UNIT - I

OPERATING SYSTEM OBJECTIVES AND FUNCTIONS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.

It can be thought of as having three objectives:

- **Convenience :** An OS makes a computer more convenient to use.
- **Efficiency :** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions.

The Operating System as a User/Computer Interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion

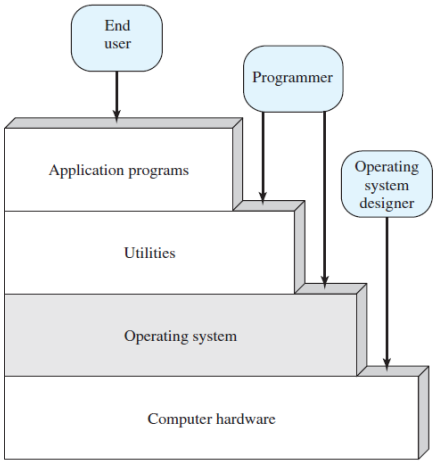


Figure 2.1 **Layers and Views of a Computer System**

The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications. An application can be expressed in a programming language and is developed by an application programmer.

Responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking. To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities. These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. The OS typically provides services in the following areas:

Program development: The OS provides a variety of facilities and services, such as editors.
Access to I/O devices: Each I/O device requires its own peculiar set of instructions
Controlled access to files: For file access, the OS must reflect an understanding of storage medium.

System access: For shared or public systems, the OS controls access to the system as a whole
Error detection and response: A variety of errors can occur while a computer system is running. OS must provide a response that clears the error condition with the least impact on running applications
Accounting: A good OS will collect usage statistics and monitor performance parameters

The Operating System as Resource Manager

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources. By managing the computer's resources, the OS is in control of the computer's basic functions. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs.

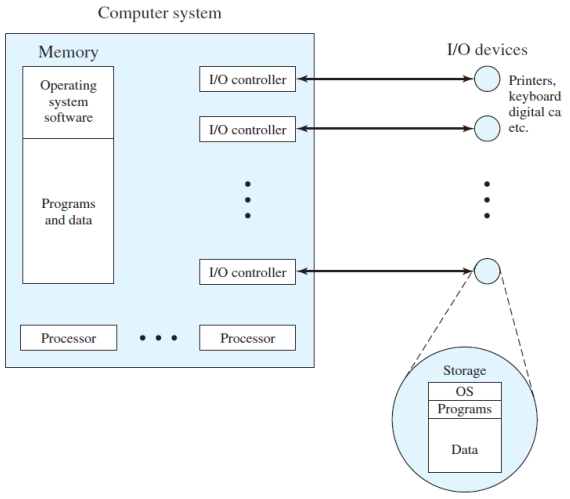


Figure 2.2 The Operating System as Resource Manager

Ease of Evolution of an Operating System

Hardware upgrades plus new types of hardware: For example, early versions of UNIX and the Macintosh operating system did not employ a paging mechanism because they were run on processors without paging hardware.1 Subsequent versions of these operating systems were modified to exploit paging capabilities.
New services: In response to user demand or in response to the needs of system managers, the OS expands to offer new service
Fixes: Any OS has faults. These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults

The need to change an OS regularly places certain requirements on its design. An obvious statement is that the system should be modular in construction, with clearly defined interfaces between the modules, and that it should be well documented.

THE EVOLUTION OF OPERATING SYSTEMS

Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer.

These early systems presented two main problems:

• **Scheduling:** Most installations used a hardcopy sign-up sheet to reserve computer time. Typically, a user could sign up for a block of time in multiples of a half hour or so. **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. This mode of operation could be termed *serial processing*, reflecting the fact that users have access to the computer in series.

Simple Batch Systems

Early computers were very expensive, and therefore it was important to maximize processor utilization. To improve utilization, the concept of a batch operating system was developed. It appears that the first batch operating system (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor**. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor

Monitor point of view: The monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 2.3).That portion is referred to as the **resident monitor**.

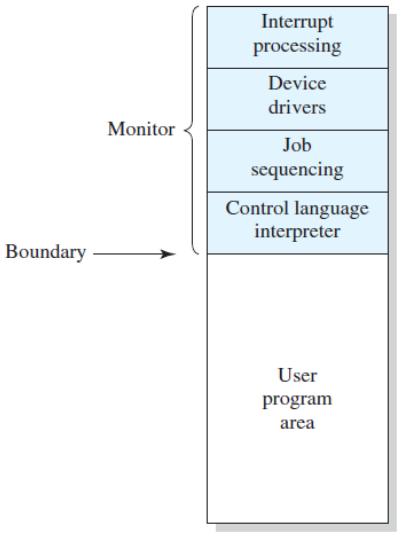


Figure 2.3 Memory Layout for a Resident Monitor

Processor point of view: At a certain point, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read into another portion of main memory.

With each job, instructions are included in a primitive form of **job control language** (JCL). This is a special type of programming language used to provide instructions to the monitor. During the execution of the user program, any input instruction causes one line of data to be read. The input instruction in the user program causes an input routine that is part of the OS to be invoked.

Memory protection: Program is executing it must not alter memory area containing the monitor.
Timer: A timer is used to prevent a single job from monopolizing the system.
Privileged instructions: Machine level instructions are designated privileged
Interrupts :OS more flexibility in relinquishing control and regaining control from user programs.

Considerations of memory protection and privileged instructions lead to the concept of modes of operation. A user program executes in a **user mode**, in which certain areas of memory are protected from the user's use. The monitor executes in a system mode, or what has come to be called **kernel mode**, in which privileged instructions may be executed and in which protected areas of memory may be accessed.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

Percent CPU Utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 2.4 System Utilization Example

Multiprogrammed Batch Systems
Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor.

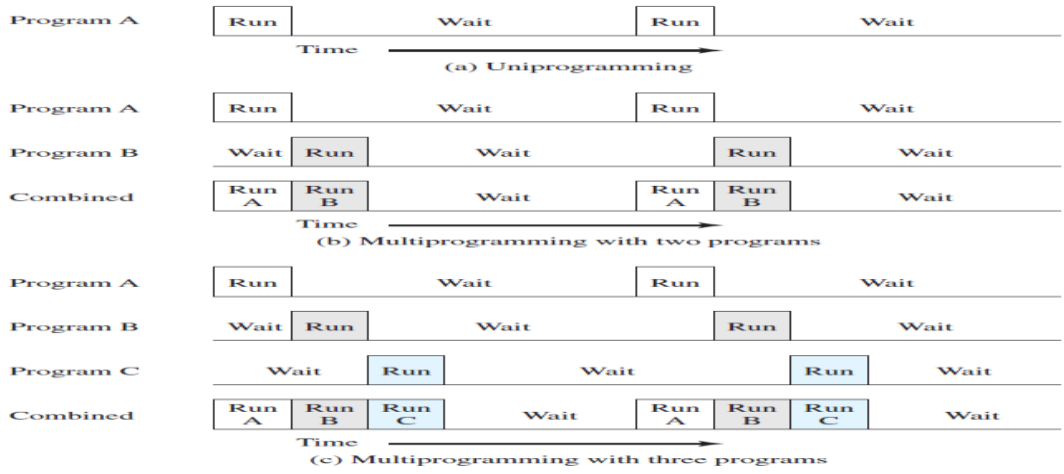


Figure 2.5 Multiprogramming Example

Where we have a single program, referred to as uniprogramming. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O , the approach is known as **multiprogramming**, or **multitasking**.

Table 2.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

To have several jobs ready to run, they must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as **time sharing**, because processor time is shared among multiple users.

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS) [CORB62], developed at MIT by a group known as Project MAC (Machine-Aided Cognition, or Multiple-Access Computers). At each clock interrupt, the OS regained control and could assign the processor to another user. This technique is known as **time slicing**.**STE**

To minimize disk traffic, user memory was only written out when the incoming program would overwrite it. This principle is illustrated in Figure 2.7. Assume that there are four interactive users with the following memory requirements, in words:

• JOB1: 15,000
• JOB2: 20,000
• JOB3: 5000
• JOB4: 10,000

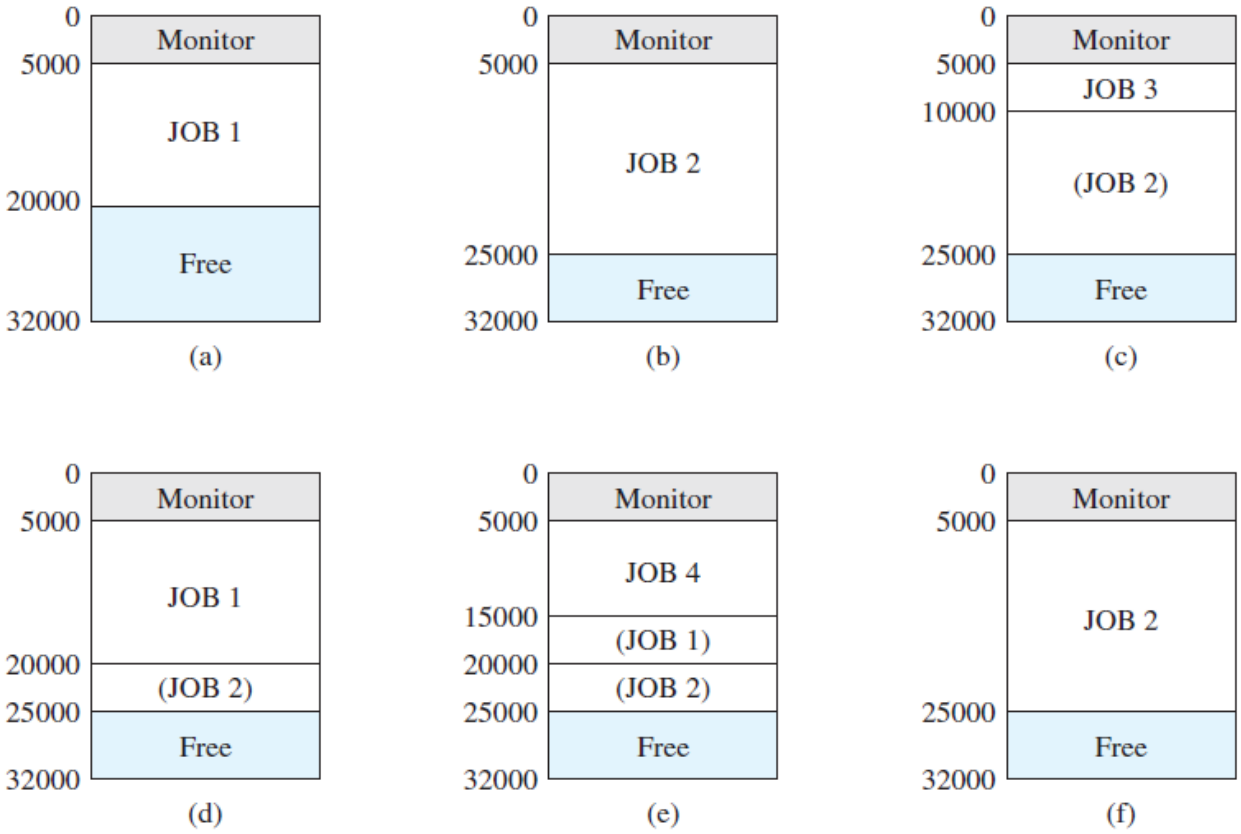


Figure 2.7 CTSS Operation

The CTSS approach is primitive compared to present-day time sharing.

- Initially, the monitor loads JOB1 and transfers control to it (a).
- Later, the monitor decides to transfer control to JOB2. Because JOB2 requires more memory than JOB1, JOB1 must be written out first, and then JOB2 can be loaded (b).
- Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of JOB2 can remain in memory, reducing disk write time (c).
- Later, the monitor decides to transfer control back to JOB1.An additional portion of JOB2 must be written out when JOB1 is loaded back into memory (d).
- When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (e).
- At this point, if eitherJOB1 or JOB2 is activated, only a partial load will be required. In this example, it is JOB2 that runs next. This requires that JOB4 and the remaining resident portion of JOB1 be written out and that the missing portion of JOB2 be read in (f).

WHAT IS A PROCESS?

A computer platform consists of a collection of hardware resources Computer applications are developed to perform some task It is inefficient for applications to be written directly for a given hardware platform. The OS was developed to provide a convenient, feature-rich, secure, and consistent interface for applications to use. We can think of the OS as providing a uniform, abstract representation of resources.

PROCESSES AND PROCESS CONTROL BLOCKS

Suggested several definitions of the term *process*, including
• A program in execution
• An instance of a program running on a computer
• The entity that can be assigned to and executed on a processor
• A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.

Two essential elements of a process are **program code** (which may be shared with other processes that are executing the same program) and a **set of data** associated with that code. Process can be uniquely characterized by a number of elements, including the following:

• **Identifier:** A unique identifier associated with this process to distinguish it from other processes.
• **State:** If the process is currently executing, it is in the running state.
• **Priority:** Priority level relative to other processes.
• **Program counter:** The address of the next instruction in the program to be executed.
• **Memory pointers:** Includes pointers to the program code and data associated with this process,
• **Context data:** Data that are present in registers in the processor while the process is executing.
• **I/O status information:** Includes outstanding I/O requests, I/O devices .
• **Accounting information:** It include the amount of processor time and clock time used and so on.

The information in the preceding list is stored in a data structure, typically called a **process control block that** is created and managed by the OS.

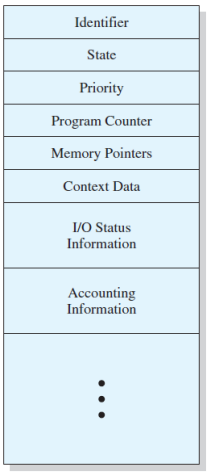


Figure 3.1 **Simplified Process Control Block**

The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block

PROCESS STATES

From the processor's point of view, it executes instructions from its repertoire in some sequence dictated by the changing values in the program counter register. We can characterize the behavior of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a **trace** of the process. In addition, there is a small **dispatcher** program that switches the processor from one process to another.

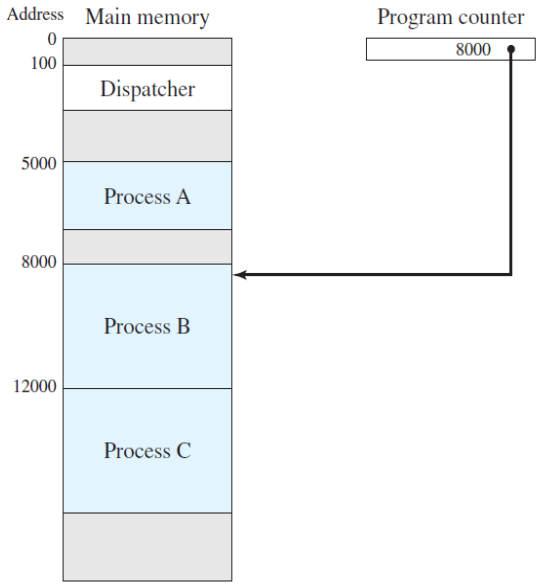


Figure 3.2 **Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

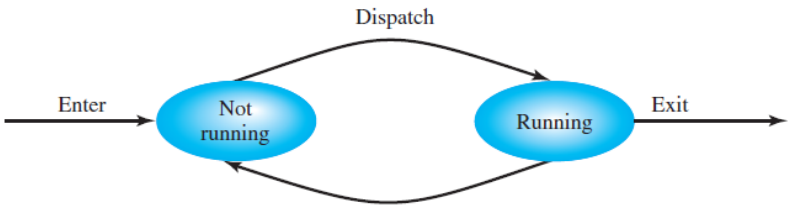
5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 **Traces of Processes of Figure 3.2**

In the figure3.2 the **shaded areas** represent code executed by the dispatcher. The same sequence of instructions is executed by the dispatcher in each instance because the same functionality of the dispatcher is being executed. We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles the first six instructions of process A are executed, followed by a time-out and the execution of some code in the dispatcher. 2After four instructions are executed, process B requests an I/O action for which it must wait. Therefore, the processor stops executing process B and moves on, via the dispatcher, to process C

A Two-State Process Model

The operating system's principal responsibility is controlling the execution of processes. This includes determining the interleaving pattern for execution and allocating resources to processes. We can construct the simplest possible model by observing that, at any time, a process is either being executed by a processor or not. In this model, a process may be in one of two states: Running or Not Running, as shown in below Figure.



(a) State transition diagram

When the OS creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state. The process exists, is known to the OS, and is waiting for an opportunity to execute. From this simple model Each process must be represented in some way so that the OS can keep track of it.

The Creation and Termination of Processes

The life of a process is bounded by its creation and termination. **Process Creation** When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.

Table 3.1 Reasons for Process Creation

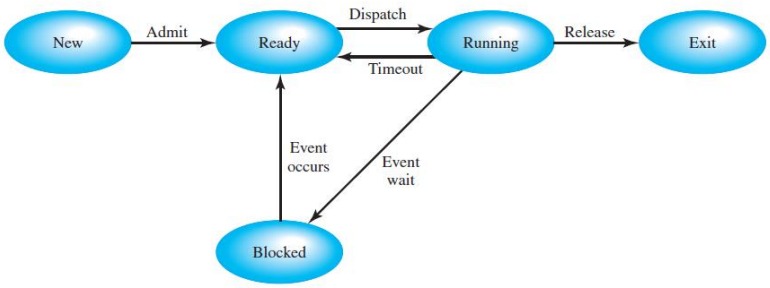
New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

When the OS creates a process at the explicit request of another process, the action is referred to as **process spawning**. When one process spawns another, the former is referred to as the **parent process**, and the spawned process is referred to as the **child process**.

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Process Termination : The above table summarizes typical reasons for process termination.Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instruction or an explicit OS service call for termination. some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. To handle this situation is to split the Not Running state into two states: Ready and



Blocked. **Figure 3.6 Five-State Process Model**

In above figure, we have added two additional states that will prove useful. The five states in this
new diagram are as follows:

• **Running:** The process that is currently being executed
• **Ready:** A process that is prepared to execute when given the opportunity.

• **Blocked/Waiting:** A process that cannot execute until some event occurs.
• **New:** A process that has just been created but has not yet been admitted to the pool
• **Exit:** A process that has been released from the pool of executable processes

An identifier is associated with the process. Any tables that will be needed to manage the process are allocated and built. At this point, the process is in the New state. the types of events that lead to each state transition for a process. The possible transitions are as follows:
• **Null →New:** A new process is created to execute a program.
• **New →Ready:** The OS □□will move a process from the New state to the Ready state
Ready → Running: The OS chooses one of the processes in the Ready state
Running →Exit: The currently running process is terminated by the OS
Running → Ready: Running process has reached the maximum allowable time
Running →Blocked: A process is put in the Blocked state if it requests something
Blocked →Ready: A process in the Blocked state is moved to the Ready state
Ready →Exit: For clarity, this transition is not shown on the state diagram. In some systems, a parent may terminate a child process at any time access is put in the Blocked state.

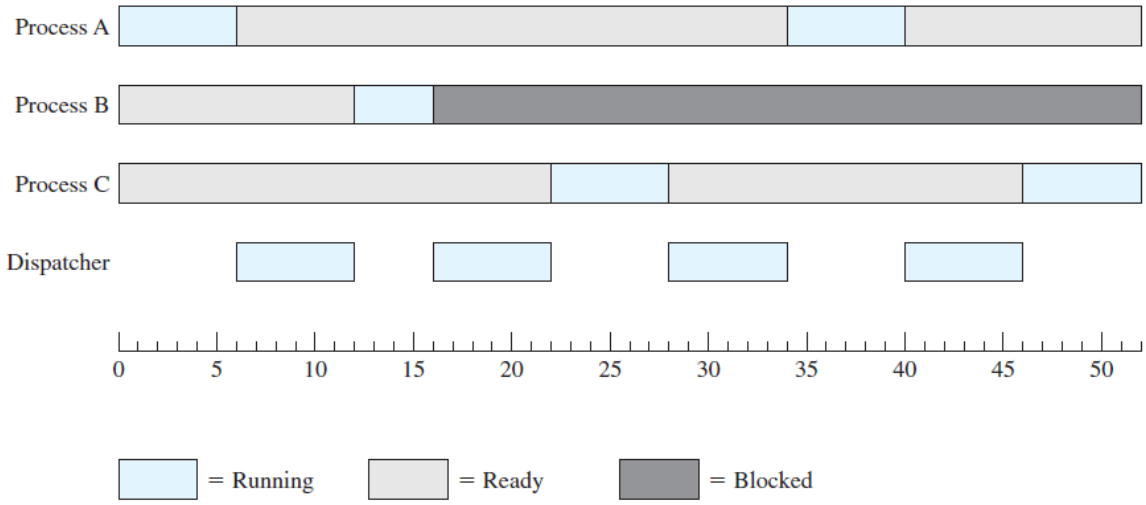


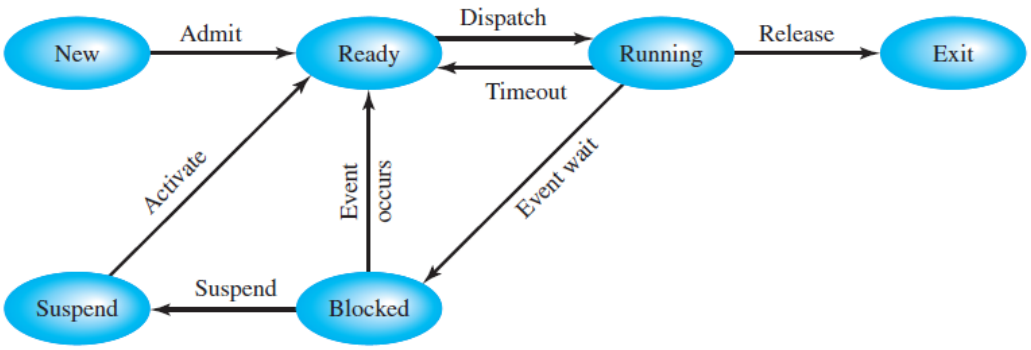
Figure 3.7 Process States for the Trace of Figure 3.4

Suspended Processes

The Need for Swapping

The three principal states just described (Ready,Running,Blocked) provide a systematic way of modeling the behavior of processes the processor is so much faster than I/O that it will be common for all of the processes in memory to be waiting for I/O. Each process to be executed must be loaded fully into main memory. Main memory could be expanded to accommodate more processes.

Another solution is swapping, which involves moving part or all of a process from main memory to disk. When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out onto disk into a suspend queue. With the use of swapping as just described, one other state must be added to our process behavior model : **the Suspend state**



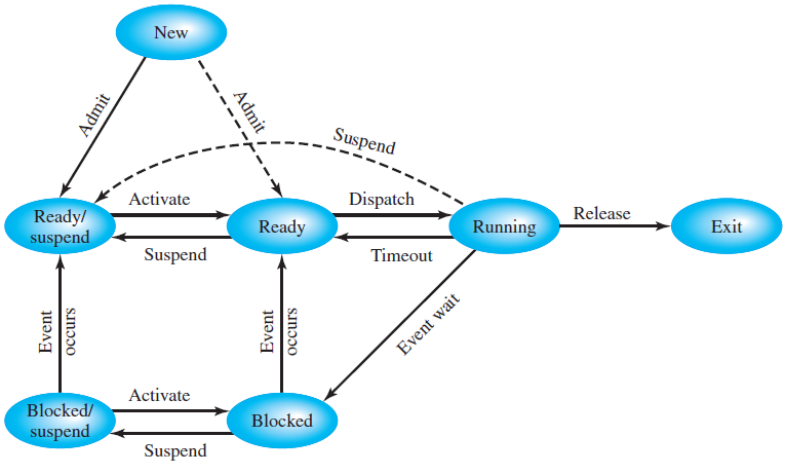
(a) With one suspend state

we need to rethink this aspect of the design. There are two independent concepts here:

1. whether a process is waiting on an event (blocked or not)
2. whether a process has been swapped out of main memory (suspended or not).

To accommodate this 2 * 2 combination, we need four states:

• **Ready:** The process is in main memory and available for execution.
• **Blocked:** The process is in main memory and awaiting an event.
• **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
• **Ready/Suspend:** The process is in secondary memory



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

In above figure 3.9 b, at the state transition model that we havedeveloped. (The dashed lines in the figure indicate possible but not necessary transitions.) Important new transitions are the following:

Blocked →Blocked/Suspend

If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked.

Blocked/Suspend →Ready/Suspend:

A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs.

Ready/Suspend → Ready

When there are no ready processes in main memory, the OS will need to bring.

Ready →Ready/Suspend

One in to continue execution the OS may choose to suspend a lower-priority ready process rather than a higher priority blocked process

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Suspended process as having the following characteristics:

1. The process is not immediately available for execution.
2. The process may or may not be waiting on an event. If it is, this blocked condition is independent of the suspend condition, and occurrence of the blocking event does not enable the process to be executed immediately.
3. The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution.
4. The process may not be removed from this state until the agent explicitly orders the removal.

.

PROCESS DESCRIPTION

The OS controls events within the computer system. It schedules and dispatches processes for execution by the processor, allocates resources to processes, and responds to requests by user processes for basic services.

Operating System Control Structures

If the OS is to manage processes and resources, it must have information about the current status of each process and resource. The universal approach to providing this information is straightforward: The OS constructs and maintains tables of information about each entity that it is managing. A general idea of the scope of this effort is indicated in Figure 3.11, which shows four different types of tables maintained
by the OS: memory, I/O, file, and process

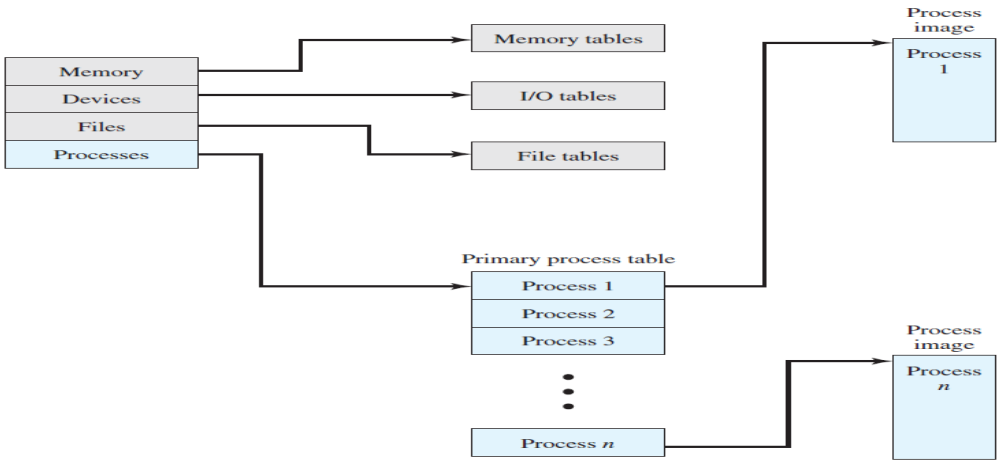


Figure 3.11 **General Structure of Operating System Control Tables**

Memory tables are used to keep track of both main (real) and secondary (virtual) memory.

The memory tables must include the following information:

- The allocation of main memory to processes
- The allocation of secondary memory to processes
- Any protection attributes of blocks of main or virtual memory, such as which processes may access certain shared memory regions Any information needed to manage virtual memory

I/O tables are used by the OS to manage the I/O devices and channels of the

- computer system.At any given time, an I/O device may be available or assigned to
- a particular process. If an I/O operation is in progress, the OS needs to know the
- status of the I/O operation and the location in main memory being used as the
- source or destination of the I/O transfer

The OS may also maintain **file tables**.These tables provide information about

- The existence of files, their location on secondary memory, their current status, and other attributes.

Finally, the OS must maintain **process tables** to manage processes. The remainder of this section is devoted to an examination of the required process tables. Memory, I/O, and files are managed on behalf of processes, so there must be some reference to these resources, directly or indirectly, in the process tables.

Process Control Structures

The OS must know if it is to manage and control a process First, it must know where the process is located, and second, it must know the attributes of the process that are necessary for its management (e.g., process ID and process state).

Process Location

Before we can deal with the questions of where a process is located or what its attributes are, we need to address an even more fundamental question: What is the physical manifestation of a process? At a minimum, a process must include a program or set of programs to be executed. Associated with these programs is a set of data locations for local and global variables and any defined constants.

Finally, each process has associated with it a number of attributes that are used by the OS for process control. Typically, the collection of attributes is referred to as a **process control block**. We can refer to this collection of program, data, stack, and attributes as the **process image.**

Table 3.5 Typical Elements of a Process Control Block

Process Identification

Identifiers
Numeric identifiers that may be stored with the process control block include
- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers
A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers
These are a variety of processor registers that are employed to control the operation of the processor. These include
- *Program counter:* Contains the address of the next instruction to be fetched
- *Condition codes:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- *Status information:* Includes interrupt enabled/disabled flags, execution mode

Stack Pointers
Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Information

Scheduling and State Information
This is information that is needed by the operating system to perform its scheduling function. Typical items of information:
- *Process state:* Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- *Priority:* One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- *Scheduling-related information:* This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- *Event:* Identity of event the process is awaiting before it can be resumed.

Data Structuring
A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication
Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges
Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process Attributes

A sophisticated multiprogramming system requires a great deal of information about each process. As was explained, this information can be considered to reside in a process control block.

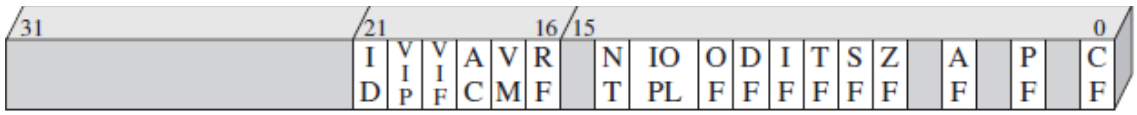
We can group the process control block information into three general categories:

1. Process identification
2. Processor state information
3. Process control information

With respect to **process identification**, in virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table. otherwise there must be a mapping that allows the OS to locate the

appropriate tables based on the process identifier. When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication. When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process.

Processor state information consists of the contents of processor registers. While a process is running, of course, the information is in the registers. When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution.
all processor designs include a register or set of registers, often known as the program status word (PSW), that contains status information. The PSW typically contain condition codes plus other status information. A good example of a processor status word is that on Pentium processors, referred to as the EFLAGS register.



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Figure 3.12 Pentium II EFLAGS Register

The third major category of information in the process control block can be called, for want of a better name, **process control information**. This is the additional information needed by the OS to control and coordinate the various active processes. In the below Figure suggests the structure of process images in virtual memory. Each process image consists of a process control block, a user stack, the private address space.

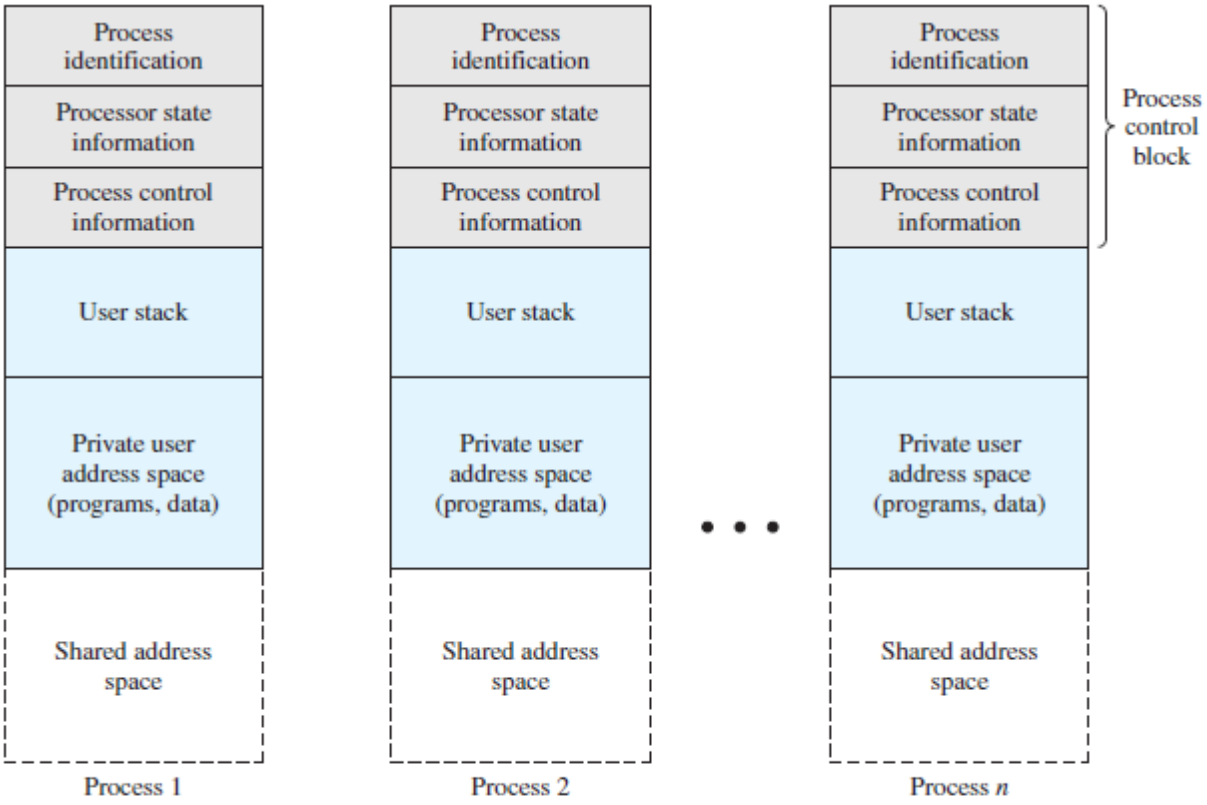


Figure 3.13 User Processes in Virtual Memory

The Role of the Process Control Block

The process control block is the most important data structure in an OS.Each process control block contains all of the information about a process that is needed by the OS. The blocks are read and/or modified by virtually every module in the OS, including those involved with scheduling, resource allocation, interrupt processing, and performance monitoring and Analysis.

PROCESS CONTROL

Modes of Execution

we need to distinguish between the mode of processor execution normally associated with the OS and that normally associated with user programs. Most processors support at least two modes of execution.

Certain instructions can only be executed in the more-privileged mode. The less-privileged mode is often referred to as the **user mode**, because user programs typically would execute in this mode. The more-privileged mode is referred to as the **system mode**, **control mode**, or **kernel mode**

Table 3.7 Typical Functions of an Operating System Kernel

Process Management
• Process creation and termination
• Process scheduling and dispatching
• Process switching
• Process synchronization and support for interprocess communication
• Management of process control blocks
Memory Management
• Allocation of address space to processes
• Swapping
• Page and segment management
I/O Management
• Buffer management
• Allocation of I/O channels and devices to processes
Support Functions
• Interrupt handling
• Accounting
• Monitoring

There is a bit in the program status word (PSW) that indicates the mode of execution. This bit is changed in response to certain events. Typically, when a user makes a call to an operating system service or when an interrupt triggers execution of an operating system routine, the mode is set to the kernel mode and, upon return from the service to the user process, the mode is set to user mode.

Process Creation

To create a new process, OS an proceed as follows:

1. **Assign a unique process identifier to the new process.** At this time, a new entry is added to the primary process table, which contains one entry per process.
2. **Allocate space for the process.** This includes all elements of the process image. Thus, the OS must know how much space is needed for the private user address space (programs and data) and the user stack.
3. **Initialize the process control block.** The process identification portion contains the ID of this process plus other appropriate IDs, such as that of the parent process.
4. **Set the appropriate linkages.** For example, if the OS maintains each scheduling queue as a linked list, then the new process must be put in the Ready or Ready/Suspend list.
5. **Create or expand other data structures.** For example, the OS may maintain an accounting file on each process to be used subsequently for billing and/or performance assessment purposes

Process Switching

The function of process switching would seem to be straightforward. At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process.

When to Switch Processes

A process switch may occur any time that the OS has gained control from the currently running process. Table 3.8 suggests the possible events that may give control to the OS.

Table 3.8 Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

With an ordinary **interrupt**, control is first transferred to an interrupt handler, which does some basic housekeeping and then branches to an OS routine **Clock interrupt:** The OS determines whether the currently running process has been executing for the maximum allowable unit of time, referred to as a **time slice**. That is, a time slice is the maximum amount of time that a process
can execute before being interrupted.

Interrupt: The OS determines what I/O action has occurred. If the I/O action constitutes an event for which one or more processes are waiting, then the OS moves all of the corresponding blocked processes to the Ready state

Memory fault: The processor encounters a virtual memory address reference for a word that is not in main memory. With a **trap**, the OS determines if the error or exception condition is fatal. If so,
then the currently running process is moved to the Exit state and a process switch occurs.

Finally, the OS may be activated by a **supervisor call** from the program being executed. For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open.

Mode Switching : Change of Process State

It is clear, then, that the mode switch is a concept distinct from that of the process switch.10 A mode switch may occur without changing the state of the process that is currently in the Running state. In that case, the context saving and subsequent restoral involve little overhead.

The OS must make substantial changes in its environment. The steps involved in a
full process switch are as follows:

1. Save the context of the processor, including program counter and other registers.
2. Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states.

3. Move the process control block of this process to the appropriate queue
4. Select another process for execution
5. Update the process control block of the process selected. This includes changing the state of this process to Running.
6. Update memory management data structures. This may be required, depending on how address translation is managed.

Restore the context of the processor to that which existed at the time the selected process was last switched out of the Running state Thus, the process switch, which involves a state change, requires more effort than a mode switch.

UNIT – II

PROCESSES AND THREADS

Multithreading: *Multithreading* refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

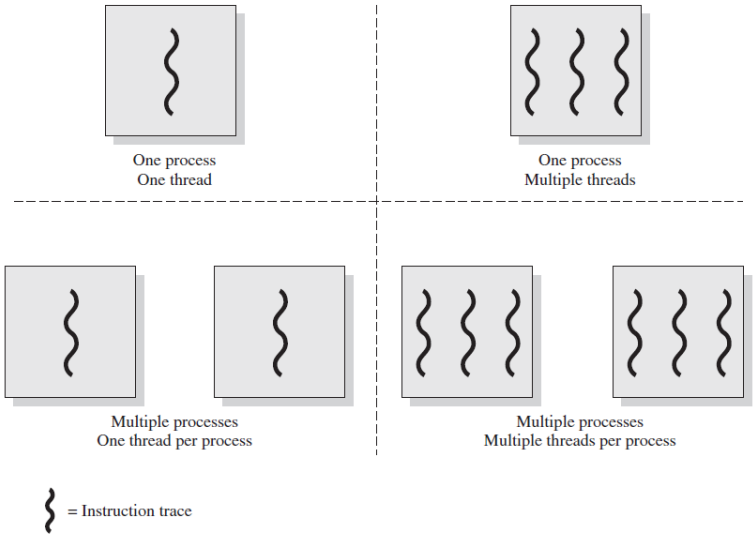


Figure 4.1 **Threads and Processes [ANDE97]**

The two arrangements shown in the left half of Figure 4.1 are single-threaded approaches. MS-DOS is an example of an OS that supports a single user process and a single thread. Other operating systems, such as some variants of UNIX, support multiple user processes but only support one thread per process. The right half of Figure 4.1 depicts multithreaded approaches. A Java run-time environment is an example of a system of one process with multiple threads. In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection.

The following are associated with processes:

• A virtual address space that holds the process image

• Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels)

Within a process, there may be one or more threads, each with the following:

• A thread execution state (Running, Ready, etc.).

• A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.

• An execution stacks some per-thread static storage for local variables.

• Access to the memory and resources of its process, shared with all other threads in that process.

The distinction between threads and processes from the point of view of process management is in a single-threaded process model (i.e., there is no distinct concept of thread), the representation of a process includes its process control block and user address

space, as well as user and kernel stacks to manage the call/return behavior of the execution of the process.

In a multithreaded environment, there is still a single process control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread-related state information.

Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data. When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

The key benefits of threads derive from the performance implications:

1. It takes far less time to create a new thread in an existing process than to create a brand-new process.
2. It takes less time to terminate a thread than a process.
3. It takes less time to switch between two threads within the same process than to switch between processes.
4. Threads enhance efficiency in communication between different executing Programs

Thread Functionality

Like processes, threads have execution states and may synchronize with one another.

Thread States: As with processes, the key states for a thread are Running, Ready, and Blocked.

There are **four basic thread operations** associated with a change in **thread state**

- **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned.
- **Block :** When a thread needs to wait for an event, it will block
- **Unblock :** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
- **Finish :** When a thread completes, its register context and stacks are deallocated

Thread Synchronization: All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures.

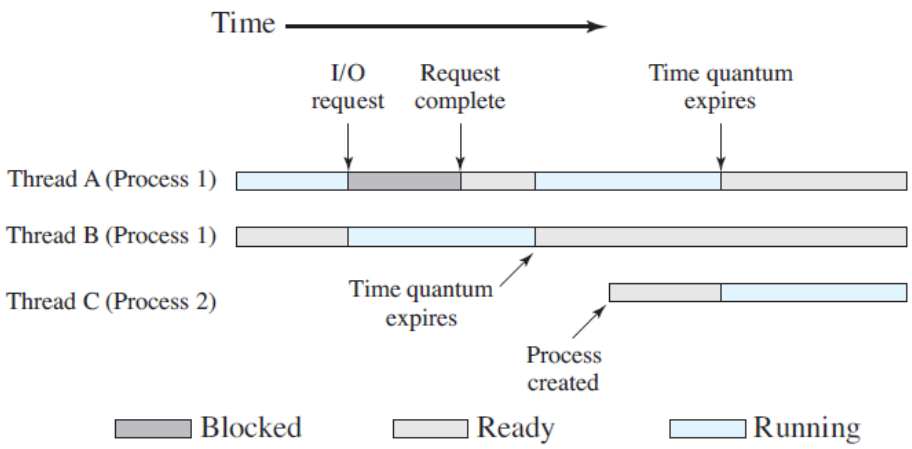


Figure 4.4 Multithreading Example on a Uniprocessor

Example—Adobe PageMaker4

An example of the use of threads is the Adobe PageMaker application running under a shared system. PageMaker is a writing, design, and production tool for desktop publishing. The thread structure for PageMaker used in the operating system OS/2, shown in Figure 4.5

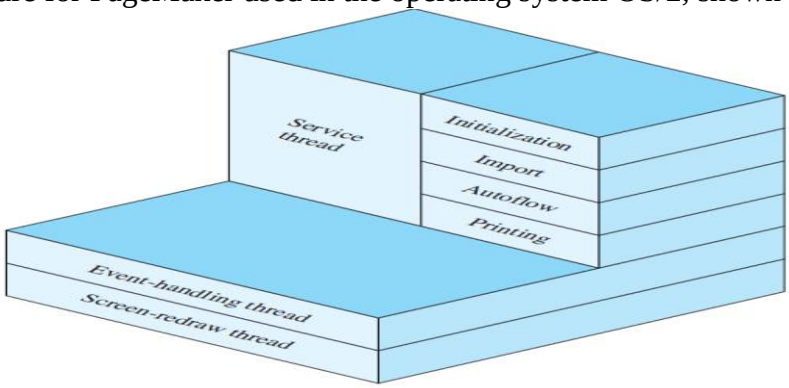


Figure 4.5 Thread Structure for Adobe PageMaker

Three threads are always active: an event-handling thread, a screen-redraw thread, and a service thread. For example, calling a subroutine to print a page while processing a print command would prevent the system from dispatching any further message to any applications, slowing performance.

To meet this criterion, time-consuming user operations in PageMaker printing, importing data, and flowing text are performed by a service thread. Synchronizing the service thread and event-handling thread is complicated because a user may continue to type or move the mouse, which activates the event handling thread, while the service thread is still busy. The service thread sends a message to the event-handling thread to indicate completion of its task. Until this occurs, user activity in PageMaker is restricted.

The screen redraw function is handled by a separate thread. This is done for two reasons:

PageMaker does not limit the number of objects appearing on a page. Using a separate thread allows the user to abort drawing. In this case, when the user rescales a page, the redraw can proceed immediately. Dynamic scrolling redrawing the screen as the user drags the scroll indicator is also possible. The event-handling thread monitors the scroll bar and redraws the margin rulers.

User-Level and Kernel-Level Threads

There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs).

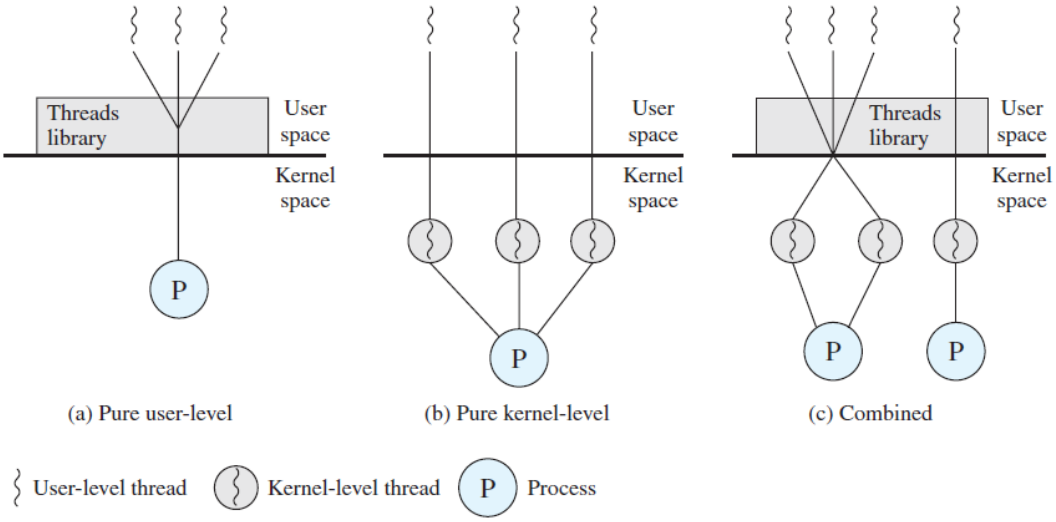


Figure 4.6 User-Level and Kernel-Level Threads

User-Level Threads: In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. Figure 4.6a illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

There are a number of advantages to the use of ULTs instead of KLTs, including the following:

1. Thread switching does not require kernel mode privileges because all of the thread management data structures are within the user address space of a single process. This saves the overhead of two mode switches (user to kernel; kernel back to user).

2. Scheduling can be application specific.One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm.The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.

3. ULTs can run on any OS. No changes are required to the underlying kernel to support ULTs.The threads library is a set of application-level functions shared by all applications.

There are two distinct disadvantages of ULTs compared to KLTs:

1. In a typical OS, many system calls are blocking. As a result, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked

2. In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time.

Therefore, only a single thread within a process can execute at a time. In effect, we have application-level multiprogramming within a single process. A way to overcome the problem of blocking threads is to use a technique referred to as **jacketing**. The purpose of jacketing is to convert a blocking system call into a non blocking system call. Within this jacket routine is code that checks to determine if the I/O device is busy. If it is, the thread enters the Blocked state and passes control (through the threads library) to another thread. When this thread later is given control again, the jacket routine checks the I/O device again.

Kernel-Level Threads : In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility. Windows is an example of this approach.

The kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the kernel is done on a thread basis. This approach overcomes the two principal drawbacks of the ULT approach. First, the kernel can simultaneously schedule multiple threads from the same process on multiple processors. Second, if one thread in a process is blocked, the kernel can schedule another thread of the same process.

Another advantage of the KLT approach is that kernel routines themselves can be multithreaded. The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combined Approaches: Some operating systems provide a combined ULT/KLT Facility. In a combined system, thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best overall results. Recently, there has been much interest in providing for multiple threads within a single process, which is a many-to-one relationship.

Table 4.2 Relationship between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

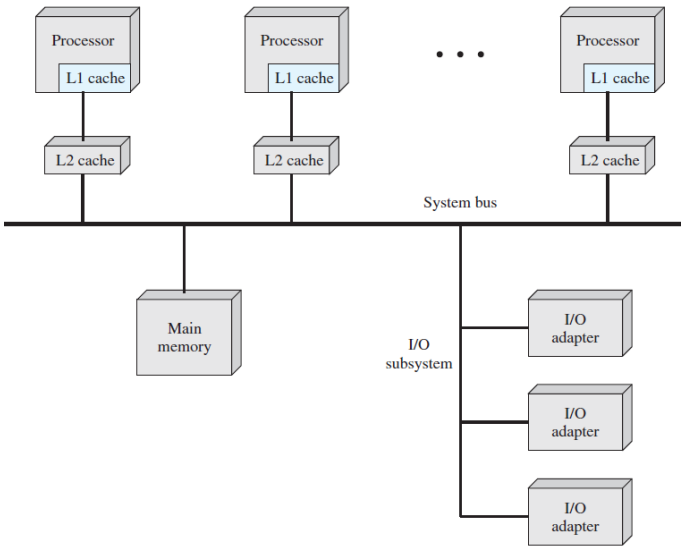
As with the multithreading approaches discussed so far, multiple threads may execute in a single domain, providing the efficiency it is also possible for a single user activity, or application, to be performed in multiple domains. In this case, a thread exists that can move from one domain to another. In a multiprogramming environment that allows user-spawned processes, the main program could generate a new process to handle I/O and then continue to execute.

SYMMETRIC MULTIPROCESSING

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism.

SMP Organization

The general organization of an SMP, There are multiple processors, each of which contains its own control unit, arithmetic-logic unit, and registers.



The two most popular approaches to providing parallelism by replicating processors: symmetric multiprocessors (SMPs) and clusters.

SMP Architecture

It is useful to see where SMP architectures fit into the overall category of parallel Processors.

Flynn proposed the following categories of computer systems:

• **Single instruction single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory.
• **Single instruction multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis.
Multiple instruction single data (MISD) stream: A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence.
Multiple instruction multiple data (MIMD) stream: A set of processors simultaneously execute different instruction sequences on different data sets.
In a **symmetric multiprocessor (SMP)**, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. It must ensure that two processors do not choose the same process and that processes are not somehow lost from the queue.

Symmetric Multiprocessor Organization

Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in shared address spaces).

Multiprocessor Operating System Design Considerations

An SMP operating system manages processor and other computer resources so that the user may view the system in the same fashion as a multiprogramming uniprocessor system. Thus a multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors.

s**imultaneous concurrent processes or threads:** Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously.

Scheduling: Scheduling may be performed by any processor, so conflicts must be avoided.

Synchronization: With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization.

Memory management: Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers.

Reliability and fault tolerance: The OS should provide graceful degradation in the face of processor failure.

PRINCIPLES OF CONCURRENCY
In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. In a multiple-processor system, it is possible not only to interleave the execution of multiple processes but also to overlap them

The relative speed of execution of processes cannot be predicted. It depends on the activities of other processes, the way in which the OS handles interrupts, and the scheduling policies of the OS.
The following difficulties arise:

1. The sharing of global resources is fraught with peril.
2. It is difficult for the OS to manage the allocation of resources optimally
3. It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. As a first example, suppose that two processes, P1 and P2, share the global variable a. At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2.Thus, the two tasks are in a race to write variable a For our second example, consider two process, P3 and P4, that share global variables b and c, with initial values b = 1 and c = 2. At some point in its execution, P3 executes the assignment b = b + c, and at some point in its execution, P4 executes the assignment c = b + c. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments.

Operating System Concerns

List the following concerns:
1. The OS must be able to keep track of the various processes
2. The OS must allocate and deallocate various resources for each active process. At times, multiple processes want access to the same resource,
3. The OS must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes

Process Interaction
We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence.

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Processes unaware of each other: These are independent processes that are not intended to work together.
Processes indirectly aware of each other: These are processes that are not necessarily aware of each other by their respective process IDs
Processes directly aware of each other: These are processes that are able to communicate with each other by process ID
Competition among Processes for Resources Concurrent processes come into conflict with each other when they are competing for the use of the same resource. Each process is unaware of the existence of other processes, and each is to be unaffected by the execution of the other processes.
In particular, if two processes both wish access to a single resource, then one process will be allocated that resource by the OS, and the other will have to wait. Therefore, the process that is denied access will be slowed down. In the case of competing processes three control problems must be faced. First is the need for **mutual exclusion.**

Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a **critical resource**, and the portion of the program that uses it a **critical section** of the program. The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock,** the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

A final control problem is **starvation,** Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical

section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Cooperation among Processes by Sharing : The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data.Thus the processes must cooperate to ensure that the data they share
are properly managed.

Cooperation among Processes by Communication : Typically, communication can be characterized as consisting of messages of some sort. Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel. Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation. However, the problems of deadlock and starvation are still present. As an example of deadlock, two processes may be blocked, each waiting for a communication from theother.

Requirements for Mutual Exclusion

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.

2. A process that halts in its noncritical section must do so without interfering with other processes.

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.

4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

5. No assumptions are made about relative process speeds or number of processors.

6. A process remains inside its critical section for a finite time only.

MUTUAL EXCLUSION: HARDWARE SUPPORT

A number of software algorithms for enforcing mutual exclusion have been developed.

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution. They can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted.Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts.

Special Machine Instructions

In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors

behave independently in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based. **Compare&Swap Instruction** The compare&swap instruction, also called a compare and exchange instruction. A **compare** is made between a memory value and a test value; if the values differ a **swap** occurs.The entire compare&swap function is carried out atomically; that is, it is not subject to interruption. Another version of this instruction returns a Boolean value: true if the swap occurred false otherwise.

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

(a) Compare and swap instruction

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

a mutual exclusion protocol based on the use of this instruction. A shared variable bolt is initialized to 0. The only process that may enter its critical section is one that finds bolt equal to 0. All other processes at enter their critical section go into a busy waiting mode.

The term **busy waiting**, or **spin waiting**, refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance. When a process leaves its critical section, it resets *bolt* to 0; at this point one and only one of the waiting processes is granted access to its critical section. The choice of process depends on which process happens to execute the compare&swap instruction next.

Exchange Instruction : The exchange instruction can be defined as the instruction exchanges the contents of a register with that of a memory location. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction. A shared variable *bolt* is initialized to 0. Each process uses a local variable *key* that is initialized to 1.The only process that may enter its critical section is one that finds *bolt* equal to 0. It excludes all other processes from the critical section by setting *bolt* to 1.When a process leaves its critical section, it resets *bolt* to 0, allowing another process to gain access to its critical section. If *bolt* = 0, then no process is in its critical section. If *bolt* = 1, then exactly one process is in its critical section, namely the process whose *key* value equals 0.

Properties of the Machine-Instruction Approach The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

• It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
• It is simple and therefore easy to verify.
• It can be used to support multiple critical sections; each critical section can be defined by its own variable.
There are some serious disadvantages:

• **Busy waiting is employed.** Thus, while a process is waiting for access to a critical section, it continues to consume processor time.
• **Starvation is possible.** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus,some process could indefinitely be denied access.
• **Deadlock is possible.** Consider the following scenario on a single-processor system. Process P1 executes the special instruction (e.g., compare&swap,exchange) and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the
same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2. Because of the drawbacks of both the software and hardware solutions just outlined, we need to look for other mechanisms.

SEMAPHORES

Semaphore : An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place
until it has received a specific signal.

To transmit a signal via semaphore s, a process executes the primitive semSignal(s).To receive a signal via semaphore s, a process executes the primitive semWait(s); if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:
1. A semaphore may be initialized to a nonnegative integer value.
2. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
3. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.
The semWait and semSignal primitives are assumed to be atomic. A more restricted version, known as the **binary semaphore**.

1. A binary semaphore may only take on the values 0 and 1 and can be defined by the following
   three operations:
   A binary semaphore may be initialized to 0 or 1.

```
struct semaphore {
int count;
queueType queue;
};
void semWait(semaphore s)
{
s.count--;
if (s.count < 0) {
/* place this process in s.queue */;
/* block this process */;
}
}
void semSignal(semaphore s)
{
s.count++;
if (s.count <= 0) {
/* remove a process P from s.queue */;
/* place process P on ready list */;
}
}
```

A Definition of Semaphore Primitives

2. The semWaitB operation checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals zero). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

To contrast the two types of semaphores, the nonbinary semaphore is often referred to as either a **counting semaphore** or a **general semaphore**. A concept related to the binary semaphore is the **mutex**. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

The fairest removal policy is first-in-first-out (FIFO):The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore**. A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore.**

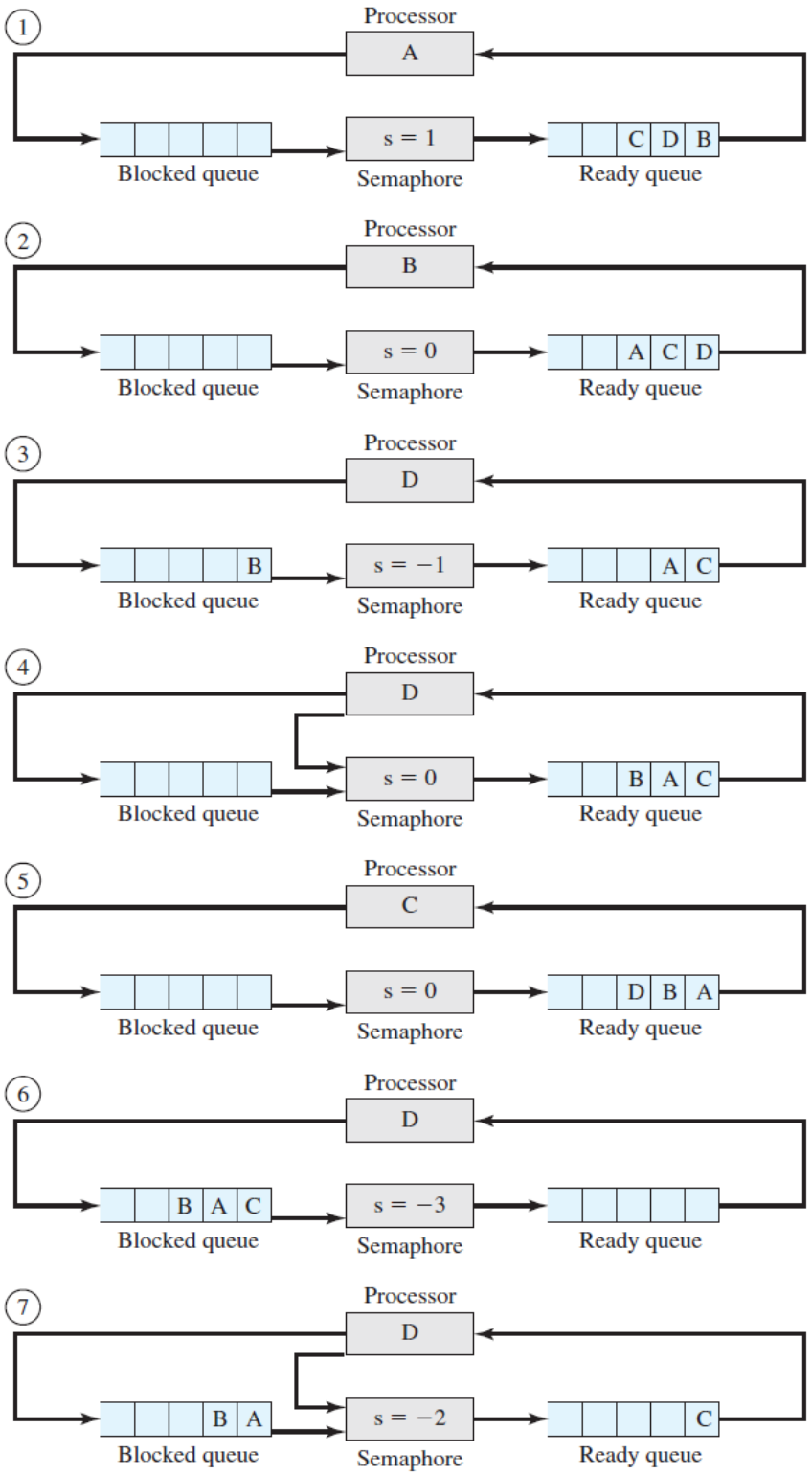


Figure 5.5 **Example of Semaphore Mechanism**

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
while (true) {
semWait(s);
/* critical section */;
semSignal(s);
/* remainder */;
}
}
void main()
{
parbegin (P(1), P(2), . . ., P(n));
}
```

Mutual Exclusion Using Semaphores

Mutual Exclusion

The semaphore is initialized to 1. Thus, the first process that executes a semWait will be able to enter the critical section immediately, setting the value of s to 0. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of s to -1. Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of s.When the process that initially entered its critical section departs, s is incremented and one of the blocked processes (if any) is removed from the queue of blocked
processes associated with the semaphore and put in a Ready state.When it is next scheduled by the OS, it may enter the critical section.

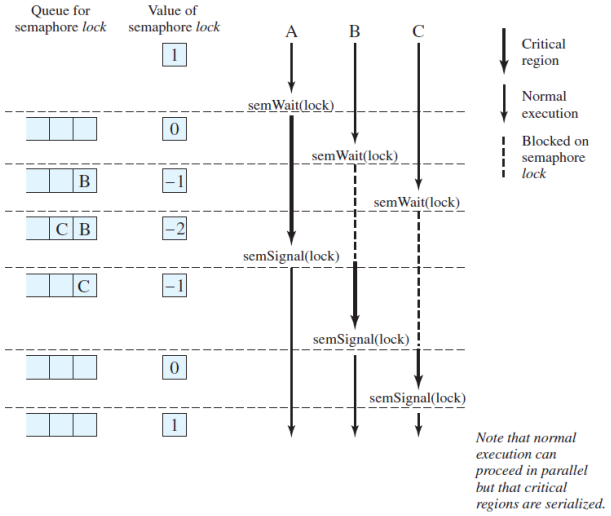


Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

The Producer/Consumer Problem

We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem. The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. The buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

```
producer:
while (true) {
   /* produce item v */;
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /* do nothing */;
   w = b[out];
   out++;
   /* consume item w */;
}
```

Implementation of Semaphores

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

Two Possible Implementations of Semaphores

Figure 5.14a shows the use of a compare & swap instruction.

In this implementation, the semaphore is again a structure, as in Figure 5.3, but now includes a new integer component, *s.flag*. Admittedly, this involves a form of busy waiting. However, the semWait and semSignal operations are relatively short, so the amount of busy waiting involved should be minor. For a single-processor system, it is possible to inhibit interrupts for the duration of a semWait or semSignal operation, as suggested in Figure 5.14b. Once again, the relatively short duration of these operations means that this approach is reasonable.

UNIT – III

PRINCIPLES OF DEADLOCK

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. All deadlocks involve conflicting needs for resources by two or more processes. A common example is the traffic deadlock. Let us now look at a depiction of deadlock involving processes and computer resources. Refer to as a **joint progress diagram**, which illustrates the progress of two processes competing for two resources. Each process needs exclusive use of both resources for a certain period of time.

Two processes, P and Q, have the following general form:

Process P Process Q
• • • • • •
Get A Get B
• • • • • •
Get B Get A
• • • • • •
Release A Release B
• • • • • •
Release B Release A
• • • • • •

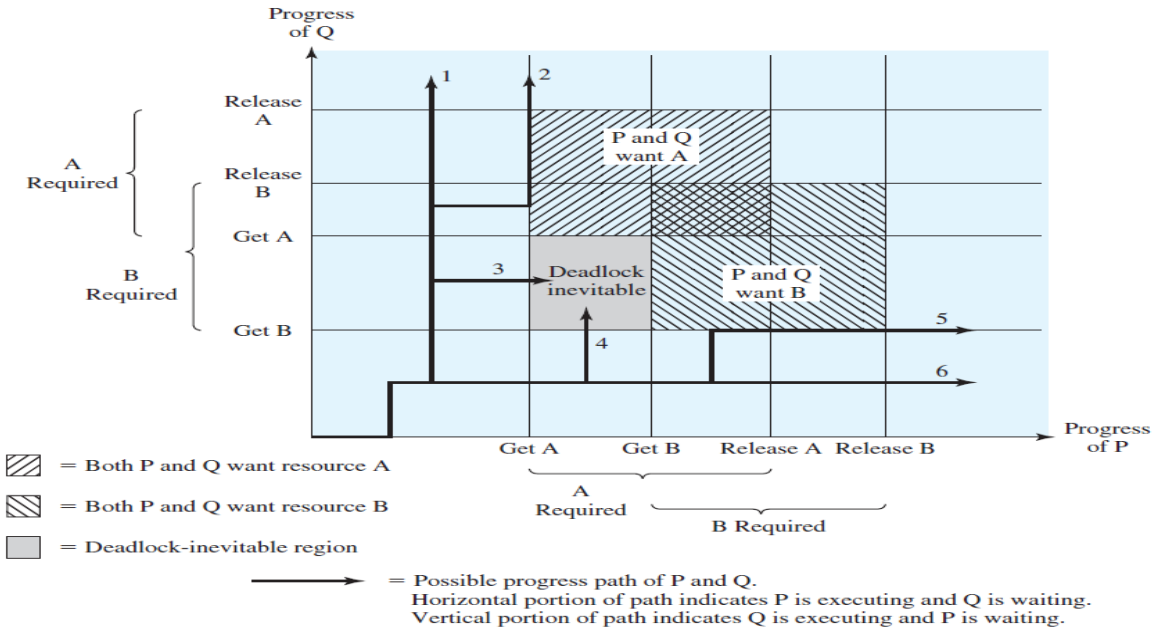


Figure 6.2 Example of Deadlock

The figure shows six different execution paths. These can be summarized as follows:

1. Q acquires B and then A and then releases B and A.When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A.Q releases B and A.When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A.Deadlock is inevitable, because as execution proceeds,Q will block on A and P will block on B.
4. P acquires A and then Q acquires B.Deadlock is inevitable, because as execution proceeds,Q will block on A and P will block on B.
5. P acquires A and then B.Q executes and blocks on a request for B. P releases A and B.When Q resumes execution, it will be able to acquire both resources.
6. P acquires A and then B and then releases A and B.When Q resumes execution, it will be able to acquire both resources.

The gray-shaded area of Figure 6.2 , which can be referred to as a **fatal region**, applies to the commentary on paths 3 and 4. If an execution path enters this fatal region, then deadlock is inevitable. Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application. For example, suppose that P does not need both resources at the same time so that the two processes have the following form:

Process P Process Q
• • • • • •
Get A Get B
• • • • • •
Release A Get A
• • • • • •
Get B Release B
• • • • • •
Release B Release A
• • • • • •

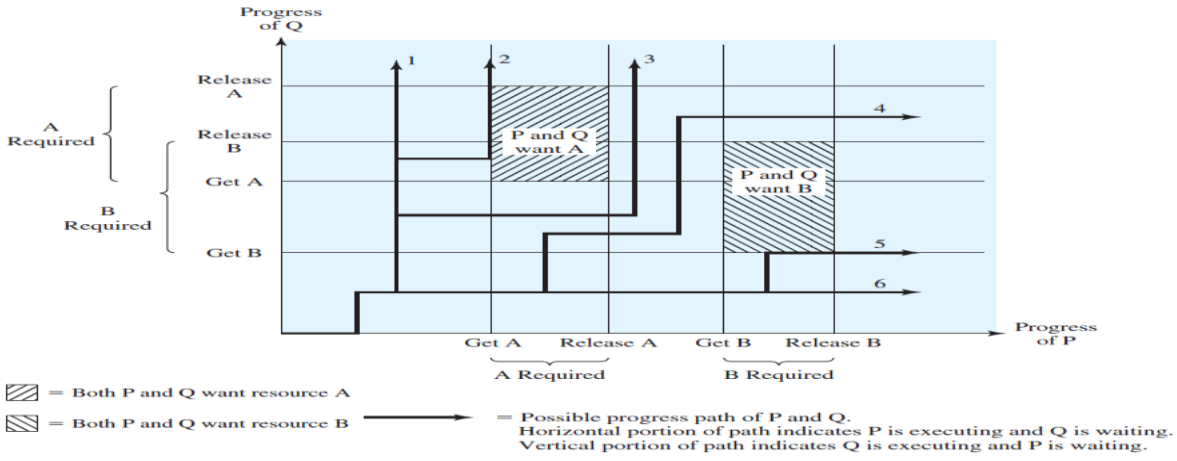


Figure 6.3 Example of No Deadlock [BACO03]

Two general categories of resources can be distinguished: reusable and consumable.

Reusable Resources

A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtain resource units that they later release for reuse by other processes. Examples of reusable resources include processors, I/O channels, main and secondary memory, devices, and data structures.

Example of deadlock with a reusable resource has to do with requests for main memory. Suppose the space available for allocation is 200 Kbytes, and the following sequence of requests occurs:

P1 P2

... ...

Request 80 Kbytes; Request 70 Kbytes;

... ...

Request 60 Kbytes; Request 80 Kbytes;

Deadlock occurs if both processes progress to their second request.

Consumable Resources

A consumable resource is one that can be created (produced) and destroyed (consumed).

Typically, there is no limit on the number of consumable resources of a particular type. An unblocked producing process may create any number of such resources. When a resource is acquired by a consuming process, the resource ceases to exist. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

As an example of deadlock involving consumable resources, consider the following pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1 P2

... ...

Receive (P2); Receive (P1);

... ...

Send (P2, M1); Send (P1, M2);

Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

Resource Allocation Graphs

A useful tool in characterizing the allocation of resources to processes is the **resource allocation graph**, The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. A graph

edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted (Figure 6.5a).Within a resource node, a dot is shown for each

instance of that resource. A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.

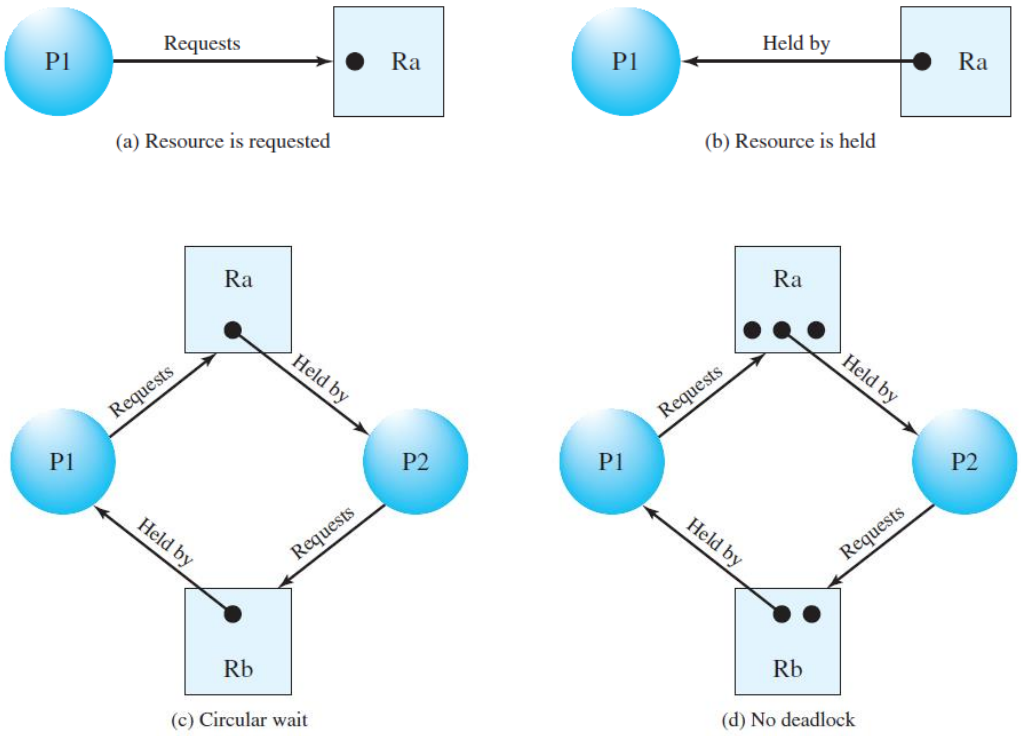


Figure 6.5 **Examples of Resource Allocation Graphs**

The resource allocation graph of Figure 6.6 corresponds to the deadlock situation in Figure 6.1b. Note that in this case, we do not have a simple situation in which two processes each have one resource the other needs. Rather, in this case, there is a circular chain of processes and resources that results in deadlock.

The Conditions for Deadlock

Three conditions of policy must be present for a deadlock to be possible:

1. Mutual exclusion. Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

2. Hold and wait. A process may hold allocated resources while awaiting assignment of other resources.

3. No preemption. No resource can be forcibly removed from a process holding it.

In many ways these conditions are quite desirable. For example, mutual exclusion is needed to ensure consistency of results and the integrity of a database. Similarly, preemption should not be done arbitrarily. For example, when data resources are involved, preemption must be supported by a rollback recovery mechanism, which restores a process and its resources to a suitable previous state from which the process can eventually repeat its actions. The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required:

4. Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain .

Thus, all four conditions listed above are sufficient for deadlock. To summarize.

Virtually all textbooks simply list these four conditions as the conditions needed for deadlock, but such a presentation obscures some of the subtler issues. Item 4, the circular wait condition, is fundamentally different from the other three conditions. Items 1 through 3 are policy decisions, while item 4 is a circumstance that might occur depending on the sequencing of requests and releases by the involved processes. Linking circular wait with the three necessary conditions leads to inadequate distinction between prevention and avoidance.

Possibility of Deadlock	**Existence of Deadlock**
1. Mutual exclusion	**1.** Mutual exclusion
2. No preemption	**2.** No preemption
3. Hold and wait	**3.** Hold and wait
	4. Circular wait

Three general approaches exist for dealing with deadlock. First, one can **prevent** deadlock by adopting a policy that eliminates one of the conditions (conditions 1 through 4). Second, one can **avoid** deadlock by making the appropriate dynamic choices based on the current state of resource allocation. Third, one can attempt to **detect** the presence of deadlock (conditions 1 through 4 hold) and take action to recover.

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

DEADLOCK PREVENTION

An indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously. A direct method of deadlock prevention is to prevent the occurrence of a circular wait

Mutual Exclusion

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.

Hold and Wait

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.

No Preemption

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource

Circular Wait

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type *R*, then it may subsequently request only those resources of types following *R* in the ordering.

DEADLOCK AVOIDANCE

On the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.

Process Initiation Denial

Consider a system of n processes and m different types of resources. Let us define the following vectors and matrices:

[Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

The matrix Claim gives the maximum requirement of each process for each resource, with one row dedicated to each process. This information must be declared in advance by a process for deadlock avoidance to work. Similarly, the matrix Allocation gives the current allocation to each process. The following relationships hold:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j	All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i, j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i, j	No process is allocated more resources of any type than the process originally claimed to need.

With these quantities defined, we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock. Start a new process P*n*+1 only if

$$R_j \geq C_{(n+1)\,j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$$

That is, a process is only started if the maximum claim of all current processes plus those of the new process can be met. This strategy is hardly optimal, because it assumes the worst: that all processes will make their maximum claims together.

DEADLOCK DETECTION

Solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.

Deadlock Detection Algorithm

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: it leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system.

The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the Allocation matrix of all zeros.

2. Initialize a temporary vector **W** to equal the Available vector.

3. Find an index *i* such that process *i* is currently unmarked and the *i*th row of **Q** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.

4. If such a row is found, mark process *i* and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

A deadlock exists if and only if there are unmarked processes at the end of the algorithm. Each unmarked process is deadlocked.

UNIPROCESSOR SCHEDULING.

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response

time, throughput, and processor efficiency. The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved.

Table 9.1 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

TYPES OF PROCESSOR SCHEDULING

Long-term scheduling is performed when a new process is created. This is a decision whether to add a new process to the set of processes that are currently active. Medium-term scheduling is a part of the swapping function. This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution. Short-term scheduling is the actual decision of which ready process to execute next.

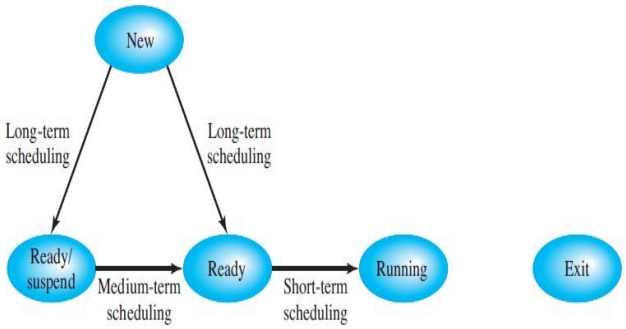


Figure 9.1 Scheduling and Process State Transitions

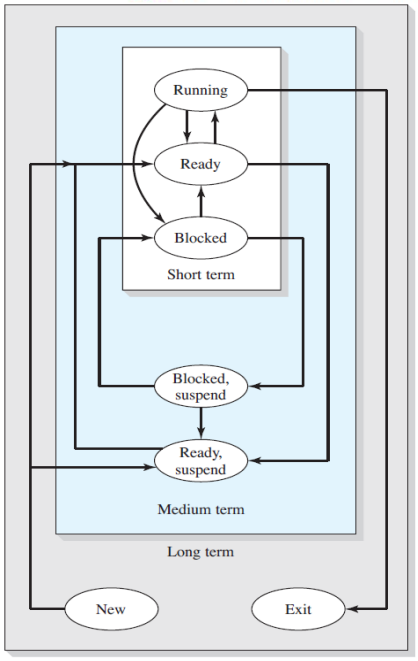


Figure 9.2 Levels of Scheduling

Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming. Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler.

The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide when the operating system can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. Thus, the long-term scheduler may limit the degree of multiprogramming to provide satisfactory service to the current set of processes.

The criteria used may include priority, expected execution time, and I/O requirements. For example, if the information is available, the scheduler may attempt to keep a mix of process or bound and I/O-bound processes.2 Also, the decision may be made depending on which I/O resources are to be requested, in an attempt to balance I/O usage.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function the swapping-in decision is based on the need to manage the degree of multiprogramming. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another.

SCHEDULING ALGORITHMS

Short-Term Scheduling Criteria

The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior. The commonly used criteria can be categorized along two dimensions. **First,** we can make a distinction between user-oriented and system-oriented criteria. Table 9.2 summarizes key scheduling criteria.

Table 9.2 Scheduling Criteria

User Oriented, Performance Related
Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.
Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.
Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.
User Oriented, Other
Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.
System Oriented, Performance Related
Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.
Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.
System Oriented, Other
Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.
Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.
Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

The Use of Priorities

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one of lower priority. Figure 9.4 illustrates the use of priorities.

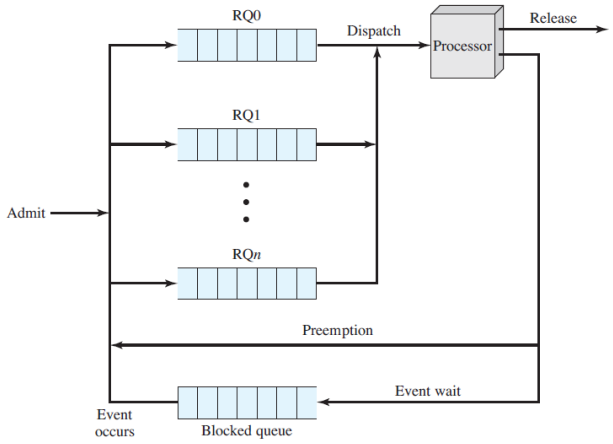


Figure 9.4 Priority Queuing

Instead of a single ready queue,we provide a set of queues, in **descending order of priority:RQ0, RQ1, . . . RQ*n***, with priority[RQ*i*] !priority[RQ*j*] for *i*"*j*.When a scheduling selection is to be made, the scheduler will start at the **highest-priority ready queue** (RQ0). If there are one or more processes in the queue, a process is selected using some scheduling policy. If RQ0 is empty, then RQ1 is examined, and so on. One problem with a pure priority scheduling scheme is that lower-priority processes may suffer **starvation**.

Alternative Scheduling Policies

The **selection function** determines which process, among ready processes, is selected next for execution. The function may be based on priority, resource requirements, or the execution characteristics of the process.

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

The **decision mode** specifies the instants in time at which the selection function is exercised. There are **two general** categories:

• **Nonpreemptive:** In this case, once a process is in the Running state, it continues to execute until (a) it terminates or (b) it blocks itself to wait for I/O or to request some operating system service.

• **Preemptive:** The currently running process may be interrupted and moved to the Ready state by the operating system.

For the example of Table 9.4, Figure 9.5 shows the execution pattern for each policy for one cycle, and Table 9.5 summarizes some key results. First, the finish time of each process is determined. from this, we can determine the turnaround time. In terms of the queuing model, **turnaround time** (TAT) is the residence time ***Tr***, or total time that the item spends in the system (**waiting time plus service time**).

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-Come-First-Served

The simplest scheduling policy is first-come-first served (FCFS), also known as **first-in-first-out (FIFO)** or a strict queuing scheme. As each process becomes ready, it joins the ready queue. When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running. **FCFS performs much better for long processes than short ones**. **Another difficulty with FCFS is that it tends to favor processor-bound processes over I/O-bound processes**. If the processor-bound process is also blocked, the processor becomes idle. Thus, FCFS may result in inefficient use of both the processor and the I/O devices.

Round Robin

A straightforward way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock. The simplest such policy is round robin. A clock interrupt is generated at periodic intervals. **When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis. This technique is also** known as **time slicing**, because each process is given a slice of time before being preempted. With round robin, the principal design issue is the length of the time quantum, or slice, to be used. If the quantum is very short, then short processes will move through the system relatively quickly.

Round robin is particularly effective in a general-purpose time-sharing system or transaction processing system. One drawback to round robin is its relative treatment of

processor-bound and I/O-bound processes. **processor-bound process generally uses a complete time quantum while executing and immediately returns to the ready queue. Thus, processor-bound processes tend to receive an unfair portion of processor time, which results in poor performance for I/O-bound processes, inefficient use of I/O devices, and an increase in the variance of response time**. A refinement to round robin that he refers to as a virtual round robin (VRR) and that avoids this unfairness. Figure 9.7 illustrates the scheme.

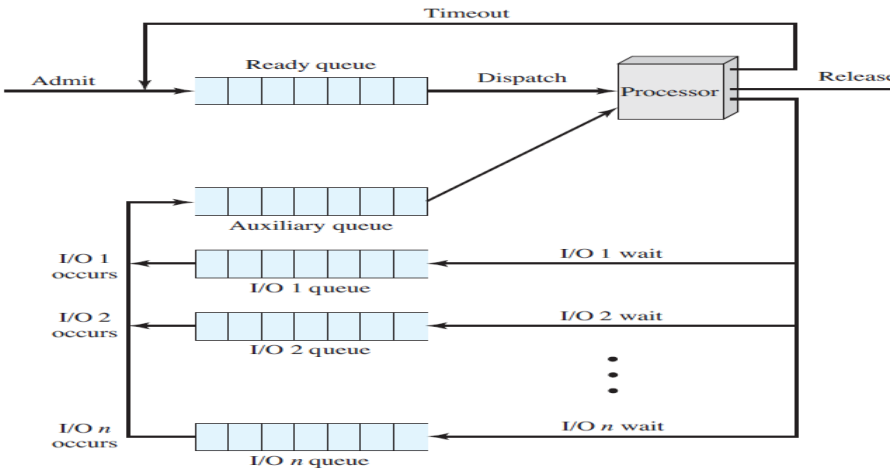


Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

The **new feature is an FCFS auxiliary queue** to which processes are moved after being released from an I/O block. When a dispatching decision is to be made, processes in the auxiliary queue get **preference over** those in the main ready queue.

Shortest Process Next

Another approach to reducing the bias in favor of long processes inherent in FCFS is the **Shortest Process Next (SPN)** policy. This is **a non-preemptive policy** in which the process with the shortest expected processing time is selected next. Thus a short process will jump to the head of the queue past longer jobs.

One difficulty with the SPN policy is the need to know or at least estimate the required processing time of each process. For batch jobs, the system may require the programmer to estimate the value and supply it to the operating system. A risk with SPN is the possibility of starvation for longer processes, as long as there is a steady supply of shorter processes. On the other hand, although SPN reduces the bias in favor of longer jobs, it still is not desirable for a time-sharing or transaction processing environment because of the lack of preemption.

Shortest Remaining Time

The shortest remaining time (SRT) policy is a preemptive version of SPN. In this case, the scheduler always chooses the process that has the **shortest expected remaining processing time**. When a new process joins the ready queue, **it may in fact have a shorter**

remaining time than the currently running process. Accordingly, the scheduler may preempt the current process when a new process becomes ready.

As with SPN, the scheduler must have an estimate of processing time to perform the selection function, and there is a risk of starvation of longer processes. SRT does not have the bias in favor of long processes found in FCFS. Unlike round robin, no additional interrupts are generated, reducing overhead.

Highest Response Ratio Next

In Table 9.5, we have used the normalized turnaround time, which is the ratio of turnaround time to actual service time, as a figure of merit. For each individual process, we would like to minimize this ratio, and we would like to minimize the average value over all processes.

Consider the following ratio:

$$R = \frac{w + s}{s}$$

where
R = response ratio
W = time spent waiting for the processor
s = expected service time

If the process with this value is dispatched immediately, R is equal to the normalized turnaround time. Note that the minimum value of R is 1.0, which occurs when a process first enters the system.

Thus, our scheduling rule becomes the following:

When the current process completes or is blocked, choose the ready process with the greatest value of R. As with SRT and SPN, the expected service time must be estimated to use highest response ratio next (HRRN).

Feedback

If we have no indication of the relative length of various processes, then none of SPN, SRT, and HRRN can be used. Another way of establishing a preference for shorter jobs is to penalize jobs that have been running longer. Figure 9.10 illustrates the feedback scheduling mechanism by showing the path that a process will follow through the various queues.

This approach is known as **multilevel feedback**, meaning that the operating system allocates the processor to a process and, when the process blocks or is preempted, feeds it back into one of several priority queues.

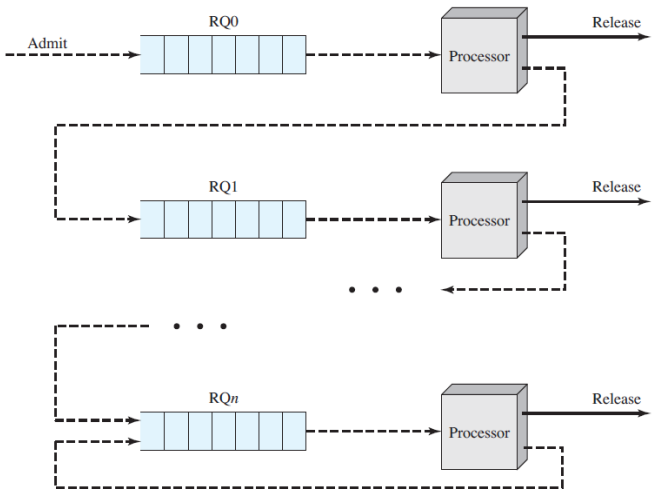


Figure 9.10 Feedback Scheduling

In general, a process scheduled from **RQ*i* is allowed to execute 2*i* time units** before preemption. This scheme is illustrated for our example in Figure 9.5 and Table 9.5. Even with the allowance for greater time allocation at lower priority, a longer process may still suffer starvation. A possible remedy is to promote a process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue.

Performance Comparison

Clearly, the performance of various scheduling policies is a critical factor in the choice of a scheduling policy. However, it is impossible to make definitive comparisons because relative performance will depend on a variety of factors, including the probability distribution of service times of the various processes, the efficiency of the scheduling and context switching mechanisms, and the nature of the I/O demand and the performance of the I/O subsystem.

Queuing Analysis

In this section, we make use of basic queuing formulas, with the common assumptions of Poisson arrivals and exponential service times.First, we make the observation that any such scheduling discipline that chooses the next item to be served independent of service time obeys the following relationship:

$$\frac{T_r}{T_s} = \frac{1}{1 - \rho}$$

where
Tr = turnaround time or residence time; total time in system, waiting plus execution
Ts =average service time; average time spent in Running state
p= processor utilization

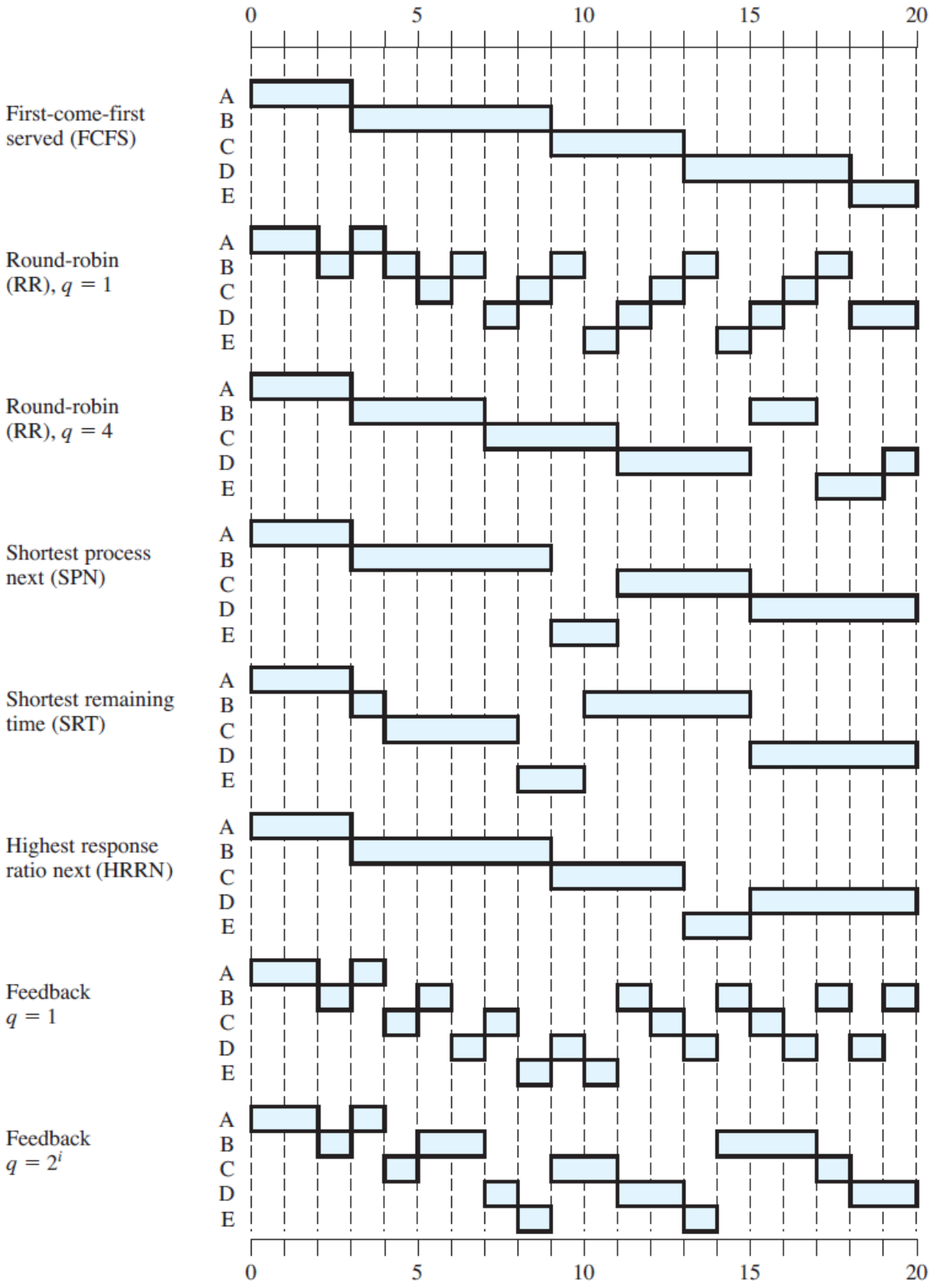


Figure 9.5 **A Comparison of Scheduling Policies**

Table 9.5 A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$						
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

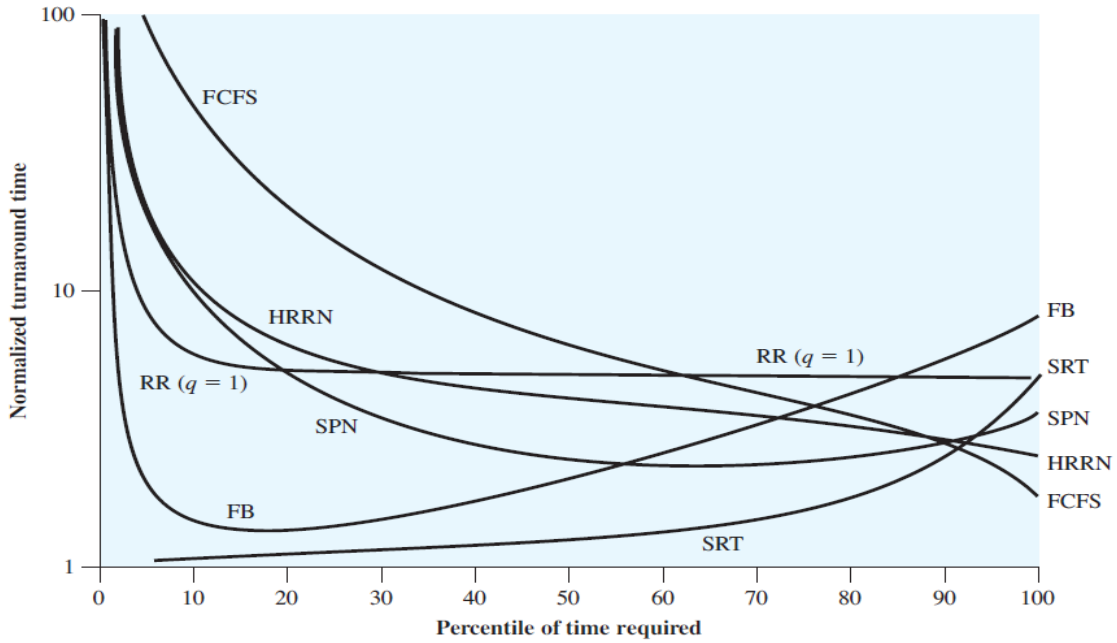


Figure 9.14 Simulation Result for Normalized Turnaround Time

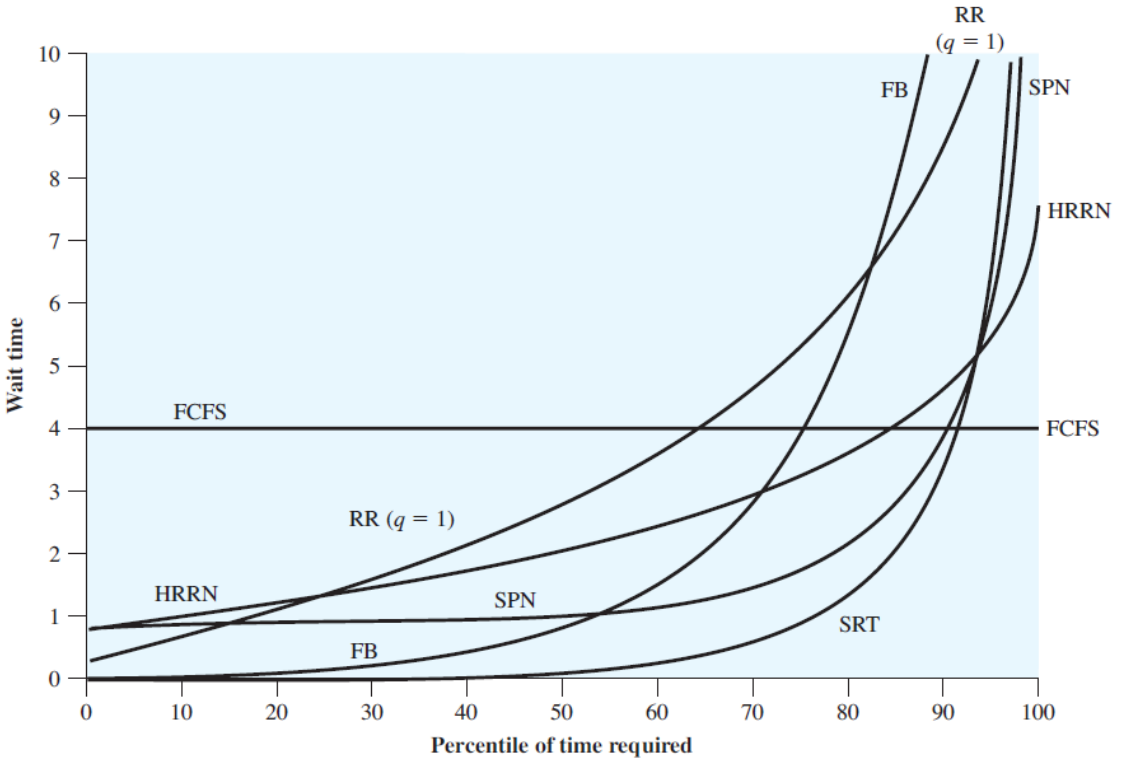


Figure 9.15 Simulation Result for Waiting Time

UNIT – IV

MEMORY MANAGEMENT

In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

MEMORY MANAGEMENT REQUIREMENTS

The requirements that memory management is intended to satisfy.
- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

Relocation

In a multiprogramming system, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. There may need to **relocate** the process to a different area of memory.

Protection

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process for reading or writing purposes.

Sharing

Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory. For example, if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy.

Logical Organization

Programs are organized into modules. Modules can be written and compiled independently, with all references from one module to another

Physical Organization

Computer memory is organized into at least two levels, referred to as main memory and secondary memory. In this two-level scheme, the organization of the flow of information betweenmain and secondary memory.

MEMORY PARTITIONING

Partition the memory into number of blocks for loading the running process. There are six different partitioningtechniques is used for memory management.

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation

Fixed Partitioning

Description :Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.

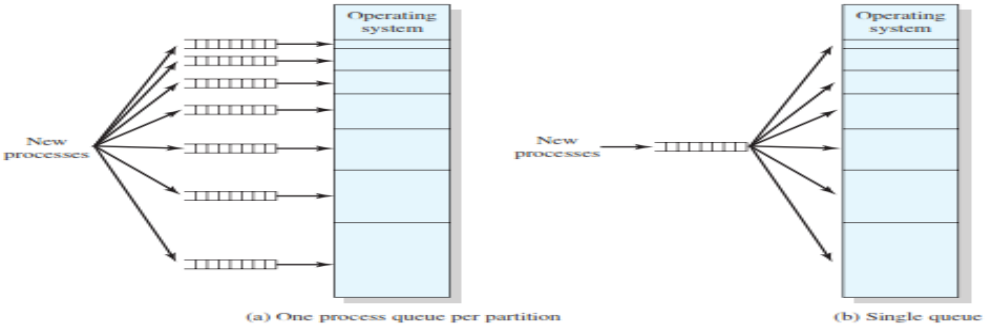


Figure 7.3 **Memory Assignment for Fixed Partioning**

Strengths: Simple to implement hence, little operating system overhead.
Weaknesses:Inefficient use of memory due to internal fragmentation maximum number ofactive processes is fixed.

Dynamic Partitioning

Description :Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.

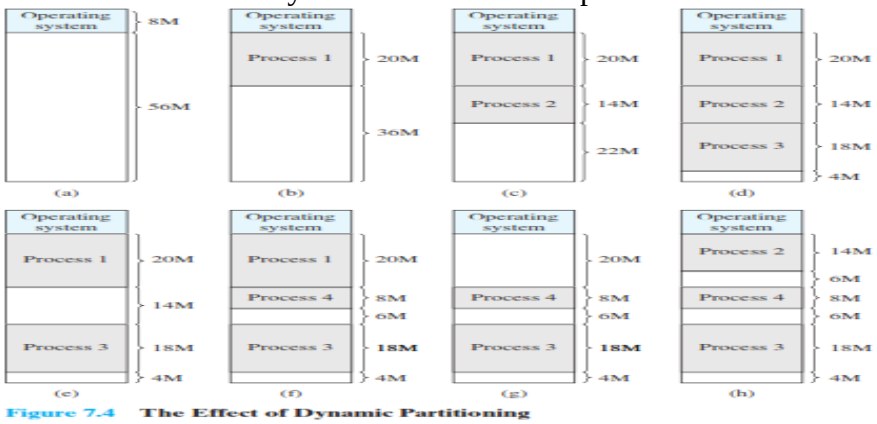


Figure 7.4 **The Effect of Dynamic Partitioning**

Strengths : No internal fragmentation, more efficient use of main memory.
Weaknesses: Inefficient use of processor due to the need for compaction to counter external fragmentation.

Three placement algorithms that might be considered are best-fit, first-fit, and next-fit. All, of course, are limited to choosing among free blocks of main memory that are equal to or larger than the process to be brought in. **Best-fit** chooses the block that is closest in size to the request. **First-fit** begins to scan memory from the beginning and chooses the first available block that is large enough. **Next-fit** begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.

Simple Paging

Description:Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.
Strengths : No external fragmentation.
Weaknesses:A small amount of internal fragmentation.

Simple Segmentation

Description:Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.
Strengths :No internal fragmentation, improved memory utilization and reduced overhead compared to dynamic partitioning.
Weaknesses:External fragmentation.

BUDDY SYSTEM

Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently.

In a buddy system, memory blocks are available of size 2^K words, $L \leq K \leq U$, where

2^L = smallest size block that is allocated
2^U = largest size block that is allocated; generally 2U is the size of the entire memory available for allocation

To begin, the entire space available for allocation is treated as a single block of size 2^U. If a request of size s such that $2^{U-1} < s \leq 2^U$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If $2^{U-2} < s \leq 2^{U-1}$, then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split in half again. This process continues until the smallest block greater than or equal to s is generated and allocated to the request. At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i. A hole may be removed from the $(i + 1)$ list by splitting it in half to create two buddies of size 2^i in the i list. Whenever a pair of buddies on the i list both become unallocated, they are removed from that list and coalesced into a single block on the $(i + 1)$ list. Presented

The following recursive algorithm is used to find a hole of size 2

```
void get_hole(int i)
{
  if (i == (U + 1)) <failure>;
  if (<i_list empty>) {
      get_hole(i + 1);
      <split hole into buddies>;
      <put buddies on i_list>;
  }
  <take first hole on i_list>;
}
```

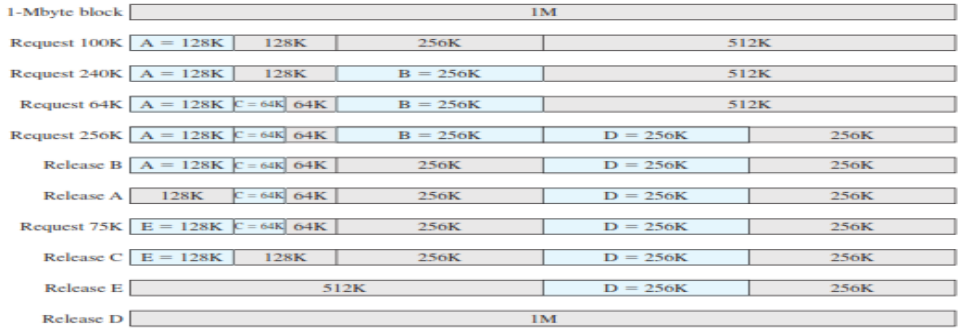


Figure 7.6 **Example of Buddy System**

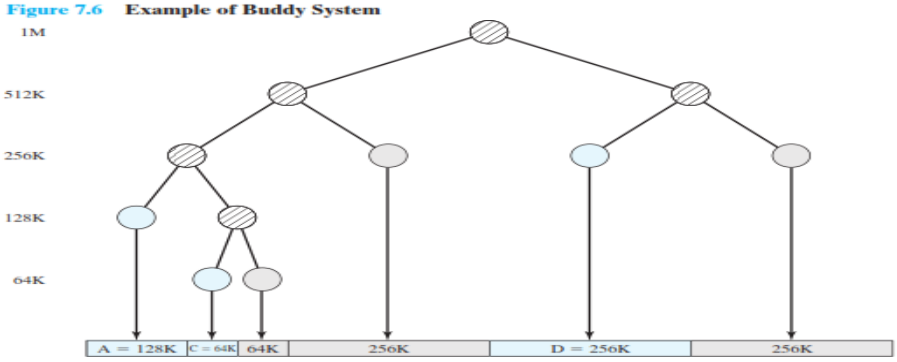


Figure 7.7 **Free Representation of Buddy System**

Relocation

A distinction is made among several types of addresses. A **logical address** is a reference to a memory location independent of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved. A **physical address**,or absolute address, is an actual location in main memory.

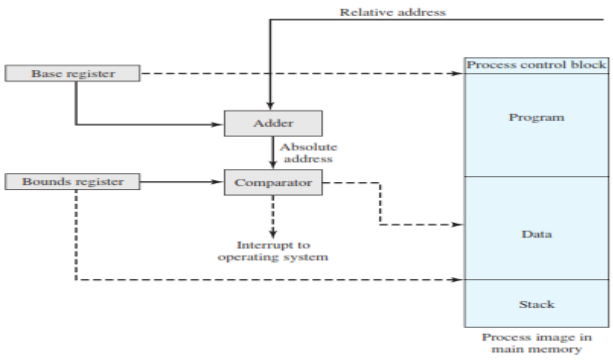


Figure 7.8 **Hardware Support for Relocation**

PAGING

Main memory is partitioned into equal fixed-sizechunks that are relatively small, and that each process is also divided into smallfixed-size chunks of the same size. Then the chunks of a process, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames.

The operating system maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page.

Presented with a logicaladdress (page number, offset), the processor uses the page table to produce a physicaladdress (frame number,offset).

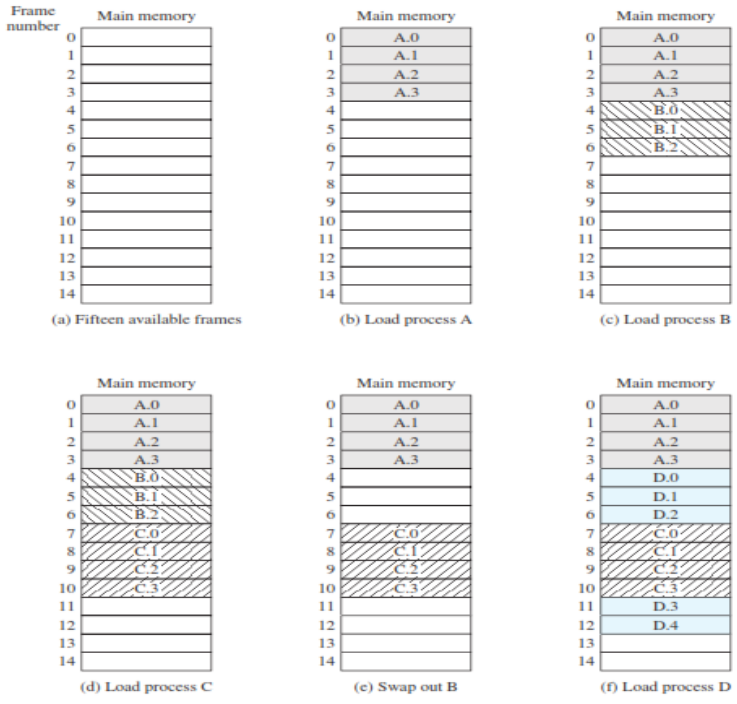
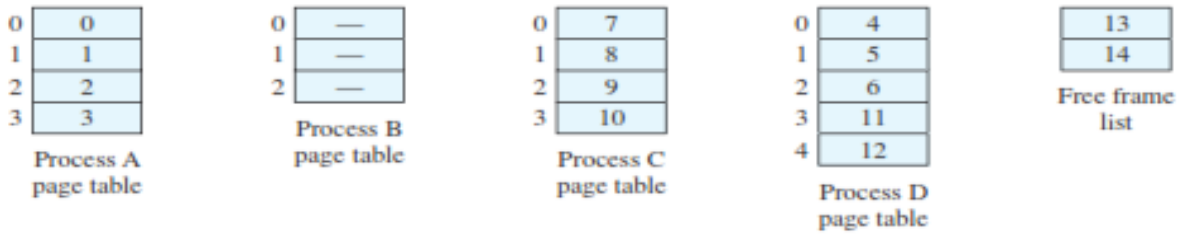


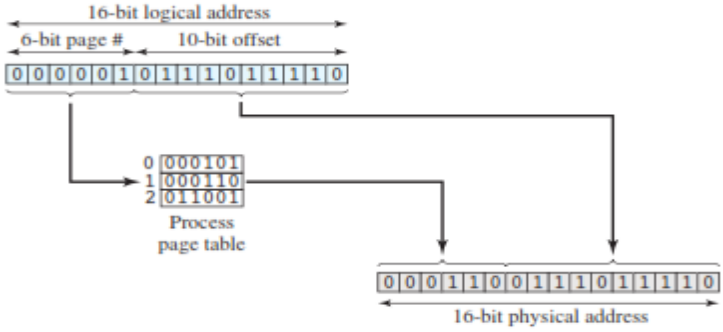
Figure 7.9 **Assignment of Process to Free Frames**



Consider an address of $n + m$ bits, where the leftmost n bits are the page number and the rightmost m
bits are the offset. In our example $n = 6$ and $m = 10$.

The following steps are needed for address translation:
• Extract the page number as the leftmost n bits of the logical address.
• Use the page number as an index into the process page table to find the framenumber, k.
• The starting physical address of the frame is $k \times 2^m$and the physical address ofthe referenced byte is that number plus the offset. This physical address neednot be calculated; it is easily constructed by appending the frame number to the offset.

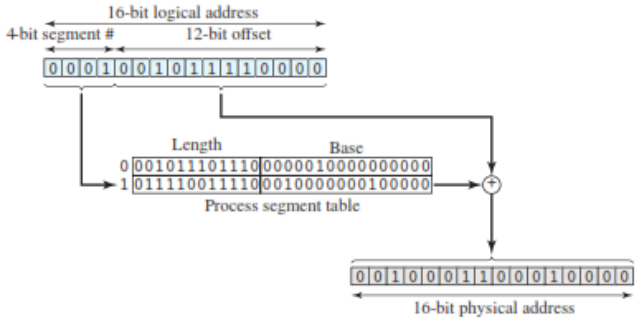
In our example, we have the logical address 0000010111011110, which is pagenumber 1, offset 478. Suppose that this page is residing in main memory frame 6 = binary 000110. Then the physical address is frame number 6, offset 478 = 0001100111011110



(a) Paging

SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments. Segmentation scheme would make use of a segment table for each process and a list of free blocks of main memory. Each segment table entry would have to give the starting address in main memory of the corresponding segment. Consider an address of n + mbits, where the leftmost n bits are the segment number and the rightmost mbits are the offset.



(b) Segmentation

In our example $n = 4$ and $m = 12$. Thus the maximum segment size is $2^{12} = 4096$. The following steps are needed for address translation.

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find thestarting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of thesegment plus the offset.

In our example, we have the logical address 0001001011110000, which is segmentnumber 1, offset 752. Suppose that this segment is residing in main memory starting atphysical address 0100000000100000. Then the physical address is 0010000000100000 + 001011110000 = 0010001100010000

VIRTUAL MEMORY

HARDWARE AND CONTROL STRUCTURES

Locality and Virtual Memory

Virtual memory, based on either paging or paging plus segmentation, has become an essential component of contemporary operating systems. Thus, at any one time, only a few pieces of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pieces are not swapped in and out of memory.

The principle of locality states that program and data references within a process tend to cluster. Hence, the assumption that only a few pieces of a process will be needed over a short period of time is valid.

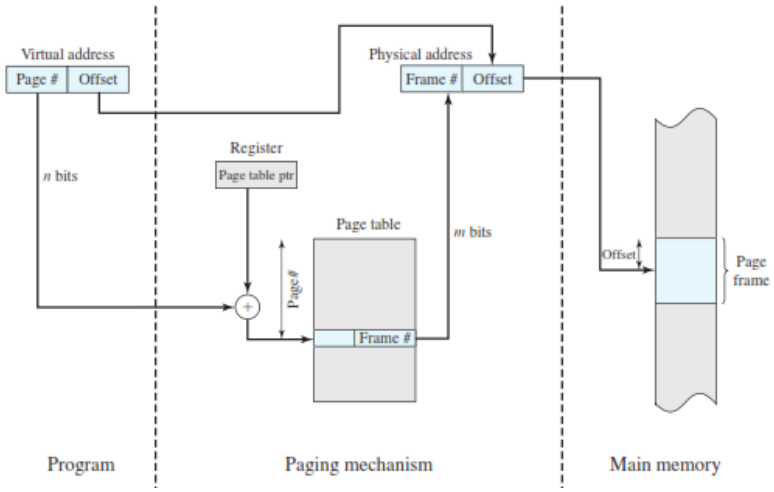
Paging

Paging is used to achieve virtual memory. A page table is also needed for a virtual memory scheme based on paging. Again, it is typical to associate a unique page table with each process. A bit is needed in each page table entry to indicate whether the corresponding

page is present (P) in main memory or not. If the bit indicates thatthe page is in memory, then the entry also includes the frame number of that page.

Page Table Structure

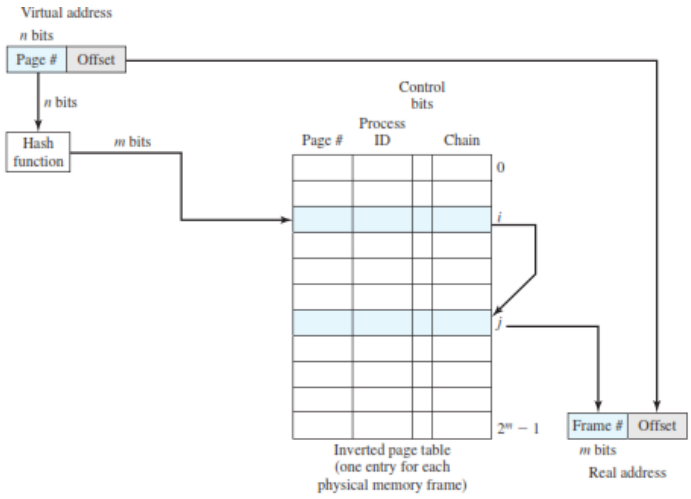
The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table.



The above suggests a hardware implementation. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number.

Inverted Page Table

A drawback of the type of page tables that we have been discussing is that their size is proportional to that of the virtual address space. An alternative approach to the use of one or multiple-level page tables is the use of an **inverted page table** structure.



In this approach, the page number portion of a virtual address is mapped into a hash value using a simple hashing function. The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the inverted page table for each real memory page frame rather than one per virtual page.**More than one virtual address** may map into the same hash table entry, a **chaining technique** is used for managing the overflow.

Each entry in the page table includes the following:

- **Page number** **:** This is the page number portion of the virtual address.
- **Process identifier :**The process that owns this page.
- **Control bits** **:** This field includes flags, such as valid, referenced, and modified; and protection and locking information
- **Chain pointer** **:** This field is null (perhaps indicated by a separate bit) if there are no chained entries for this entry. Otherwise, the field contains the index value (number between 0 and 2^m- 1) of the next entry in the chain

Translation Lookaside Buffer

Every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry and one to fetch the desired data. To overcome theproblem of doubling the memory access time.

To overcome this problem, most virtual memory schemes make use of a special high-speed cache for page table entries, usually called a **translation lookaside buffer** (TLB).

Given a virtual address, the processor will first examine the TLB. If the desired page table entry is present (***TLB hit*****),** then the frame number is retrieved and the real address is formed. If the desired page table entry is not found **(*TLB miss***), then the processor uses the page number to index the process page table and examine the corresponding page table entry.

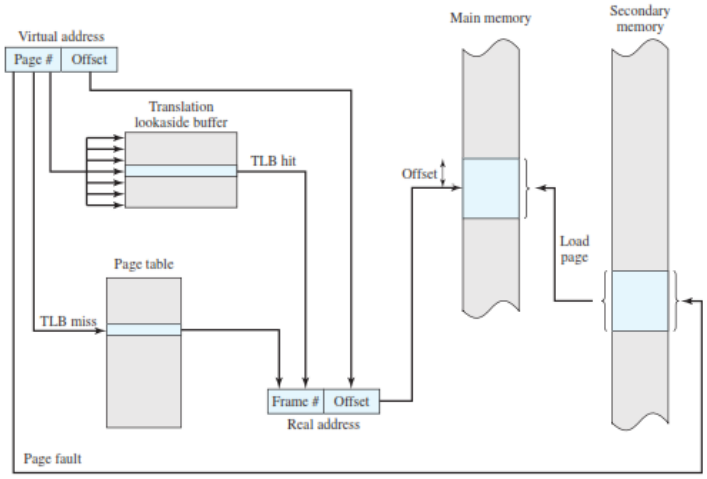


Figure 8.7 **Use of a Translation Lookaside Buffer**

The “present bit” is set, then the page is in main memory and if the “present bit” is not set, then the desired page is not in main memory and a memory access fault, called a **page fault**, is issued.

Direct versus Associative Lookup for Page Table Entries

Each entry in the TLB must include the page number as well as the complete page table entry. The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number. This technique is referred to as **associative mapping** and is contrasted with the direct mapping, or indexing, used for lookup in the page table . The design of the TLB also must consider the way in which entries are organized in the TLB and which entry to replace when a new entry is brought in.

Segmentation

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. This organization has a number of advantages to the programmer over a
nonsegmented address space:

- It simplifies the handling of growing data structures
- It allows programs to be altered and recompiled independently
- It lends itself to sharing among processes
- It lends itself to protection

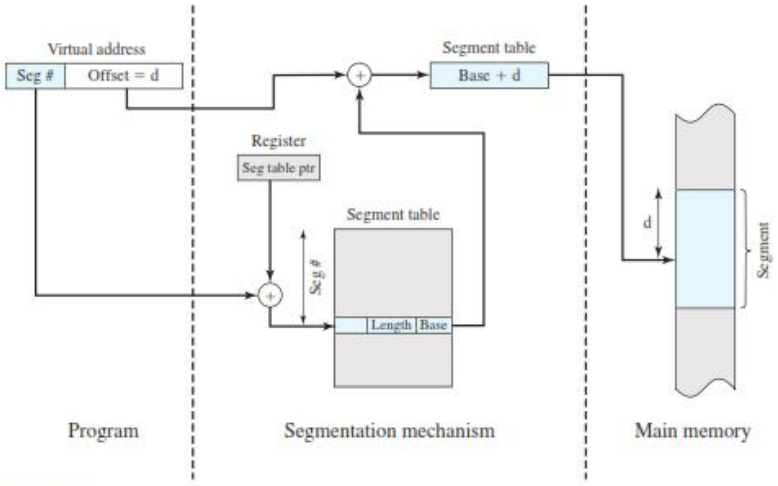


Figure 8.12 **Address Translation in a Segmentation System**

The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of segment number and offset, into a physical address, using a segment table. When a particular process is running, a register holds the starting address of the segment table for that process. The segment number of a virtual address is used to index that table and look up the corresponding main memory address for the start of the segment. This is added to the offset portion of the virtual address to produce the desired real address.

Combined Paging and Segmentation

In a combined paging/segmentation system, a user's address space is broken up into a number of segments Each segment is, in turn, broken up into a number of fixed-size pages, which are equal in length to a main memory frame.

OPERATING SYSTEM SOFTWARE

The design of the memory management portion of an operating system depends on three fundamental areas of choice:

- Whether or not to use virtual memory techniques
- The use of paging or segmentation or both
- The algorithms employed for various aspects of memory management. The choices related to the third item are the domain of operating system software

Table 8.4 Operating System Policies for Virtual Memory

Fetch Policy Demand Prepaging **Placement Policy** **Replacement Policy** Basic Algorithms Optimal Least recently used (LRU) First-in-first-out (FIFO) Clock Page buffering	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local **Cleaning Policy** Demand Precleaning **Load Control** Degree of multiprogramming

Fetch Policy

The fetch policy determines when a page should be brought into main memory. The two common alternatives are demand paging and prepaging. With **demand paging**, a page is brought into main memory only when a reference is made to a location on that page. With **prepaging**, pages other than the one demanded by a page fault are brought in.

Placement Policy

The placement policy determines where in real memory a process piece is to reside. In a pure segmentation system, the placement policy is an important design issue policies such as best-fit, first-fit.

Replacement Policy

The "replacement policy," deals with the selection of a page in main memory to be replaced when a new page must be brought in. When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, the replacement

policy determines which page currently in memory is to be replaced. There are certain basic algorithms that are used for the selection of a page to replace.

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock

The **optimal** policy selects for replacement that page for which the time tothe next reference is the longest. This policy results in the fewest number of page faults.this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events.

The **least recently used** (LRU) policy replaces the page in memory that hasnot been referenced for the longest time. he LRU policy does nearly as well as the optimal policy. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data.

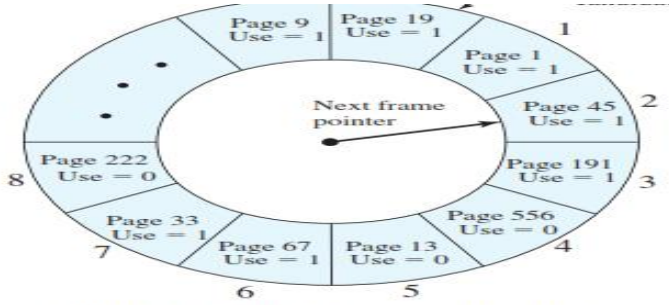
The **first-in-first-out** (FIFO) policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. All that is required is a pointer that circles through the page frames of the process. This is therefore one of the simplest page replacement policies to implement. The logic behind this choice, other than its simplicity, is that one is replacing the page that has been in memory the longest.

The simplest form of **clock policy** requires the association of an additional bit with each frame, referred to as the use bit. When a page is first loaded into a frame in memory, the use bit for that frame is set to 1. Whenever the page is subsequently referenced its use bit is set to 1.

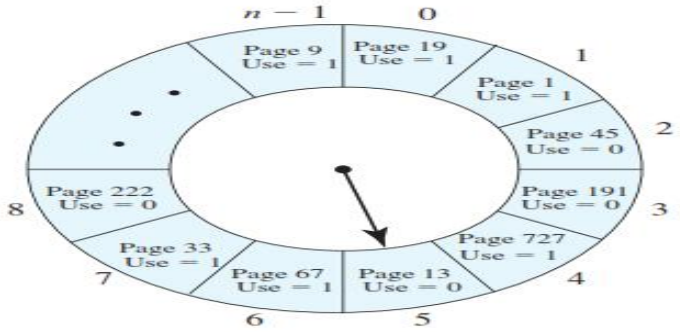
When it comes time to replace a page, the operating system scans the buffer to find a frame with a use bit set to zero. Each time it encounters a frame with a use bit of 1, it resets that bit to zero and continues on.

Page ac

s



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

CL

Figure 8

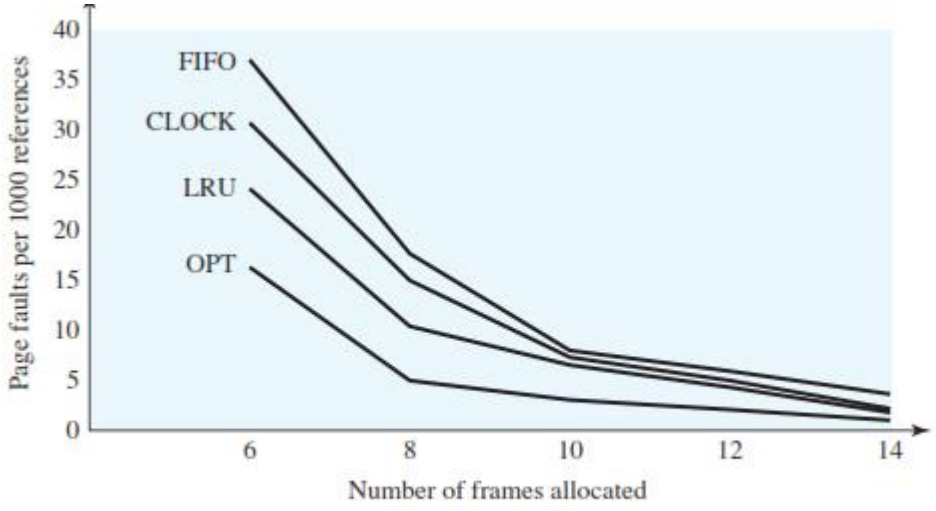


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Page Buffering

The free and modified page lists act as a cache of pages. The modified page list serves another useful function. Modified pages are written out in clusters rather than one at a time.

Replacement Policy and Cache Size

With a large cache, the replacement of virtual memory pages can have a performance impact. If the page frame selected for replacement is in the cache, then that cache block is lost as well as the page that it holds.

UNIT – V

I/O MANAGEMENT AND DISK SCHEDULING

I/O DEVICES

External devices that engage in I/O with computer systems can be roughly grouped into three categories:

- **Human readable:** Suitable for communicating with the computer user. Examples include printers and terminals,

- **Machine readable:** Suitable for communicating with electronic equipment.Examples are disk drives, USB keys, sensors

- **Communication:** Suitable for communicating with remote devices. Examples are digital line drivers and modems.

The operating system differs in transfer of I/O by the following characteristics

- **Data rate:** The speed of data bits transferred per second is vary from one operating system to another.
- **Application:** The use to which a device is put has an influence on the software and policies in the operating system and supporting utilities.
- **Complexity of control:** A printer requires a relatively simple control interface. A disk is much more complex.
- **Unit of transfer:** Data may be transferred as a stream of bytes or characters (e.g., terminal I/O) or in larger blocks (e.g., disk I/O).
- **Data representation:** Different data encoding schemes are used by different devices, including differences in character code.
- **Error conditions:** The nature of errors, the way in which they are reported, their consequences,

ORGANIZATION OF THE I/O FUNCTION

There are three techniques for performing I/O :

Table 11.1 I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Programmed I/O: The processor issues an I/O command, on behalf of a process, to an I/O module.

Interrupt-driven I/O: The processor issues an I/O command on behalf of a process. There are then two possibilities. If the I/O instruction from the process is nonblocking, then the processor continues to execute instructions from the process that issued the I/O command. If

the I/O instruction is blocking, then the next instruction that the processor executes is from the OS, which will put the current process in a blocked state and schedule another process.

Direct memory access (DMA): A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

The Evolution of the I/O Function

The evolutionary steps can be summarized as follows:

1. The processor directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The processor uses programmed I/O without interrupts.With this step, the processor becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as step 2 is used, but now interrupts are employed. The processor need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.
4. The I/O module is given direct control of memory via DMA. It can now move a block of data to or from memory without involving the processor, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a separate processor, with a specialized instruction set tailored for I/O.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled,with minimal processor involvement.

Direct Memory Access

Figure 11.2 indicates, in general terms, the DMA logic. The DMA unit is capable of mimicking the processor and, indeed, of taking over control of the system bus just like a processor. It needs to do this to transfer data to and from memory over the system bus.

The DMA technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module.

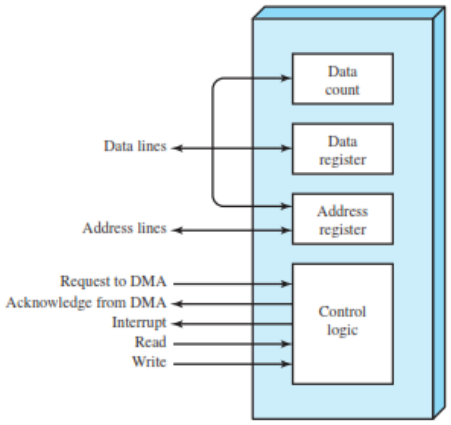


Figure 11.2 Typical DMA Block Diagram

- The address of the I/O device involved, communicated on the data lines The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
- The number of words to be read or written, again communicated via the data lines and stored in the data count register
- The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor.

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. The system bus that the DMA module shares with the processor and main memory is used by the DMA module only to exchange data with memory and to exchange control signals with the processor.

OPERATING SYSTEM DESIGN ISSUES

Design Objectives

Two objectives are paramount in designing the I/O facility: efficiency and generality. **Efficiency** is important because I/O operations often form a bottleneck in a computing system.

Logical Structure of the I/O Function

The hierarchical philosophy is that the functions of the operating system should be separated according to their complexity, their characteristic time scale, and their level of abstraction. Following this approach leads to an organization of the operating system into a series of layers. Each layer performs a related subset of the functions required of the operating system. It relies on the next lower layer to perform more primitive functions and to conceal the details of those functions. Ideally, the layers should be defined so that changes in one layer do not require changes in other layers. The details of the organization will depend on the type of device and the application. The three most important logical structures are presented in the below figure.

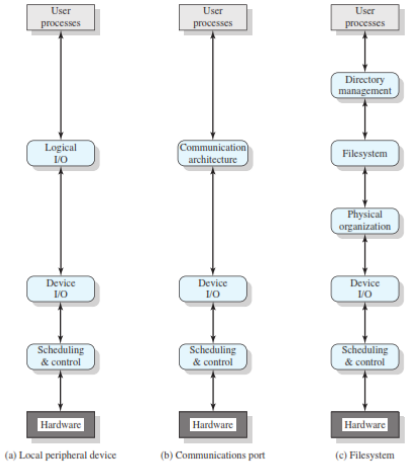


Figure 11.4 **A Model of I/O Organization**

The following layers are involved:

- **Logical I/O:** The logical I/O module deals with the device as a logical resource and is not concerned with the details of actually controlling the device. The logical I/O module is concerned with managing general I/O functions on behalf of user processes, allowing them to deal with the device in terms of a device identifier and simple commands such as open, close, read, write.

- **Device I/O:** The requested operations and data (buffered characters, records, etc.) are converted into appropriate sequences of I/O instructions, channel commands, and controller orders. Buffering techniques may be used to improve utilization.

- **Scheduling and control:** The actual queuing and scheduling of I/O operations occurs at this layer, as well as the control of the operations. Thus, interrupts are handled at this layer and I/O status is collected and reported. This is the layer of software that actually interacts with the I/O module and hence the device hardware.

Figure 11.4c shows a representative structure for managing I/O on a secondarystorage device that supports a file system. The three layers not previously discussedare as follows :

- **Directory management:** At this layer, symbolic file names are converted to identifiers that either reference the file directly or indirectly through a file descriptor or index table.This layer is also concerned with user operations thataffect the directory of files, such as add, delete, and reorganize.

- **File system:** This layer deals with the logical structure of files and with the operationsthatcanbespecifiedbyusers,such as open, close, read, write. Accessrights are also managed at this layer.

- **Physical organization:** Just as virtual memory addresses must be convertedinto physical main memory addresses, taking into account the segmentationand paging structure, logical references to files and records must be convertedto physical secondary storage addresses.

I/O BUFFERING

Suppose that a user process wishes to read blocks of data from a disk one at a time,with each block having a length of 512 bytes. There are two problems with this approach. First, the program is hung up waiting for the relatively slow I/O to complete.The second problem is that this approach to I/O interferes with swapping decisions by the operating system.

The various approaches to buffering, it is sometimes important to make a distinction between two types of I/O devices: block oriented and stream oriented. A **block-oriented** device stores information in blocks that are usually of fixed size, and transfers are made one block at a time. Generally, it is possible to reference data by its block number. Disks and USB keys are examples of block-oriented devices.

A **stream-oriented** device transfers data in and out as a stream of bytes, with no block structure. Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are stream oriented.

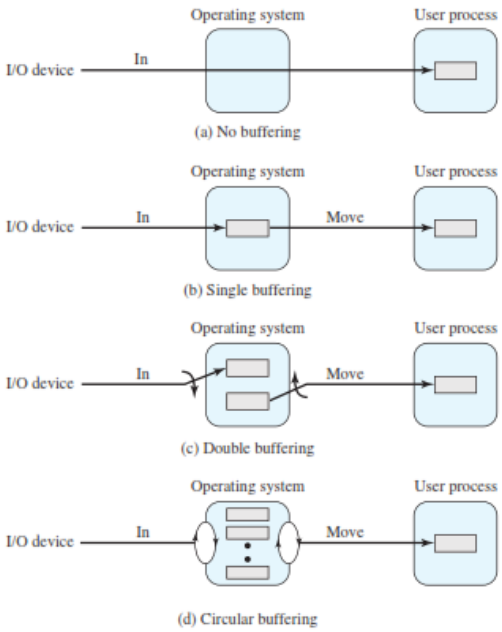


Figure 11.5 **I/O Buffering Schemes (input)**

Single Buffer

The simplest type of support that the operating system can provide is single buffering(Figure11.5b).When a user process issues an I/O request, the operating systemassigns a buffer in the system portion of main memory to the operation. This approach will generally provide a speedup compared to the lack of systembuffering.

The user process can be processing one block of data while the nextblock is being read in. The operating system is able to swap the process out becausethe input operation is taking place in system memory rather than user process memory.

Double Buffer

An improvement over single buffering can be had by assigning two system buffers to the operation (Figure 11.5c). A process now transfers data to (or from) one buffer while the operating system empties (or fills) the other. This technique is known as **double buffering** or **buffer swapping**.

Circular Buffer

A double-buffer scheme should smooth out the flow of data between an I/O device and a process. If the performance of a particular process is the focus of our concern, then we would like for the I/O operation to be able to keep up with the process. Double buffering may be inadequate if the process performs rapid bursts of I/O.

The Utility of Buffering

Buffering is a technique that smoothes out peaks in I/O demand. However, no amount of buffering will allow an I/O device to keep pace with a process indefinitely when the average demand of the process is greater than the I/O device can service. Even with multiple buffers, all of the buffers will eventually fill up and the process will have to wait after processing each chunk of data.

DISK SCHEDULING

Disk Performance Parameters

The actual details of disk I/O operation depend on the computer system, the operatingsystem,and the nature of the I/O channel and disk controller hardware. On amovable-head system, the time it takes to position the head at the track is known as**seek time**. In either case, once the track is selected, the disk controller waits until theappropriate sector rotates to line up with the head.

The time it takes for the beginningof the sector to reach the head is known as **rotational delay**, or rotational latency. Thesum of the seek time, if any, and the rotational delay equals the **access time**, which isthe time it takes to get into position to read or write. the time required for the transfer is the**transfer time**.

Seek Time -Seek time is the time required to move the disk arm to the requiredtrack.
Rotational Delay- Rotational delay is the time required for the addressed areaof the disk to rotate into a position where it is accessible by the read/write head.

Transfer Time The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$$T = \frac{b}{rN}$$

where

T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

where T_s is the average seek time.

Consider a disk with an advertised average seek time of 4 ms, rotation speed of 7500 rpm, and 512-byte sectors with 500 sectors per track. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

First, let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 5 adjacent tracks (5 tracks × 500 sectors/track = 2500 sectors). This is known as *sequential organization*. The time to read the first track is as follows:

Average seek	4 ms
Rotational delay	4 ms
Read 500 sectors	8 ms
	16 ms

Suppose that the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep up with the flow from the disk. Then, at most, we need to deal with rotational delay for each succeeding track. Thus, each successive track is read in 4 + 8 = 12 ms. To read the entire file;

$$\text{Total time} = 16 + (4 \times 12) = 64 \text{ ms} = 0.064 \text{ seconds}$$

Now let us calculate the time required to read the same data using random access rather than sequential access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have

Average seek	4 ms
Rotational delay	4 ms
Read 1 sector	0.016 ms
	8.016 ms

$$\text{Total time} = 2500 \times 8.016 = 20{,}040 \text{ ms} = 20.04 \text{ seconds}$$

Disk Scheduling Policies

First-In-First-Out : The simplest form of scheduling is first-in-first-out (FIFO)scheduling, which processes items from the queue in sequential order. This strategyhastheadvantageofbeingfair,because every request is honored and the requestsare honored in the order received. With FIFO, if there are only a few processes that require access and if manyof the requests are to clustered file sectors, then we can hope for good performance.

Priority : With a system based on priority (PRI), the control of the scheduling isoutside the control of disk management software. Such an approach is not intendedto optimize disk utilization but to meet other objectives within the operating system.Often short batch jobs and interactive jobs are given higher priority than longer jobsthat require longer computation.

Last In First Out : Surprisingly, a policy of always taking the most recent requesthas some merit. In transaction processing systems, giving the device to the most recentusershouldresultinlittleornoarmmovementformovingthroughasequentialfile.Taking advantage of this locality improves throughput and reduces queuelengths.

Shortest Service Time First: The SSTF policy is to select the disk I/O requestthat requires the least movement of the disk arm from its current position. Thus, wealways choose to incur the minimum seek time.

SCAN: With the exception of FIFO, all of the policies described so far can leavesome request unfulfilled until the entire queue is emptied. That is, there may alwaysbe new requests arriving that will be chosen before an existing request. A simple alternative that prevents this sort of starvation is the SCAN algorithm, also known asthe elevator algorithm.

With SCAN, the arm is required to move in one direction only, satisfying alloutstanding requests en route, until it reaches the last track in that direction or untilthere are no more requests in that direction.

C-SCAN The C-SCAN (circular SCAN) policy restricts scanning to one directiononly. Thus, when the last track has been visited in one direction, the arm is returnedto the opposite end of the disk and the scan begins again. This reduces the maximumdelay experienced by new requests.

N-step-SCAN and FSCAN With SSTF, SCAN, and C-SCAN, it is possiblethat the arm may not move for a considerable period of time. For example, if one ora few processes have high access rates to one track, they can monopolize the entiredevice by repeated requests to that track.

Table 11.2 Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

Table 11.3 Disk Scheduling Algorithms

Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of N records at a time	Service guarantee
FSCAN	N-step-SCAN with N = queue size at beginning of SCAN cycle	Load sensitive

DISK CACHE

The term *cache memory* is usually used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor.

Design Considerations

Several design issues are of interest. First, when an I/O request is satisfied from the disk cache, the data in the disk cache must be delivered to the requesting process. This can be done either by transferring the block of data within main memory from the disk cache to memory assigned to the user process A second design issue has to do with the replacement strategy. When a new sector is brought into the disk cache, one of the existing blocks must be replaced Another possibility is **least frequently used (LFU)**: Replace that block in the set that has experienced the fewest references To overcome this difficulty with LFU, a technique known as frequency-based replacement is proposed.

FILE ORGANIZATION AND ACCESS

The term *file organization* to refer to the logical structuring of the records as determined by the way in which they are accessed In choosing a file organization, several criteria are important:

- Short access time
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability

The five organizations are
- The pile
- The sequential file
- The indexed sequential file
- The indexed file
- The direct, or hashed, file

The Pile

The least-complicated form of file organization may be termed the *pile*. Data are collected in the order in which they arrive. Each record consists of one burst of data. The purpose of the pile is simply to accumulate the mass of data and save it. Records may have different fields, or similar fields in different orders.

The Sequential File

The most common form of file structure is the sequential file. In this type of file, a fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length and position of each field are known, only the values of fields need to be stored.

The field name and length for each field are attributes of the file structure. One particular field, usually the first field in each record, is referred to as the **key field**. The key field uniquely identifies the record; thus key values for different records are always different

The Indexed Sequential File

A popular approach to overcoming the disadvantages of the sequential file is the indexed sequential file. The indexed sequential file maintains the key characteristic of the sequential file: records are organized in sequence based on a key field. Two features are added: an index to the file to support random access, and an overflow file.

The indexed sequential file greatly reduces the time required to access a single record, without sacrificing the sequential nature of the file. To process the entire file sequentially, the records of the main file are processed in sequence until a pointer to the overflow file is found.

The Indexed File

The indexed sequential file retains one limitation of the sequential file: effective processing is limited to that which is based on a single field of the file. For example, when it is necessary to search for a record on the basis of some other attribute than the key field, both forms of sequential file are inadequate.

In some applications, the flexibility of efficiently searching by various attributes is desirable. To achieve this flexibility, a structure is needed that employs multiple indexes, one for each type of field that may be the subject of a search. In the general indexed file, the concept of sequentiality and a single key are abandoned. Records are accessed only through their indexes.

The Direct or Hashed File

The direct, or hashed, file exploits the capability found on disks to access directly any block of a known address. As with sequential and indexed sequential files, a key field is required in each record.

FILE DIRECRTORIES

The following are the elements present in the file directory.

Table 12.2 Information Elements of a File Directory

	Basic Information
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory.
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
	Address Information
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
	Access Control Information
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, transmitting over a network
	Usage Information
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:

- **Search:** When a user or application references a file, the directory must be searched to find the entry corresponding to that file.
- **Create file:** When a new file is created, an entry must be added to the directory.
- **Delete file:** When a file is deleted, an entry must be removed from the directory.
- **List directory:** All or a portion of the directory may be requested. Generally, this request is made by a user and results in a listing of all files owned by that user, plus some of the attributes of each file (e.g., type, access control information, usage information).
- **Update directory:** Because some file attributes are stored in the directory, a change in one of these attributes requires a change in the corresponding directory entry.

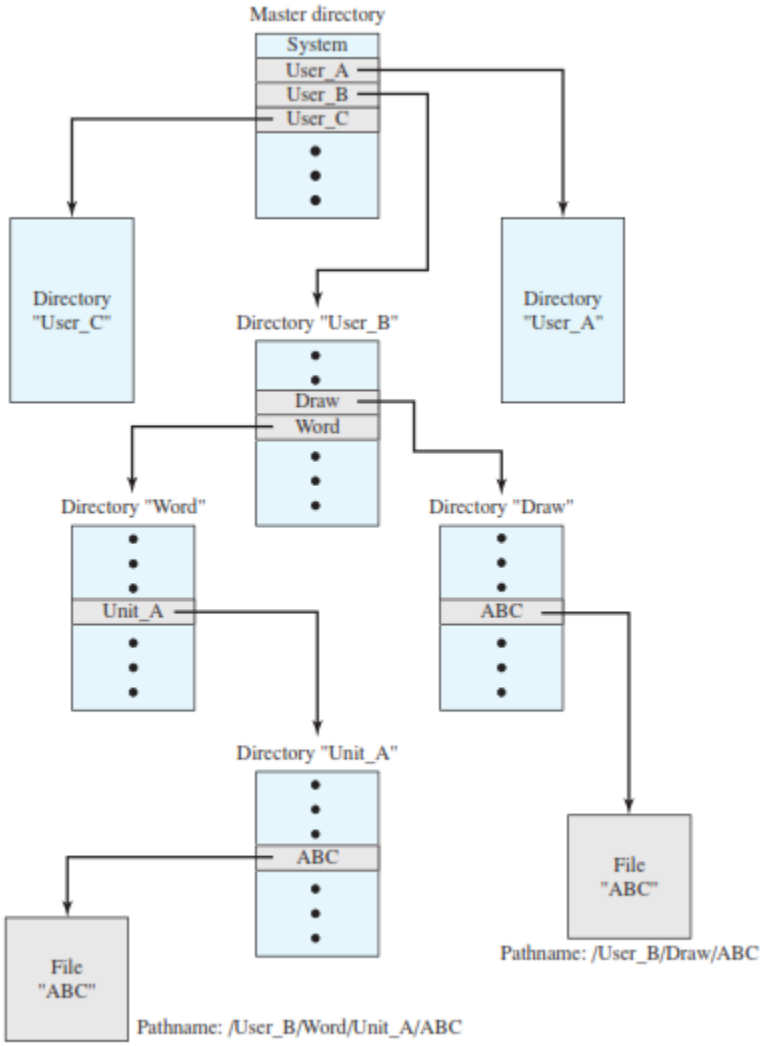


Figure 12.5 **Example of Tree-Structured Directory**

FILE SHARING

Access Rights

The file system should provide a flexible tool for allowing extensive file sharing among users. The file system should provide a number of options so that the way in which a particular file is accessed can be controlled. Typically, users or groups of users are granted certain access rights to a file. A wide range of access rights has been used. The following list is representative of access rights that can be assigned to a particular user for a particular file:

- **None:** The user may not even learn of the existence of the file, much less access it. To enforce this restriction, the user would not be allowed to read the user directory that includes this file.
- **Knowledge:** The user can determine that the file exists and who its owner is. The user is then able to petition the owner for additional access rights.
- **Execution:** The user can load and execute a program but cannot copy it. Proprietary programs are often made accessible with this restriction.
- **Reading:** The user can read the file for any purpose, including copying and execution. Some systems are able to enforce a distinction between viewing and copying. In the former case, the contents of the file can be displayed to the user, but the user has no means for making a copy.
- **Appending:** The user can add data to the file, often only at the end, but cannot modify or delete any of the file's contents. This right is useful in collecting data from a number of sources.
- **Updating:** The user can modify, delete, and add to the file's data. This normally includes writing the file initially, rewriting it completely or in part, and removing all or a portion of the data. Some systems distinguish among different degrees of updating.
- **Changing protection:** The user can change the access rights granted to other users. Typically, this right is held only by the owner of the file. In some systems, the owner can extend this right to others. To prevent abuse of this mechanism, the file owner will typically be able to specify which rights can be changed by the holder of this right.
- **Deletion:** The user can delete the file from the file system.

Simultaneous Access

When access is granted to append or update a file to more than one user, the operating system or file management system must enforce discipline. A brute-force approach is to allow a user to lock the entire file when it is to be updated.

RECORD BLOCKING

Given the size of a block, there are three methods of blocking that can be used:

Fixed blocking: Fixed-length records are used, and an integral number of records are stored in a block. There may be unused space at the end of each block. This is referred to as internal fragmentation.

Variable-length spanned blocking: Variable-length records are used and are packed into blocks with no unused space. Thus, some records must span two blocks, with the continuation indicated by a pointer to the successor block.

Variable-length unspanned blocking: Variable-length records are used, but spanning is not employed. There is wasted space in most blocks because of the inability to use the remainder of a block if the next record is larger than the remaining unused space.

SECONDARY STORAGE MANAGEMENT

On secondary storage, a file consists of a collection of blocks.The operating systemor file management system is responsible for allocating blocks to files. This raises two management issues. First, space on secondary storage must be allocated to files, and second, it is necessary to keep track of the space available for allocation

File Allocation

Several issues are involved in file allocation:

1. When a new file is created, is the maximum space required for the file allocated at once?
2. Space is allocated to a file as one or more contiguous units,which we shall refer to as portions.
3. What sort of data structure or table is used to keep track of the portions assigned to a file?

Preallocation versus Dynamic Allocation

A preallocation policy requires that the maximum size of a file be declared at the time of the file creation requestthere are advantages to the use of dynamic allocation, which allocates space to a file in portions as needed.

Portion Size: The second issue listed is that of the size of the portion allocated to a file. At one extreme, a portion large enough to hold the entire file is allocated. At the other extreme, space on the disk is allocated one block at a time. In choosing a portion size, there is a tradeoff between efficiency from the point of view of a single file versus overall system efficiency.

The following are possible alternative strategies:

- **First fit:** Choose the first unused contiguous group of blocks of sufficient size from a free block list.
- **Best fit:** Choose the smallest unused group that is of sufficient size.
- **Nearest fit:** Choose the unused group of sufficient size that is closest to the previous allocation for the file to increase locality.

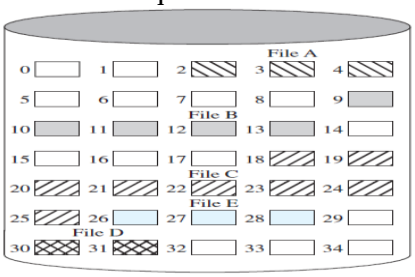
Table 12.3 File Allocation Methods

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

File Allocation Methods

Three methods are in common use: **contiguous, chained, and indexed**.

With **contiguous allocation**, a single contiguous set of blocks is allocated to a file at the time of file creation (Figure 12.7). Thus, this is a preallocation strategy, using variable-size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Contiguous allocation is the best from the point of view of the individual sequential file.

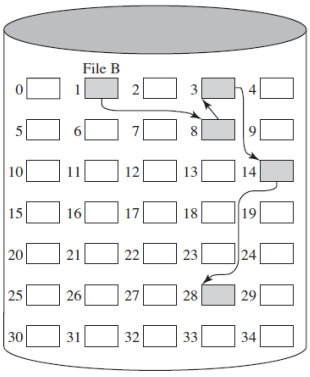


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.7 Contiguous File Allocation

At the opposite extreme from contiguous allocation is **chained allocation** (Figure 12.9). Typically, allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again, the file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Although preallocation is possible, it is more common simply to allocate blocks as needed. The selection of blocks is now a simple matter: any free block can be added to a chain.



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Figure 12.9 Chained Allocation

Indexed allocation addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file; the index has one entry for each portion allocated to the file.Typically, the file indexes are not physically stored as part of the file allocation table. Rather, the file index for a file is kept in a separate block, and the entry for the file in the file allocation table points to that block.Allocation may be on the basis of either fixed-size blocks (Figure 12.11) or variable-size portions (Figure 12.12).

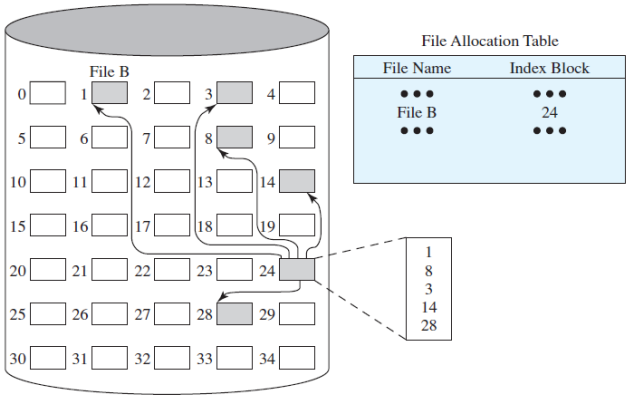


Figure 12.11 **Indexed Allocation with Block Portions**

FREE SPACE MANAGEMENT

To perform any of the file allocation techniques described previously, it is necessary to know what blocks on the disk are available. Thus we need a **disk allocation table** in addition to a file allocation table.

Here a number of techniques that have been implemented for free space management

Bit Tables : This method uses a vector containing one bit for each block on the disk. Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use. For example, for the disk layout of Figure 12.7, a vector of length 35 is needed and would have the following value:3.

00111000011111000011111111111011000

A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks. Thus, a bit table works well with any of the file allocation methods outlined. Another advantage is that it is as small as possible.

Chained Free Portions : The free portions may be chained together by using a pointer and length value in each free portion. This method has negligible spaceoverhead because there is no need for a disk allocation table, merely for a pointer to the beginning of the chain and the length of the first portion. This method is suited to all of the file allocation methods.

If allocation is a block at a time, simply choose the free block at the head of the chain and adjust the first pointer or length value. If allocation is by variable-length portion, a first-fit algorithm may be used:The headers from the portions are fetched one at a time to determine the next suitable free portion in the chain. Again, pointer and length values are adjusted.

Indexing : The indexing approach treats free space as a file and uses an index table as described under file allocation. For efficiency, the index should be on the basis of variable-size portions rather than blocks.Thus, there is one entry in the table for every free portion on the disk.

Free Block List : In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved portion of the disk.

Volume : A collection of addressable sectors in secondary memory that an OS or application can use for data storage.The sectors in a volume need not be consecutive on a physical storage device; instead they need only appear that way to the OS or application. A volume may be the result of assembling and merging smaller volumes.