognizes when a job has concluded and should be terminated.

The Process Scheduler is the low-level scheduler that assigns the CPU to execute the processes of those jobs placed on the READY queue by the Job Scheduler. This becomes a crucial function 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 neighbor.

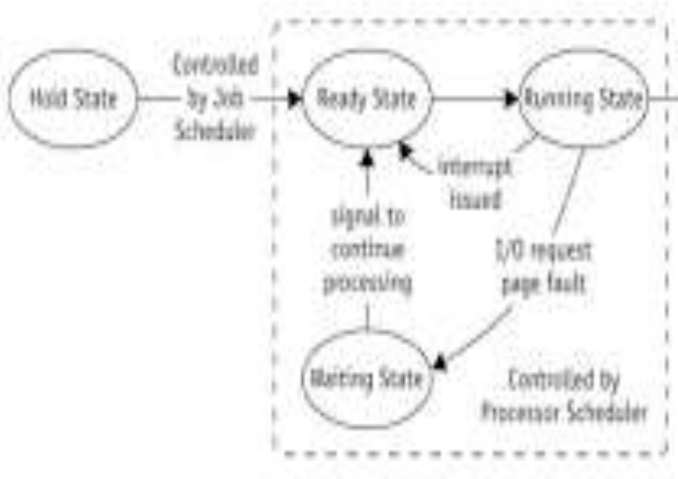
To schedule the CPU, the Process Scheduler takes advantage of a common trait among most computer programs: they alternate between CPU cycles and I/O cycles.

In a highly interactive environment, there's also a third layer of the Processor Manager 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 jobs to be completed faster. The jobs that are swapped out and eventually swapped back in are managed by the middle-level scheduler.

In a single-user environment, there's no distinction made between job and process scheduling because only one job is active in the system at any given time. So the CPU and all other resources are dedicated to that job, and to each of its processes in turn, until the job is completed.

Figure 4.2 : A typical job (or process) changes status as it moves through the system from HOLD to FINISHED.

STET WOMEN'S COLLEGE


Hold State
Controlled by Job Scheduler
Ready State
Running State
Controlled by Job Scheduler
Finished
interrupt issued
signal to continue processing
I/O request page fault
Waiting State
Controlled by Processor Scheduler

(figure 4.2)

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

Job and Process Status

As a job moves through the system, it's always in one of five states (or at least three) as it changes from HOLD to READY to RUNNING to WAITING and eventually to FINISHED as shown in Figure 4.2. These are called the job status or the process status.

From HOLD, the job moves to READY when it's ready to run but is waiting for the CPU. In some systems, the job (or process) might be placed on the READY list 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. Upon completion, the job is FINISHED and returned to the user.

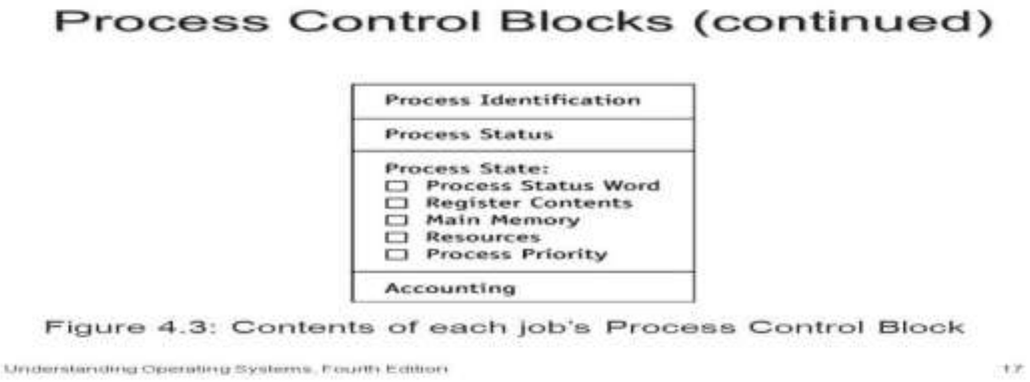
The transition from one job or process status to another is initiated by either the Job Scheduler or the Process Scheduler:

- The transition from HOLD to READY is initiated by the Job Scheduler according to some predefined policy. 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 some predefined algorithm (i.e., FCFS, SJN, priority scheduling, SRT, or round robin—all of which will be discussed shortly).
- The transition from RUNNING back to READY is handled by the Process Scheduler according to some predefined time limit or other criterion, for example a priority interrupt.
- The transition from RUNNING to WAITING is handled by the Process Scheduler and is initiated by an instruction in the job such as a command to READ, WRITE, or other I/O request, or one that requires a page fetch.
- 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 on the READY queue.
- Eventually, the transition from RUNNING to FINISHED is initiated by the Process Scheduler or the Job Scheduler either when (1) the job is successfully completed and it ends execution or (2) the operating system indicates that an error has occurred and the job is being terminated prematurely.

Process Control Blocks

Each process in the system is represented by a data structure called a Process Control Block (PCB) that performs the same function as a traveler's passport. The PCB (illustrated in Figure 4.3) contains the basic information about the job, including what it is, where it's going, how much of its processing has been completed, where it's stored, and how much it has spent in using resources.

Figure 4.3: Contents of each job's Process Control Block.



Process Control Blocks (continued)

Process Identification
Process Status
Process State: □ Process Status Word □ Register Contents □ Main Memory □ Resources □ Process Priority
Accounting

Figure 4.3: Contents of each job's Process Control Block

Understanding Operating Systems, Fourth Edition 17

Process Identification

Each job is uniquely identified by the user's identification and a pointer connecting it to its descriptor (supplied by the Job Scheduler when the job first enters the system and is placed on HOLD).

Process Status

This indicates the current status of the job—HOLD, READY, RUNNING, or WAITING —and the resources responsible for that status.

Process State

This contains all of the information needed to indicate the current state of the job such as:

- Process Status Word—the current instruction counter and register contents when the job isn't running but is either on HOLD or is READY or WAITING. If the job is RUNNING, this information is left undefined.
- Register Contents—the contents of the register if the job has been interrupted and is waiting to resume processing.
- Main Memory—pertinent information, including the address where the job is stored and, in the case of virtual memory, the mapping between virtual and physical memory locations.
- Resources—information about all resources allocated to this job. Each resource has an identification field listing its type and a field describing details of its allocation, such as the sector address on a disk. These resources can be hardware units (disk drives or printers, for example) or files.
- Process Priority—used by systems using a priority scheduling algorithm to select which job will be run next.

PCBs and Queueing

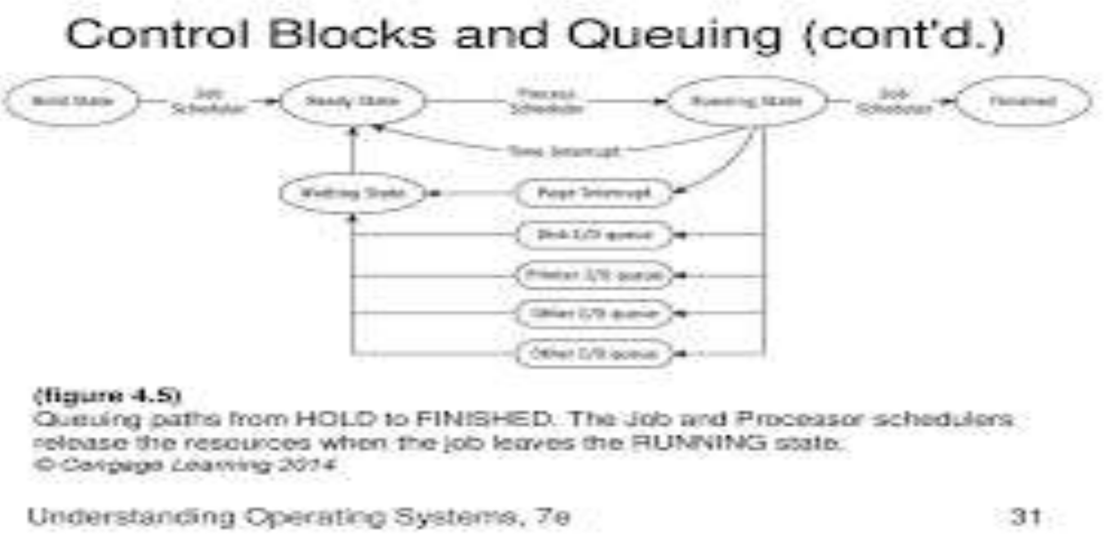
A job's PCB is created when the Job Scheduler accepts the job and is updated as the job progresses from the beginning to the end of its execution.

Queues use PCBs to track jobs the same way customs officials use passports to track international visitors. The PCB contains all of the data about the job needed by the operating system to manage the processing of the job. As the job moves through the system, its progress is noted in the PCB.

The PCBs, not the jobs, are linked to form the queues as shown in Figure 4.4. Although each PCB is not drawn in detail, the reader should imagine each queue as a linked list of PCBs. The PCBs for every ready job are linked on the READY queue, and all of the PCBs for the jobs just entering the system are linked on the HOLD queue.

Figure 4.4: Queuing paths from HOLD to FINISHED. The Job and Processor schedulers release the resources when the job leaves the RUNNING state.

SIET WOMEN'S COLLEGE



Control Blocks and Queuing (cont'd.)

(figure 4.5)
Queuing paths from HOLD to FINISHED. The Job and Processor schedulers release the resources when the job leaves the RUNNING state.
© Cengage Learning 2014

Understanding Operating Systems, 7e 31

The jobs that are WAITING, however, are linked together by “reason for waiting,” so the PCBs for the jobs in this category are linked into several queues. For example, the PCBs for jobs that are waiting for I/O on a specific disk drive are linked together, while those waiting for the printer are linked in a different queue. These queues need to be managed in an orderly fashion and that’s determined by the process scheduling policies and algorithms.

Process Scheduling Policies

In a multiprogramming environment, there are usually more jobs 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: (1) there are a finite number of resources (such as disk drives, printers, and tape drives); (2) some resources, once they’re allocated, can’t be shared with another job (e.g., printers); and (3) some resources require operator intervention—that is, they can’t be reassigned automatically from job to job (such as tape drives).

What’s a good process scheduling policy? Several criteria come to mind, but notice in the list below that some 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 interruptions.
- Minimize response time. Quickly turn around interactive requests. This could be done by running only interactive jobs and letting the batch 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 can be grouped to run more efficiently than interactive jobs).
- Minimize waiting time. Move jobs out of the READY queue as quickly as possible. This could only be done by reducing the number of users allowed on the system so the CPU would 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 (and 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 not giving special treatment to any job, regardless of its processing characteristics or priority.

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 will be discussed later), this extensive use of the CPU will build up the READY queue while emptying out the I/O queues, which creates an unacceptable imbalance in the system.

To solve this problem, the Process Scheduler often uses a timing mechanism and periodically interrupts running processes when a predetermined slice of time has expired. When that 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 happens: 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).

A scheduling strategy that interrupts the processing of a job and transfers the CPU to another job is called a preemptive scheduling policy; it is widely used in time-sharing environments. The alternative, of course, is a nonpreemptive 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 (with exceptions made for infinite loops, which are interrupted by both preemptive and nonpreemptive policies).

Process Scheduling Algorithms

The Process Scheduler relies on a process scheduling algorithm, based on a specific policy, to allocate the CPU and move jobs through the system. Early operating systems used nonpreemptive policies designed to move batch jobs through the system as efficiently as possible. Most current systems, with their emphasis on interactive use and response time, use an algorithm that takes care of the immediate requests of interactive users.

Here are six process scheduling algorithms that have been used extensively

First-come, first-served (FCFS) is a nonpreemptive scheduling algorithm that handles jobs 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 FIFO queue. This algorithm is fine for most batch systems, but it is unacceptable for interactive systems because interactive users expect quick response times.

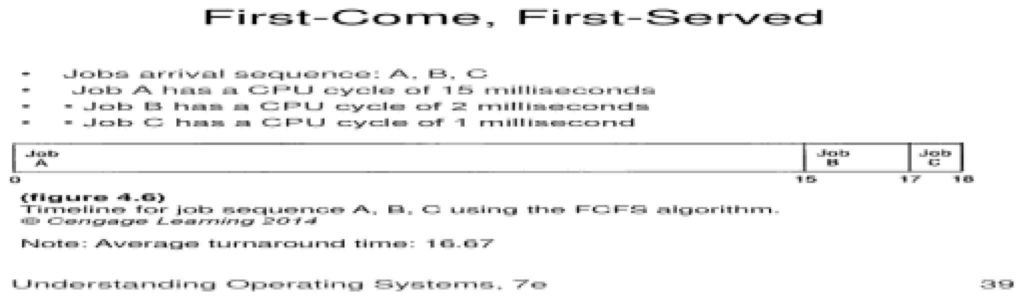
With FCFS, as a new job enters the system its PCB is linked to the end of the READY queue and it is removed from the front of the queue when the processor becomes available—that is, after it has processed all of the jobs before it in the queue. In a strictly FCFS system there are no WAIT queues (each job is run to completion), although there may be systems in which control (context) is switched on a natural wait (I/O request) and then the job resumes on I/O completion.

The following examples presume a strictly FCFS environment (no multiprogramming). Turnaround time is unpredictable with the FCFS policy; consider the following three jobs:

- Job A has a CPU cycle of 15 milliseconds.
- Job B has a CPU cycle of 2 milliseconds.
- Job C has a CPU cycle of 1 millisecond.

For each job, the CPU cycle contains both the actual CPU usage and the I/O requests. That is, it is the total run time. Using an FCFS algorithm with an arrival sequence of A, B, C, the timeline is shown in Figure 4.5.

Figure 4.5 : Timeline for job sequence A, B, C using the FCFS algorithm


First-Come, First-Served

- Jobs arrival sequence: A, B, C
- Job A has a CPU cycle of 15 milliseconds
- • Job B has a CPU cycle of 2 milliseconds
- • Job C has a CPU cycle of 1 millisecond

(figure 4.6)
Timeline for job sequence A, B, C using the FCFS algorithm.
© Cengage Learning 2014

Note: Average turnaround time: 16.67

Understanding Operating Systems, 7e 39

If all three jobs arrive almost simultaneously, we can calculate that the turnaround time for Job A is 15, for Job B is 17, and for Job C is 18. So the average turnaround time is:

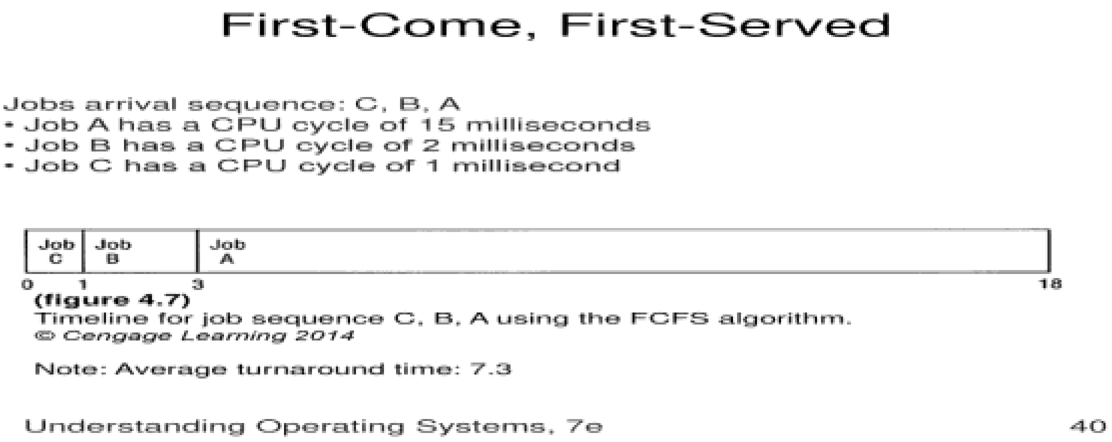
WOMEN'S COLLEGE

15 + 17 + 18/ 3 = 16.67

However, if the jobs arrived in a different order, say C, B, A, then the results using the same FCFS algorithm would be as shown in Figure 4.6.

Figure 4.6: Timeline for job sequence C, B, A using the 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 is:



First-Come, First-Served

Jobs arrival sequence: C, B, A
- Job A has a CPU cycle of 15 milliseconds
- Job B has a CPU cycle of 2 milliseconds
- Job C has a CPU cycle of 1 millisecond

(figure 4.7)
Timeline for job sequence C, B, A using the FCFS algorithm.
© Cengage Learning 2014

Note: Average turnaround time: 7.3

Understanding Operating Systems, 7e 40

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 is:

18 + 3 + 1/ 3 = 7.3

That's quite an improvement over the first sequence. Unfortunately, these two examples illustrate the primary disadvantage of using the FCFS concept—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, A). With four jobs the odds fall to 1 in 24, and so on.

In a strictly FCFS algorithm, neither situation occurs. However, the turnaround time is variable (unpredictable). For this reason, FCFS is a less attractive algorithm than one that would serve t
he shortest job first, as the next scheduling algorithm does, even in a non multiprogramming environment.

Shortest Job Next

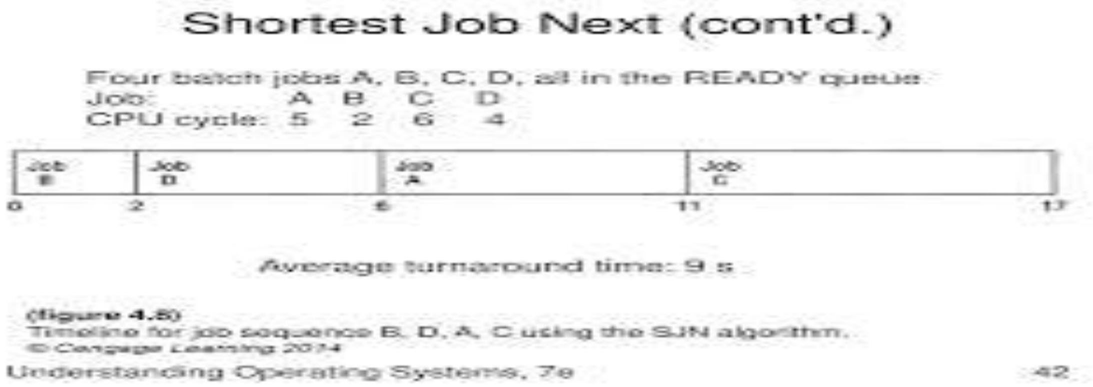
Shortest job next (SJN) is a nonpreemptive scheduling algorithm (also known as shortest job first, or SJF) that handles jobs based on the length of their CPU cycle time. It's easiest to implement in batch environments where the estimated CPU time required to run the job is given in advance by each user at the start of each job. However, it doesn't work in interactive systems because users don't estimate in advance the CPU time required to run their jobs.

For example, here are four batch jobs, all in the READY queue, for which the CPU cycle, or run time, is estimated as follows:

Job: A B C D
CPU cycle: 5264

The SJN algorithm would review the four jobs and schedule them for processing in this order: B, D, A, C. The timeline is shown in Figure 4.7.

Figure 4.7: Timeline for job sequence B, D, A, C using the SJN algorithm.



Shortest Job Next (cont'd.)

Four batch jobs A, B, C, D, all in the READY queue

Job:	A	B	C	D
CPU cycle:	5	2	6	4

Average turnaround time: 9 s

(figure 4.8)
Timeline for job sequence B, D, A, C using the SJN algorithm.
© Cengage Learning 2014

Understanding Operating Systems, 7e 42

The average turnaround time is:

2 + 6 + 11 + 17/ 4 = 9.0

Let's take a minute to see why this algorithm can be proved to be optimal and will consistently give the minimum average turnaround time. We'll use the previous example to derive a general formula.

If we look at Figure 4.7, we can see that Job B finishes in its given time (2), Job D finishes in its given time plus the time it waited for B to run (4 + 2), Job A finishes in its given time plus D's time plus B's time (5 + 4 + 2), and Job C finishes in its given time plus that of the previous three (6 + 5 + 4 + 2). So when calculating the average we have:

(2) + (4 + 2) + (5 + 4 + 2) + (6 + 5 + 4 + 2)/4 = 9.0

The time for the first job appears in the equation four times—once for each job. Similarly, the time for the second job appears three times (the number of jobs minus one). The time for the third job appears twice (number of jobs minus 2) and the time for the fourth job appears only once (number of jobs minus 3).

4 * 2 + 3 * 4 + 2 * 5 + 1 * 6/4 = 9.0

Because the time for the first job appears in the equation four times, it has four times the effect on the average time than does the length of the fourth job, which appears only once. Therefore, if the first job requires the shortest computation time, followed in turn by the other jobs, ordered from shortest to longest, then the result will be the smallest possible average. The formula for the average is as follows

t1(n) + t2(n – 1) + t3(n – 2) + … + tn(n(1)) /n

where n is the number of jobs in the queue and tj (j = 1, 2, 3,…,n) is the length of the CPU cycle for each of the jobs. However, the SJN algorithm is optimal only when all of the jobs are available at the same time and the CPU estimates are available and accurate.

Priority Scheduling

Priority scheduling is a nonpreemptive algorithm and one of the most common scheduling algorithms in batch systems, even though it may give slower turnaround to some users. This algorithm gives preferential treatment to important jobs. It allows the programs with the highest priority to be processed first, and they aren't interrupted until their CPU cycles (run times) are completed or a natural wait occurs. If two or more jobs with equal priority are present in the READY queue, the processor is allocated to the one that arrived first (first-come, first-served within priority).

Priorities can also be determined by the Processor Manager based on characteristics intrinsic to the jobs such as:

• Memory requirements. Jobs requiring large amounts of memory could be allocated lower priorities than those requesting small amounts of memory, or vice versa.

• Number and type of peripheral devices. Jobs requiring many peripheral devices would be allocated lower priorities than those requesting fewer devices.

• Total CPU time. Jobs having a long CPU cycle, or estimated run time, would be given lower priorities than those having a brief estimated run time.

• Amount of time already spent in the system. This is the total amount of elapsed time since the job was accepted for processing. Some systems increase the priority of jobs that have been in the system for an unusually long time to expedite their exit. This is known as aging.

SIET WOMEN'S COLLEGE

These criteria are used to determine default priorities in many systems. The default priorities can be overruled by specific priorities named by users. There are also preemptive priority schemes. These will be discussed later in this chapter in the section on multiple queues.

Shortest Remaining Time

Shortest remaining time (SRT) is the preemptive version of the SJN algorithm. The processor is allocated to the job closest to completion—but even this job can be preempted if a newer job in the READY queue has a time to completion that's shorter. This algorithm can't be implemented in an interactive system because it requires advance knowledge of the CPU time required to finish each job. It is often used in batch environments, when it is desirable to give preference to short jobs, even though SRT involves more overhead than SJN because the operating system has to frequently monitor the CPU time for all the jobs in the READY queue and must perform context switching for the jobs being swapped (switched) at preemption time

The example in Figure 4.8 shows how the SRT algorithm works with four jobs that arrived in quick succession (one CPU cycle apart).

Arrival time: 0 1 2 3

Job: A B C D

CPU cycle: 6 3 1 4

In this case, the turnaround time is the completion time of each job minus its arrival time:

Job: A B C D

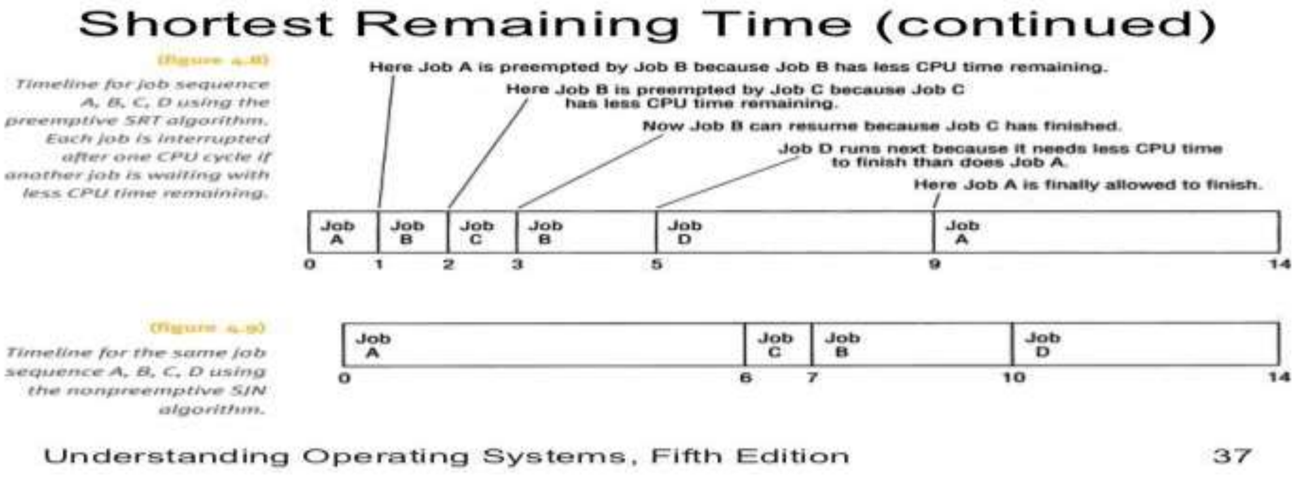
Turnaround: 14 4 1 6

So the average turnaround time is:

14 + 4 + 1 + 6/4 = 6.25

Figure 4.8 : Timeline for job sequence A, B, C, D using the preemptive SRT algorithm. Each job is interrupted after one CPU cycle if another job is waiting with less CPU time remaining & (figure 4.9) Timeline for the same job sequence A, B, C, D using the nonpreemptive SJN algorithm

WOMEN'S COLLEGE



Shortest Remaining Time (continued)

(figure 4.8)

Timeline for job sequence A, B, C, D using the preemptive SRT algorithm. Each job is interrupted after one CPU cycle if another job is waiting with less CPU time remaining.

Here Job A is preempted by Job B because Job B has less CPU time remaining.

Here Job B is preempted by Job C because Job C has less CPU time remaining.

Now Job B can resume because Job C has finished.

Job D runs next because it needs less CPU time to finish than does Job A.

Here Job A is finally allowed to finish.

(figure 4.9)

Timeline for the same job sequence A, B, C, D using the nonpreemptive SJN algorithm.

Understanding Operating Systems, Fifth Edition 37

How does that compare to the same problem using the non preemptive SJN policy? Figure 4.9 shows the same situation using SJN.

In this case, the turnaround time is:

Job:	A B C D
Turnaround:	6 9 5 11

So the average turnaround time is:

6 + 9 + 5 + 11/4 = 7.75

Note in Figure 4.9 that initially A is the only job in the READY queue so it runs first and continues until it's finished because SJN is a nonpreemptive algorithm. The next job to be run is C because when Job A is finished (at time 6), all of the other jobs (B, C, and D) have arrived. Of those three, C has the shortest CPU cycle, so it is the next one run, then B, and finally D.

Therefore, with this example, SRT at 6.25 is faster than SJN at 7.75. However, we neglected to include the time required by the SRT algorithm to do the context switching. Context switching is required by all preemptive algorithms. When Job A is preempted, all of its processing information must be saved in its PCB for later, when Job A's execution is to be continued, and the contents of Job B's PCB are loaded into the appropriate registers so it can start running again; this is a context switch. Later, when Job A is once again assigned to the processor, another context switch is performed. This time the information from the preempted job is stored in its PCB, and the contents of Job A's PCB are loaded into the appropriate registers.

Round Robin Round robin is a preemptive process scheduling algorithm that is used extensively in interactive systems. It's easy to implement and isn't based on job characteristics but on a predetermined slice of time that's given to each job to ensure that the CPU is equally shared among all active processes and isn't monopolized by any one job.

This time slice is called a time quantum and its size is crucial to the performance of the system. It usually varies from 100 milliseconds to 1 or 2 seconds. Jobs are placed in the READY queue using a first-come, first-served scheme and the Process Scheduler selects the first job from the front of the queue, sets the timer to the time quantum, and allocates the CPU to this job. If processing isn't finished when time expires, the job is preempted and put at the end of the READY queue and its information is saved in its PCB.

Round robin is a preemptive process scheduling algorithm that is used extensively in interactive systems. It's easy to implement and isn't based on job characteristics but on a predetermined slice of time that's given to each job to ensure that the CPU is equally shared among all active processes and isn't monopolized by any one job. This time slice is called a time quantum and its size is crucial to the performance of the system. It usually varies from 100 milliseconds to 1 or 2 seconds. Jobs are placed in the READY queue using a first-come, first-served scheme and the Process Scheduler selects the first job from the front of the queue, sets the timer to the time quantum, and allocates the CPU to this job. If processing isn't finished when time expires, the job is preempted and put at the end of the READY queue and its information is saved in its PCB.

In the event that the job's CPU cycle is shorter than the time quantum, one of two actions will take place:

(1) If this is the job's last CPU cycle and the job is finished, then all resources allocated to it are released and the completed job is returned to the user;

STET WOMENS COLLEGE

(2) if the CPU cycle has been interrupted by an I/O request, then information about the job is saved in its PCB and it is linked at the end of the appropriate I/O queue. Later, when the I/O request has been satisfied, it is returned to the end of the READY queue to await allocation of the CPU.

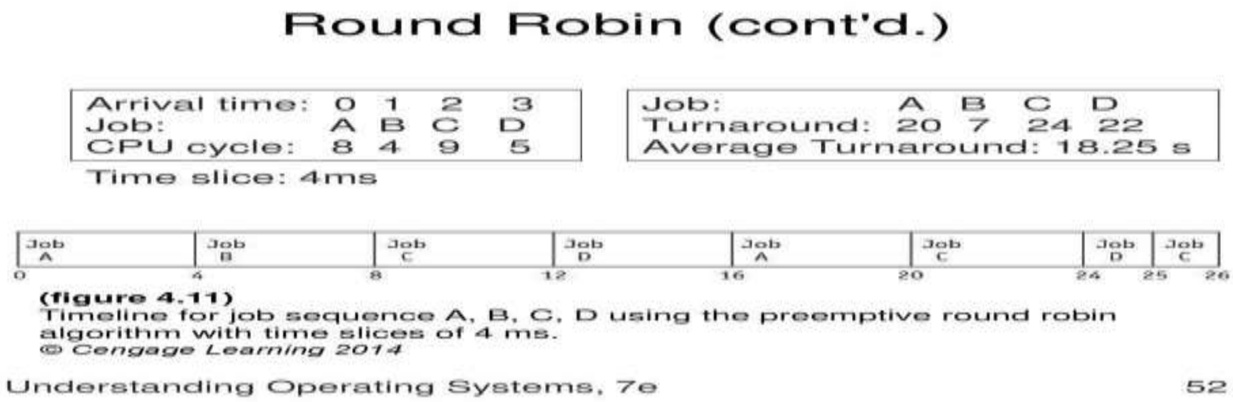
The example in Figure 4.10 illustrates a round robin algorithm with a time slice of 4 milliseconds (I/O requests are ignored):

Arrival time: 0 1 2 3

Job: A B C D

CPU cycle: 8 4 9 5

Figure 4.10: Timeline for job sequence A, B, C, D using the preemptive round robin algorithm with time slices of 4 ms



Round Robin (cont'd.)

Arrival time: 0 1 2 3
Job: A B C D
CPU cycle: 8 4 9 5
Time slice: 4ms

Job: A B C D
Turnaround: 20 7 24 22
Average Turnaround: 18.25 s

(figure 4.11)
Timeline for job sequence A, B, C, D using the preemptive round robin algorithm with time slices of 4 ms.
© Cengage Learning 2014

Understanding Operating Systems, 7e 52

The turnaround time is the completion time minus the arrival time:

Job: A B C D

Turnaround: 20 7 24 22

So the average turnaround time is:

20 + 7 + 24 + 22/4 = 18.25

Note that in Figure 4.10, Job A was preempted once because it needed 8 milliseconds to complete its CPU cycle, while Job B terminated in one time quantum. Job C was preempted twice because it needed 9 milliseconds to complete its CPU cycle, and Job D was preempted once because it needed 5 milliseconds. In their last execution or swap into memory, both Jobs D and C used the CPU for only 1 millisecond and terminated before their last time quantum expired, releasing the CPU sooner.

In Figure 4.11, the first case (a) has a time quantum of 10 milliseconds and there is no context switching (and no overhead). The CPU cycle ends shortly before the time

SIET WOMEN'S COLLEGE

Figure 4.11: Context switches for three different time quantums. In (a), Job A (which requires only 8 cycles to run to completion) finishes before the time quantum of 10 expires. In (b) and (c), the time quantum expires first, interrupting the jobs.

quantum expires and the job runs to completion. For this job with this time quantum, there is no difference between the round robin algorithm and the FCFS algorithm. In the second case (b), with a time quantum of 5 milliseconds, there is one context switch. The job is preempted once when the time quantum expires, so there is some overhead for context switching and there would be a delayed turnaround based on the number of other jobs in the system. In the third case (c), with a time quantum of 1 millisecond, there are 10 context switches because the job is preempted every time the time quantum expires; overhead becomes costly and turnaround time suffers accordingly.

Here are two general rules of thumb for selecting the proper time quantum: (1) it should be long enough to allow 80 percent of the CPU cycles to run to completion, and (2) it should be at least 100 times longer than the time required to perform one context switch. These rules are used in some systems, but they are not inflexible.

Multiple-Level Queues

Multiple-level queues isn't really a separate scheduling algorithm but works in conjunction with several of the schemes already discussed and is found in systems with jobs that can be grouped according to a common characteristic. We've already introduced at least one kind of multiple-level queue—that of a priority-based system with different queues for each priority level.

Another kind of system might gather all of the CPU-bound jobs in one queue and all I/O-bound jobs in another. The Process Scheduler then alternately selects jobs from each queue to keep the system balanced.
Multiple-level queues raise some interesting questions:

- Is the processor allocated to the jobs in the first queue until it is empty before moving to the next queue, or does it travel from queue to queue until the last job on the last queue has been served and then go back to serve the first job on the first queue, or something in between?
- Is this fair to those who have earned, or paid for, a higher priority?
- Is it fair to those in a low-priority queue?
- If the processor is allocated to the jobs on the first queue and it never empties out, when will the jobs in the last queues be served?
- Can the jobs in the last queues get "time off for good behavior" and eventually move to better queues?

The answers depend on the policy used by the system to service the queues. There are four primary methods to the movement: not allowing movement between queues, moving jobs from queue to queue, moving jobs from queue to queue and increasing the time quantums for lower queues, and giving special treatment to jobs that have been in the system for a long time (aging).

Case 1: No Movement Between Queues No movement between queues is a very simple policy that rewards those who have high-priority jobs. The processor is allocated to the jobs in the high-priority queue in FCFS fashion and it is allocated to jobs in low-priority queues only when the highpriority queues are empty. This policy can be justified if there are relatively few users with high-priority jobs so the top queues quickly empty out, allowing the processor to spend a fair amount of time running the low-priority jobs.

Case 2: Movement Between Queues

Movement between queues is a policy that adjusts the priorities assigned to each job: High-priority jobs are treated like all the others once they are in the system. (Their initial priority may be favorable.) When a time quantum interrupt occurs, the job is preempted and moved to the end of the next lower queue. A job may also have its priority increased; for example, when it issues an I/O request before its time quantum has expired. This policy is fairest in a system in which the jobs are handled according to their computing cycle characteristics: CPU-bound or I/O-bound. This assumes that a job that exceeds its time quantum is CPU-bound and will require more CPU allocation than one that requests I/O before the time quantum expires. Therefore, the CPU-bound jobs are placed at the end of the next lower-

SIET WOMEN'S COLLEGE

level queue when they're preempted because of the expiration of the time quantum, while I/O-bound jobs are returned to the end of the next higher-level queue once their I/O request has finished. This facilitates I/Obound jobs and is good in interactive systems.

Case 3: Variable Time Quantum Per Queue Variable time quantum per queue is a variation of the movement between queues policy, and it allows for faster turnaround of CPU-bound jobs. In this scheme, each of the queues is given a time quantum twice as long as the previous queue. The highest queue might have a time quantum of 100 milliseconds. So the second-highest queue would have a time quantum of 200 milliseconds, the third would have 400 milliseconds, and so on. If there are enough queues, the lowest one might have a relatively long time quantum of 3 seconds or more. If a job doesn't finish its CPU cycle in the first time quantum, it is moved to the end of the next lower-level queue; and when the processor is next allocated to it, the job executes for twice as long as before. With this scheme a CPU-bound job can execute for longer and longer periods of time, thus improving its chances of finishing faster.

Case 4: Aging Aging is used to ensure that jobs in the lower-level queues will eventually complete their execution. The operating system keeps track of each job's waiting time and when a job gets too old—that is, when it reaches a certain time limit—the system moves the job to the next highest queue, and so on until it reaches the top queue. A more drastic aging policy is one that moves the old job directly from the lowest queue to the end of the top queue. Regardless of its actual implementation, an aging policy guards against the indefinite postponement of unwieldy jobs. As you might expect, indefinite postponement means that a job's execution is delayed for an undefined amount of time because it is repeatedly preempted so other jobs can be processed.

A Word About Interrupts

another type of interrupt that occurs when the time quantum expires and the processor is deallocated from the running job and allocated to another one. There are other interrupts that are caused by events internal to the process. I/O interrupts are issued when a READ or WRITE command is issued.

Internal interrupts, or synchronous interrupts, also occur as a direct result of the arithmetic operation or job instruction currently being processed.

Illegal arithmetic operations, such as the following, can generate interrupts:

- Attempts to divide by zero
- Floating-point operations generating an overflow or underflow
- Fixed-point addition or subtraction that causes an arithmetic overflow Illegal job instructions, such as the following, can also generate interrupts: • Attempts to access protected or nonexistent storage locations
- Attempts to use an undefined operation code
- Operating on invalid data
- Attempts to make system changes, such as trying to change the size of the time quantum

The control program that handles the interruption sequence of events is called the interrupt handler. When the operating system detects a nonrecoverable error, the interrupt handler typically follows this sequence:

1. The type of interrupt is described and stored—to be passed on to the user as an error message.
2. The state of the interrupted process is saved, including the value of the program counter, the mode specification, and the contents of all registers.
3. The interrupt is processed: The error message and state of the interrupted process are sent to the user; program execution is halted; any resources allocated to the job are released; and the job exits the system.
4. The processor resumes normal operation.

Deadlock

Deadlock is more serious than indefinite postponement or starvation because it affects more than one job. Because resources are being tied up, the entire system (not just a few programs) is affected. The example most often used to illustrate deadlock is a traffic jam.

As shown in Figure 5.1, 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 the rear of a line moves back. Obviously it requires outside intervention to remove one of the four vehicles from an intersection or to make a line move back. Only then can the deadlock be resolved. Deadlocks became prevalent with the introduction of interactive systems, which generally improve the use of resources through dynamic resource sharing, but this capability also increases the possibility of deadlocks.

SIET WOMEN'S COLLEGE

Seven Cases of Deadlock

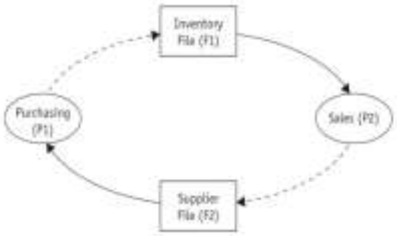
A deadlock usually occurs when nonsharable, nonpreemptable resources, such as files, printers, or scanners, are allocated to jobs that eventually require other nonsharable, nonpreemptive resources—resources that have been locked by other jobs.

Directed graphs visually represent the system's resources and processes, and show how they are deadlocked. Using a series of squares (for resources) and circles (for processes), and connectors with arrows (for requests), directed graphs can be manipulated to understand how deadlocks occur.

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a deadlock can occur as the simplified directed graph shown in Figure graphically illustrates

Figure : Case 1. These two processes, shown as circles, are each waiting for a resource, shown as rectangles, that has already been allocated to the other process, thus creating a deadlock



(figure 5.2)

Case 1. These two processes, shown as circles, are each waiting for a resource, shown as rectangles, that has already been allocated to the other process, thus creating a deadlock.

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. 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 until the process is completed. 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.

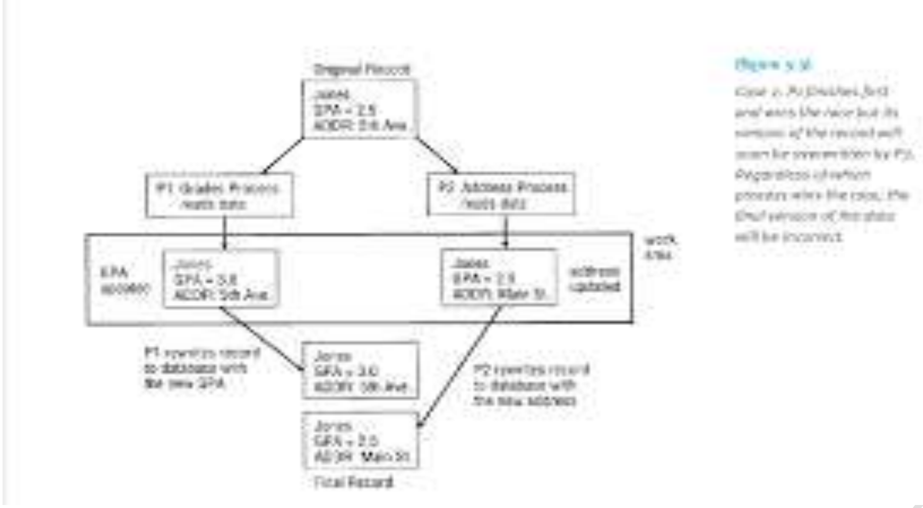
Here's a system that locks each record when it is accessed until the process is completed. There are two processes (P1 and P2), each of which needs to update two records (R1 and R2), and the following sequence leads to a deadlock:

1. P1 accesses R1 and locks it.
2. P2 accesses R2 and locks it.
3. P1 requests R2, which is locked by P2.
4. P2 requests R1, which is locked by P1.

An alternative, of course, is to avoid the use of locks—but that leads to other difficulties. If locks are not used to preserve their integrity, the updated 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. This is known as a race between

STET WOMEN'S COLLEGE

processes and is illustrated in the following example and Figure **Case 2**


Original Record
Jones
GPA = 2.5
ADDR: 5th Ave.
P1 Grades Process
reads data
P2 Address Process
reads data
work area
GPA added
Jones
GPA = 3.0
ADDR: 5th Ave.
Jones
GPA = 2.5
ADDR: Main St.
address updated
P1 rewrites record to database with the new GPA
Jones
GPA = 3.0
ADDR: 5th Ave.
P2 rewrites record to database with the new address
Jones
GPA = 2.5
ADDR: Main St.
Final Record
Figure 5.x
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 you are a student of a university that maintains most of its files on a database that can be accessed by several different programs, including one for grades and another listing home addresses. You’ve just moved so you send the university a change of address form at the end of the fall term, shortly after grades are submitted. And one fateful day, both programs race to access your record in the database:
1. The grades process (P1) is the first to access your record (R1), and it copies the record to its work area
2. The address process (P2) accesses your record (R1) and copies it to its work area.
3. P1 changes your student record (R1) by entering your grades for the fall term and calculating your new grade average.
4. P2 changes your record (R1) by updating the address field.
5. P1 finishes its work first and rewrites its version of your record back to the database. Your grades have been updated, but your address hasn’t. 6. P2 finishes and rewrites its updated record back to the database. Your address has been changed, but your grades haven’t. According to the database, you didn’t attend school this term.

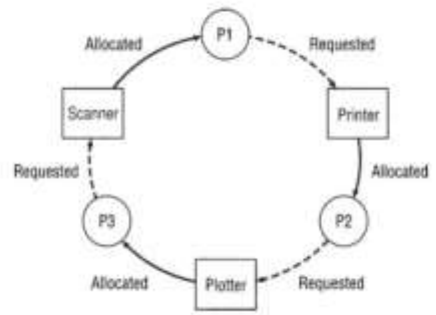
Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as a cluster of DVD read/write drives, can also deadlock the system. Let’s say two users from the local board of education are each running a program (P1 and P2), and both programs will eventually need two DVD drivers to copy files from one disc to another. The system is small, however, and when the two programs are begun, only two DVD-R drives are available and they’re allocated on an “as requested” basis. Soon the following sequence transpires:
1. P1 requests drive 1 and gets it.
2. P2 requests drive 2 and gets it.
3. P1 requests drive 2 but is blocked.
4. P2 requests drive 1 but is blocked.
Neither job can continue because each is waiting for the other to finish and release its drive—an event that will never occur. A similar series of events could deadlock any group of dedicated devices.

Case 4: Deadlocks in Multiple Device Allocation Deadlocks aren’t restricted to processes 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

Figure: 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.

STET WOMEN'S COLLEGE



(figure 5.4)

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.

onsider the case of an engineering design firm with three programs (P1, P2, and P3) and three dedicated devices: scanner, printer, and plotter. The following sequence of events will result in deadlock:

1. P1 requests and gets the scanner.
2. P2 requests and gets the printer.
3. P3 requests and gets the plotter.
4. P1 requests the printer but is blocked.
5. P2 requests the plotter but is blocked.
6. P3 requests the scanner but is blocked. As in the earlier examples, none of the jobs can continue because each is waiting for a resource being held by another.

Case 5: Deadlocks in Spooling

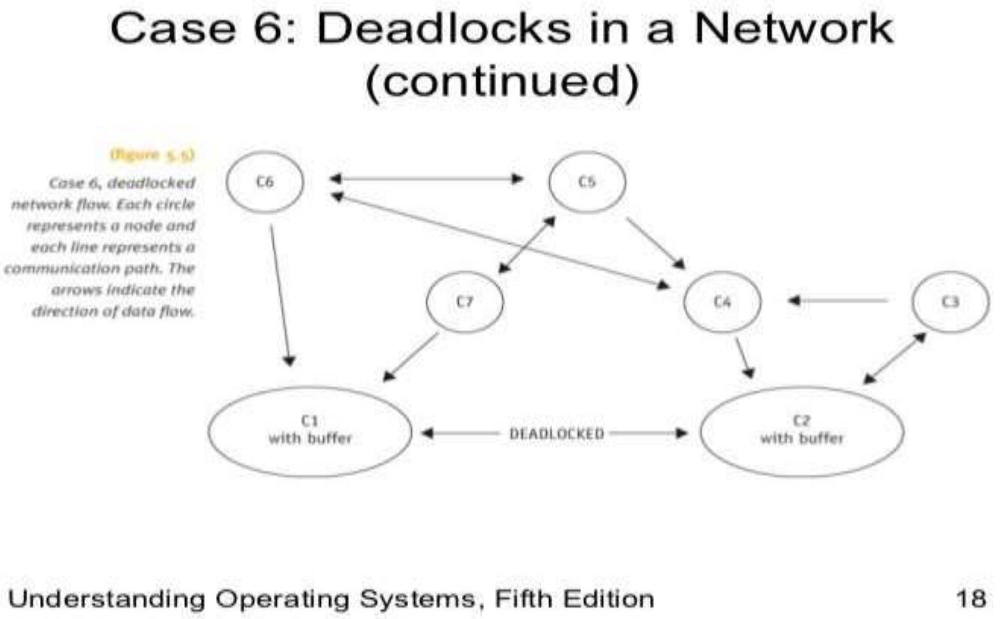
Although in the previous example the printer was a dedicated device, printers are usually sharable devices, called virtual devices, that use high-speed storage to transfer data between it and the CPU. The spooler accepts output from several users and acts as a temporary storage area for all output until the printer is ready to accept it. This process is called spooling. 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 happens like this. Let's say it's one hour before the big project is due for a computer class. Twenty-six frantic programmers key in their final changes and, with only minutes to spare, all issue print commands. The spooler receives the pages one at a time from each of the students but the pages are received separately, several page ones, page twos, etc. 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 programs but the last page for none of them. Alas, the spooler is full of partially completed output so no other pages can be accepted, but none of the jobs can be printed out (which would release their disk space) because the printer only accepts completed output files. It's an unfortunate state of affairs. This scenario isn't limited to printers. Any part of the system that relies on spooling, such as one that handles incoming jobs or transfers files over a network, is vulnerable to such a deadlock.

Case 6: Deadlocks in a Network

A network that's congested or has filled a large percentage of its I/O buffer space can become deadlocked if it doesn't have protocols to control the flow of messages through the network as shown in Figure .

Figure : 5.5 : Case 6, deadlocked network flow. Notice that only two nodes, C1 and C2, have buffers. Each circle represents a node and each line represents a communication path. The arrows indicate the direction of data flow.

STET WOMEN'S COLLEGE



Case 6: Deadlocks in a Network (continued)

(Figure 5.5) Case 6, deadlocked network flow. Each circle represents a node and each line represents a communication path. The arrows indicate the direction of data flow.

Understanding Operating Systems, Fifth Edition 18

or example, a medium-sized word-processing center has seven computers on a network, each on different nodes. C1 receives messages from nodes C2, C6, and C7 and sends messages to only one: C2. C2 receives messages from nodes C1, C3, and C4 and sends messages to only C1 and C3. The direction of the arrows in Figure 5.5 indicates the flow of messages.

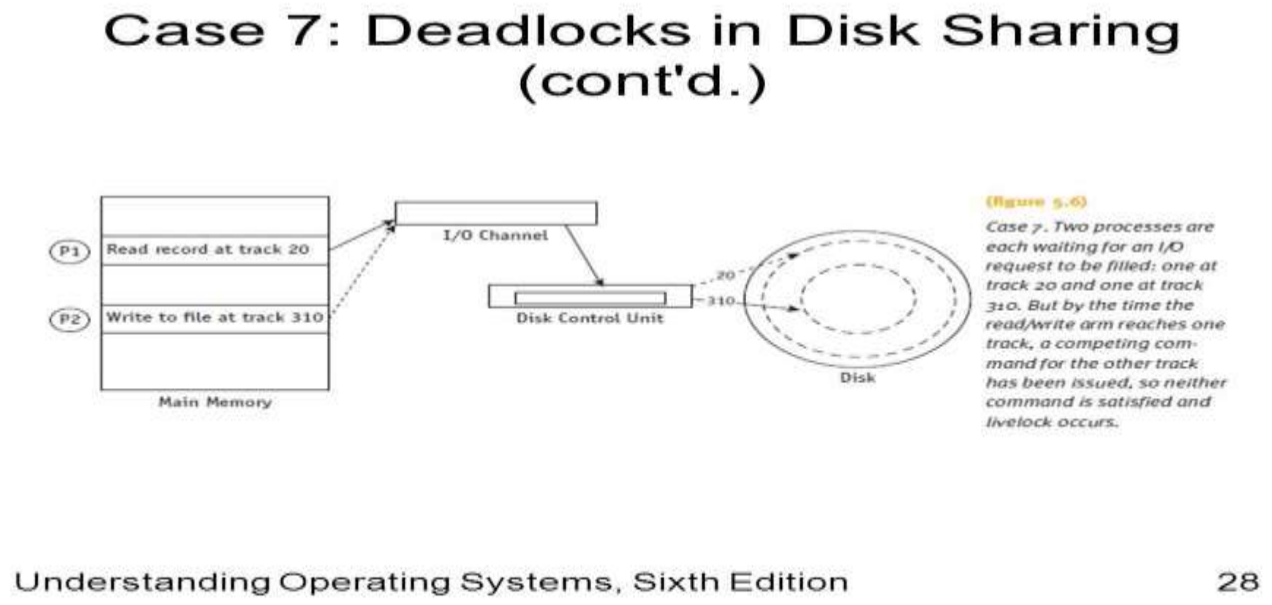
Messages received by C1 from C6 and C7 and destined for C2 are buffered in an output queue. Messages received by C2 from C3 and C4 and destined for C1 are buffered in an output queue. As the traffic increases, the length of each output queue increases until all of the available buffer space is filled. At this point C1 can't accept any more messages (from C2 or any other computer) because there's no more buffer space available to store them. For the same reason, C2 can't accept any messages from C1 or any other computer, not even a request to send. The communication path between C1 and C2 becomes deadlocked; and because C1 can't send messages to any other computer except C2 and can only receive messages from C6 and C7, those C7047_05_Ch05.qxd 1/12/10 4:50 PM Page 147 routes also become deadlocked. C1 can't send word to C2 about the problem and so the deadlock can't be resolved without outside intervention.

Case 7: Deadlocks in Disk Sharing

Disks are designed to be shared, so it's not uncommon for two processes to be accessing different areas of the same disk. This ability to share creates an active type of deadlock, known as livelock. Processes use a form of busy-waiting that's different from a natural wait. In this case, it's waiting to share a resource but never actually gains control of it. In Figure 5.6, two competing processes are sending conflicting commands, causing livelock. Notice that neither process is blocked, which would cause a deadlock. Instead, each is active but never reaches fulfillment.

Figure : Case 7. Two processes are each waiting for an I/O request to be filled: one at track 20 and one at track 310. But by the time the read/write arm reaches one track, a competing command for the other track has been issued, so neither command is satisfied and livelock occurs.

STET WOMEN'S COLLEGE



Case 7: Deadlocks in Disk Sharing (cont'd.)

(figure 5.6)

Case 7. Two processes are each waiting for an I/O request to be filled: one at track 20 and one at track 310. But by the time the read/write arm reaches one track, a competing command for the other track has been issued, so neither command is satisfied and livelock occurs.

Understanding Operating Systems, Sixth Edition 28

For example, at an insurance company the system performs many daily transactions.

Conditions for Deadlock

In each of these seven cases, the deadlock (or livelock) involved the interaction of several processes and resources, but each deadlock was preceded by the simultaneous occurrence of four conditions that the operating system (or other systems) could have recognized: mutual exclusion, resource holding, no preemption, and circular wait.

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 deadlock.

When two people meet between landings, they can't pass because 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 (a dedicated resource), is the first condition for deadlock.

When two people meet on the stairs and each one holds ground and waits for the other to retreat, that is an example of resource holding (as opposed to resource sharing), the second condition for deadlock.

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 preemption, the lack of temporary reallocation of resources, and is the third condition for deadlock.

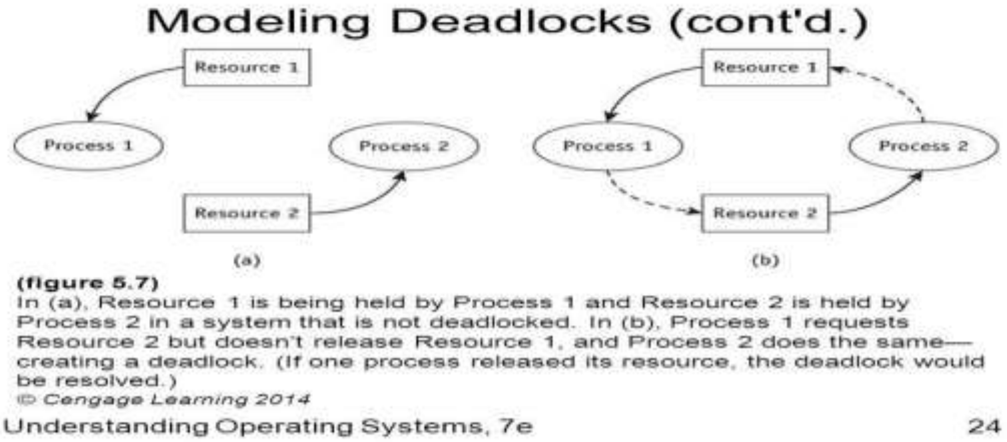
These three lead to the fourth condition of circular wait in which each person (or process) involved in the impasse is waiting for another to voluntarily release the step (or resource) so that at least one will be able to continue on and eventually arrive at the destination.

Modeling Deadlocks

Holt showed how the four conditions can be modeled using directed graphs. (We used modified directed graphs in Figure 5.2 and Figure 5.4.) These graphs use two kinds of symbols: processes represented by circles and resources represented by squares. A solid arrow from a resource to a process, shown in Figure 5.7(a), means that the process is holding that resource. A dashed line with an arrow from a process to a resource, shown in Figure 5.7(b), means that the process is waiting for that resource. The direction of the arrow indicates the flow. If there's a cycle in the graph then there's a deadlock involving the processes and the resources in the cycle.

STET WOMENS CO

Figure 5.7:



Modeling Deadlocks (cont'd.)

(figure 5.7)
In (a), Resource 1 is being held by Process 1 and Resource 2 is held by Process 2 in a system that is not deadlocked. In (b), Process 1 requests Resource 2 but doesn't release Resource 1, and Process 2 does the same—creating a deadlock. (If one process released its resource, the deadlock would be resolved.)
© Cengage Learning 2014

Understanding Operating Systems, 7e 24

The following system has three processes—P1, P2, P3—and three resources—R1, R2, R3—each of a different type: printer, disk drive, and plotter. Because there is no specified order in which the requests are handled, we'll look at three different possible scenarios using graphs to help us detect any deadlocks.

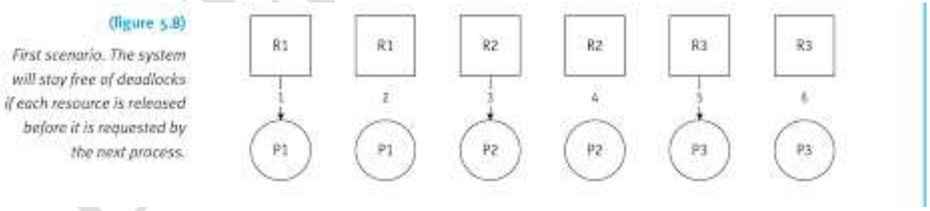
Scenario 1

The first scenario's sequence of events is shown in Table 5.1. The directed graph is shown in Figure 5.8.

Event	Action
1	P1 requests and is allocated the printer R1.
2	P1 releases the printer R1.
3	3 P2 requests and is allocated the disk drive R2.
4	4 P2 releases the disk R2.
5	5 P3 requests and is allocated the plotter R3.
6	6 P3 releases the plotter R3

Notice in the directed graph that there are no cycles. Therefore, we can safely conclude that a deadlock can't occur even if each process requests every resource if the resources are released before the next process requests them.

Figure 5.8 : First scenario. The system will stay free of deadlocks if each resource is released before it is requested by the next process.



(figure 5.8)
First scenario. The system will stay free of deadlocks if each resource is released before it is requested by the next process.

Scenario 2

Now, consider a second scenario's sequence of events shown in Table 5.2.

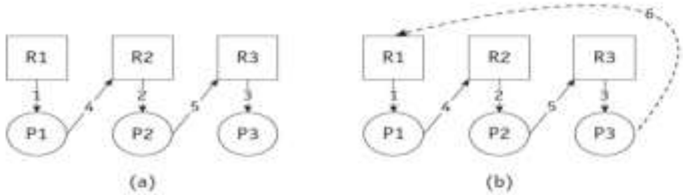
Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
	P2 requests R3.

SLI WOMEN'S COLLEGE

5	P3 requests R1.

The progression of the directed graph is shown in Figure 5.9. A deadlock occurs because every process is waiting for a resource that is being held by another process, but none will be released without intervention.

(figure 5.9) Second scenario. The system (a) becomes deadlocked (b) when P3 requests R1. Notice the circular wait.



(figure 5.9)

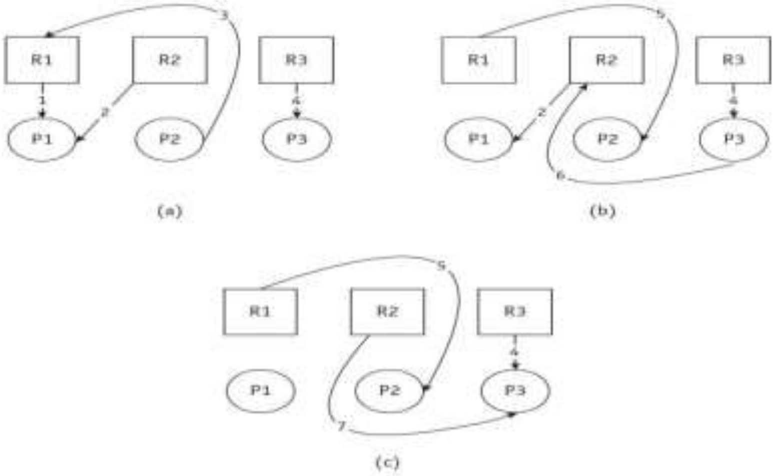
Second scenario. The system (a) becomes deadlocked (b) when P3 requests R1. Notice the circular wait.

Scenario 3

The third scenario is shown in Table 5.3. As shown in Figure 5.10, the resources are released before deadlock can occur.

Event	Action
1	P1 requests and is allocated R1.
2	P1 requests and is allocated R2.
3	P2 requests R1.
4	P3 requests and is allocated R3.
5	P1 releases R1, which is allocated to P2.
6	P3 requests R2. 7 P1 releases R2, which is allocated to P3

Figure 5.10



(figure 5.10)

The third scenario. After event 4, the directed graph looks like (a) and P2 is blocked because P1 is holding on to R1. However, event 5 breaks the deadlock and the graph soon looks like (b). Again there is a blocked process, P3, which must wait for the release of R2 in event 7 when the graph looks like (c).

Strategies for Handling Deadlocks

As these examples show, the requests and releases are received in an unpredictable order, which makes it very difficult to design a foolproof preventive policy. In general, operating systems use one of three 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 and recover from it gracefully (detection).

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

Mutual exclusion is necessary in any computer system because some resources such as memory, CPU, and dedicated devices must be exclusively allocated to one user at a time. In the case of I/O devices, such as printers, the mutual exclusion may be bypassed by spooling, which allows the output from many jobs to be stored in separate temporary spool files at the same time, and each complete output file is then selected for printing when the device is ready. However, we may be trading one type of deadlock (Case 3: Deadlocks in Dedicated Device Allocation) for another (Case 5: Deadlocks in Spooling).

Resource holding, where a job holds on to one resource while waiting for another one that's not yet available, could be sidestepped by forcing each job to request, at creation time, every resource it will need to run to completion. For example, if every job in a batch system is given as much memory as it needs, then the number of active jobs will be dictated by how many can fit in memory—a policy that would significantly decrease the degree of multiprogramming. In addition, peripheral devices would be idle because they would be allocated to a job even though they wouldn't be used all the time. As we've said before, this was used successfully in batch environments although it reduced the effective use of resources and restricted the amount of multiprogramming. But it doesn't work as well in interactive systems.

No preemption could be bypassed by allowing the operating system to deallocate resources from jobs. This can be done if the state of the job can be easily saved and restored, as when a job is preempted in a round robin environment or a page is swapped to secondary storage in a virtual memory system. On the other hand, preemption of a dedicated I/O device (printer, plotter, tape drive, and so on), or of files during the modification process, can require some extremely unpleasant recovery tasks.

This scheme of "hierarchical ordering" removes the possibility of a circular wait and therefore guarantees the removal of deadlocks. It doesn't require that jobs state their maximum needs in advance, but it does require that the jobs anticipate the order in hich they will request resources. From the perspective of a system designer, one of the difficulties of this scheme is discovering the best order for the resources so that the needs of the majority of the users are satisfied. Another difficulty is that of assigning a ranking to nonphysical resources such as files or locked database records where there is no basis for assigning a higher number to one over another.

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 presented in Figure 5.7 through Figure 5.11, there exists at least one allocation of resources sequence that will allow jobs to continue without becoming deadlocked. One such algorithm was proposed by Dijkstra in 1965 to regulate resource allocation to avoid deadlocks. The Banker's Algorithm is based on a bank with a fixed amount of capital that operates on the following principles: • No customer will be granted a loan exceeding the bank's total capital.

- All customers will be given a maximum credit limit when opening an account.
- No customer will be allowed to borrow over the limit.
- The sum of all loans won't exceed the bank's total capital.

Under these conditions, the bank isn't required to have on hand the total of all maximum lending quotas before it can open up for business (we'll assume the bank will always have the same fixed total and we'll disregard interest charged on loans). For our example, the bank has a total capital fund of $10,000 and has three customers, C1, C2, and C3, who have maximum credit limits of $4,000, $5,000, and $8,000, respectively. Table 5.4 illustrates the state of affairs of the bank after some loans have been granted to C2 and C3. This is called a safe state because the bank still has enough money left to satisfy the maximum requests of C1, C2, or C3.

Table 5.4 : The bank started with $10,000 and has remaining capital of $4,000 after these loans. Therefore, it's in a "safe state."

STET WOMEN'S COLLEGE

Customer	Loan Amount	Maximum Credit	Remaining Credit
C1	0	4,000	4,000
C2	2,000	5,000	3,000
C3	4,000	8,000 4,000	

Total loaned: $6,000

otal capital fund: $10,000

A few weeks later after more loans have been made, and some have been repaid, the bank is in the unsafe state represented in Table 5.5.

Customer	Loan Amount	Maximum Credit	Remaining Credit
C1	0	4,000	4,000
C2	2,000	5,000	3,000
C3	4,000	8,000 4,000	4,000

Total loaned: $6,000

Total capital fund: $10,000

This is an unsafe state because with only $1,000 left, the bank can't satisfy anyone's maximum request; and if the bank lent the $1,000 to anyone, then it would be deadlocked (it can't make a loan).

An unsafe state doesn't necessarily lead to deadlock, but it does indicate that the system is an excellent candidate for one. After all, none of the customers is required to request the maximum, but the bank doesn't know the exact amount that will eventually be requested; and as long as the bank's capital is less than the maximum amount available for individual loans, it can't guarantee that it will be able to fill every loan request.

If we substitute jobs for customers and dedicated devices for dollars, we can apply the same banking principles to an operating system. In this example the system has 10 devices. Table 5.6 shows our system in a safe state and Table 5.7 depicts the same system in an unsafe state. As before, a safe state is one in which at least one job can finish because there are enough available resources to satisfy its maximum needs. Then, using the resources released by the finished job, the maximum needs of another job can be filled and that job can be finished, and so on until all jobs are done.

Table 5.6 Resource assignments after initial allocations. A safe state: Six devices are allocated and four units are still available.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	0	4	4
2	2	5	3
3	4	8	4

Total number of devices allocated: 6

Total number of devices in system: 10

Table 5.7 Resource assignments after later allocations. An unsafe state: Only one unit is available but every job requires at least two to complete its execution

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	2	4	2
2	3	5	2
3	4	8	4

Total number of devices allocated: 6

Total number of devices in system: 10

STET WOMEN'S COLLEGE

The Banker’s Algorithm has been used to avoid deadlocks in systems with a few resources, it isn’t always practical for most systems for several reasons:

• As they enter the system, jobs must predict the maximum number of resources needed. As we’ve said before, this isn’t practical in interactive systems.

• The number of total resources for each class must remain constant. If a device breaks and becomes suddenly unavailable, the algorithm won’t work (the system may already be in an unsafe state).

• The number of jobs must remain fixed, something that isn’t possible in interactive systems where the number of active jobs is constantly changing.

• The overhead cost incurred by running the avoidance algorithm can be quite high when there are many active jobs and many devices because it has to be invoked for every request.

• Resources aren’t well utilized because the algorithm assumes the worst case and, as a result, keeps vital resources unavailable to guard against unsafe states.

• Scheduling suffers as a result of the poor utilization and jobs are kept waiting for resource allocation. A steady stream of jobs asking for a few resources can cause the indefinite postponement of a more complex job requiring many resources.

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, only when the operator notices that throughput has deteriorated, or when an angry user complains.

The detection algorithm can be explained by using directed resource graphs and “reducing” them. Begin with a system that is in use, as shown in Figure 5.12(a). The steps to reduce a graph are these:

1. Find a process that is currently using a resource and not waiting for one. This process can be removed from the graph (by disconnecting the link tying the resource to the process, such as P3 in Figure 5.12(b)), and the resource can be returned to the “available list.” This is possible because the process would eventually finish and return the resource.
2. Find a process that’s waiting only for resource classes that aren’t fully allocated (such as P2 in Figure 5.12(c)). This process isn’t contributing to deadlock since it would eventually get the resource it’s waiting for, finish its work, and return the resource to the “available list” as shown in Figure 5.12(c).
3. Go back to step 1 and continue with steps 1 and 2 until all lines connecting resources to processes have been removed, eventually reaching the stage shown in Figure 5.12(d).

If there are any lines left, this indicates that the request of the process in question can’t be satisfied and that a deadlock exists. Figure 5.12 illustrates a system in which three processes—P1, P2, and P3—and three resources—R1, R2, and R3—aren’t deadlocked

Figure 5.12 shows the stages of a graph reduction from (a), the original state. In (b), the link between P3 and R3 can be removed because P3 isn’t waiting for any other resources to finish, so R3 is released and allocated to P2 (step 1). In (c), the links between P2 and R3 and between P2 and R2 can be removed because P2 has all of its requested resources and can run to completion—and then R2 can be allocated to P1. Finally, in (d), the links between P1 and R2 and between P1 and R1 can be removed because P1 has all of its requested resources and can finish successfully. Therefore, the graph is completely resolved. However, Figure 5.13 shows a very similar situation that is deadlocked because of a key difference: P2 is linked to R1.

The deadlocked system in Figure 5.13 can’t be reduced. In (a), the link between P3 and R3 can be removed because P3 isn’t waiting for any other resource, so R3 is released and allocated to P2. But in (b), P2 has only two of the three resources it needs to finish and it is waiting for R1. But R1 can’t be released by P1 because P1 is waiting for R2, which is held by P2; moreover, P1 can’t finish because it is waiting for P2 to finish (and release R2), and P2 can’t finish because it’s waiting for R1. This is a circular wait.

STET WOMEN'S COLLEGE

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.

The first and simplest recovery method, and the most drastic, is to terminate every job that's active in the system and restart them from the beginning.

The second method is to terminate only the jobs involved in the deadlock and ask their users to resubmit them. The third method is to identify which jobs are involved in the deadlock and terminate them one at a time, checking to see if the deadlock is eliminated after each removal, until the deadlock has been resolved. Once the system is freed, the remaining jobs are allowed to complete their processing and later the halted jobs are started again from the beginning.

The fourth method can be put into effect only if the job keeps a record, a snapshot, of its progress so it can be interrupted and then continued without starting again from the beginning of its execution. The snapshot is like the landing in our staircase example: Instead of forcing the deadlocked stair climbers to return to the bottom of the stairs, they need to retreat only to the nearest landing and wait until the others have passed. Then the climb can be resumed. In general, this method is favored for long-running jobs to help them make a speedy recovery.

Several factors must be considered to select the victim that will have the least-negative effect on the system. The most common are:

- The priority of the job under consideration—high-priority jobs are usually untouched
- CPU time used by the job—jobs close to completion are usually left alone
- The number of other jobs that would be affected if this job were selected as the victim

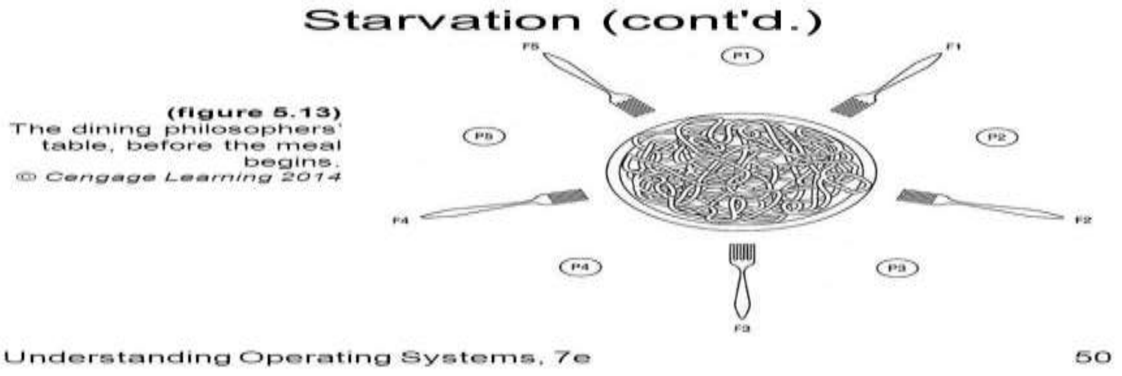
Starvation

Starvation occurs when a low priority program is requesting for a system resource, but are not able to execute because a higher priority program is utilizing that resource for an extended period. A scheduler is needed to help juggle all the processes trying to use the resources from the CPU.

Five philosophers are sitting at a round table, each deep in thought, and in the center lies a bowl of spaghetti that is accessible to everyone. There are forks on the table—one between each philosopher, as illustrated in Figure 5.14. Local custom dictates that each philosopher must use two forks, the forks on either side of the plate, to eat the spaghetti, but there are only five forks—not the 10 it would require for all five thinkers to eat at once—and that's unfortunate for Philosopher 2.

When they sit down to dinner, Philosopher 1 (P1) is the first to take the two forks (F1 and F5) on either side of the plate and begins to eat. Inspired by his colleague,

Figure 5.14: The dining philosophers' table, before the meal begins.


Starvation (cont'd.)

(figure 5.13) The dining philosophers' table, before the meal begins. © Cengage Learning 2014

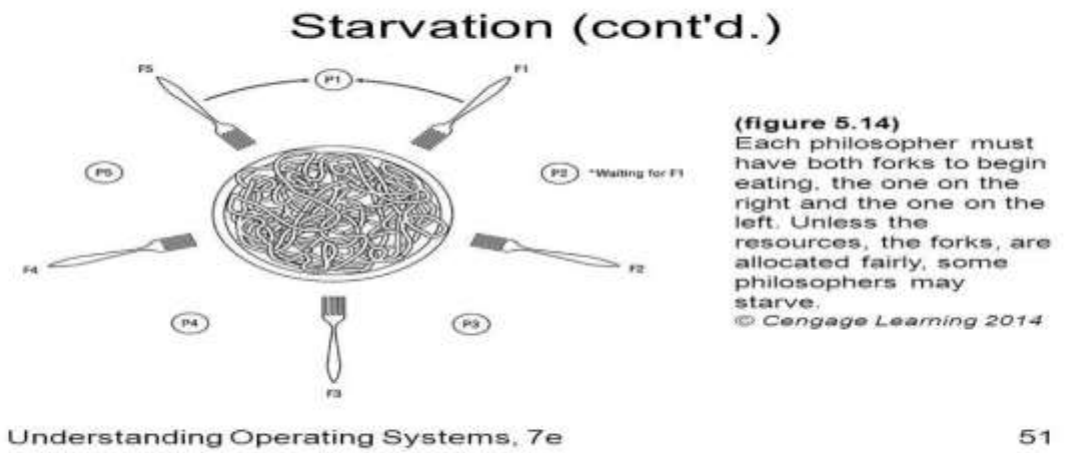
Understanding Operating Systems, 7e

50

Philosopher 3 (P3) does likewise, using F2 and F3. Now Philosopher 2 (P2) decides to begin the meal but is unable to start because no forks are available: F1 has been allocated to P1, and F2 has been allocated to P3, and the only remaining fork can be used only by P4 or P5. So (P2) must wait.

4 and P5 are quietly thinking and P1 is still eating when P3 (who should be full) decides to eat some more; and because the resources are free, he is able to take F2 and F3 once again. Soon thereafter, P1 finishes and releases F1 and F5, but P2 is still not able to eat because F2 is now allocated.

Figure 5.15 : Each philosopher must have both forks to begin eating, the one on the right and the one on the left. Unless the resources, the forks, are allocated fairly, some philosophers may starve.



Starvation (cont'd.)

(figure 5.14)
Each philosopher must have both forks to begin eating, the one on the right and the one on the left. Unless the resources, the forks, are allocated fairly, some philosophers may starve.
© Cengage Learning 2014

Understanding Operating Systems, 7e 51

This scenario could continue forever; and as long as P1 and P3 alternate their use of the available resources, P2 must wait. P1 and P3 can eat any time they wish while P2 starves—only inches from nourishment.

In a computer environment, the resources are like forks and the competing processes are like dining philosophers. If the resource manager doesn't watch for starving processes and jobs, and plan for their eventual completion, they could remain in the system for ever waiting for the right combination of resources.

To address this problem, an algorithm designed to detect starving jobs can be implemented, which tracks how long each job has been waiting for resources (this is the same as aging, described in Chapter 4). Once starvation has been detected, the system can block new jobs until the starving jobs have been satisfied. This algorithm must be monitored closely: If monitoring is done too often, then new jobs will be blocked too frequently and throughput will be diminished. If it's not done often enough, then starving jobs will remain in the system for an unacceptably long period of time.

Concurrent Processes

Multiprogramming systems that use only one CPU, one processor, which is shared by several jobs or processes. This is called multiprogramming. Multiprocessing systems include single computers with multiple cores as well as linked computing systems with only one processor each to share processing among them.

What Is Parallel Processing?

Parallel processing, one form of multiprocessing, is a situation in which two or more processors operate in unison. That means two or more CPUs are executing instructions simultaneously. In multiprocessing systems, the Processor Manager has to coordinate the activity of each processor, as well as synchronize cooperative interaction among the CPUs. There are two primary benefits to parallel processing systems: increased reliability and faster processing.

The reliability stems from the availability of more than one CPU: If one processor fails, then the others can continue to operate and absorb the load. This isn't simple to do; the system must be carefully designed so that, first, the failing processor can inform other processors to take over and, second, the operating system can restructure its resource allocation strategies so the remaining processors don't become overloaded.

The increased processing speed is often achieved because sometimes instructions can be processed in parallel, two or more at a time, in one of several ways. Some systems allocate a CPU to each program or job. Others allocate

STET WOMEN'S COLLEGE

a CPU to each working set or parts of it. Still others subdivide individual instructions so that each subdivision can be processed simultaneously (which is called concurrent programming).

The complexities of the Processor Manager's task when dealing with multiple processors or multiple processes are easily illustrated with an example: You're late for an early afternoon appointment and you're in danger of missing lunch, so you get in line for the drive-through window of the local fast-food shop. When you place your order, the order clerk confirms your request, tells you how much it will cost, and asks you to drive to the pickup window where a cashier collects your money and hands over your order. All's well and once again you're on your way—driving and thriving. You just witnessed a well-synchronized multiprocessing system. Although you came in contact with just two processors—the order clerk and the cashier—there were at least two other processors behind the scenes who cooperated to make the system work—the cook and the bagger .

A fast-food lunch spot is similar to the six-step information retrieval system below. It is described in a different way in Table 6.1.

a) Processor 1 (the order clerk) accepts the query, checks for errors, and passes the request on to Processor 2 (the bagger).
b) Processor 2 (the bagger) searches the database for the required information (the hamburger).
c) Processor 3 (the cook) retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage.
d) Once the data is gathered (the hamburger is cooked), it's placed where Processor 2 can get it (in the hamburger bin).
e) Processor 2 (the bagger) passes it on to Processor 4 (the cashier). f) Processor 4 (the cashier) routes the response (your order) back to the originator of the request—you.

Table 6.1 The six steps of the fast-food lunch stop.

Originator	**Action**	**Receiver**
Processor 1 Accepts the query, checks for errors,	Accepts the query, checks for errors, and passes the request on to =>	Processor 2 (the bagger)
Processor 2(the bagger)	Searches the database for the required information (the hamburger)	
Processor 3	Retrieves the data from the database (the cook)	
Processor 3	Once the data is gathered (the hamburger is cooked), it's placed where the receiver => can get it (in the hamburger bin)	Processor 2(the bagger)
Processor 2(the bagger)	Passes it on to =>	Processor 4(the Cashier)
Processor 4(the Cashier)	Routes the response (your order) back to the originator of the request =>	You

Synchronization is the key to the system's success because many things can go wrong in a multiprocessing system.

Evolution of Multiprocessors

Multiprocessing can take place at several different levels, each of which requires a different frequency of synchronization, as shown in Table 6.2. Notice that at the job level, multiprocessing is fairly benign. It's as if each job is running on its own workstation with shared system resources. On the other hand, when multiprocessing takes

STET WOMEN'S COLLEGE

place at the thread level, a high degree of synchronization is required to disassemble each process, perform the thread's instructions, and then correctly reassemble the process.

Table 6.2 Levels of parallelism and the required synchronization among processors.

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor synchronization and all processes and threads are run by that same processor	No explicit required.
Process Level	Unrelated processes, regardless of job, are assigned to any available processor.	Moderate amount of synchronization required to track processes.
Thread Level	Threads are assigned to available processors.	High degree of synchronization required, often requiring explicit instructions from the programmer.

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. Each transistor switches between two positions—0 and 1—as the computer conducts its binary arithmetic at increasingly fast speeds. However, as transistors reached nano-sized dimensions and the space between transistors became ever closer, the quantum physics of electrons got in the way.

In a nutshell, here's the problem. When transistors are placed extremely close together, electrons have the ability to 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 the 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 ever 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 dualcore 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 predictions, as of this writing, that 80 or more cores will be placed on a single chip.

Does this hardware innovation affect the operating system? Yes, because it must manage multiple processors, multiple RAMs, 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

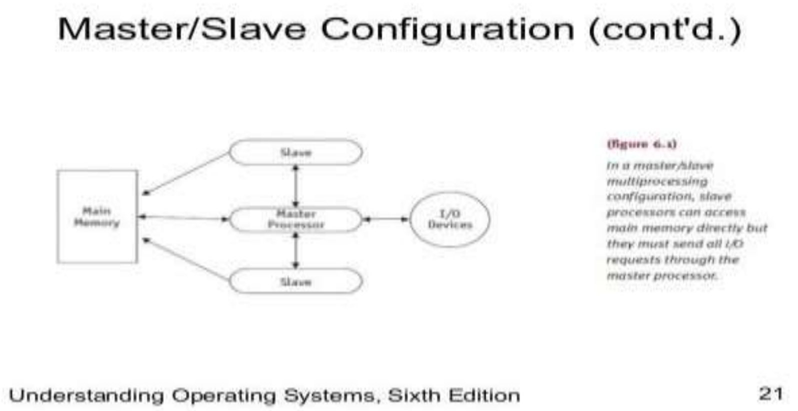
Master/Slave Configuration The master/slave configuration is an asymmetric multiprocessing system. Think of it as a single-processor system with additional slave processors, each of which is managed by the primary master processor as shown in Figure 6.1.

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 processors; in these cases, the

STET WOMEN'S COLLEGE

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.

Figure 6.1: In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all I/O requests through the master processor.



Master/Slave Configuration (cont'd.)

(figure 6.1)
In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all I/O requests through the master processor.

Understanding Operating Systems, Sixth Edition 21

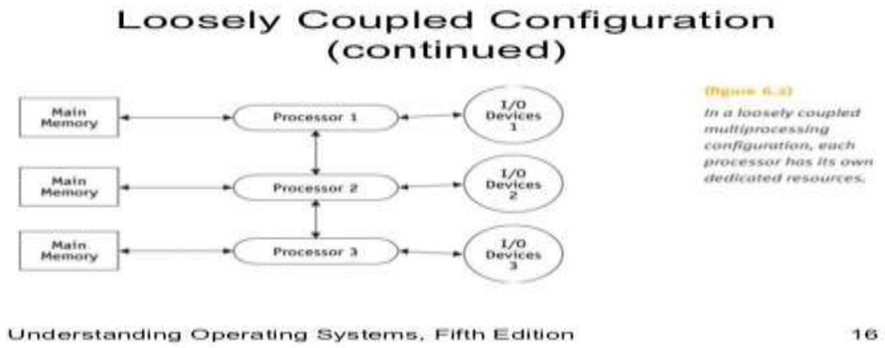
The primary advantage of this configuration is its simplicity. However, it has three serious disadvantages:

- Its reliability is no higher than for a single-processor system because if the master processor fails, 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 that 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 multiprocessing configuration, each processor has its own dedicated resources.



Loosely Coupled Configuration (continued)

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

Understanding Operating Systems, Fifth Edition 16

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 to which processor 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 the best combination of output devices available.

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

Symmetric Configuration

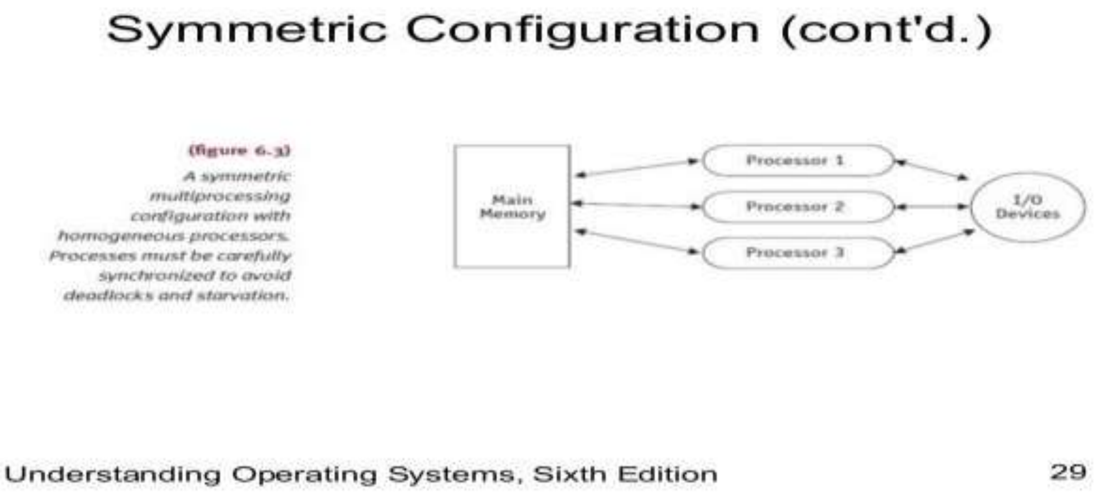
The symmetric configuration (also called tightly coupled) has four advantages over loosely coupled configuration:

- It's more reliable.
- It uses resources effectively.
- It can balance loads well.
- It can degrade gracefully in the event of a failure.

However, it is the most difficult configuration to implement because the processes must be well synchronized to avoid the problems of races and deadlocks.

In a symmetric configuration (as depicted in Figure 6.3), processor scheduling is decentralized. 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 every processor has access to it. Each processor uses the same scheduling algorithm to select which process it will run next.

Figure 6.3 : A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.



Symmetric Configuration (cont'd.)

(figure 6.3)

A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.

Understanding Operating Systems, Sixth Edition 29

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.

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 printers and other I/O devices, a location in storage, or a data file. In essence, the used resource must be locked away from other processes until it is released. Only when it is released is a waiting process allowed to use the resource. This is where synchronization is critical. A mistake could leave a job waiting indefinitely (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 numbers on the wall are changed by the clerks who pull a chain to increment them as they attend to each customer. But what happens when there is no synchronization 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 pulls the chain so the sign says “Now Serving #75.” But just then the clerk is called to the telephone and leaves the building, never to return (an interrupt). Meanwhile, Clerk 1 pulls the chain and proceeds to wait on #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—an unacceptable state of affairs.

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) processing system. It is called a critical region because it is a critical section and its execution must be handled as a unit.

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 performed 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—and each could 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 is a single, indivisible machine instruction known simply as TS and was introduced by IBM for its multiprocessing System 360/370 computers. In a single machine cycle it tests to see if the key is available and, if it is, sets it to unavailable.

The actual key is a single bit in a storage location that can contain a 0 (if it’s free) or a 1 (if busy). We can consider TS to be a function subprogram that has one parameter (the storage location) and returns one value (the condition code: busy/free), with the exception that it takes only one machine cycle.

Therefore, a process (Process 1) would test the condition code using the TS instruction before entering a critical region. If no other process was in this critical region, then Process 1 would be allowed to proceed and the condition code would be changed from 0 to 1. Later, when Process 1 exits the critical region, the condition code is reset to 0 so another process can enter. On the other hand, if Process 1 finds a busy condition code, then it’s placed in a waiting loop where it continues to test the condition code and waits until it’s free.

It’s a simple procedure to implement, and it works well for a small number of processes, test-and-set has two major drawbacks. First, when many processes are waiting to enter a critical region, starvation could occur because the processes gain access in an arbitrary fashion. Unless a first-come, first-served policy were set up, some processes could be favored over others. A second drawback is that the waiting processes remain in unproductive, resource- consuming wait loops, requiring context switching. This is known as busy waiting—which not only consumes valuable processor time but also relies on the competing processes to test the key, something that is best handled by the operating system or the hardware.

STET WOMEN'S COLLEGE

WAIT and SIGNAL

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

WAIT is activated when the process encounters a busy condition code. WAIT sets the process's process control block (PCB) to the blocked state and links it to the queue of processes waiting to enter this particular critical region. The Process Scheduler then selects another process for execution. SIGNAL is activated when a process exits the critical region and the condition code is set to "free." It checks the queue of processes waiting to enter this critical region and selects one, setting it to the READY state. Eventually the Process Scheduler will choose this process for running. The addition of the operations WAIT and SIGNAL frees the processes from the busy waiting dilemma and returns control to the operating system, which can then run other jobs while the waiting processes are idle (WAIT).

Semaphores

A semaphore is a non-negative integer variable that's used as a binary signal, a flag. One of the most well-known semaphores was the signaling device, shown in Figure 6.4, used by railroads 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, up or down (on or off).

In an operating system, a semaphore performs a similar function: It signals if and when a resource is free and can be used by a process. 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 and increment.

Here's how they work. If we let s be a semaphore variable, then the V operation on s is simply to increment s by 1. The action can be stated as:

V(s): s: = s + 1

This in turn necessitates a fetch, increment, and store sequence. Like the test-and-set operation, the increment operation must be performed as a single indivisible action to avoid deadlocks. And that means that s cannot be accessed by any other process during the operation.

The operation P on s is to test the value of s and, if it's not 0, to decrement it by 1. The action can be stated as:

P(s): If s > 0 then s: = s – 1

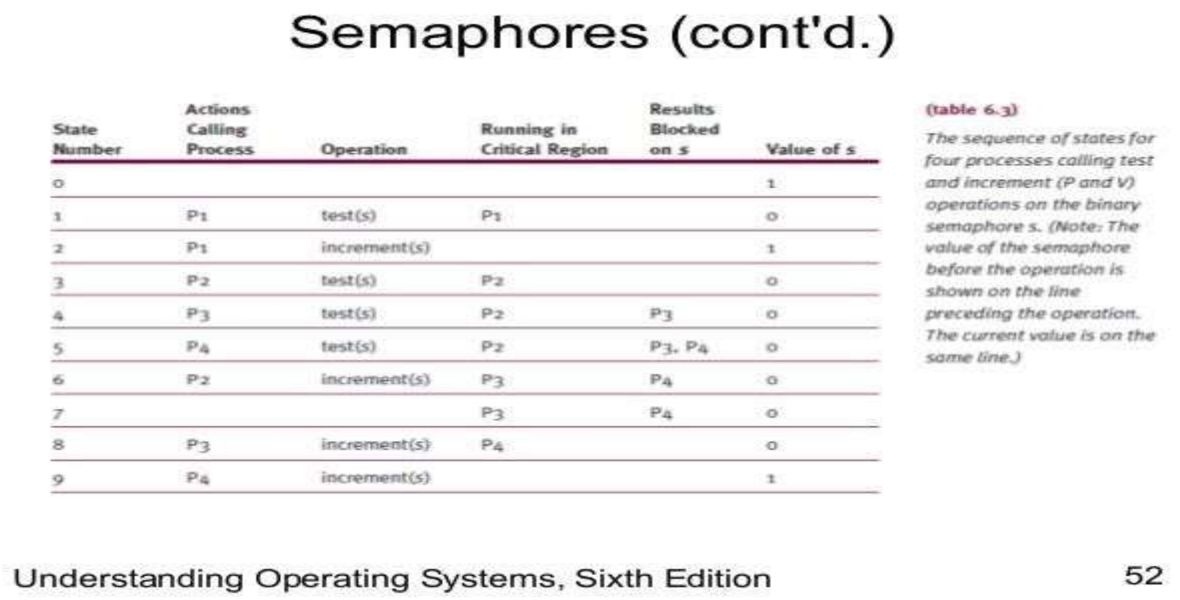
This involves a test, fetch, decrement, and store sequence. Again this sequence must be performed as an indivisible action in a single machine cycle or be arranged so that the process cannot take action until the operation (test or increment) is finished.

The operations to test or increment are executed by the operating system in response to calls issued by any one process naming a semaphore as parameter (this alleviates the process from having control). If s = 0, it means that the critical region is busy and the process calling on the test operation must wait until the operation can be executed and that's not until s > 0.

As shown in Table 6.3, P3 is placed in the WAIT state (for the semaphore) on State 4. As also shown in Table 6.3, for States 6 and 8, when a process exits the critical region, the value of s is reset to 1 indicating that the critical region is free. This, in turn, triggers the awakening of one of the blocked processes, its entry into the critical region, and the resetting of s to 0. In State 7, P1 and P2 are not trying to do processing in that critical region and P4 is still blocked.

Table 6.3 :The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

STET WOMEN'S COLLEGE



Semaphores (cont'd.)

State Number	Actions Calling Process	Operation	Running in Critical Region	Results Blocked on s	Value of s
0					1
1	P1	test(s)	P1		0
2	P1	increment(s)			1
3	P2	test(s)	P2		0
4	P3	test(s)	P2	P3	0
5	P4	test(s)	P2	P3, P4	0
6	P2	increment(s)	P3	P4	0
7			P3	P4	0
8	P3	increment(s)	P4		0
9	P4	increment(s)			1

(table 6.3)

The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

Understanding Operating Systems, Sixth Edition 52

After State 5 of Table 6.3, the longest waiting process, P3, was the one selected to enter the critical region, but that isn't necessarily the case unless the system is using a first-in, first-out selection policy. In fact, the choice of which job will be processed next depends on the algorithm used by this portion of the Process Scheduler.

As you can see from Table 6.3, test and increment operations on semaphore s enforce the concept of mutual exclusion, which is necessary to avoid having two operations attempt to execute at the same time. The name traditionally given to this semaphore in the literature is mutex and it stands for MUTual EXclusion. So the operations become:

test(mutex): if mutex > 0 then mutex: = mutex – 1
increment(mutex): mutex: = mutex + 1

In sequential computations mutual exclusion is achieved automatically because each operation is handled in order, one at a time. However, in parallel computations the order of execution can change, so mutual exclusion must be explicitly stated and maintained. In fact, the entire premise of parallel processes hinges on the requirement that all operations on common variables consistently exclude one another over time.

Process Cooperation

There are occasions when several processes work directly together to complete a common task. Two famous examples are the problems of producers and consumers, and of readers and writers. Each case requires both mutual exclusion and synchronization, and each is implemented by using semaphores.

Producers and Consumers

The classic problem of producers and consumers is one in which one process produces some data that another process consumes later. Although we'll describe the case with one producer and one consumer, it can be expanded to several pairs of producers and consumers.

Let's return for a moment to the fast-food framework at the beginning of this chapter because the synchronization between two of the processors (the cook and the bagger) represents a significant problem in operating systems. The cook produces hamburgers that are sent to the bagger (consumed). Both processors have access to one common area, the hamburger bin, which can hold only a finite number of hamburgers (this is called a buffer area). The bin is a necessary storage area because the speed at which hamburgers are produced is independent from the speed at which they are consumed.

Problems arise at two extremes: when the producer attempts to add to an already full bin (as when the cook tries to put one more hamburger into a full bin) and when the consumer attempts to draw from an empty bin (as when the bagger tries to take a hamburger that hasn't been made yet). In real life, the people watch the bin and if it's

STET WOMENS COLLEGE

empty or too full the problem is recognized and quickly resolved. However, in a computer system such resolution is not so easy.

onsider the case of the prolific CPU. The CPU can generate output data much faster than a printer can print it. Therefore, since this involves a producer and a consumer of two different speeds, we need a buffer where the producer can temporarily store data that can be retrieved by the consumer at a more appropriate speed. Figure 6.5 shows three typical buffer states.

Because the buffer can hold only a finite amount of data, the synchronization process must delay the producer from generating more data when the buffer is full. It must also be prepared to delay the consumer from retrieving data when the buffer is empty. This task can be implemented by two counting semaphores—one to indicate the number of full positions in the buffer and the other to indicate the number of empty positions in the buffer.
A third semaphore, mutex, will ensure mutual exclusion between processes.

Table 6.4 Definitions of the Producers and Consumers processes.

Producer	**Consumer**
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	consume data

Table 6.5 Definitions of the elements in the Producers and Consumers Algorithm.

Variables, Functions	**Definitions**
full defined as a	semaphore
empty	defined as a semaphore
mutex	defined as a semaphore
n	the maximum number of positions in the buffer
V (x)	x: = x + 1 (x is any variable defined as a semaphore)
P (x)	if x > 0 then x: = x – 1
mutex = 1	means the process is allowed to enter the critical region
COBEGIN	the delimiter that indicates the beginning of concurrent processing
COEND	the delimiter that indicates the end of concurrent Processing

Given the definitions in Table 6.4 and Table 6.5, the Producers and Consumers Algorithm shown below synchronizes the interaction between the producer and consumer.

Producers and Consumers Algorithm

```
empty:          = n
full:           = 0
mutex:          = 1
COBEGIN
    repeat until no more data PRODUCER
    repeat until buffer is empty CONSUMER
COEND
```

STET WOMEN'S COLLEGE

The processes (PRODUCER and CONSUMER) then execute as described. Try the code with n = 3 or try an alternate order of execution to see how it actually works.

The concept of producers and consumers can be extended to buffers that hold records or other data, as well as to other situations in which direct process-to-process communication of messages is required.

Readers and Writers

The problem of readers and writers was first formulated by Courtois, Heymans, and Parnas (1971) and arises when two types of processes need to access a shared resource such as a file or database. They called these processes readers and writers.

An airline reservation system is a good example. The readers are those who want flight information. They're called readers because they only read the existing data; they don't modify it. And because no one is changing the database, the system can allow many readers to be active at the same time—there's no need to enforce mutual exclusion among them.

The writers are those who are making reservations on a particular flight. Writers must be carefully accommodated because they are modifying existing data in the database. The system can't allow someone to be writing while someone else is reading (or writing). Therefore, it must enforce mutual exclusion if there are groups of readers and a writer, or if there are several writers, in the system. Of course the system must be fair when it enforces its policy to avoid indefinite postponement of readers or writers.

To prevent either type of starvation, Hoare (1974) proposed the following combination priority policy. When a writer is finished, any and all readers who are waiting, or on hold, are allowed to read. Then, when that group of readers is finished, the writer who is on hold can begin, and any new readers who arrive in the meantime aren't allowed to start until the writer is finished.

The state of the system can be summarized by four counters initialized to 0:

- Number of readers who have requested a resource and haven't yet released it (R1 = 0)
- Number of readers who are using a resource and haven't yet released it (R2 = 0)
- Number of writers who have requested a resource and haven't yet released it (W1 = 0)
- Number of writers who are using a resource and haven't yet released it (W2 = 0)

This can be implemented using two semaphores to ensure mutual exclusion between readers and writers. A resource can be given to all readers, provided that no writers are processing (W2 = 0). A resource can be given to a writer, provided that no readers are reading (R2 = 0) and no writers are writing (W2 = 0).

Readers must always call two procedures: the first checks whether the resources can be immediately granted for reading; and then, when the resource is released, the second checks to see if there are any writers waiting. The same holds true for writers. The first procedure must determine if the resource can be immediately granted for writing, and then, upon releasing the resource, the second procedure will find out if any readers are waiting.

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 multiprocessors. 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.

Applications of Concurrent Programming

Most programming languages are serial in nature—instructions are executed one at a time. Therefore, to resolve an arithmetic expression, every operation is done in sequence following the order prescribed by the programmer and compiler. Table 6.6 shows the steps to compute the following expression:

Table 6.6 : The sequential computation of the expression requires several steps. (In this example, there are seven steps, but each step may involve more than one machine operation.)

Step No.	Operation	Result
1	(F – G)	Store difference in T1
2	(D + E)	Store sum in T2
3	(T2) ** (T1)	Store power in T1

SIET WOMEN'S COLLEGE

4	4 / (T1)	Store quotient in T2
5	3 * B	Store product in T1
6	(T1) * C	Store product in T1
7	(T1) + (T2)	Store sum in A

All equations follow a standard order of operations, which states that to solve an equation you first perform all calculations in parentheses. Second, you calculate all exponents. Third, you perform all multiplication and division. Fourth, you perform the addition and subtraction. For each step you go from left to right. If you were to perform the calculations in some other order, you would run the risk of finding the incorrect answer.

For many computational purposes, serial processing is sufficient; it's easy to implement and fast enough for most users.

However, arithmetic expressions can be processed differently if we use a language that allows for concurrent processing. Let's revisit two terms—COBEGIN and COEND— that will indicate to the compiler which instructions can be processed concurrently. Then we'll rewrite our expression to take advantage of a concurrent processing compiler.

```
COBEGIN
    T1 = 3 * B
    T2 = D + E
    T3 = F – G
COEND
 COBEGIN
     T4 = T1 * C
     T5 = T2 ** T3
COEND
        A = T4 + 4 / T5
```

As shown in Table 6.7, to solve A = 3 * B * C + 4 / (D + E) ** (F – G), the first three operations can be done at the same time if our computer system has at least three processors. The next two operations are done at the same time, and the last expression is performed serially with the results of the first two steps.

Table 6.7 : With concurrent processing, the sevenstep procedure can be processed in only four steps, which reduces execution time.

Step No.	**Processor**	**Operation**	**Result**
1	1	3 * B	Store product in T1
	2	(D + E)	Store sum in T2
	3	(F – G)	Store difference in T3
2	1	(T1) * C	Store product in T4
	2	(T2) ** (T3)	Store power in T5
3	1	4 / (T5)	Store quotient in T1
4	1	(T4) + (T1)	Store sum in A

With this system we've increased the computation speed, but we've also increased the complexity of the programming language and the hardware (both machinery and communication among machines).

In fact, we've also placed a large burden on the programmer—to explicitly state which instructions can be executed in parallel. This is explicit parallelism.

The automatic detection by the compiler of instructions that can be performed in parallel is called implicit parallelism.

STET WOMEN'S COLLEGE

With a true concurrent processing system, the example presented in Table 6.6 and Table 6.7 is coded as a single expression. It is the compiler that translates the algebraic expression into separate instructions and decides which steps can be performed in parallel and which in serial mode.

For example, the equation Y = A + B * C + D could be rearranged by the compiler as A + D + B * C so that two operations A + D and B * C would be done in parallel, leaving the final addition to be calculated last. As shown in the four cases that follow, concurrent processing can also dramatically reduce the complexity of working with array operations within loops, of performing matrix multiplication, of conducting parallel searches in databases, and of sorting or merging files. Some of these systems use parallel processors that execute the same type of tasks.

Case 1: Array Operations

To perform an array operation within a loop in three steps, the instruction might say:

```
  for(j = 1; j <= 3; j++)
  a(j) = b(j) + c(j);
```

If we use three processors, the instruction can be performed in a single step like this:

Processor #1:	Processor #2:	Processor #3:
a(1) = b(1) + c(1)	a(2) = b(2) + c(2)	a(3) = b(3) + c(3)

Case 2: Matrix Multiplication

Matrix multiplication requires many multiplication and addition operations that can take place concurrently, such as this equation: Matrix C = Matrix 1 * Matrix 2.

Matrix C = Matrix 1 * Matrix 2

$$\begin{matrix} Z & y & x & & A & B & C & * & K & L \\ W & v & u & = & D & E & F & & M & N \\ T & s & r & & & & & & O & P \end{matrix}$$

To find z in Matrix C, you multiply the elements in the first column of Matrix 1 by the corresponding elements in the first row of Matrix 2 and then add the products. Therefore, one calculation is this: z = (A * K) + (D * L). Likewise, x = (C * K) + (F * L) and r = (C * O) + (F * P).

Using one processor, the answer to this equation can be computed in 27 steps. By multiplying several elements of the first row of Matrix 1 by corresponding elements in Matrix 2, three processors could cooperate to resolve this equation in fewer steps. The actual number of products that could be computed at the same time would depend on the number of processors available. With two processors it takes only 18 steps to perform the calculations in parallel. With three, it would require even fewer. Notice that concurrent processing does not necessarily cut processing activity in direct proportion to the number of processors available. In this example, by doubling the number of processors from one to two, the number of calculations was reduced by one-third—not by one-half.

Case 3: Searching Databases

Database searching is a common non-mathematical application for concurrent processing systems. For example, if a word is sought from a dictionary database or a part number from an inventory listing, the entire file can be split into discrete sections with one processor allocated to each section. This results in a significant reduction in search time. Once the item is found, all processors can be deallocated and set to work on the next task. Even if the item sought is in the last record of the database, a concurrent search is faster than if a single processor was allocated to search the database.

Case 4: Sorting or Merging Files

By dividing a large file into sections, each with its own processor, every section can be sorted at the same time. Then pairs of sections can be merged together until the entire file is whole again—and sorted.

Threads and Concurrent Programming

The cooperation and synchronization of traditional processes, also known as heavyweight processes, which have the following characteristics:

STET WOMEN'S COLLEGE

- 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.

To minimize this overhead time, most current operating systems support the implementation of threads, or lightweight processes, which have become part of numerous application packages. Threads are supported at both the kernel and user level and can be managed by either the operating system or the application that created them.

Threads share the same resources as the process that created them which now becomes a more passive element because the thread is the unit that uses the CPU and is scheduled for execution. Processes might have from one to several active threads, which can be created, scheduled and synchronized more efficiently because the amount of information needed is reduced. When running a process with multiple threads in a computer system with a single CPU, the processor switches very quickly from one thread to another, giving the impression that the threads are executing in parallel. However, it is only in systems with multiple CPUs that the multiple threads in a process are actually executed in parallel.

Each active thread in a process has its own processor registers, program counter, stack and status, but shares the data area and the resources allocated to its process. Each thread has its own program counter and stack, which is used to store variables dynamically created by a process. For example, function calls in C might create variables that are local to the function. These variables are stored in the stack when the function is invoked and are destroyed when the function is exited. Since threads within a process share the same space and resources they can communicate more efficiently, increasing processing performance.

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 6.6. 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 Thread Scheduler, whose function is similar to that of the Process Scheduler, assigns it a processor.

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. This is similar to the WAIT and SIGNAL process synchronization algorithm.

Figure 6.6: A typical thread changes states from READY to FINISHED as it moves through the system.

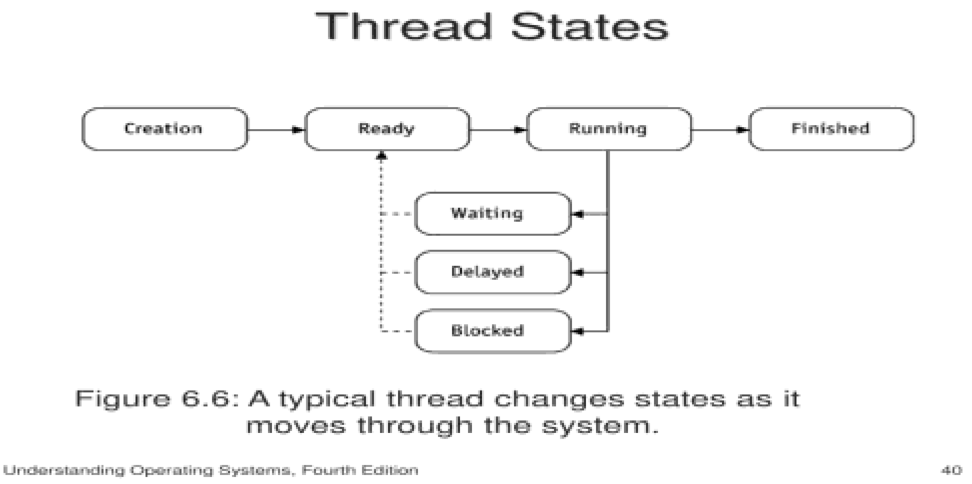


Figure 6.6: A typical thread changes states as it moves through the system.

Understanding Operating Systems, Fourth Edition 40

When an application such as a word processor 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, the thread that periodically backs up a current document to disk will be delayed for a period of time after it has completed the backup. After the time has expired, it performs the next backup 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 backup, wasting processor time and reducing system performance. Setting up a delay state avoids the problems of the test-and-set process synchronization algorithm.

A thread transitions from RUNNING to BLOCKED when an application issues an I/O request. After the I/O is completed, the thread returns to the READY state. When a thread transitions from RUNNING to FINISHED, all its resources are released and it can exit the system.

As you can see, the same operations are performed on both traditional processes and threads. Therefore, 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 on a WAIT state 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

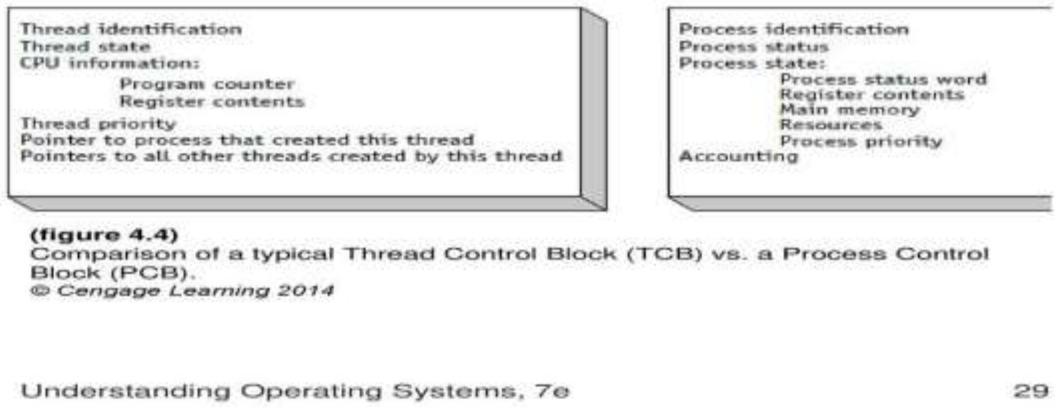
To do so, the operating system needs to track the critical information for each thread.

Thread Control Block

Just as processes are represented by Process Control Blocks (PCBs), so threads are represented by Thread Control Blocks (TCBs), which contain basic information about a thread such as its ID, state, and a pointer to the process that created it. Figure 6.7 shows the contents of a typical TCB:

- A thread ID, a unique identifier assigned by the operating system when the thread is created
- The thread state, which changes as the thread progresses through its execution; state changes, as well as all other entries in the TCB, apply individually to each thread
- CPU information, which contains everything that the operating system needs to know about how far the thread has executed, which instruction is currently being performed, and what data is being used
- Thread priority, used to indicate the weight of this thread relative to other threads and used by the Thread Scheduler when determining which thread should be selected from the READY queue
- A pointer to the process that created the thread
- Pointers to other threads created by this thread

Figure 6.7: Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB)



Thread identification
Thread state
CPU information:
Program counter
Register contents
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread

Process identification
Process status
Process state:
Process status word
Register contents
Main memory
Resources
Process priority
Accounting

(figure 4.4)
Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB).
© Cengage Learning 2014

Understanding Operating Systems, 7e 29

Concurrent Programming Languages

Early programming languages did not support the creation of threads or the existence of concurrent processes. Typically, they gave programmers the possibility of creating a single process or thread of control. The Ada programming language, developed in the late 1970s, was one of the first languages to provide specific concurrency commands. Java, developed by Sun Microsystems, Inc., was designed as a universal software platform for Internet applications and has been widely adopted.

Java

Java was released in 1995 as the first software platform that allowed programmers 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.

Java uses both a compiler and an interpreter. The source code of a Java program is first compiled into an intermediate language called Java byte codes, which are platform-independent. This means that one can compile a Java program on any computer that has a Java compiler, and the bytecodes can then be run on any computer that has a Java interpreter. The interpreter is designed to fit in with the hardware and operating system specifications of the computer that will run the Java byte codes. Its function is to parse and run each bytecode instruction on that computer. This combination of compiler and interpreter makes it easy to distribute Java applications because they don't have to be rewritten to accommodate the characteristics of every computer system. Once the program has been compiled it can be ported to, and run on, any system with a Java interpreter.

The Java Platform

Typically a computer platform contains the hardware and software where a program runs. The Java platform is a software-only platform that runs on top of other hardware-based platforms. It has two components: the Java Virtual Machine (Java VM), and the Java Application Programming Interface (Java API). Java VM is the foundation for the Java platform and contains the Java interpreter, which runs the byte codes provided by the compiler. Java VM sits on top of many different hardware-based platforms, as shown in Figure 6.8.

The Java API is a collection of software modules that programmers can use in their applications. The Java API is grouped into libraries of related classes and interfaces. These libraries, also known as packages, provide well-tested objects ranging from basic data types to I/O functions, from network interfaces to graphical user interface kits.

The Java Language Environment

The Java Language Environment Java was designed to make it easy for experienced programmers to learn. Its syntax is familiar because it looks and feels like C++. It is object-oriented, which means it takes advantage of modern software development methodologies and fits well into distributed client-server applications.

One of Java's features is that memory allocation is done at run time, unlike C and C++ where memory allocation is done at compilation time. Java's compiled code references memory via symbolic "handles" that are translated into real memory addresses at run time by the Java interpreter. This means that the memory allocation and referencing models are not visible to programmers, but are controlled entirely by the underlying run-time platform.

When a programmer declares some methods within a class to be synchronized, they are not run concurrently. These synchronized methods are under the control of monitors that ensure that variables remain in a consistent state. When a synchronized method begins to run it is given a monitor for the current object, which does not allow any other synchronized method in that object to execute. The monitor is released when a synchronized method exits, which allows other synchronized methods within the same object to run.

Java technology continues to be popular with programmers for several reasons:

- It offers the capability of running a single program on various platforms without having to make any changes.
- It offers a robust set of features such as run-time memory allocation, security, and multithreading.
- It is used for many Web and Internet applications, and integrates well with browsers that can run Java applets with audio, video, and animation directly in a Web page.

STET WOMEN'S COLLEGE

Unit IV

Device Management : Hardware devices typically provide the ability to **input** data into the computer or **output** data from the computer. To simplify the ability to support a variety of hardware devices, standardized application programming interfaces (API) are used.

Despite the multitude of devices that appear (and disappear) in the marketplace and the swift rate of change in device technology, the Device Manager must manage every peripheral device of the system. To do so, it must maintain a delicate balance of supply and demand—balancing the system's finite supply of devices with users' almostinfinite demand for them.

Device management involves four basic functions:

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

Types of Devices

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

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 tape drives, printers, and plotters, demand this kind of allocation scheme, because it would be awkward to let several users share them. A shared plotter might produce half of one user's graph and half of another. The disadvantage of dedicated devices is that they must be allocated to a single user for the duration of a job's execution, which can be quite inefficient, especially when the device isn't used 100 percent of the time. And some devices can be shared or virtual.

Shared devices can be assigned to several processes. For instance, a disk, or any other direct access storage device (often shortened to DASD), can be shared by several processes at the same time 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 each need to read from the same disk—must be resolved based on predetermined policies to decide which request will be 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) are converted into sharable devices through a spooling program that reroutes all print requests to 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.

For example, 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 (assisted by USB hubs) can accommodate up to 127 different devices, including flash memory, cameras, scanners, musical keyboards, etc. Each device is identified by the USB host controller with a unique identification number, which allows many devices to exchange data with the computer using the same USB connection. The USB controller assigns 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.

STET WOMEN'S COLLE

- Lowest priority is assigned to bulk transfers or exchanges that can accommodate slower data flow, such as printers or scanners.

Sequential Access Storage Media

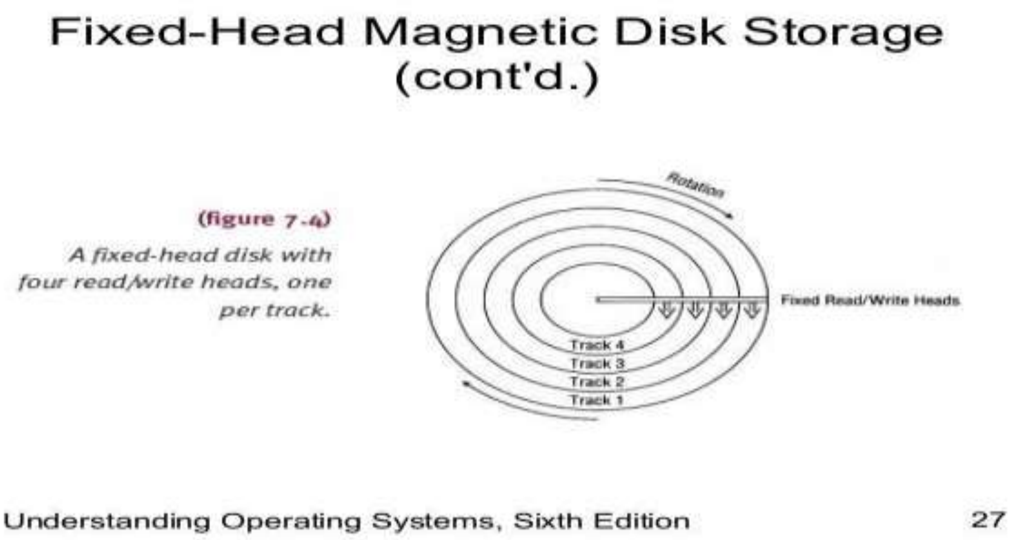
Direct access storage devices (DASDs) are devices that can directly read or write to a specific place. DASDs can be grouped into three categories: magnetic disks, optical discs, and flash memory. Although the variance in DASD access times isn't as wide as with magnetic tape, the location of the specific record still has a direct effect on the amount of time required to access it.

Fixed-Head Magnetic Disk Storage

A fixed-head magnetic disk looks like a large CD or DVD covered with magnetic film that has been formatted, usually on both sides, into concentric circles. Each circle is a track. Data is recorded serially on each track by the fixed read/write head positioned over it.

A fixed-head disk, shown in Figure 7.4, is also very fast—faster than the movable-head disks we'll talk about in the next section. However, its major disadvantages are its high cost and its reduced storage space compared to a movable-head disk (because the tracks must be positioned farther apart to accommodate the width of the read/write heads). These devices have been used when speed is of the utmost importance, such as space flight or aircraft applications.

Figure 7.4: A fixed-head disk with four read/write heads, one per track.



Fixed-Head Magnetic Disk Storage (cont'd.)

(figure 7.4)

A fixed-head disk with four read/write heads, one per track.

Understanding Operating Systems, Sixth Edition 27

Movable-Head Magnetic Disk Storage

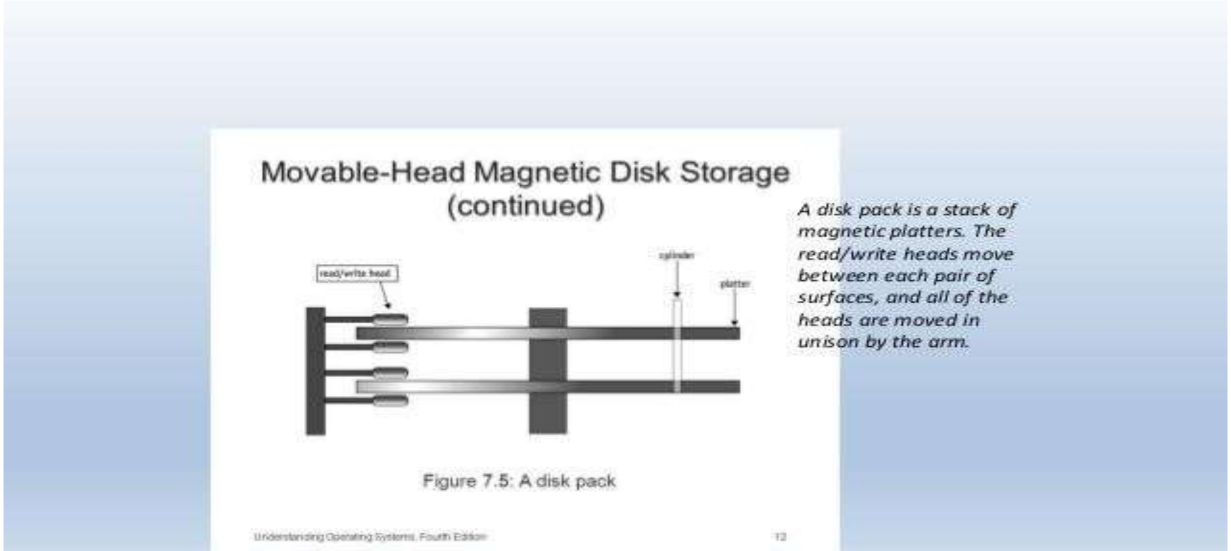
Movable-head magnetic disks, such as computer hard drives, have one read/write head that floats over each surface of each disk. Disks can be a single platter, or part of a disk pack, which is a stack of magnetic platters. Figure 7.5 shows a typical disk pack—several platters stacked on a common central spindle, separated by enough space to allow the read/write heads to move between each pair of disks.

As shown in Figure 7.5, each platter (except those at the top and bottom of the stack) has two surfaces for recording, and each surface is formatted with a specific number of concentric tracks where the data is recorded. The number of tracks varies from manufacturer to manufacturer, but typically there are a thousand or more on a highcapacity hard disk. Each track on each surface is numbered: Track 0 identifies the outermost concentric circle on each surface; the highest-numbered track is in the center .

The arm, shown in Figure 7.6, moves two read/write heads between each pair of surfaces: one for the surface above it and one for the surface below. 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—in other words, they're all positioned on the same track but on their respective surfaces creating a virtual cylinder.

This raises some interesting questions: Is it more efficient to write a series of records on surface one and, 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?

Figure 7.5 : A disk pack is a stack of magnetic platters. The read/write heads move between each pair of surfaces, and all of the heads are moved in unison by the arm.



Movable-Head Magnetic Disk Storage (continued)

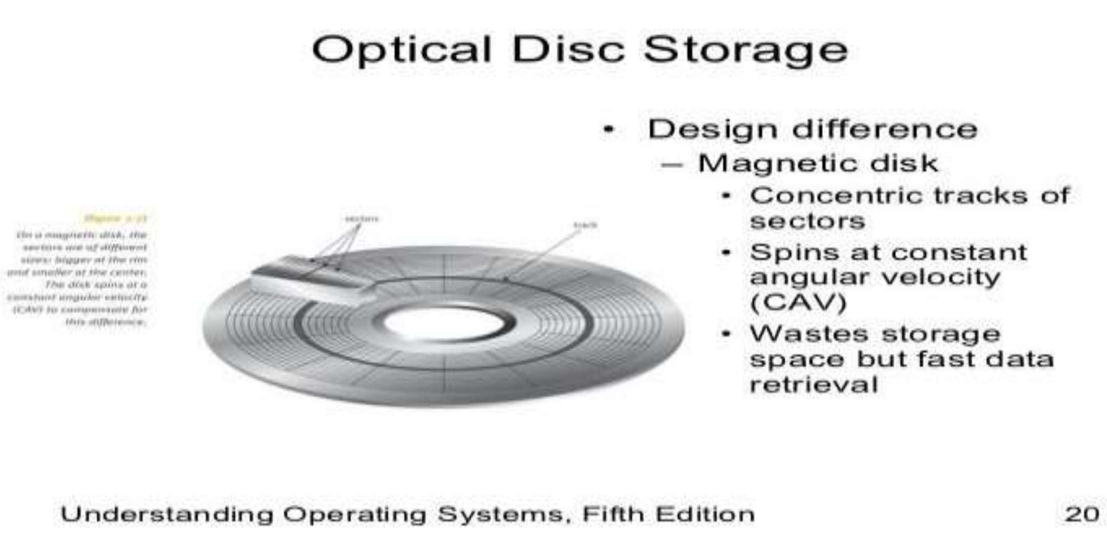
A disk pack is a stack of magnetic platters. The read/write heads move between each pair of surfaces, and all of the heads are moved in unison by the arm.

Figure 7.5: A disk pack

Understanding Operating Systems, Fourth Edition 12

To access any given record, the system needs three things: its cylinder number, so the arm can move the read/write heads to it; its surface number, so the proper read/write head is activated; and its sector number, as shown in Figure 7.7, so the read/write head knows the instant when it should begin reading or writing.

(figure 7.7)



Optical Disc Storage

- Design difference
  - Magnetic disk
    - Concentric tracks of sectors
    - Spins at constant angular velocity (CAV)
    - Wastes storage space but fast data retrieval

Figure 7.7

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

Understanding Operating Systems, Fifth Edition 20

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/track coincides with the term track or head used in some other texts.

Optical Disc Storage

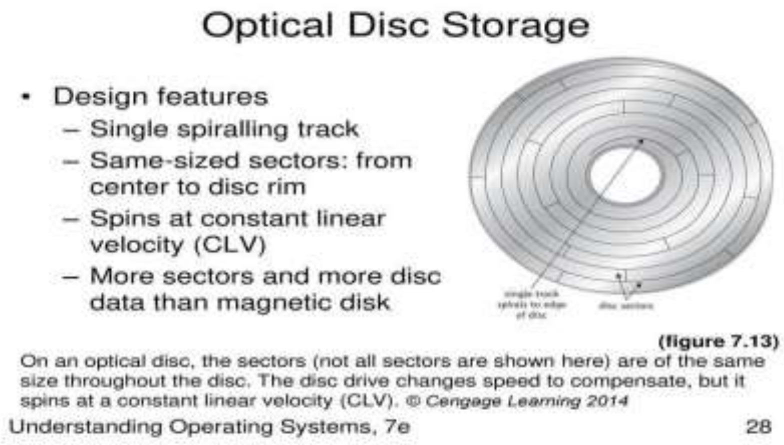
The advent of optical disc storage was made possible by developments in laser technology. Among the many differences between an optical disc and a magnetic disk is the design of the disc track and sectors.

A magnetic disk, which consists of concentric tracks of sectors, spins at a constant speed—this is called constant angular velocity (CAV). Because the sectors at the outside of the disk spin faster past the read/write head than the inner sectors, outside sectors are much larger than sectors located near the center of the disk. This format wastes storage space but maximizes the speed with which data can be retrieved.

On the other hand, an optical disc consists of a single spiraling track of same-sized sectors running from the center to the rim of the disc, as shown in Figure 7.8. This single track also has sectors, but all sectors are the same size regardless of their locations on the disc. This design allows many more sectors, and much more data, to fit on an optical disc compared to a magnetic disk of the same size. The disc drive adjusts the speed of the disc's spin to compensate for the sector's location on the disc—this is called constant linear velocity (CLV). Therefore, the disc spins faster to read sectors located at the center of the disc, and slower to read sectors near the outer edge. If you listen to a disc drive in action, you can hear it change speeds as it makes these adjustments.

Two of the most important measures of optical disc drive performance are sustained data transfer rate and average access time. The data transfer rate is measured in megabytes per second and refers to the speed at which massive amounts of data can be read from the disc. This factor is crucial for applications requiring sequential access, such as for audio and video playback. For example, a DVD with a fast data transfer rate will drop fewer frames when playing back a recorded video segment than will a unit with a slower transfer rate. This creates an image that's much smoother.

Figure 7.8



Optical Disc Storage

- Design features
  - Single spiralling track
  - Same-sized sectors: from center to disc rim
  - Spins at constant linear velocity (CLV)
  - More sectors and more disc data than magnetic disk

(figure 7.13)

On an optical disc, the sectors (not all sectors are shown here) are of the same size throughout the disc. The disc drive changes speed to compensate, but it spins at a constant linear velocity (CLV). © *Cengage Learning 2014*

Understanding Operating Systems, 7e 28

There are several types of optical-disc systems, depending on the medium and the capacity of the discs: CDs, DVDs, and Blu-ray as shown in Figure 7.9. To put data on an optical disc, a high-intensity laser beam burns indentations on the disc that are called pits. These pits, which represent 0s, contrast with the unburned flat areas, called lands, which represent 1s. The first sectors are located in the center of the disc and the laser moves outward reading each sector in turn. If a disc has multiple layers, the laser's course is reversed to read the second layer with the arm moving from the outer edge to the inner.

Figure 7.9

CD and DVD Technology

In the CD or DVD player, data is read back by focusing a low-powered red laser on it, which shines through the protective layer of the disc onto the CD track (or DVD tracks) where data is recorded. Light striking a land is reflected into a photo detector while light striking a pit is scattered and absorbed. The photo detector then converts the intensity of the light into a digital signal of 1s and 0s.

Recordable CD and DVD disc drives require more expensive disc controllers than the read-only disc players because they need to incorporate write mechanisms specific to each medium. For example, a CD consists of several layers, including a gold reflective layer and a dye layer, which is used to record the data. The write head uses a high-powered laser beam to record data. A permanent mark is made on the dye when the energy from the laser beam is absorbed into it and it cannot be erased after it is recorded. When it is read, the existence of a mark on the dye will cause the laser beam to scatter and light is not returned back to the read head. However, when there are no marks on the dye, the gold layer reflects the light right back to the read head. This is similar to the process of reading pits and lands. The software used to create a recordable CD (CD-R) uses a standard format, such as ISO 9096, which automatically checks for errors and creates a table of contents, used to keep track of each file’s location.

Similarly, recordable and rewritable CDs (CD-RWs) use a process called phase change technology to write, change, and erase data. The disc’s recording layer uses an alloy of silver, indium, antimony, and tellurium. The recording layer has two different phase states: amorphous and crystalline. In the amorphous state, light is not reflected as well as in the crystalline state.

To record data, a laser beam heats up the disc and changes the state from crystalline to amorphous. When data is read by means of a low-power laser beam, the amorphous state scatters the light that does not get picked up by the read head. This is interpreted as a 0 and is similar to what occurs when reading pits. On the other hand, when the light hits the crystalline areas, light is reflected back to the read head, and this is similar to what occurs when reading lands and is interpreted as a 1. To erase data, the CD-RW drive uses a low-energy beam to heat up the pits just enough to loosen the alloy and return it to its original crystalline state.

Although DVDs use the same design and are the same size and shape as CDs, they can store much more data. A dual-layer, single-sided DVD can hold the equivalent of 13 CDs; its red laser, with a shorter wavelength than the CD’s red laser, makes smaller pits and allows the spiral to be wound tighter. When the advantages of compression technology (discussed in the next chapter) are added to the high capacity of a DVD, such as MPEG video compression, a single-sided, double-layer DVD can hold 8.6GB, more than enough space to hold a two-hour movie with enhanced audio.

Blu-ray Disc Technology

A Blu-ray disc is the same physical size as a DVD or CD but the laser technology used to read and write data is quite different. As shown in Figure 7.9, the pits (each representing a 1) on a Blu-ray disc are much smaller and the tracks are wound much tighter than they are on a DVD or CD. Although Blu-ray products can be made backward compatible so they can accommodate the older CDs and DVDs, the Blu-ray’s blue- violet laser (405nm) has a shorter wavelength than the CD/DVD’s red laser (650nm). This allows data to be packed more tightly and stored in less space.

In addition, the blue-violet laser can write on a much thinner layer on the disc, allowing multiple layers to be written on top of each other and vastly increasing the amount of data that can be stored on a single disc. The disc's format was created to further the commercial prospects for high-definition video, and to store large amounts of data, particularly for games and interactive applications via the Java programming language. Blu-ray players execute Java programs for menus and user interaction.

As of this writing, each Blu-ray disc can hold much more data (50GB for a two-layer disc) than can a similar DVD (8.5GB for a two-layer disc). (Pioneer Electronics has reported that its new 20-layer discs can hold 500GB.) Reading speed is also much faster with the fastest Blu-ray players featuring 432 Mbps (comparable DVD players reach 168.75 Mbps). Like CDs and DVDs, Blu-ray discs are available in several

Flash Memory Storage

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). It's a nonvolatile removable medium that emulates random access memory, but, unlike RAM, stores data securely even when it's removed from its power source. Historically, flash memory was primarily used to store startup (boot up) information for computers, but is now used to store data for cell phones, mobile devices, music players, cameras, and more.formats: read-only (BD-ROM), recordable (BD-R), and rewritable (BD- RE).

Flash memory uses a phenomenon (known as Fowler-Nordheim tunneling) to send electrons through a floating gate transistor where they remain even after power is turned off. Flash memory allows users to store data. It is sold in a variety of configurations, including compact flash, smart cards, and memory sticks, and they often connect to the computer through the USB port.

Flash memory gets its name from the technique used to erase its data. To write data, an electric charge is sent through one transistor, called the floating gate, then through a metal oxide layer, and into a second transistor called the control gate where the charge is stored in a cell until it's erased. To reset all values, a strong electrical field, called a flash, is applied to the entire card. However, flash memory isn't indestructible. It has two limitations: The bits can be erased only by applying the flash to a large block of memory and, with each flash erasure, the block becomes less stable. In time (after 10,000 to 1,000,000 uses), a flash memory device will no longer reliably store data.

Magnetic Disk Drive 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. Obviously, seek time doesn't apply to devices with fixed read/write heads because each track has its own read/write head. 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. Transfer time is the fastest of the three; that's when the data is actually transferred from secondary storage to main memory.

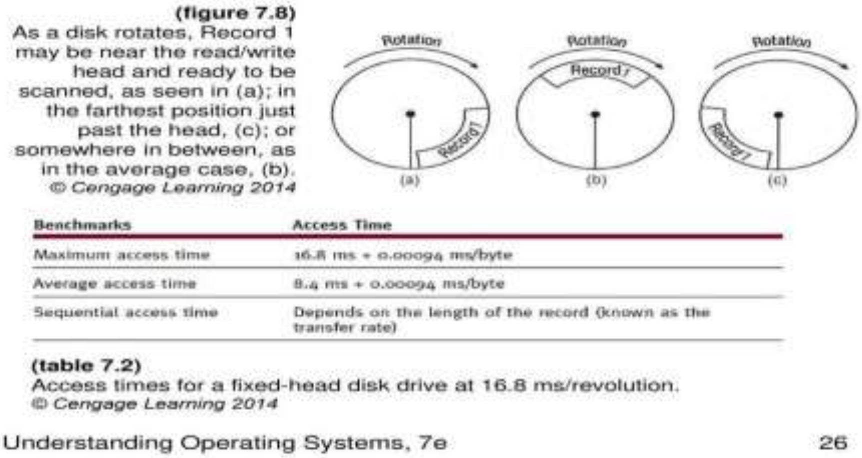
Fixed-Head Drives

Fixed-head disk drives are fast. The total amount of time required to access data depends on the rotational speed, which varies from device to device but is constant within each device, and 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 7.10(a) shows the best possible situation because the record is next to the read/write head when the I/O command is executed; this gives a rotational delay of zero. Figure 7.10(b) shows the average situation because the record is directly opposite the read/write head when the I/O command is executed; this gives a rotational delay of t/2 where t (time) is one full rotation. Figure 7.10(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 7.10& Figure 7.11



(figure 7.8)
As a disk rotates, Record 1 may be near the read/write head and ready to be scanned, as seen in (a); in the farthest position just past the head, (c); or somewhere in between, as in the average case, (b).
© Cengage Learning 2014

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)
Access times for a fixed-head disk drive at 16.8 ms/revolution.
© Cengage Learning 2014

Understanding Operating Systems, 7e 26

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 7.10(b), is 8.4 ms. The data-transfer time varies from device to device, but a typical value is 0.00094 ms per byte—the size of the record dictates this value. For example, if it takes 0.094 ms (almost 0.1 ms) to transfer a record with 100 bytes, then the resulting access times are shown in Table 7.2.

Table 7.2 : Access times for a fixed-head disk drive at 16.8 ms/revolution.

Benchmarks	Access Time
Maximum access	16.8 ms + 0.00094 ms/byte
Average access	8.4 ms + 0.00094 ms/byte
Sequential access	Depends on the length of the record; generally less than 1 ms (known as the transfer rate)

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:

access time = 8.4 + 0.094 = 8.494 ms for one record

total access time = 10(8.4 + 0.094) = 84.940 ms for 10 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:

access time = 8.4 + (0.094 * 10) = 8.4 + 0.94 = 9.34 ms for 10 records in one block

Once the block is in memory, the software that handles blocking and deblocking takes over. But, 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.

Movable-Head Devices

Movable-head disk drives add a third time element to the computation of access time. Seek time is the time required to move the arm into position over the proper track. So now the formula for access time is:

seek time (arm movement) search time (rotational delay) + transfer time (data transfer) access time

Of the three components of access time in this equation, seek time is the longest. We'll examine several seek strategies in a moment. The calculations to figure search time (rotational delay) and transfer time are the same as those presented for fixed-head drives. The maximum seek time, which is the maximum time required to move the arm, is typically 50 ms. Table 7.3 compares typical access times for movable-head drives.

Benchmarks	**Access Time**
Maximum access	50 ms + 16.8 ms + 0.00094 ms/byte
Average access	25 ms + 8.4 ms + 0.00094 ms/byte
Sequential access	Depends on the length of the record, generally less than 1 ms

The variance in access time has increased in comparison to that of the fixed-head drive but it's relatively small—especially when compared to tape access, which varies from milliseconds to minutes. Again, blocking is a good way to minimize access time. If we use the same example as for fixed-head disks and consider the average case with 10 seeks followed by 10 searches, we would get:

access time = 25 + 8.4 + 0.094 = 33.494 ms for one record

total access time = 10 * 33.494 = 334.94 ms for 10 records (about 1 ⁄3 of a second)

But when we put the 10 records into one block, the access time is significantly decreased:

total access time = 25 + 8.4 + (0.094 * 10) = 33.4 + 0.94 = 34.34 ms for

10 records (about 1 ⁄29 of a second)

We stress that these figures wouldn't apply in an actual operating environment. For instance, we haven't taken into consideration what else is happening in the system while I/O is taking place. Therefore, although we can show the comparable performance of these components of the system, we're not seeing the whole picture.

Components of the I/O Subsystem

The pieces of the I/O subsystem all have to work harmoniously, and they work in a manner similar to the mythical “Flynn Taxicab Company” shown in Figure 7.11.

Many requests come in from all over the city to the taxi company dispatcher. It’s the dispatcher’s job to handle the incoming calls as fast as they arrive and to find out who needs transportation, where they are, where they’re going, and when. Then the dispatcher organizes the calls into an order that will use the company’s resources as efficiently as possible. That’s not easy, because the cab company has a variety of vehicles at its disposal: ordinary taxicabs, station wagons, vans, limos, and a minibus. These are serviced by specialized mechanics. A mechanic handles only one type of vehicle, which is made available to many drivers. Once the order is set, the dispatcher calls the mechanic who, ideally, has the vehicle ready for the driver who jumps into the appropriate vehicle, picks up the waiting passengers, and delivers them quickly to their respective destinations.

The I/O subsystem’s components perform similar functions. The channel plays the part of the dispatcher in this example. Its job is to keep up with the I/O requests from the CPU and pass them down the line to the appropriate control unit. The control units play the part of the mechanics. The I/O devices play the part of the vehicles.

I/O channels are programmable units placed between the CPU and the control units. Their job is to synchronize the fast speed of the CPU with the slow speed of the I/O device, and they make it possible to overlap I/O operations with processor operations so the CPU and I/O can process concurrently. Channels use I/O channel programs, which can range in size from one to many instructions. Each channel program specifies the action to be performed by the devices and controls the transmission of data between main memory and the control units.

At the start of an I/O command, the information passed from the CPU to the channel is this:

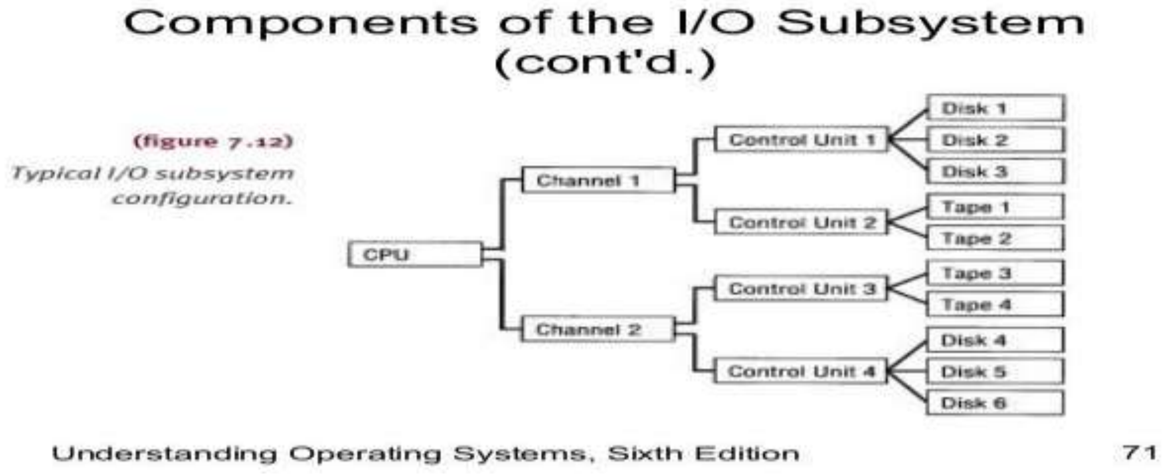
- I/O command (READ, WRITE, REWIND, etc.)
- Channel number
- Address of the physical record to be transferred (from or to secondary storage)
- Starting address of a memory buffer from which or into which the record is to be transferred

The system shown in Figure 7.12 requires that the entire path be available when an I/O command is initiated. However, there’s some flexibility built into the system because each unit can end independently of the others, as will be explained in the next section. This figure also shows the hierarchical nature of the interconnection and the one-to-one correspondence between each device and its transmission path.

Additional flexibility can be built into the system by connecting more than one channel to a control unit or by connecting more than one control unit to a single device. That’s the same as if the mechanics of the Flynn Taxicab Company could also make repairs for the ABC Taxicab Company, or if its vehicles could be used by ABC drivers (or if the drivers in the company could share vehicles).

These multiple paths increase the reliability of the I/O subsystem by keeping communication lines open even if a component malfunctions. Figure 7.13 shows the same system presented in Figure 7.12, but with one control unit connected to two channels and one device connected to two control units.

Figure 7.12 : Typical I/O subsystem configuration



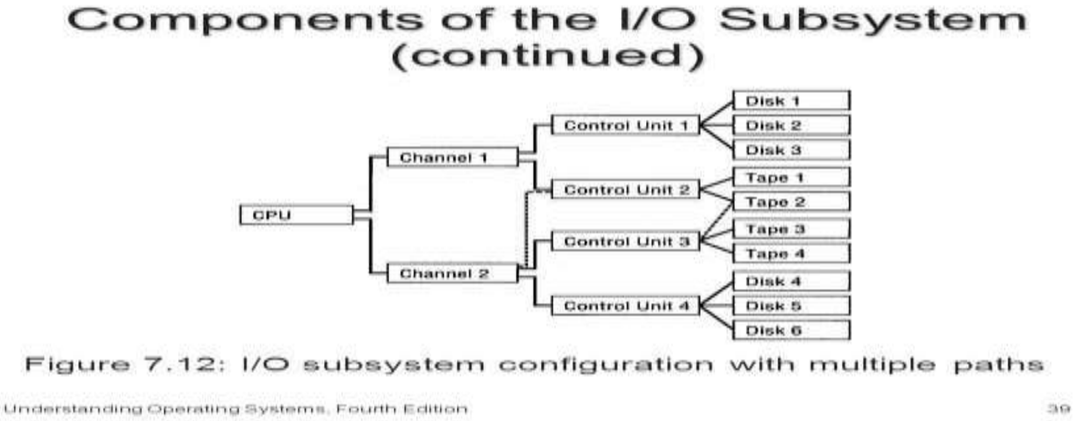
Components of the I/O Subsystem (cont'd.)

(figure 7.12)

Typical I/O subsystem configuration.

Understanding Operating Systems, Sixth Edition 71

Figure 7.13 :



Components of the I/O Subsystem (continued)

Figure 7.12: I/O subsystem configuration with multiple paths

Understanding Operating Systems, Fourth Edition 39

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 problems that must be resolved: • 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 buffering records and queuing requests.

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 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. This flag is made up of three bits and resides in the Channel Status Word (CSW), which is in a predefined location in main memory and 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 and vice versa. There are two common ways to perform this test—polling and using interrupts.

Polling uses a special machine instruction to test the flag. For example, the CPU periodically tests the channel status bit (in the CSW). If the channel 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 having the CPU test the flag, a hardware mechanism does the test as part of every machine instruction executed 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, which 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 current situation because it's not unusual for more than one unit to have caused the I/O interrupt. So 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.

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

Buffers are used extensively to better synchronize the movement of data between the relatively slow I/O devices and the very fast CPU. Buffers are temporary storage areas residing in three convenient locations throughout the system: main memory, channels, and control units. They're used to store data read from an input device before it's needed by the processor and 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 important, to maximize their throughput, the technique of double buffering is used, as shown in Figure 7.14. In this system, two buffers are present in main memory, channels, and 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.

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

Management of I/O Requests

Although most users 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 I/O subsystem. The I/O traffic controller watches the status of all devices, control units, and channels. The I/O scheduler 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.

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: (1) it must determine if there's at least one path 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 shown in Table 7.4.

Table 7.4 Each control block contains the information it needs to manage its part of the I/O subsystem.

Channel Control Block	**Control Unit Control Block**	**Device Control Block**
• Channel identification	• Control unit identification	• Device Identification
• Status	• Status	• Status
• List of control units connected to it	• List of channels connected to it	• List of control units connected to it
• List of processes waiting for it	• List of devices connected to it	

	• List of processes waiting for it	• List of processes waiting for it

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 (actually its Process Control Block, or PCB) 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 quickly selects the first PCB from the queue for that path.

The I/O scheduler performs the same job as the Process Scheduler 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 preemption 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 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.

Device Handler Seek Strategies

A seek strategy for the I/O device handler is the predetermined policy that the device handler uses to allocate access to the device among the many processes that may be waiting for it. It determines the order in which the processes get the device, and the goal is to keep seek time to a minimum. We'll look at some of the most commonly used seek strategies—first-come, first-served (FCFS); shortest seek time first (SSTF); and SCAN and its variations LOOK, N-Step SCAN, C-SCAN, and C-LOOK.

Every scheduling algorithm should do the following: • Minimize arm movement • Minimize mean response time • Minimize the variance in response time

These goals are only a guide. In actual systems, the designer must choose the strategy that makes the system as fair as possible to the general user population while using the system's resources as efficiently as possible.

First-come, first-served (FCFS) is the simplest device-scheduling algorithm; it is easy to program and essentially fair to users. However, on average, it doesn't meet any of the three goals of a seek strategy. To illustrate, consider a single-sided disk with one recordable surface where the tracks are numbered from 0 to 49. It takes 1 ms to travel from one track to the next adjacent one. For this example, let's say that while retrieving data from Track 15, the following list of requests has arrived: Tracks 4, 40, 11, 35, 7, and 14. Let’s also assume that once a requested track has been reached, the entire track is read into main memory.

FCFS has an obvious disadvantage of extreme arm movement: from 15 to 4, up to 40, back to 11, up to 35, back to 7, and, finally, up to 14. Remember, seek time is the most time-consuming of the three functions performed here, so any algorithm that can minimize it is preferable to FCFS.

Shortest seek time first (SSTF) uses the same underlying philosophy as Shortest Job Next (described in Chapter 4), where the shortest jobs are processed first and longer jobs are made to wait. With SSTF, the request with the track closest to the one being served (that is, the one with the shortest distance to travel) is the next to be satisfied, thus minimizing overall seek time. Again, without considering search time and data transfer time, it took 47 ms to satisfy all requests—which is about one third of the time required by FCFS. That’s a substantial improvement.

But SSTF has its disadvantages. Remember that the Shortest Job Next (SJN) process scheduling algorithm had a tendency to favor the short jobs and postpone the long, unwieldy jobs. The same holds true for SSTF. It favors easy-to-reach requests and postpones traveling to those that are out of the way

SCAN uses a directional bit to indicate whether the arm is moving toward the center of the disk or away from it. The algorithm moves the arm methodically from the outer to the inner track, servicing every request in its path. When it reaches the innermost track, it reverses direction and moves toward the outer tracks, again servicing every request in its path. The most common variation of SCAN is LOOK, sometimes known as the elevator algorithm, in which the arm doesn’t necessarily go all the way to either edge unless there are requests there. In effect, it “looks” ahead for a request before going to service it.

Again, without adding search time and data transfer time, it took 61 ms to satisfy all requests, 14 ms more than with SSTF. Does this make SCAN a less attractive algorithm than SSTF? For this particular example, the answer is yes. But for the overall system, the answer is no because it eliminates the possibility of indefinite postponement of requests in out-of-the-way places—at either edge of the disk. Also, as requests arrive, each is incorporated in its proper place in the queue and serviced when the arm reaches that track. Therefore, if Track 11 is being served when the request for Track 13 arrives, the arm continues on its way to Track 7 and then to Track 1. Track 13 must wait until the arm starts on its way back, as does the request for Track 16. This eliminates a great deal of arm movement and saves time in the end. In fact, SCAN meets all three goals for seek strategies. Variations of SCAN, in addition to LOOK, are N-Step SCAN, C-SCAN, and C-LOOK.

N-Step SCAN holds all new requests until the arm starts on its way back. Any requests that arrive while the arm is in motion are grouped for the arm’s next sweep.

With C-SCAN (an abbreviation for Circular SCAN), the arm picks up requests on its path during the inward sweep. When the innermost track has been reached, it immediately returns to the outermost track and starts servicing requests that arrived during its last inward sweep. With this modification, the system can provide quicker service to those requests that accumulated for the low-numbered tracks while the arm was moving inward. The theory here is that by the time the arm reaches the highestnumbered tracks, there are few requests immediately behind it. However, there are many requests at the far end of the disk and these have been waiting the longest. Therefore, C-SCAN is designed to provide a more uniform wait time.

C-LOOK is an optimization of C-SCAN, just as LOOK is an optimization of SCAN. In this algorithm, the sweep inward stops at the last high-numbered track request, so the arm doesn’t move all the way to the last track unless it's required to do so. In addition, the arm doesn't necessarily return to the lowest-numbered track; it returns only to the lowest-numbered track that's requested.

some broad generalizations can be made based on simulation studies:

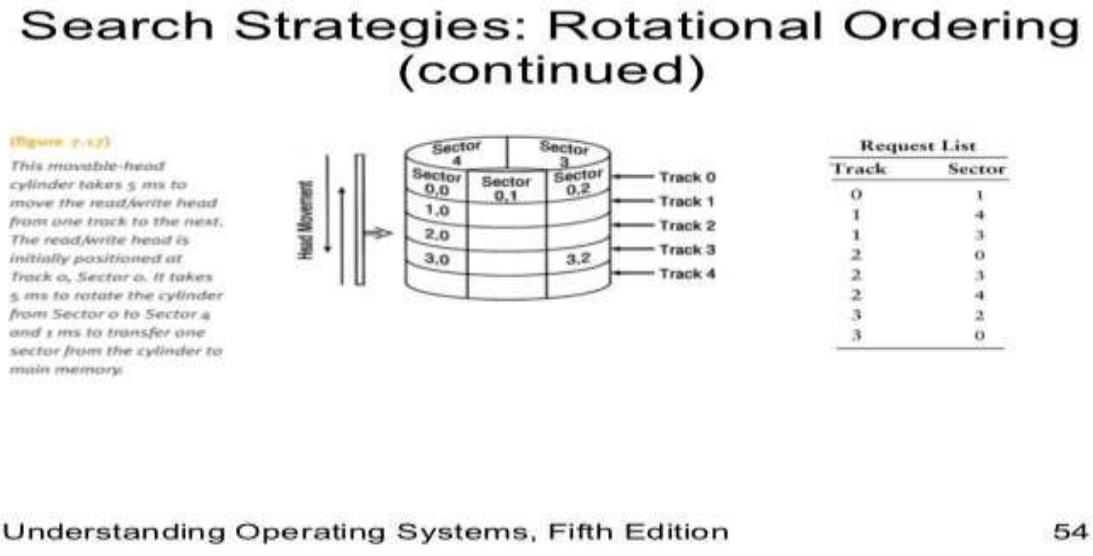
- FCFS works well with light loads; but as soon as the load grows, service time becomes unacceptably long.
- SSTF is quite popular and intuitively appealing. It works well with moderate loads but has the problem of localization under heavy loads.
- SCAN works well with light to moderate loads and eliminates the problem of indefinite postponement. SCAN is similar to SSTF in throughput and mean service times.
- C-SCAN works well with moderate to heavy loads and has a very small variance in service times.

Search Strategies: Rotational Ordering

Look at a way to optimize search times by ordering the requests once the read/write heads have been positioned. This search strategy is called rotational ordering.

To help illustrate the abstract concept of rotational ordering, let's consider a virtual cylinder with a movable read/write head. Figure 7.18 illustrates the list of requests arriving at this cylinder for different sectors on different tracks. For this example, we'll assume that the cylinder has only five tracks, numbered 0 through 4, and that each track contains five sectors, numbered 0 through 4. We'll take the requests in the order in which they arrive.

Figure 7.18 : This movable-head cylinder takes 5 ms to move the read/write head from one track to the next. The read/write head is initially positioned at Track 0, Sector 0. It takes 5 ms to rotate the cylinder from Sector 0 to Sector 4 and 1 ms to transfer one sector from the cylinder to main memory.



Search Strategies: Rotational Ordering (continued)

(figure 7.17)

This movable-head cylinder takes 5 ms to move the read/write head from one track to the next. The read/write head is initially positioned at Track 0, Sector 0. It takes 5 ms to rotate the cylinder from Sector 0 to Sector 4 and 1 ms to transfer one sector from the cylinder to main memory.

Request List

Track	Sector
0	1
1	4
1	3
2	0
2	3
2	4
3	2
3	0

Understanding Operating Systems, Fifth Edition 54

Each request is satisfied as it comes in. The results are shown in Table 7.5.

Table 7.5 : It takes 36 ms to fill the eight requests on the movable-head cylinder shown in Figure 7.18

Request (Track, Sector)		Seek Time	Search Time	Data Transfer	Total Time
1.	0,1	0	1	1	2

2.	1,4	5	2	1	8
3.	1,3	0	3	1	4
4.	2,0	5	1	1	7
5.	2,3	0	2	1	3
6.	2,4	0	0	1	1
7.	3,2	5	2	1	8
8.	3,0	0	2	1	3
	TOTALS	15 ms +	13 ms +	8 ms	= 36 ms

If the requests are ordered within each track so that the first sector requested on the second track is the next number higher than the one just served, rotational delay will be minimized, as shown in Table 7.6.

To properly implement this algorithm, the device controller must provide rotational sensing so the device driver can see which sector is currently under the read/write head. Under heavy I/O loads, this kind of ordering can significantly increase throughput, especially if the device has fixed read/write heads rather than movable heads.

Disk pack cylinders are an extension of the previous example. Once the heads are positioned on a cylinder, each surface has its own read/write head shown in Figure 7.5. So rotational ordering can be accomplished on a surface-by-surface basis, and the read/write heads can be activated in turn with no additional movement required.

(table 7.6) It takes 28 ms to fill the same eight requests shown in Table 7.5 after the requests are ordered to minimize search time, reducing it from 13 ms to 5 ms

Request (Track, Sector)		**Seek Time**	**Search Time**	**Data Transfer**	**Total Time**
1.	0,1	0	1	1	2
2.	1,3	5	1	1	7
3.	1,4	0	0	1	1
4.	2,0	5	0	1	6
5.	2,3	0	2	1	3
6.	2,4	0	0	1	1
7.	3,0	5	0	1	6
8.	3,2	0	1	1	2
	TOTALS	15 ms +	5 ms +	8 ms	= 28 ms

Only one read/write head can be active at any one time, so the controller must be ready to handle mutually exclusive requests such as Request 2 and Request 5 in Table 7.6. They're mutually exclusive because both are requesting Sector 3, one at Track 1 and the other at Track 2, but only one of the two read/write heads can be transmitting at any given time. So the policy could state that the tracks will be processed from low-numbered to high-numbered and then from high-numbered to low-numbered in a sweeping motion such as that used in SCAN.

Therefore, to handle requests on a disk pack, there would be two orderings of requests: one to handle the position of the read/write heads making up the cylinder and the other to handle the processing of each cylinder.

UNIT-V

File Management

The File Manager controls every file in the system. In this chapter we'll 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.

The efficiency of the File Manager depends on how the system's files are organized (sequential, direct, or indexed sequential); how they're stored (contiguously, non contiguously, or indexed); how each file's records are structured (fixed-length or variable length); and how access to these files is controlled.

The File Manager

The File Manager (also called 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 to online users, for spooling operations, and for interactive computing. These functions are performed in collaboration with the Device Manager.

Responsibilities of the File 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 efficient access to the files.
3. Allocate each file when a user has been cleared for access to it, then record its use.
4. Deallocate the file when the file is to be returned to storage, and communicate its availability to others who may be

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, magazines, books-on-tape, DVDs, maps, and videos. 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 reshelves it.

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 each file.

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.

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.

Definitions

Before we continue, let's take a minute to define some basic file elements, illustrated in Figure 8.1, that relate to our discussion of the File Manager.

A field is a group of related bytes that can be identified by the user with a name, type, and size. A record is a group of related fields.

A file is a group of related records that contains information to be used by specific application programs to generate reports. This type of file contains data and is sometimes called a flat file because it has no connections to other files; unlike databases, it has no dimensionality.

A database appears to the File Manager to be a type of file, but databases are more complex because they're actually groups of related files that are interconnected at various levels to give users flexibility of access to the data stored. If the user's database requires a specific structure, the File Manager must be able to support it.

Program files contain instructions and data files contain data; but as far as storage is concerned, the File Manager treats them exactly the same way.

Directories are special files with listings of filenames and their attributes. Data collected to monitor system performance and provide for system accounting is collected into files. In fact, every program and data file accessed by the computer system, as well as every piece of computer software, is treated as a file.

Interacting with the File Manager

The user communicates with the File Manager, which responds to specific commands.

Some examples displayed in Figure 8.2 are OPEN, DELETE, RENAME, and COPY. Actually, files can be created with other system-specific terms: for example, the first time a user gives the command to save a file, it's actually the CREATE command. In other operating systems, the OPEN NEW command within a program indicates to the File Manager that a file must be created.

These commands and many more were designed to be very simple to use, so they're devoid of the detailed instructions required to run the device (information in the device driver) where the file may be stored. That is, they're device independent. Therefore, to access a file, the user doesn't need to know its exact physical location on the disk pack (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. That's fortunate because file access is 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 user's program issues a command to read a record from a disk the READ instruction has to be decomposed into 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 to main memory. 5. Set a flag to indicate that the device is free to satisfy another request.

While all of this is going on, the system must check for possible error conditions. The File Manager does all of this, freeing the user from including in each program the lowlevel instructions for every device to be used: the terminal, keyboard, printer, CD, disk drive, etc. Without the File Manager, every program would need to include instructions to operate all of the different types of devices and every model within each type. Considering the rapid development and increased sophistication of I/O devices, it would be impractical to require each program to include these minute operational details. That's the advantage of device independence.

Typical Volume Configuration

Normally the active files for a computer system reside on secondary storage units. Some devices accommodate removable storage units—such as CDs, DVDs, floppy disks, USB devices, and other removable media—so files that aren't frequently used can be stored offline and mounted only when the user specifically requests them. Other devices feature integrated storage units, such as hard disks and non removable disk packs.

Each storage unit, whether it's removable or not, is considered a volume, and each volume can contain several files, so they're called "multifile volumes." However, some files are extremely large and are contained in several volumes; not surprisingly, these are called "multivolume files."

Each volume in the system is given a name. The File Manager writes this name and other descriptive information, as shown in Figure 8.3, on an easy-to-access place on each unit: the innermost part of the CD or DVD, the beginning of the tape, or the first sector of the outermost track of the disk pack. Once identified, the operating system can interact with the storage unit.

Figure 8.3: The volume descriptor, which is stored at the beginning of each volume, includes this vital information about the storage unit.

The master file directory (MFD) is stored immediately after the volume descriptor and lists the names and characteristics of every file contained in that volume. The filenames in the MFD can refer to program files, data files, and/or system files. And if the File Manager supports subdirectories, they're listed in the MFD as well. The remainder of the volume is used for file storage. The first operating systems supported only a single directory per volume. This directory was created by the File Manager and contained the names of files, usually organized in alphabetical, spatial, or chronological order. Although it was simple to implement and maintain, this scheme had some major disadvantages:

- It would take a long time to search for an individual file, especially if the MFD was organized in an arbitrary order.
- If the user had more than 256 small files stored in the volume, the directory space (with a 256 filename limit) would fill up before the disk storage space filled up. The user would then receive a message of "disk full" when only the directory itself was full.
- Users couldn't create subdirectories to group the files that were related.
- Multiple users couldn't safeguard their files from other users because the entire directory was freely made available to every user in the group on request.
- Each program in the entire directory needed a unique name, even those directories serving many users, so only one person using that directory could have a program named Program1.

Introducing Subdirectories

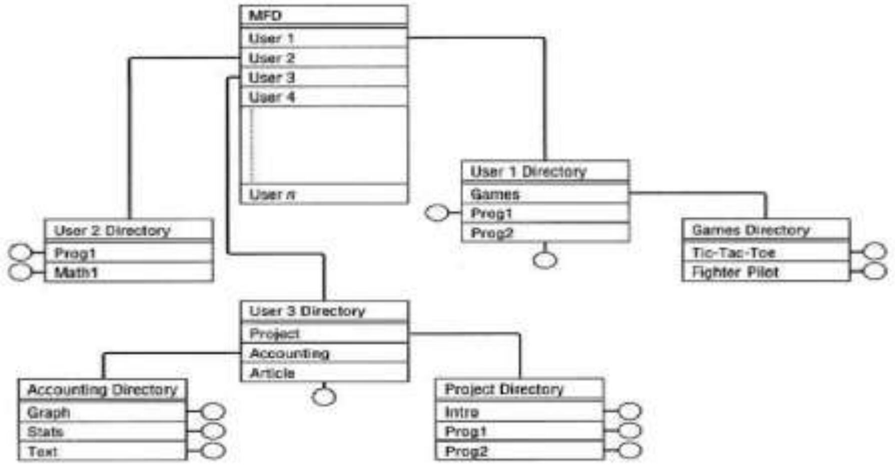
File Managers create an MFD for each volume that can contain entries for both files and subdirectories. A subdirectory is created when a user opens an account to access the computer system. Although this user directory is treated as a file, its entry in the MFD is flagged to indicate to the File Manager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

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. Actually, the MFD 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.

Regardless of the complexity of the directory structure, each file entry in every directory 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 example, files and directories) • 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
- Record size—its fixed size or its maximum size, depending on the type of record

Figure 8.4 : 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.



(figure 8.4)
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. © Cengage Learning 2014

Understanding Operating Systems,7e 21

File-Naming Conventions

A file's name can be much longer than it appears. Depending on the File Manager, it can have from two to many components. The two components common to many filenames are a relative filename and an extension.

To avoid confusion, in the following discussion we'll use the term "complete filename" to identify the file's absolute filename (that's the long name that includes all path information), and "relative filename" to indicate the name without 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. Generally, the relative filename can vary in length from one to many characters and can include letters of the alphabet, as well as digits. However, every operating system has specific rules that affect the length of the relative name and the types of characters allowed. Most operating systems allow names with dozens of characters including spaces, hyphens, underlines, and certain other keyboard characters.

Some operating systems require an extension that's appended to the relative filename. It's usually two or three characters long and is separated from the relative name by a period, and its purpose is to identify the type of file or its contents. For example, in a Windows operating system, a typical relative filename with extension would be BASIA_TUNE.MP3. Similarly, TAKE OUT MENU.RTF and TAKE OUT MENU.DOC both indicate that they can be opened with a word processing application. What happens if an extension is incorrect or unknown? Most ask for guidance from the user, as shown in Figure 8.5.

(figure 8.5) :To open a file with an unrecognized extension, Windows asks the user to choose an application to associate with that type of file.

STET WOMEN'S

Some extensions (such as EXE, BAT, COB, and FOR) 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.

There may be other components required for a file's complete name. Here's how a file named INVENTORY_COST.DOC is identified by different operating systems:

1. Using a Windows operating system and a personal computer with three disk drives, the file's complete name is composed of its relative name and extension, preceded by the drive label and directory name: C:\IMFST\FLYNN\INVENTORY_COST.DOC

This indicates to the system that the file INVENTORY_COST.DOC requires a word processing application program, and it can be found in the directory; IMFST; subdirectory FLYNN in the volume residing on drive C.

2. A UNIX or Linux system might identify the file as: /usr/imfst/flynn/inventory_cost.doc

The first entry is represented by the forward slash (/). This represents a special master directory called the root. Next is the name of the first subdirectory, usr/imfst, followed by a sub-subdirectory, /flynn, in this multiple directory system. The final entry is the file's relative name, inventory_cost.doc. (Notice that UNIX and Linux filenames are case sensitive and often expressed in lowercase.)

Why don't users see the complete file name when accessing a file? First, the File Manager selects a directory for the user when the interactive session begins, so all file operations requested by that user start from this "home" or "base" directory. Second, from this home directory, the user selects a subdirectory, which is called a current directory or working directory. Thereafter, the files are presumed to be located in this current directory. Whenever a file is accessed, the user types in the relative name, and the File Manager adds the proper prefix. As long as users refer to files in the working directory, they can access their files without entering the complete name.

The concept of a current directory is based on the underlying hierarchy of a tree structure, as shown in Figure 8.4, and allows programmers to retrieve a file by typing only its relative filename…

INVENTORY_COST.DOC …

and not its complete filename: C:\IMFST\FLYNN\INVENTORY_COST.DOC

File Organization

When we discuss file organization, we are talking about the arrangement of records within a file because all files are composed of records. 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

All files are composed of records. When a user gives a command to modify the contents of a file, it's actually 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 of variable length, as shown in Figure 8.6. And these records, regardless of their format, can be blocked or not blocked.

Fixed-length records are the most common because they're the easiest to access directly. That's why they're ideal for data files. The critical aspect of fixed-length records is the size of the record. If it's too small—smaller than the number of characters to be stored in the record—the leftover characters are truncated. But if the record size is too large—larger than the number of characters to be stored—storage space is wasted.

Variable-length records don't leave empty storage space and don't truncate any characters, thus eliminating the two disadvantages of fixed-length records. But while they can easily be read (one after the other), they're difficult to access directly because it's hard to calculate exactly where the record is located. That's why they're used most frequently in files that are likely to be accessed sequentially, such as text files and program files or files that use an index to access their records. The record format, how it's blocked, and other related information is 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

STET WOMEN'S COLLEGE

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: sequential, direct, or indexed sequential. To select the best of these file organizations, the programmer or analyst usually 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 processed 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 (This is especially crucial when doing time-sensitive searches)

Sequential record organization is by far the easiest to implement because records are stored and retrieved serially, one after the other. To find a specific record, the file is searched from its beginning until the requested record is found. To speed 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, in which case 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 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. The user identifies a field (or combination of fields) in the record format and designates it as the key field because it uniquely identifies each record. The program used to store the data follows a set of instructions, called a hashing algorithm, that 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.

Direct access files can be updated more quickly than sequential files because records can be quickly rewritten to their original addresses after modifications have been made. And there's no need to preserve the order of the records so adding or deleting them takes very little time.

For example, data for a telephone mail-order firm must be accessed quickly so customer information can be retrieved quickly. To do so, they can use hashing algorithms to directly access their data. Let's say you're placing an order, and you're asked for your postal code and street number (let's say they are 15213 and 2737). The program that retrieves information from the data file uses that key in a hashing algorithm to calculate the logical address where your record is stored. So when the order clerk types 152132737, the screen soon shows a list of all current customers whose customer numbers generated the same logical address. If you're in the database, the operator knows right away. If not, you will be soon.

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

STET WOMEN'S COLLEGE

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.

The maximum size of the file is established when it's created, and eventually either the file might become completely full or the number of records stored in the overflow area might become so large that the efficiency of retrieval is lost. At that time, the file must be reorganized and rewritten, which requires intervention by the File Manager.

Indexed sequential record organization combines the best of sequential and direct access. It's created and maintained through an Indexed Sequential Access Method (ISAM) application, which removes the burden of handling overflows and preserves record order from the shoulders of the programmer.

This type of organization doesn't create collisions because it doesn't use the result of the hashing algorithm to generate a record's address. Instead, it uses this information to generate an index file through which the records are retrieved. This organization divides an ordered sequential file into blocks of equal size. Their size is determined by the File Manager to take advantage of physical storage devices and to optimize retrieval strategies. Each entry in the index file contains the highest record key and the physical location of the data block where this record, and the records with smaller keys, are stored.

Therefore, to access any record in the file, the system begins by searching the index file and then goes to the physical location indicated at that entry. We can say, then, that the index file acts as a pointer to the data file. An indexed sequential file also has overflow areas, but they're spread throughout the file, perhaps every few records, so expansion of existing records can take place, and new records can be located in close physical sequence as well as in logical sequence.

Another overflow area is located apart from the main data area but is used only when the other overflow areas are completely filled. We call it the overflow of last resort. This last-resort overflow area can store records added during the lifetime of the file. The records are kept in logical order by the software package without much effort on the part of the programmer. Of course, when too many records have been added here, the retrieval process slows down because the search for a record has to go from the index to the main data area and eventually to the overflow area.

When retrieval time becomes too slow, the file has to be reorganized. That's a job that, although it's not as tedious as reorganizing direct access files, is usually performed by maintenance software. For most dynamic files, indexed sequential is the organization of choice because it allows both direct access to a few requested records and sequential access to many.

Physical Storage Allocation

The File Manager must work with files not just as whole units but also as logical units or records. 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 the operating system. An exception is made for those systems that are heavily oriented to 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? At this stage the File Manager and Device Manager have to cooperate to ensure successful storage and retrieval of records.

STET WOMEN'S COLLEGE

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, so the C7047_08_Ch08.qxd 1/13/10 4:37 PM Page 263 directory is 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 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 created. If there's not enough 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.

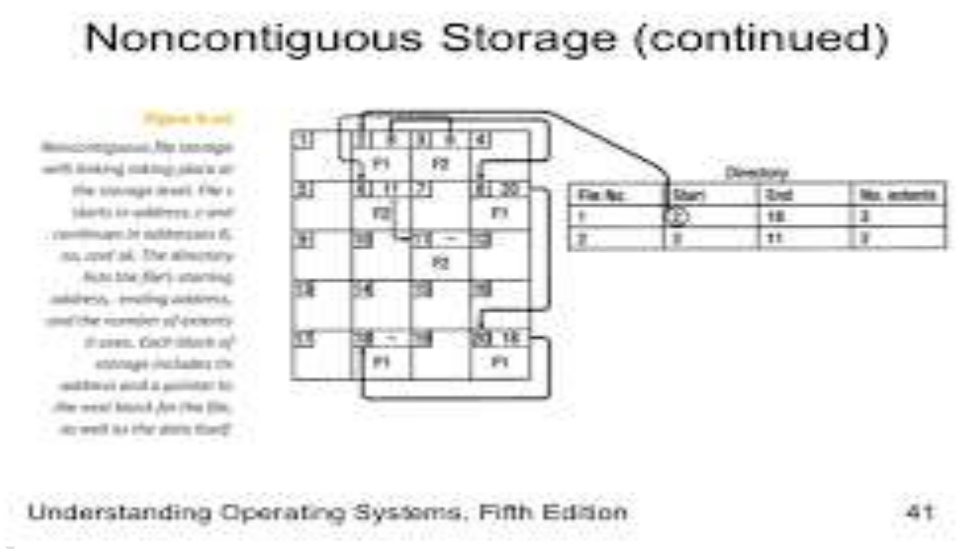
The File Manager keeps track of the empty storage areas by treating them as files— they're entered in the directory but are flagged to differentiate them from real files. Usually the directory is kept in order by sector number, so adjacent empty areas can be combined into one large free space.

Noncontiguous Storage

Noncontiguous 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—or another 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.

Figure 8.10 : Noncontiguous file storage with linking taking place at the storage level. File 1 starts in address 2 and continues in addresses 8, 20, and 18. The directory lists the file's starting address, ending address, and the number of extents it uses. Each block of storage includes its address and a pointer to the next block for the file, as well as the data itself


Noncontiguous Storage (continued)

File No.	Start	End	No. extents
1	2	18	3
2	3	11	3

Understanding Operating Systems, Fifth Edition 41

The alternative is for the linking to take place at the directory level, as shown in Figure 8.11. Each extent is listed with its physical address, its size, and a pointer to the next extent. A null pointer, shown in Figure 8.11 as a hyphen (-), indicates that it's the last one.

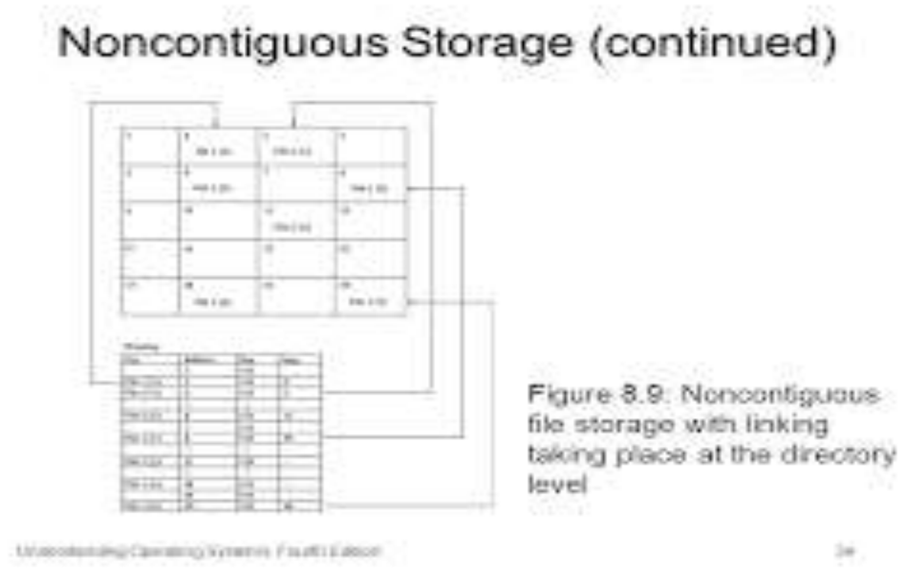
Although both noncontiguous allocation schemes eliminate external storage fragmentation and the need for compaction, they don’t support direct access because there’s no easy way to determine the exact location of a specific record.

Files are usually declared to be either sequential or direct when they’re created, so the File Manager can select the most efficient method of storage allocation: contiguous for direct files and noncontiguous 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 that specific purpose.

Indexed Storage

Indexed storage allocation allows direct record access by bringing together the pointers linking every extent of that file into an index block. Every file has its own index block, which consists of the addresses of each disk sector that make up the file. The index lists each entry in the same order in which the sectors are linked, as shown in Figure 8.12. For example, the third entry in the index block corresponds to the third sector making up the file.

Figure 8.11: Noncontiguous storage allocation with linking taking place at the directory level for the files shown in Figure 8.10.



Noncontiguous Storage (continued)

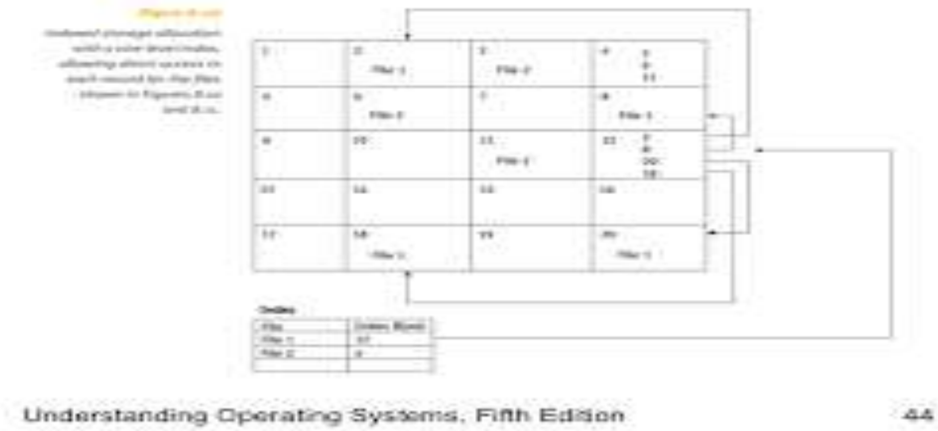
Figure 8.9: Noncontiguous file storage with linking taking place at the directory level

Understanding Operating Systems, Fourth Edition

34

When a file is created, the pointers in the index block are all set to null. Then, as each sector is filled, the pointer is set to the appropriate sector address—to be precise, the address is removed from the empty space list and copied into its position in the index block. This scheme supports both sequential and direct access, but it doesn’t necessarily improve the use of storage space because each file must have an index block—usually the size of one disk sector. For larger files with more entries, several levels of indexes

(figure 8.12) Indexed storage allocation with a one-level index, allowing direct access to each record for the files shown in Figures 8.10 and 8.11



Understanding Operating Systems, Fifth Edition 44

can be generated; in which case, to find a desired record, the File Manager accesses the first index (the highest level), which points to a second index (lower level), which points to an even lower-level index and eventually to the data record.

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.

A file that has been organized in sequential fashion can support only sequential access to its records, and these records can be of either fixed or variable length, as shown in Figure 8.6. 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.

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:

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 to the CBA.

CBA = CBA + N + RL

Direct Access

If a file is organized in direct fashion, it can be accessed easily in either direct or sequential order if the records are of fixed length. 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 – 1) * 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. An alternative is for the File Manager to keep a table of record numbers and their CBAs. Then, to fill a request, this table is searched for the exact storage location of the desired record, so the direct

STET WOMEN'S COLLEGE

access reduces to a table lookup. To avoid dealing with this problem, many systems force users to have their files organized for fixed-length records, if the records are to be accessed directly.

Records in an indexed sequential file can be accessed either sequentially or directly, so either of the procedures to compute the CBA presented in this section would apply but with one extra step: the index file must be searched for the pointer to the block where the data is stored. Because the index file is smaller than the data file, it can be kept in main memory, and a quick search can be performed to locate the block where the desired record is located. Then, the block can be retrieved from secondary storage, and the beginning byte address of the record can be calculated. In systems that support several levels of indexing to improve access to very large files, the index at each level must be searched before the computation of the CBA can be done. The entry point to this type of data file is usually through the index file.

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. The highest level module is called the "basic file system," and 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.

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, which 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 8.1.

Table 8.1: A typical list of files stored in the directory called journal

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

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 transforms the record number to its byte address using the familiar formula:

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

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 that block, it computes the record's offset within that block using these formulas:

STET WOMEN'S COLLEGE

block number = integers(byte address/ physical block size)+ address of the first physical record

offset = remainder (byte address/ physical block size)

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, here's where the device-scheduling algorithms come into play, as 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.

we used a READ command for our example, a WRITE command is handled in exactly the same way until the process reaches the device handler. At that point, the portion of the device interface module that handles allocation of free space, the allocation module, 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.

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

Access Control Verification Module

The first operating systems couldn't support file sharing among users. For instance, early systems needed 10 copies of a compiler to serve 10 users. Today's systems require only a single copy to serve everyone, regardless of the number of active programs in the system. In fact, any file can be shared—from data files and user-owned program files to system files. The advantages of file sharing are numerous. In addition to saving space, it allows for synchronization of data updates, as when two applications are updating the same data file. It also improves the efficiency of the system's resources because if files are shared in main memory, then there's a reduction of I/O operations.

However, as often happens, progress brings problems. The disadvantage of file sharing is that the integrity of each file must be safeguarded; that calls for control over who is allowed to access the file and what type of access is permitted. There are five possible actions that can be performed on a file—the ability to READ only, WRITE only, EXECUTE only, DELETE only, or some combination of the four. Each file management system has its own method to control file access.

Access Control Matrix

The access control matrix is intuitively appealing and easy to implement, but because of its size it only works well for systems with a few files and a few users. In the matrix, each column identifies a user and each row identifies a file. The intersection of the row and column contains the access rights for that user to that file, as Table 8.2 illustrates.

Table 8.2 :The access control matrix showing access rights for each user for each file. User 1 is allowed unlimited access to File 1 but is allowed only to read and execute File 4 and is denied access to the three other files. R = Read Access, W = Write Access, E = Execute Access, D = Delete Access, and a dash (-) = Access Not Allowed.

	User 1	**User 2**	**User 3**	**User 4**	**User 5**
File 1	RWED	R-E	- ----	RWE	- --E
File 2	----	R-E-	R-E-	--E-	----
File 3	----	RWED	----	--E-	----
File 4	R-E-	----	----	----	RWED
File 5	----	----	----	----	RWED

STET WOMEN'S COLLEGE

In the actual implementation, the letters RWED are represented by bits 1 and 0: a 1 indicates that access is allowed, and a 0 indicates access is denied. Therefore, as shown in Table 8.3, the code for User 2 for File 1 would read "1010" and not "R-E-".

Table 8.3 :The five access codes for User 2 from Table 8.2. The resulting code for each file is created by assigning a 1 for each checkmark, and a 0 for each blank space.

Access	R	W	E	D	Resulting Code
R-E-					1010
R-E-					1010
RWED					1111
----					0000
----					0000

The access control matrix is a simple method; but as the numbers of files and users increase, the matrix becomes extremely large—sometimes too large to store in main memory. Another disadvantage is that a lot of space is wasted because many of the entries are all null, such as in Table 8.2, where User 3 isn't allowed into most of the files, and File 5 is restricted to all but one user. A scheme that conserved space would have only one entry for User 3 or one for File 5, but that's incompatible with the matrix format.

Access Control Lists

The access control list is a modification of the access control matrix. Each file is entered in the list and contains the names of the users who are allowed to access it and the type of access each is permitted. To shorten the list, only those who may use the file are named; those denied any access are grouped under a global heading such as WORLD, as shown in Table 8.4.

Table 8.4: An access control list showing which users are allowed to access each file. This method requires less storage space than an access control matrix

File	Access				
File 1	USER1 (RWED),	USER2 (R-E-),	USER4 (RWE-),	USER5 (--E-),	WORLD (---)
File 2	USER2 (R-E-),	USER3 (R-E-),	USER4 (--E-),	WORLD (---)	
File 3	USER2 (RWED),	USER4 (--E-),	WORLD (---)		
File 4	USER1 (R-E-),	USER5 (RWED),	WORLD (---)		
File 5	USER5 (RWED),	WORLD (---)			

Some systems shorten the access control list even more by putting every user into a category: system, owner, group, and world. SYSTEM or ADMIN is designated for system personnel who have unlimited access to all files in the system. The OWNER has absolute control over all files created in the owner's account. An owner may create a GROUP file so that all users belonging to the appropriate group have access to it. WORLD is composed of all other users in the system; that is, those who don't fall into any of the other three categories. In this system, the File Manager designates default types of access to all files at creation time, and it's the owner's responsibility to change them as needed.

STET WOMEN'S COLLEGE

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 8.5

Table 8.5 : A capability list shows files for each user and requires less storage space than an access control matrix; and when users are added or deleted from the system, is easier to maintain than an access control list.

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)

Of the three schemes described so far, the most commonly used is the access control list. However, capability lists are gaining in popularity because in operating systems such as Linux or UNIX, they control access to devices as well as to files.

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

STET WOMEN'S COLLEGE