MODULE 1

INTRODUCTION TO OPERATING SYSTEM

Structure

- Goals of an OS
- Operation of an OS
- Computational Structures
- Resource allocation techniques
- Efficiency, System Performance and User Convenience
- Classes operating System
- Batch processing,
- Multi programming
- Time Sharing Systems
- Real Time operating systems
- Distributed operating systems

1.0: Objective

On completion of this chapter student will be able to

- Discuss the concepts of operating system.
- Explain the goals and functions of OS.
- Differentiate between the different classes of OS.
- Explain Batch processing Operating System.
- Discuss the concepts of multiprogramming system.
- Discuss the operation of time sharing system
- Explain the concept of real time operating system
- Discuss the operation of distributed system.

An operating system (OS) is different things to different users. Each user's view is called an *abstract view* because it emphasizes features that are important from the viewer‘s perspective, ignoring all other features. An operating system implements an abstract view by acting as an intermediary between the user and the computer system. This arrangement not only permits an operating system to provide several functionalities at the same time, but also to change and evolve with time.

An operating system has two goals—efficient use of a computer system and user convenience. Unfortunately, user convenience often conflicts with efficient use of a computer system- Consequently; an operating system cannot provide both. It typically strikes a balance between the two that is most effective in the environment in which a computer system is used—efficient use is important when a computer system is shared by several users while user convenience is important in personal computers.

1.1 The fundamental goals of an operating system are:

1. *Efficient use:* Ensure efficient use of a computer‘s resources.
2. *User convenience:* Provide convenient methods of using a computer system.
3. *Non-interference:* Prevent interference in the activities of its users.

The goals of efficient use and user convenience sometimes conflict. For example, emphasis on quick service could mean that resources like memory have to remain allocated to a program even when the program is not in execution; however, it would lead to inefficient use of resources. When such conflicts arise, the designer has to make a trade-off to obtain the combination of efficient use and user convenience that best suits the environment. This is the notion of *effective utilization* of the computer system. We find a large number of operating systems in use because each one of them provides a different flavor of effective utilization. At one extreme we have OSs that provide fast service required by command and control applications, at the other extreme we have OSs that make efficient use of computer resources to provide low-cost computing, while in the middle we have OSs that provide different users. Such interference could be caused by both users and nonusers, and every OS must incorporate measures to prevent it. In the following, we discuss important aspects of these fundamental goals.

1.1.1 Efficient Use

An operating system must ensure efficient use of the fundamental computer system resources of memory, CPU, and I/O devices such as disks and printers. Poor efficiency can result if a program does not use a resource allocated to it, e.g., if memory or I/O devices allocated to a program remain idle. Such a situation may have a snowballing effect: Since the resource is allocated to a program, it is denied to other programs that need it. These programs cannot execute, hence resources allocated to them also remain idle. In addition, the OS itself consumes some CPU and memory resources during its own operation, and this consumption of resources constitutes an *overhead* that also reduces the resources available to user programs. To achieve good efficiency, the OS must minimize the waste of resources by programs and also minimize its own overhead. Efficient use of resources can be obtained by monitoring use of resources and performing corrective actions when necessary. However, monitoring use of resources increases the overhead, which lowers efficiency of use. In practice, operating systems that emphasize efficient use limit their overhead by either restricting their focus to efficiency of a few important resources, like the CPU and the memory, or by not monitoring the use of resources at all, and instead handling user programs and resources in a manner that guarantees high efficiency.

1.1.2 User Convenience

User convenience has many facets, as Table 1.1 indicates. In the early days of computing, user convenience was synonymous with bare necessity—the mere ability to execute a program written in a higher level language was considered adequate. Experience with early operating systems led to demands for better service, which in those days meant only fast response to a user request. Other facets of user convenience evolved with the use of computers in new fields. Early operating systems had *command-line interfaces*, which required a user to type in a command and specify values of its parameters. Users needed substantial training to learn use of the commands, which was acceptable because most users were scientists or computer professionals. However, simpler interfaces were needed to facilitate use of computers by new classes of users. Hence *graphical user interfaces* (GUIs) were evolved. These interfaces used *icons* on a screen to represent programs and files and interpreted mouse clicks on the icons and associated menus as commands concerning them

Table 1.1 Facets of User Convenience

Facet	Examples
Fulfillment of necessity	Ability to execute programs, use the file system
Good Service	Speedy response to computational requests
User friendly interfaces	Easy-to-use commands, graphical user interface (GUI)
New programming model	Concurrent programming
Web-oriented features	Means to set up complex computational structures
Evolution	Multi-threaded application servers

1.1.3 Non-interference

A computer user can face different kinds of interference in his computational activities. Execution of his program can be disrupted by actions of other persons, or the OS services which he wishes to use can be disrupted in a similar manner. The OS prevents such interference by allocating resources for exclusive use of programs and OS services, and preventing illegal accesses to resources. Another form of interference concerns programs and data stored in user files. A computer user may collaborate with some other users in the development or use of a computer application, so he may wish to share some of his files with them. Attempts by any other person to access his files are illegal and constitute interference. To prevent this form of interference, an OS has to know which files of a user can be accessed by which persons. It is achieved through the act of *authorization*, whereby a user specifies which collaborators can access what files. The OS uses this information to prevent illegal accesses to file

1.2 OPERATION OF AN OS

The primary concerns of an OS during its operation are execution of programs, use of resources, and prevention of interference with programs and resources. Accordingly, its three principal functions are:

- *Program management:* The OS initiates programs, arranges their execution on the CPU, and terminates them when they complete their execution. Since many programs exist in the system at any time, the OS performs a function called *scheduling* to select a program for execution.

- *Resource management:* The OS allocates resources like memory and I/O devices when a program needs them. When the program terminates, it deallocates these resources and allocates them to other programs that need them.

- *Security and protection:* The OS implements non-interference in users‘activities through joint actions of the security and protection functions. As an example, consider how the OS prevents illegal accesses to a file. The *security* function prevents nonusers from utilizing the services and resources in the computer system, hence none of them can access the file. The *protection* function prevents users other than the file owner or users authorized by him, from accessing the file.

Table 1.2 describes the tasks commonly performed by an operating system. When a computer system is switched on, it automatically loads a program stored on a reserved part of an I/O device, typically a disk, and starts executing the program. This program follows a software technique known as *bootstrapping* to load the software called the *boot procedure* in memory—the loads program initially loaded in memory some other programs in memory, which load other programs, and so on until the complete boot procedure is loaded. The boot procedure makes a list of all hardware resources in the system, and hands over control of the computer system to the OS

Task	When performed
Construct a list of resources	during booting
Maintain information for security	while registering new users
Verify identity of a user	at login time
Initiate execution of programs	at user commands
Maintain authorization information	When a user specifies which Collaborators can access what programs or data.
Perform resource allocation	when requested by users or programs

Table 1.2 Common Tasks Performed by Operating Systems

The following sections are a brief overview of OS responsibilities in managing programs and resources and in implementing security and protection. The following sections are a brief overview of OS responsibilities in managing programs and resources and in implementing security and protection.

1.2.1 Program Management

Modern CPUs have the capability to execute program instructions at a very high rate, so it is possible for an OS to interleave execution of several programs on a CPU and yet provide good user service. The key function in achieving interleaved execution of programs is *scheduling*, which decides which program should be given the CPU at any time. Figure 1.3 shows an abstract view of scheduling. The *scheduler*, which is an OS routine that performs scheduling, maintains a list of programs waiting to execute on the CPU, and selects one program for execution.

In operating systems that provide fair service to all programs, the scheduler also specifies how long the program can be allowed to use the CPU. The OS takes away the CPU from a program after it has executed for the specified period of time, and gives it to another program. This action is called *preemption*. A program that loses the CPU because of preemption is put back into the list of programs waiting to execute on the CPU. The scheduling policy employed by an OS can influence both efficient use of the CPU and user service. If a program is preempted after it has executed for only a short period of time, the overhead of scheduling actions would be high because of frequent preemption. However,each program would suffer only a short delay before it gets an opportunity to use the CPU, which would result in good user service. If preemption is performed after a program has executed for a longer period of time, scheduling overhead would be lesser but programs would suffer longer delays, so user service would be poorer.

1.2.2 Resource Management

Resource allocations and deallocations can be performed by using a resource table. Each entry in the table contains the name and address of a resource unit and its present status, indicating whether it is free or allocated to some program. Table 1.3 is such a table for management of I/O devices. It is constructed by the boot procedure by sensing the presence of I/O devices in the system, and updated by the operating system to reflect the allocations and deallocations made by it. Since any part of a disk can be accessed directly, it is possible to treat different parts

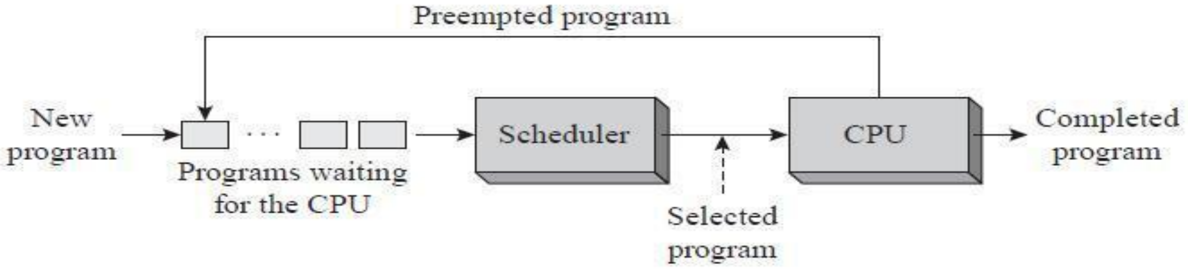


Figure 1.1 A schematic of scheduling

Table 1.3 Resource Table for I/O Devices

Resource name	Class	Address	Allocation status
printer1	Printer	101	Allocated to P1
printer2	Printer	102	Free
printer3	Printer	103	Free
disk1	Disk	201	Allocated to P1
disk2	Disk	202	Allocated to P2
cdw1	CD writer	301	Free

Virtual Resources A *virtual resource* is a fictitious resource—it is an illusion supported by an OS through use of a real resource. An OS may use the same real resource to support several virtual resources. This way, it can give the impression of having a larger number of resources than it actually does. Each use of a virtual resource results in the use of an appropriate real resource. In that sense, a virtual resource is an abstract view of a resource taken by a program.

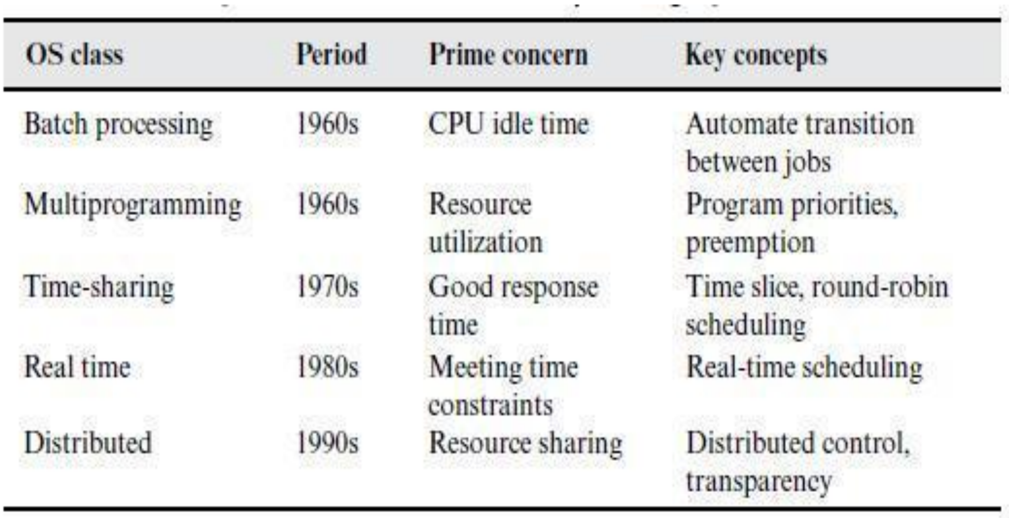
Use of virtual resources started with the use of virtual devices. To prevent mutual interference between programs, it was a good idea to allocate a device exclusively for use by one program. However, a computer system did not possess many real devices, so virtual devices were used. An OS would create a virtual device when a user needed an I/O device; e.g., the disks called disk1 and disk2 in Table 1.3 could be two virtual disks based on the real disk, which is allocated to programs P1 and P2, respectively. Virtual devices are used in contemporary operating systems as well. A print server is a common example of a virtual device.

When a program wishes to print a file, the print server simply copies the file into the print queue. The program requesting the print goes on with its operation as if the printing had been performed. The print server continuously examines the print queue and prints the files it finds in the queue. Most operating systems provide a virtual resource called *virtual memory*, which is an illusion of a memory that is larger in size than the real memory of a computer. Its use enables a programmer to execute a program whose size may exceed the size of real memory.

1.3 Classes of Operating Systems

Classes of operating systems have evolved over time as computer systems and users' expectations of them have developed; i.e., as computing environments have evolved. As we study some of the earlier classes of operating systems, we need to understand that each was designed to work with computer systems of its own historical period; thus we will have to look at architectural features representative of computer systems of the period. Table 1.4 lists five fundamental classes of operating systems that are named according to their defining features. The table shows when operating systems of each class first came into widespread use; what fundamental effectiveness criterion, or prime concern, motivated its development; and what key concepts were developed to address that prime concern.

Computing hardware was expensive in the early days of computing, so the batch processing and multiprogramming operating systems focused on efficient use of the CPU and other resources in the computer system. Computing environments were noninteractive in this era. In the 1970s, computer hardware became cheaper, so efficient use of a computer was no longer the prime concern and the focus shifted to productivity of computer users. Interactive computing environments were developed and time-sharing operating systems facilitated



OS class	Period	Prime concern	Key concepts
Batch processing	1960s	CPU idle time	Automate transition between jobs
Multiprogramming	1960s	Resource utilization	Program priorities, preemption
Time-sharing	1970s	Good response time	Time slice, round-robin scheduling
Real time	1980s	Meeting time constraints	Real-time scheduling
Distributed	1990s	Resource sharing	Distributed control, transparency

Table 1.4 Key Features of Classes of Operating Systems

better productivity by providing quick response to subrequests made to processes. The 1980s saw emergence of real-time applications for controlling or tracking of real-world activities, so operating systems had to focus on meeting the time constraints of such applications. In the 1990s, further declines in hardware costs led to development of distributed systems, in which several computer systems, with varying sophistication of resources, facilitated sharing of resources across their boundaries through networking.

The following paragraphs elaborate on key concepts of the five classes of operating systems mentioned in Table 1.4.

Batch Processing Systems: In a batch processing operating system, the prime concern is CPU efficiency. The batch processing system operates in a strict one job- at-a-time manner; within a job, it executes the programs one after another. Thus only one program is under execution at any time. The opportunity to enhance CPU efficiency is limited to efficiently initiating the next program when one program ends and the next job when one job ends, so that the CPU does not remain idle.

Multiprogramming Systems: A multiprogramming operating system focuses on efficient use of both the CPU and I/O devices. The system has several programs in a state of partial completion at any time. The OS uses *program priorities* and gives the CPU to the highest-priority program that needs it. It switches the CPU to a low-priority program when a high-priority program starts an I/O operation, and switches it back to the high-priority program at the end of the I/O operation. These actions achieve simultaneous use of I/O devices and the CPU.

Time-Sharing Systems: A time-sharing operating system focuses on facilitating quick response to subrequests made by *all* processes, which provides a tangible benefit to users. It is achieved by giving a fair execution opportunity to each process through two means: The OS services all processes by turn, which is called *round-robin scheduling*. It also prevents a process from using too much CPU time when scheduled to execute, which is called *time-slicing*. The combination of these two techniques ensures that no process has to wait long for CPU attention.

Real-Time Systems: A real-time operating system is used to implement a computer application for controlling or tracking of real-world activities. The application needs to complete its computational tasks in a timely manner to keep abreast of external events in the activity that it controls. To facilitate this, the OS permits a user to create several processes *within* an application program, and uses *real-time scheduling* to interleave the execution of processes such that the application can complete its execution within its time constraint.

Distributed Systems: A distributed operating system permits a user to access resources located in other computer systems conveniently and reliably. To enhance convenience, it does not expect a user to know the location of resources in the system, which is called *transparency*. To enhance efficiency, it may execute parts of a computation in different computer systems at the same time. It uses *distributed control*; i.e., it spreads its decision-making actions across different computers in the system so that failures of individual computers or the network does not cripple its operation.

1.4 BATCH PROCESSING SYSTEMS

Computer systems of the 1960s were non interactive. Punched cards were the primary input medium, so a job and its data consisted of a deck of cards. A computer operator would load the cards into the card reader to set up the execution of a job. This action wasted precious CPU time; batch processing was introduced to prevent this wastage.

A *batch* is a *sequence* of user jobs formed for processing by the operating system. A computer operator formed a batch by arranging a few user jobs in a sequence and inserting special marker cards to indicate the start and end of the batch. When the operator gave a command to initiate processing of a batch, the *batching kernel* set up the processing of the first job of the batch. At the end of the job, it initiated execution of the next job, and so on, until the end of the batch. Thus the operator had to intervene only at the start and end of a batch. Card readers and printers were a performance bottleneck in the 1960s, so batch processing systems employed the notion of virtual card readers and printers (described in Section 1.3.2) through magnetic tapes, to improve the system‘s throughput. A batch of jobs was first recorded on a magnetic tape, using a less powerful and cheap computer. The batch processing system processed these jobs from the tape, which was faster than processing them from cards, and wrote their results on another magnetic tape. These were later printed and released to users. Figure 1.2 shows the factors that make up the turnaround time of a job.

User jobs could not interfere with each other‘s execution directly because they did not coexist in a computer‘s memory. However, since the card reader was the only input device available to users, commands, user programs, and data were all derived from the card reader, so if a program in a job tried to read more data than provided in the job, it would read a few cards of the following job! To protect against such interference between jobs, a batch processing system required

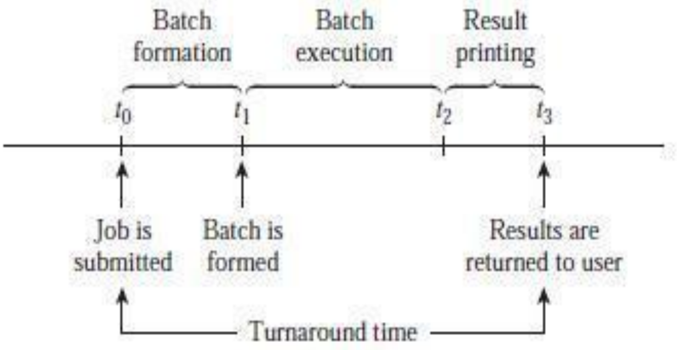


Figure 1.2 Turnaround time in a batch processing system.

// JOB → “Start of job” statement

// EXEC FORTRAN → Execute the Fortran compiler

} Fortran

program

// EXEC Execute just compiled program

} Data for

Fortran

program

/* → “End of data” statement

/& → “End of job” statement

Figure 1.3 Control statements in IBM 360/370 systems.

a user to insert a set of *control statements* in the deck of cards constituting a job. The *command interpreter*, which was a component of the batching kernel, read a card when the currently executing program in the job wanted the next card. If the card contained a control statement, it analyzed the control statement and performed appropriate actions; otherwise, it passed the card to the currently executing program. Figure 3.2 shows a simplified set of control statements used to compile and execute a Fortran program. If a program tried to read more data than provided, the command interpreter would read the /*, /& and // JOB cards. On seeing one of these cards, it would realize that the program was trying to read more cards than provided, so it would abort the job. A modern OS would not be *designed* for batch processing, but the technique is still useful in financial and scientific computation where the same kind of processing or analysis is to be performed on several sets of data. Use of batch processing in such environments would eliminate time-consuming initialization of the financial or scientific analysis separately for each set of data.

1.5 MULTIPROGRAMMING SYSTEM

Multiprogramming operating systems were developed to provide efficient resource utilization in a non interactive environment.

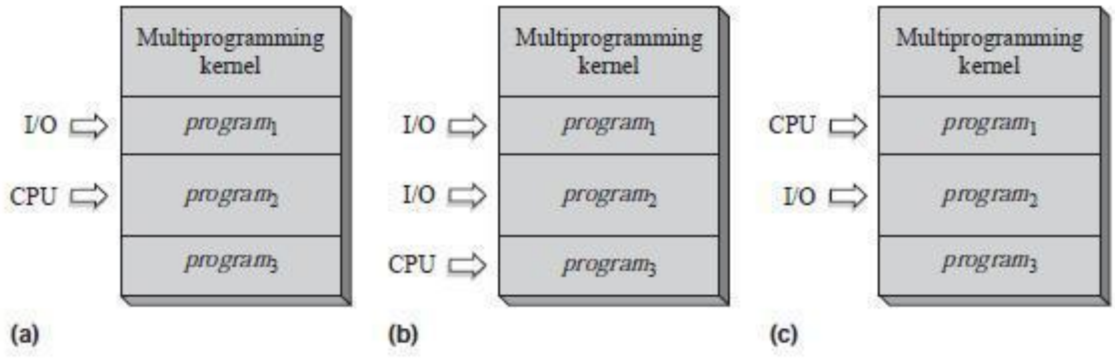


Figure 1.4 Operation of a multiprogramming system: (a) *program*2 is in execution while *program*1 is performing an I/O operation; (b) *program*2 initiates an I/O operation, *program*3 is scheduled; (c) *program*1's I/O operation completes and it is scheduled.

A multiprogramming OS has many user programs in the memory of the computer at any time, hence the name *multiprogramming*. Figure 1.4 illustrates operation of a multiprogramming OS. The memory contains three programs. An I/O operation is in progress for *program*1, while the CPU is executing *program*2. The CPU is switched to *program*3 when *program*2 initiates an I/O operation, and it is switched to *program*1 when *program*1's I/O operation completes. The multiprogramming kernel performs scheduling, memory management and I/O management.

A computer must possess the features summarized in Table 1.5 to support multiprogramming. The DMA makes multiprogramming feasible by permitting concurrent operation of the CPU and I/O devices. Memory protection prevents a program from accessing memory locations that lie outside the range of addresses defined by contents of the *base register* and *size register* of the CPU. The kernel and user modes of the CPU provide an effective method of preventing interference between programs.

The CPU initiates an I/O operation when an I/O instruction is executed. The DMA implements the data transfer involved in the I/O operation without involving the CPU and raises an I/O interrupt when the data transfer completes. Memory protection a program can access only the part of memory defined by contents of the *base register* and *size register*.

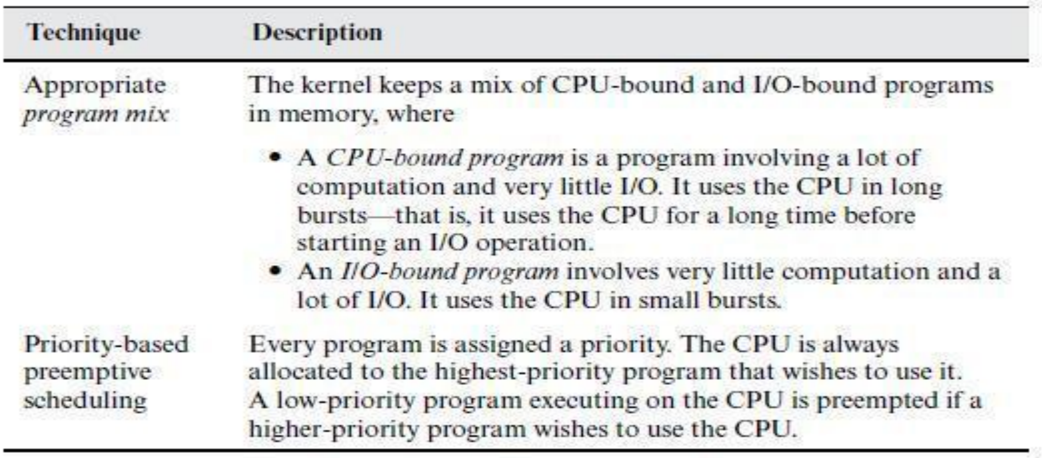
Kernel and user modes of CPU Certain instructions, called *privileged instructions,* can be performed only when the CPU is in the kernel mode. A program interrupt is raised if program tries to execute a privileged instruction when the CPU is in the user mode. The CPU is in the user mode; the kernel would abort the program while servicing this interrupt.

The turnaround time of a program is the appropriate measure of user service in a multiprogramming system. It depends on the total number of programs in the system, the manner in which the kernel shares the CPU between programs, and the program‘s own execution requirements.

1.5.1 Priority of Programs

An appropriate measure of performance of a multiprogramming OS is *throughput,* which is the ratio of the number of programs processed and the total time taken to process them. Throughput of a multiprogramming OS that processes *n* programs in the interval between times *t*0 and *tf* is *n/(tf* − *t*0*).* It may be larger than the throughput of a batch processing system because activities in several programs may take place simultaneously—one program may execute instructions on the CPU, while some other programs perform I/O operations. However, actual throughput depends on the nature of programs being processed, i.e., how much computation and how much I/O they perform, and how well the kernel can overlap their activities in time.

The OS keeps a sufficient number of programs in memory at all times, so that the CPU and I/O devices will have sufficient work to perform. This number is called the *degree of multiprogramming*. However, merely a high degree of multiprogramming cannot guarantee good utilization of both the CPU and I/O devices, because the CPU would be idle if each of the programs performed I/O operations most of the time, or the I/O devices would be idle if each of the programs performed computations most of the time. So the multiprogramming OS employs the two techniques described in Table 1.6 to ensure an overlap of CPU and I/O activities in programs: It uses an appropriate *program mix,* which ensures that some of the programs in memory are *CPU-bound programs*, which are programs that involve a lot of computation but few I/O operations, and others are *I/O-bound programs*, which contain very little computation but perform more I/O operations. This way, the programs being serviced have the potential to keep the CPU and I/O devices busy simultaneously. The OS uses the notion of *priority-based preemptive scheduling* to share the CPU among programs in a manner that would ensure good overlap of their CPU and I/O activities. We explain this technique in the following.



Technique	Description
Appropriate *program mix*	The kernel keeps a mix of CPU-bound and I/O-bound programs in memory, where • A *CPU-bound program* is a program involving a lot of computation and very little I/O. It uses the CPU in long bursts—that is, it uses the CPU for a long time before starting an I/O operation. • An *I/O-bound program* involves very little computation and a lot of I/O. It uses the CPU in small bursts.
Priority-based preemptive scheduling	Every program is assigned a priority. The CPU is always allocated to the highest-priority program that wishes to use it. A low-priority program executing on the CPU is preempted if a higher-priority program wishes to use the CPU.

Table 1.6 Techniques of Multiprogramming

The kernel assigns numeric priorities to programs. We assume that priorities are positive integers and a large value implies a high priority. When many programs need the CPU at the same time, the kernel gives the CPU to the program with the highest priority. It uses priority in a preemptive manner; i.e., it pre-empts a low-priority program executing on the CPU if a high-priority program needs the CPU. This way, the CPU is always executing the highest-priority program that needs it. To understand implications of priority-based preemptive scheduling, consider what would happen if a high-priority program is performing an I/O operation, a low-priority program is executing on the CPU, and the I/O operation of

the high-priority program completes—the kernel would immediately switch the CPU to the high-priority program. Assignment of priorities to programs is a crucial decision that can influence system throughput. Multiprogramming systems use the following priority assignment rule:

An I/O-bound program should have a higher priority than a CPU-bound program.

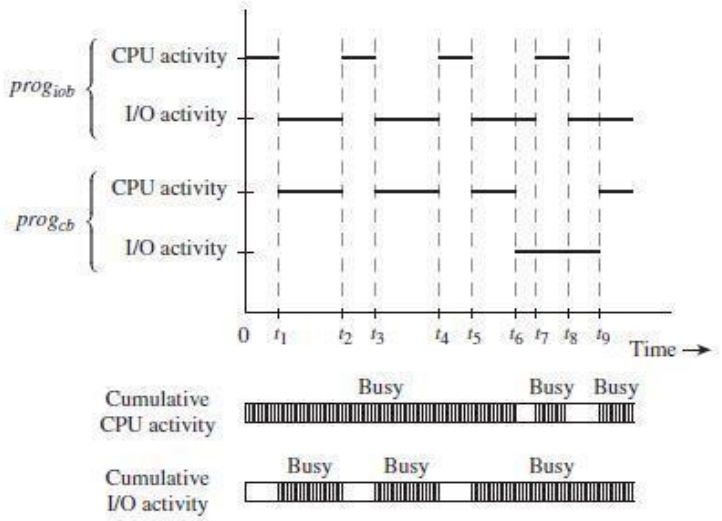
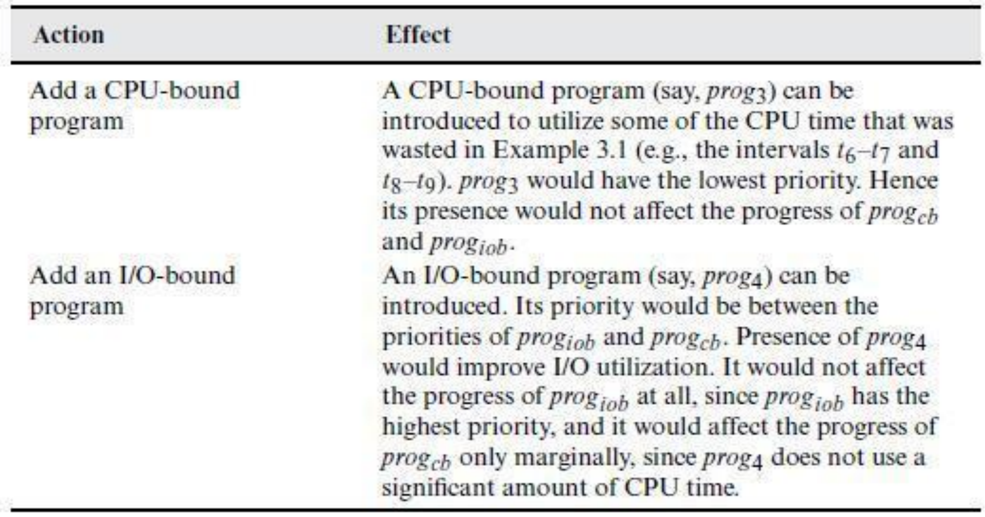


Figure 1.5 Timing chart when I/O-bound program has her priority.



Action	Effect
Add a CPU-bound program	A CPU-bound program (say, $prog_3$) can be introduced to utilize some of the CPU time that was wasted in Example 3.1 (e.g., the intervals t_6–t_7 and t_8–t_9). $prog_3$ would have the lowest priority. Hence its presence would not affect the progress of $prog_{cb}$ and $prog_{iob}$.
Add an I/O-bound program	An I/O-bound program (say, $prog_4$) can be introduced. Its priority would be between the priorities of $prog_{iob}$ and $prog_{cb}$. Presence of $prog_4$ would improve I/O utilization. It would not affect the progress of $prog_{iob}$ at all, since $prog_{iob}$ has the highest priority, and it would affect the progress of $prog_{cb}$ only marginally, since $prog_4$ does not use a significant amount of CPU time.

Table 1.7 Effect of Increasing the Degree of Multiprogramming

Table 1.7 describes how addition of a CPU-bound program can reduce CPU idling without affecting execution of other programs, while addition of an I/O-bound program can improve I/O utilization while marginally affecting execution of CPU-bound programs. The kernel can judiciously add CPU-bound or I/O-bound programs to ensure efficient use of resources.

When an appropriate program mix is maintained, we can expect that an increase in the degree of multiprogramming would result in an increase in throughput. Figure 1.6 shows how the throughput of a system actually varies with the degree of multiprogramming. When the degree of multiprogramming is 1, the throughput is dictated by the elapsed time of the lone program in the system. When more programs exist in the system, lower-priority programs also contribute to throughput. However, their contribution is limited by their opportunity to use the CPU. Throughput stagnates with increasing values of the degree of multiprogramming if low-priority programs do not get any opportunity to execute.

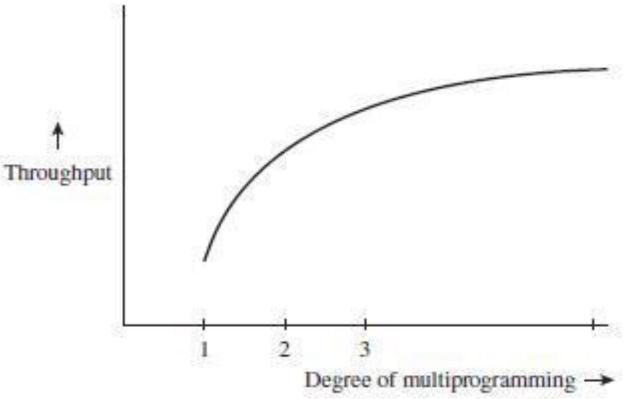


Figure 1.6 Variation of throughput with degree of multiprogramming.

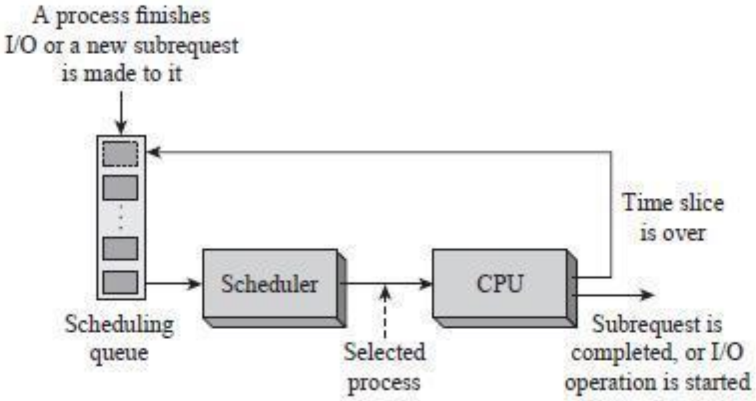

A process finishes I/O or a new subrequest is made to it
Time slice is over
Scheduler
CPU
Scheduling queue
Selected process
Subrequest is completed, or I/O operation is started

Figure 1.7 A schematic of round-robin scheduling with time-slicing.

1.6 TIME-SHARING SYSTEMS

In an interactive computing environment, a user submits a computational requirement—a subrequest—to a process and examines its response on the monitor screen. A time-sharing operating system is designed to provide a quick response to subrequests made by users. It achieves this goal by sharing the CPU time among processes in such a way that each process to which a subrequest has been made would get a turn on the CPU without much delay.

The scheduling technique used by a time-sharing kernel is called *round-robin scheduling with time-slicing*. It works as follows (see Figure 1.7): The kernel maintains a *scheduling queue* of processes that wish to use the CPU; it always schedules the process at the head of the queue. When a scheduled process completes servicing of a subrequest, or starts an I/O operation, the kernel removes it from the queue and schedules another process. Such a process would be added at the end of the queue when it receives a new subrequest, or when its I/O operation completes. This arrangement ensures that all processes would suffer comparable delays before getting to use the CPU. However, response times of processes would degrade if a process consumes too much CPU time in servicing its subrequest. The kernel uses the notion of a *time slice* to avoid this situation. We use the notation δ for the time slice.

Time Slice The largest amount of CPU time any time-shared process can consume when scheduled to execute on the CPU. If the time slice elapses before the process completes servicing of a subrequest, the kernel preempts the process, moves it to the end of the scheduling queue, and schedules another process. The preempted process would be rescheduled when it reaches the head of the queue once again.

The appropriate measure of user service in a time-sharing system is the time taken to service a subrequest, i.e., the response time (*rt*). It can be estimated in the following manner: Let the number of users using the system at any time be *n*. Let the complete servicing of each user subrequest require exactly δ CPU seconds, and let σ be the *scheduling overhead*; i.e., the CPU time consumed by the kernel to perform scheduling. If we assume that an I/O operation completes instantaneously and a user submits the next subrequest immediately after receiving a response to the previous subrequest, the response time (*rt*) and the CPU efficiency (η) are given by

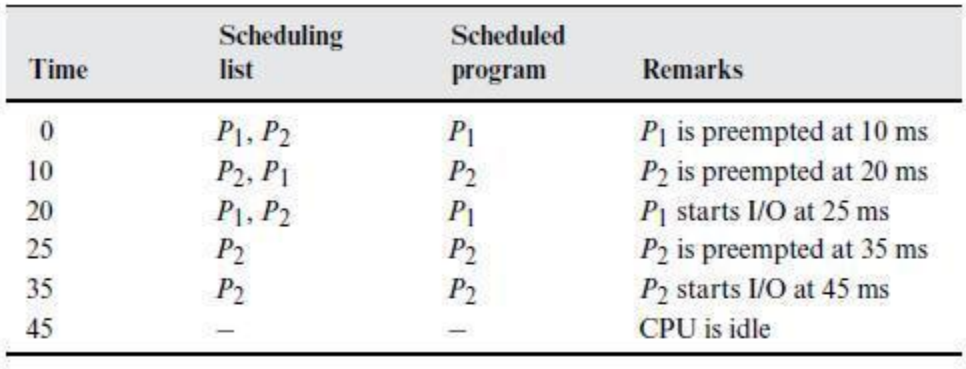
$$rt = n \times (\delta + \sigma) \qquad \textbf{(1.1)}$$

The actual response time may be different from the value of *rt* predicted by Eq. (1.1), for two reasons. First, all users may not have made subrequests to their processes. Hence *rt* would not be influenced by *n*, the total number of users in the system; it would be actually influenced by the number of active users. Second, user subrequests do not require exactly δ CPU seconds to produce a response. Hence the relationship of *rt* and η with δ is more complex than shown in Eqs (1.1) and (1.2).

1.6.1 Swapping of Programs

Throughput of subrequests is the appropriate measure of performance of a timesharing operating system. The time-sharing OS of Example 3.2 completes two subrequests in 125 ms, hence its throughput is 8 subrequests per second over the period 0 to 125 ms. However, the throughput would drop after 125 ms if users do not make the next subrequests to these processes immediately

.



Time	Scheduling list	Scheduled program	Remarks
0	P_1, P_2	P_1	P_1 is preempted at 10 ms
10	P_2, P_1	P_2	P_2 is preempted at 20 ms
20	P_1, P_2	P_1	P_1 starts I/O at 25 ms
25	P_2	P_2	P_2 is preempted at 35 ms
35	P_2	P_2	P_2 starts I/O at 45 ms
45	–	–	CPU is idle

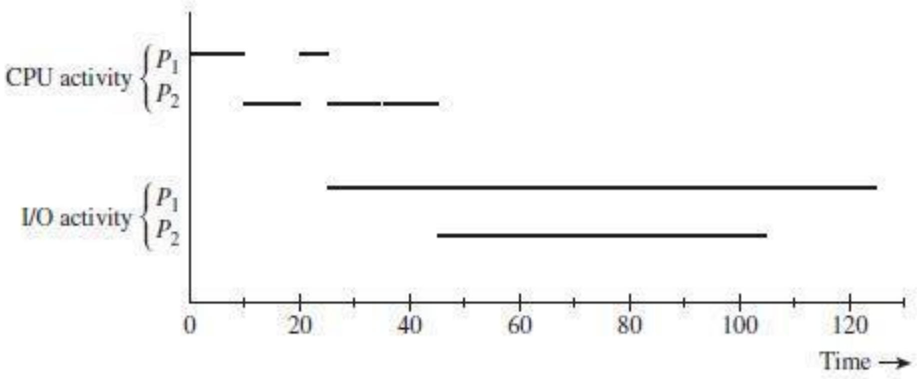


Figure 1.8 Operation of processes *P*1 and *P*2 in a time-sharing system.

The CPU is idle after 45 ms because it has no work to perform. It could have serviced a few more subrequests, had more processes been present in the system. But what if only two processes could fit in the computer‘s memory? The system throughput would be low and

response times of processes other than *P*1 and *P*2 would suffer. The technique of *swapping* is employed to service a larger number of processes than can fit into the computer‘s memory. It has the potential to improve both system performance and response times of processes.

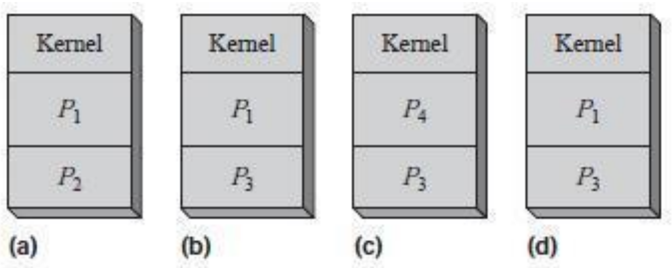


Figure 1.8 Swapping: (a) processes in memory between 0 and 105 ms; (b) *P*2 is replaced by *P*3 at 105 ms; (c) *P*1 is replaced by *P*4 at 125 ms; (d) *P*1 is swapped in to service the next subrequest made to it.

The kernel performs a *swap-out* operation on a process that is not likely to get scheduled in the near future by copying its instructions and data onto a disk. This operation frees the area of memory that was allocated to the process. The kernel now loads another process in this area of memory through a *swap-in* operation.

The kernel would overlap the swap-out and swap-in operations with servicing of other processes on the CPU, and a swapped-in process would itself get scheduled in due course of time. This way, the kernel can service more processes than can fit into the computer‘s memory. Figure 1.9 illustrates how the kernel employs swapping. Initially, processes *P*1 and

*P*2 exist in memory. These processes are swapped out when they complete handling of the subrequests made to them, and they are replaced by processes *P*3 and *P*4, respectively. The processes could also have been swapped out when they were preempted. A swapped-out process is swapped back into memory before it is due to be scheduled again, i.e., when it nears the head of the scheduling queue in Figure 1.7.

1.7 REAL-TIME OPERATING SYSTEMS

In a class of applications called *real-time applications*, users need the computer to perform some actions in a timely manner to control the activities in an external system, or to participate in them. The timeliness of actions is determined by the time constraints of the external system. Accordingly, we define a real-time application as follows:

If the application takes too long to respond to an activity, a failure can occur in the external system. We use the term *response requirement* of a system to indicate the largest value of response time for which the system can function perfectly; a timely response is one whose response time is not larger than the response requirement of the system.

Consider a system that logs data received from a satellite remote sensor. The satellite sends digitized samples to the earth station at the rate of 500 samples per second. The application process is required to simply store these samples in a file. Since a new sample arrives every two thousandth of a second, i.e., every 2 ms, the computer must respond to every ―store the sample‖ request in less than 2 ms, or the arrival of a new sample would wipe out the previous sample in the computer‘s memory. This system is a real-time application because a sample must be stored in less than 2 ms to prevent a failure. Its response requirement is 1.99 ms. The *deadline* of an action in a real-time application is the time by which the action should be performed. In the current example, if a new sample is received from the satellite at time *t*, the deadline for storing it on disk is *t* + 1.99 ms. Examples of real-time applications can be found in missile guidance, command and control applications like process control and air traffic control, data sampling and data acquisition systems like display systems in automobiles, multimedia systems, and applications like reservation and banking systems that employ large databases. The response requirements of these systems vary from a few microseconds or milliseconds for guidance and control systems to a few seconds for reservation and banking systems.

1.7.1 Hard and Soft Real-Time Systems

To take advantage of the features of real-time systems while achieving maximum cost effectiveness, two kinds of real-time systems have evolved. A *hard real-time system* is typically *dedicated* to processing real-time applications, and provably meets the response requirement of an application under all conditions. A *soft real-time system* makes the best effort to meet the response requirement of a real-time application but cannot guarantee that it will be able to meet it under all conditions. Typically, it meets the response requirements it

some probabilistic manner, say, 98 percent of the time. Guidance and control applications fail if they cannot meet the response requirement; hence they are serviced by hard real-time systems. Applications that aim at providing good quality of service, e.g., multimedia applications and applications like reservation and banking, do not have a notion of failure, so they may be serviced by soft real-time systems—the picture quality provided by a video-on-demand system may deteriorate occasionally, but one can still watch the video!

1.7.2 Features of a Real-Time Operating System

A real-time OS provides the features summarized in Table 3.7. The first three features help an application in meeting the response requirement of a system as follows: A real-time application can be coded such that the OS can execute its parts concurrently, i.e., as separate processes. When these parts are assigned priorities and priority-based scheduling is used, we have a situation analogous to multiprogramming *within* the application—if one part of the application initiates an I/O operation, the OS would schedule another part of the application. Thus, CPU and I/O activities of the application can be overlapped with one another, which help in reducing the duration of an application, i.e., its running time. *Deadline-aware scheduling* is a technique used in the kernel that schedules processes in such a manner that they may meet their deadlines.

Ability to specify *domain-specific events* and event handling actions enables a real-time application to respond to special conditions in the external system promptly.

Predictability of policies and overhead of the OS enables an application developer to calculate the worst-case running time of the application and decide whether the response requirement of the external system can be met.

A real-time OS employs two techniques to ensure continuity of operation when faults occur—*fault tolerance* and *graceful degradation*. A fault-tolerant computer system uses redundancy of resources to ensure that the system will keep functioning even if a fault occurs; e.g., it may have two disks even though the application actually needs only one disk. Graceful degradation is the ability of a system to fall back to a reduced level of service when a fault occurs and to revert to normal operations when the fault is rectified.

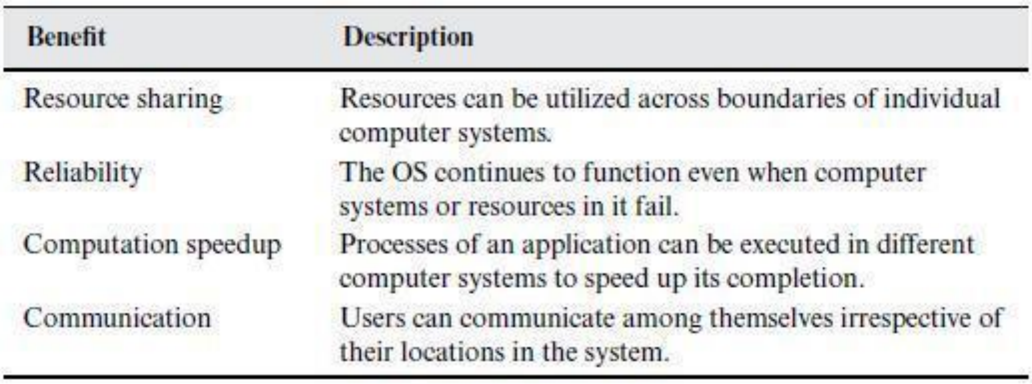
The programmer can assign high priorities to crucial functions so that they would be performed in a timely manner even when the system operates in a degraded mode.

1.8 DISTRIBUTED OPERATING SYSTEMS

A distributed computer system consists of several individual computer systems connected through a network. Each computer system could be a PC, a multiprocessor system or a *cluster*, which is itself a group of computers that work together in an integrated manner. Thus, many resources of a kind, e.g., many memories, CPUs and I/O devices, exist in the distributed system. A distributed operating system exploits the multiplicity of resources and the presence of a network to provide the benefits summarized in Table 1.9. However, the possibility of network faults or faults in individual computer systems complicates functioning of the operating system and necessitates use of special techniques in its design. Users also need to use special techniques to access resources over the network. *Resource sharing* has been the traditional motivation for distributed operating systems. A user of a PC or workstation can use resources such as printers over a local area network (LAN), and access specialized hardware or software resources of a geographically distant computer system over a wide area network (WAN).

A distributed operating system provides *reliability* through redundancy of computer systems, resources, and communication paths—if a computer system or a resource used in an application fails, the OS can switch the application to another computer system or resource, and if a path to a resource fails, it can utilize another path to the resource. Reliability can be used to offer high *availability* of resources and services, which is defined as the fraction of time a resource or service is operable. High availability of a data resource, e.g., a file, can be provided by keeping copies of the file in various parts of the system. *Computation speedup*

implies a reduction in the duration of an application, i.e., in its running time.



Benefit	Description
Resource sharing	Resources can be utilized across boundaries of individual computer systems.
Reliability	The OS continues to function even when computer systems or resources in it fail.
Computation speedup	Processes of an application can be executed in different computer systems to speed up its completion.
Communication	Users can communicate among themselves irrespective of their locations in the system.

Table 1.9 Benefits of Distributed Operating Systems

It is achieved by dispersing processes of an application to different computers in the distributed system, so that they can execute at the same time and finish earlier than if they were to be executed in a conventional OS. Users of a distributed operating system have user ids and passwords that are valid throughout the system. This feature greatly facilitates *communication* between users in two ways. First, communication through user ids automatically invokes the security mechanisms of the OS and thus ensures authenticity of communication. Second, users can be mobile within the distributed system and still be able to communicate with other users through the system.

1.8.1 Special Techniques of Distributed Operating Systems

A distributed system is more than a mere collection of computers connected to a network functioning of individual computers must be integrated to achieve the benefits summarized in Table 1.8. It is achieved through participation of all computers in the control functions of the operating system. Accordingly, we define a distributed system as follows

Distributed control is the opposite of centralized control—it implies that the control functions of the distributed system are performed by several computers in the system instead of being performed by a single computer. Distributed control is essential for ensuring that failure of a single computer, or a group of computers, does not halt operation of the entire system.

Transparency of a resource or service implies that a user should be able to access it without having to know which node in the distributed system contains it. This feature enables the OS to change the position of a software resource or service to optimize its use by applications.

For example, in a system providing transparency, a distributed file system could move a file to the node that contains a computation using the file, so that the delays involved in accessing the file over the network would be eliminated. The *remote procedure call* (RPC) invokes a procedure that executes in another computer in the distributed system. An application may employ the RPC feature to either performa part of its computation in another computer, which would contribute to computation speedup, or to access a resource located in that computer.

1.9 :Questions

1. Explain the benefits / features of distributed operating system.
2. Define an operating system . what are the different facets of user convenience?
3. Explain partition based and pool based resource allocation strategies.
4. Explain time sharing operating system with respect to
   i) Scheduling

ii) Memory management

5. What is OS? What are the common tasks performed by OS and when they are performed.
6. Explain turn around time in batch processing system.
7. Define distributed system. Give the key concepts and techniques used in distributed OS.
8. What are the two goals of an operating system ?explain briefly.
9. Describe the batch processing system and functions of scheduling and memory management for the same.
10. Why I/O bound programs should be given higher priorities in a multiprogramming environment? Illustrate with timing diagram.

1.11 FURTHER READINGS

- https://en.wikipedia.org/wiki/**operating_system**
- https:codex.cs.yale.edu/avi/os-book/OS8/os8c/slide-dir/PDF-dir/ch9.pdf
- https:www.tutorialspoint.com/operating_system/os_**operating_system**.html
- https:searchstorage.techtarget.com/definition/**classes of operating _system**

MODULE -2

PROCESS MANAGEMENT

Structure:

- OS View of Processes
- PCB
- Fundamental State Transitions,
- Threads
- Kernel and User level Threads
- Non-preemptive scheduling- FCFS and SRN
- Preemptive Scheduling- RR and LCN
- Long term, medium term and short term scheduling in a time sharing system

Objective:

On completion of this chapter student will be able to

- Discuss the concepts of process
- Discuss the concepts of PCB.
- Explain the operation fundamental state transistion.
- Explain the concepts of threads.
- Discuss the concept of kernel and user level threads.
- Discuss the operation of Non-preemptive and Preemptive Scheduling

/. This situation arises when several executions of a program are initiated, each with its own data, and when a program that is coded using concurrent programming techniques is in execution.

A programmer uses processes to achieve execution of programs in a sequential or concurrent manner as desired. An OS uses processes to organize execution of programs. Use of the process concept enables an OS to execute both sequential and concurrent programs equally easily.

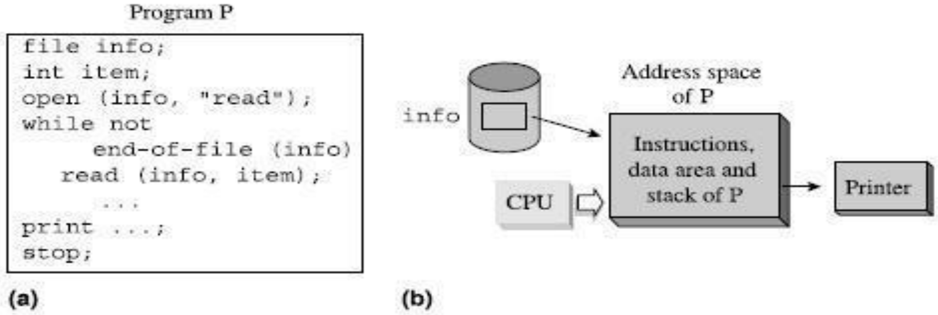
A thread is an execution of a program that uses the environment of a process, that is. its code, data and resources. If many threads use the environment of the same process, they share its code, data and resources. An OS uses this fact to reduce its overhead while switching between such threads.

2.1 OS OVERVIEW OF PROCESS

A program is a passive entity that does not perform any actions by itself; it has to be executed if the actions it calls for are to take place. A process is an execution of a program. It actually performs the actions specified in a program. An operating system shares the CPU among processes. This is how it gets user programs to execute.

To understand what a process is, let us discuss how the OS executes a program. Program P shown in Figure 2.1(a) contains declarations of file info and a variable item, and statements that read values from info, use them to perform some calculations, and print a result before coming to a halt.

During execution, instructions of this program use values in its data area and the stack to perform the intended calculations. Figure 2.1(b) shows an abstract view of its execution. The instructions, data, and stack of program P constitute its address space. To realize execution of P, the OS allocates memory to accommodate P‘s address space, allocates a printer to print its results, sets up an arrangement through which P can access file info, and schedules P for execution. The CPU is shown as a lightly shaded box because it is not always executing instructions of P—the OS shares the CPU between execution of P and executions of other programs.



Program P

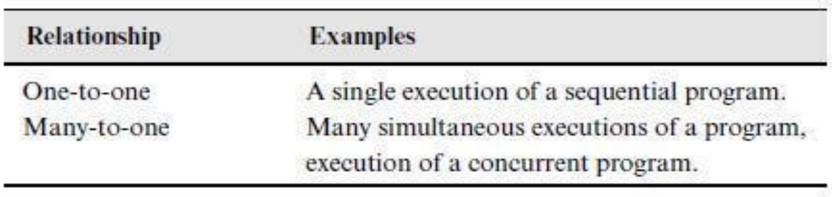
```
file info;
int item;
open (info, "read");
while not
     end-of-file (info)
   read (info, item);
      ...
print ...;
stop;
```

2.1.1 Relationships between Processes and Programs

A program consists of a set of functions and procedures. During its execution, control flows between the functions and procedures according to the logic of the program. Is an execution of a function or procedure a process? This doubt leads to the obvious question: what is the relationship between processes and programs?

The OS does not know anything about the nature of a program, including functions and procedures in its code. It knows only what it is told through system calls. The rest is under control of the program. Thus functions of a program may be separate processes, or they may constitute the code part of a single process.

Table 2.1 shows two kinds of relationships that can exist between processes and programs. A one-to-one relationship exists when a single execution of a sequential program is in progress, for example, execution of program P in Figure 2.1. A many-to-one relationship exists between many processes and a program in two cases: Many executions of a program may be in progress at the same time; processes representing these executions have a many-to-one relationship with the program. During execution, a program may make a system call to request that a specific part of its code should be executed concurrently, i.e., as a separate activity occurring at the same time. The kernel sets up execution of the specified part of the code and treats it as a separate process. The new process and the process representing execution of the program have a many-to-one relationship with the program. We call such a program a concurrent program. Processes that coexist in the system at some time are called concurrent processes. Concurrent processes may share their code, data and resources with other processes; they have opportunities to interact with one another during their execution.



Relationship	Examples
One-to-one	A single execution of a sequential program.
Many-to-one	Many simultaneous executions of a program, execution of a concurrent program.

Table 2.1 Relationships between Processes and Programs Relationship Examples

2.1.2 Child Processes

The kernel initiates an execution of a program by creating a process for it. For lack of a technical term for this process, we will call it the primary process for the program execution. The primary process may make system calls as described in the previous section to create other processes—these processes become its child processes, and the primary process becomes their parent.

A child process may itself create other processes, and so on. The parent–child relationships between these processes can be represented in the form of a process tree, which has the primary process as its root. A child process may inherit some of the resources of its parent; it could obtain additional resources during its operation through system calls.

Typically, a process creates one or more child processes and delegates some of its work to each of them. It is called multitasking within an application. Creation of child processes has the same benefits as the use of multiprogramming in an OS—the kernel may be able to interleave operation of I/O-bound and CPU-bound processes in the application, which may lead to a reduction in the duration, i.e., running time, of an application. It is called computation speedup. Most operating systems permit a parent process to assign priorities to child processes. A real-time application can assign a high priority to a child process that performs a critical function to ensure that its response requirement is met.

The third benefit, namely, guarding a parent process against errors in a child process, arises as follows: Consider a process that has to invoke an untrusted code. If the untrusted code were to be included in the code of the process, an error in the untrusted code would compel the kernel to abort the process; however, if the process were to create a child process to execute the untrusted code, the same error would lead to the abort of the child process, so the parent process would not come to any harm. The OS command interpreter uses this feature to advantage. The command interpreter itself runs as a process, and creates a child process whenever it has to execute a user program. This way, its own operation is not harmed by malfunctions in the user program.

2.1.3 Programmer view of processes

Child Processes in a Real-Time Application

Example 2.1: The real-time data logging application of Section 2.2 receives data samples from a satellite at the rate of 500 samples per second and stores them in a file. We assume that each sample arriving from the satellite is put into a special register of the computer. The primary process of the application, which we will call the data_logger process, has to perform the following three functions:

1. Copy the sample from the special register into memory.
2. Copy the sample from memory into a file.
3. Perform some analysis of a sample and record its results into another file used for future processing.

It creates three child processes named copy_sample, record_sample, and housekeeping, leading to the process tree shown in Figure 2.2(a). Note that a process is depicted by a circle and a parent–child relationship is depicted by an arrow. As shown in Figure 2.2(b), copy_sample copies the sample from the register into a memory area named buffer_area that can hold, say, 50 samples. record_sample writes a sample from buffer_area into a file. housekeeping analyzes a sample from buffer_area and records its results in another file. Arrival of a new sample causes an interrupt, and a programmer-defined interrupt servicing routine is associated with this interrupt. The kernel executes this routine whenever a new sample arrives. It activates copy_sample.

Operation of the three processes can overlap as follows: copy_sample can copy a sample into buffer_area, record_sample can write a previous sample to the file, while housekeeping can analyze it and write its results into the other file. This arrangement provides a smaller worst-case response time of the application than if these functions were to be executed sequentially. So long as buffer_area has some free space, only copy_sample has to complete before the next sample arrives. The other processes can be executed later. This possibility is exploited by assigning the highest priority to copy_sample.

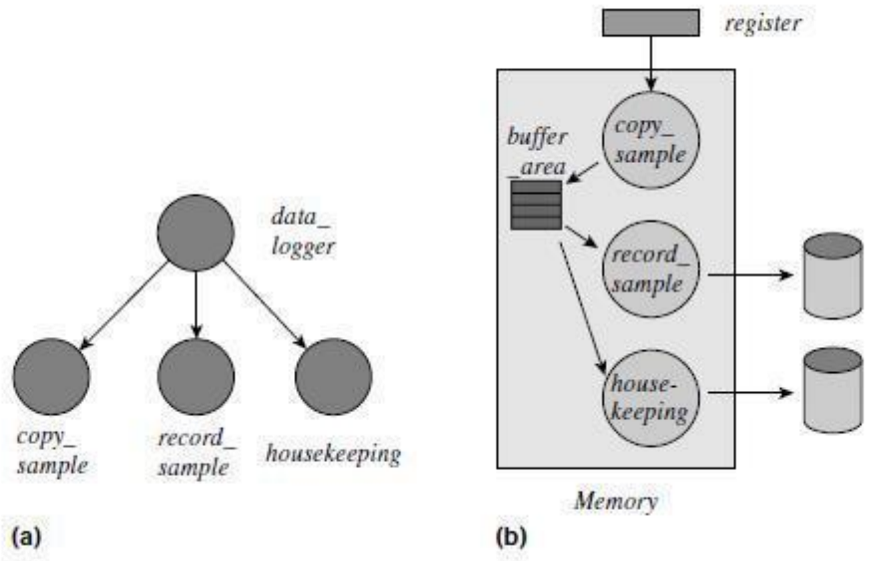


Figure 2.2 Real-time application of (a) process tree; (b) processes.

To facilitate use of child processes, the kernel provides operations for:

1. Creating a child process and assigning a priority to it
2. Terminating a child process
3. Determining the status of a child process
4. Sharing, communication, and synchronization between processes

Their use can be described as follows: In Example 2.1, the data_logger process creates three child processes. The copy_sample and record_sample processes share buffer_area. They need to synchronize their operation such that process record_sample would copy a sample out of buffer_area only after process copy_sample has written it there. The data_logger process could be programmed to either terminate its child processes before itself terminating, or terminate itself only after it finds that all its child processes have terminated.

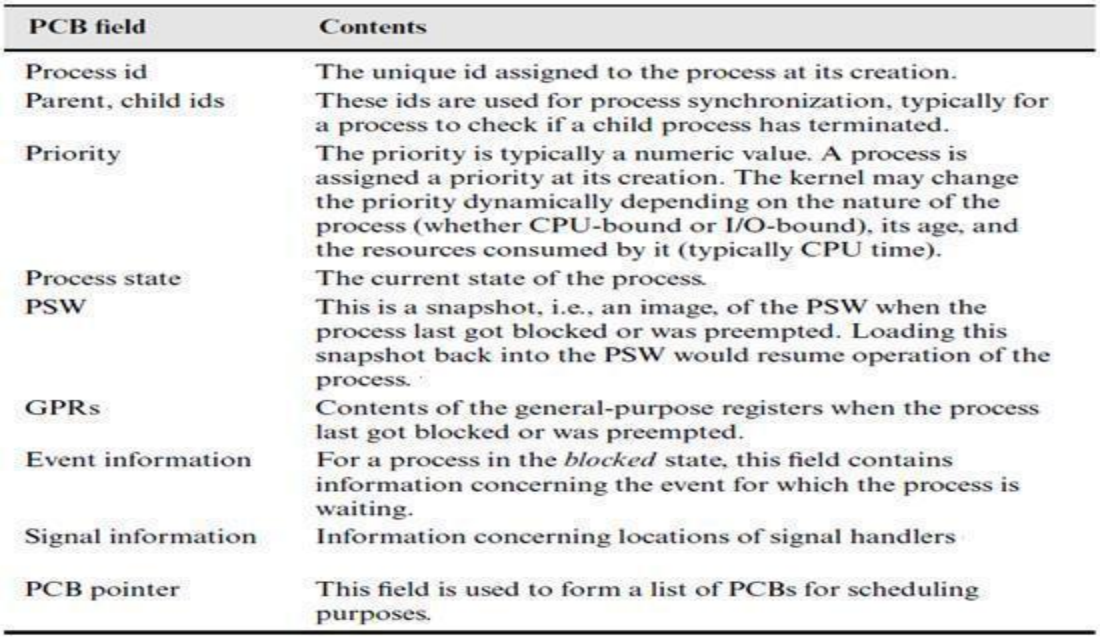
2.2 PCB

The process control block (PCB) of a process contains three kinds of information concerning the process—identification information such as the process id, id of its parent process, and id of the user who created it; process state information such as its state, and the contents of the PSW and the general-purpose registers (GPRs); and information that is useful in controlling its operation, such as its priority, and its interaction with other processes. It also contains a pointer field that is used by the kernel to form PCB lists for scheduling, e.g., a list of ready processes. Table 2.2 describes the fields of the PCB data structure.

The priority and state information is used by the scheduler. It passes the id of the selected process to the dispatcher. For a process that is not in the running state, the PSW and

GPRs fields together contain the CPU state of the process when it last got blocked or was pre-empted. Operation of the process can be resumed by simply loading this information from its PCB into the CPU. This action would be performed when this process is to be dispatched.

When a process becomes blocked, it is important to remember the reason. It is done by noting the cause of blocking, such as a resource request or an I/O operation, in the event information field of the PCB. Consider a process Pi that is blocked on an I/O operation on device d. The event information field in Pi ‘s PCB indicates that it awaits end of an I/O operation on device d. When the I/O operation on device d completes, the kernel uses this information to make the transition blocked→ready for process Pi .



PCB field	Contents
Process id	The unique id assigned to the process at its creation.
Parent, child ids	These ids are used for process synchronization, typically for a process to check if a child process has terminated.
Priority	The priority is typically a numeric value. A process is assigned a priority at its creation. The kernel may change the priority dynamically depending on the nature of the process (whether CPU-bound or I/O-bound), its age, and the resources consumed by it (typically CPU time).
Process state	The current state of the process.
PSW	This is a snapshot, i.e., an image, of the PSW when the process last got blocked or was preempted. Loading this snapshot back into the PSW would resume operation of the process.
GPRs	Contents of the general-purpose registers when the process last got blocked or was preempted.
Event information	For a process in the *blocked* state, this field contains information concerning the event for which the process is waiting.
Signal information	Information concerning locations of signal handlers
PCB pointer	This field is used to form a list of PCBs for scheduling purposes.

Table 2.2 Fields of the Process Control Block (PCB)

2.3 FUNDAMENTAL STATE TRANSITION

An operating system uses the notion of a process state to keep track of what a process is doing at any moment. The kernel uses process states to simplify its own functioning, so the number of process states and their names may vary across OSs. However, most OSs use the four fundamental states described in Table 2.3. The kernel considers a process to be in theblocked state if it has made a resource request and the request is yet to be granted, or if it is waiting for some event to occur. A CPU should not be allocated to such a process until its wait is complete. The kernel would change the state of the process to ready when the request is granted or the event for which it is waiting occurs. Such a process can be considered for scheduling. The kernel would change the state of the process to running when it is dispatched. The state would be changed to terminated when execution of the process completes or when it is aborted by the kernel for some reason.

A conventional computer system contains only one CPU, and so at most one process can be in the running state. There can be any number of processes in the blocked, ready, and terminated states. An OS may define more process states to simplify its own functioning or to support additional functionalities like swapping.

Process State Transitions A state transition for a process Pi is a change in its state. A state transition is caused by the occurrence of some event such as the start or end of an I/O operation. When the event occurs, the kernel determines its influence on activities in processes, and accordingly changes the state of an affected process.

When a process Pi in the running state makes an I/O request, its state has to be changed to blocked until its I/O operation completes. At the end of the I/O operation, Pi ‘s state is changed from blocked to ready because it now wishes to use the CPU. Similar state changes are made when a process makes some request that cannot immediately be satisfied by the OS. The process state is changed to blocked when the request is made, i.e., when the request event occurs, and it is changed to ready when the request is satisfied. The state of a ready process is changed to running when it is dispatched, and the state of a running process is changed to ready when it is preempted either because a higher-priority process became ready or because its time slice elapsed. Table 2.3 summarizes causes of state transitions.

Figure 2.3 diagrams the fundamental state transitions for a process. A new process is put in the ready state after resources required by it have been allocated. It may enter the running, blocked, and ready states a number of times as a result of events described in Table 2.4. Eventually it enters the terminated state.

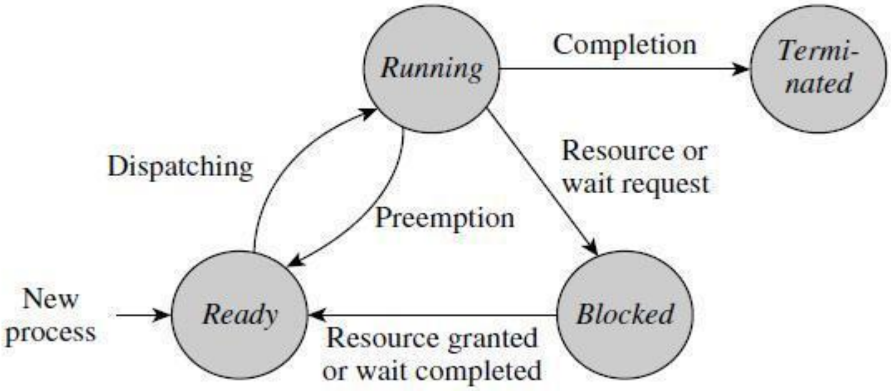
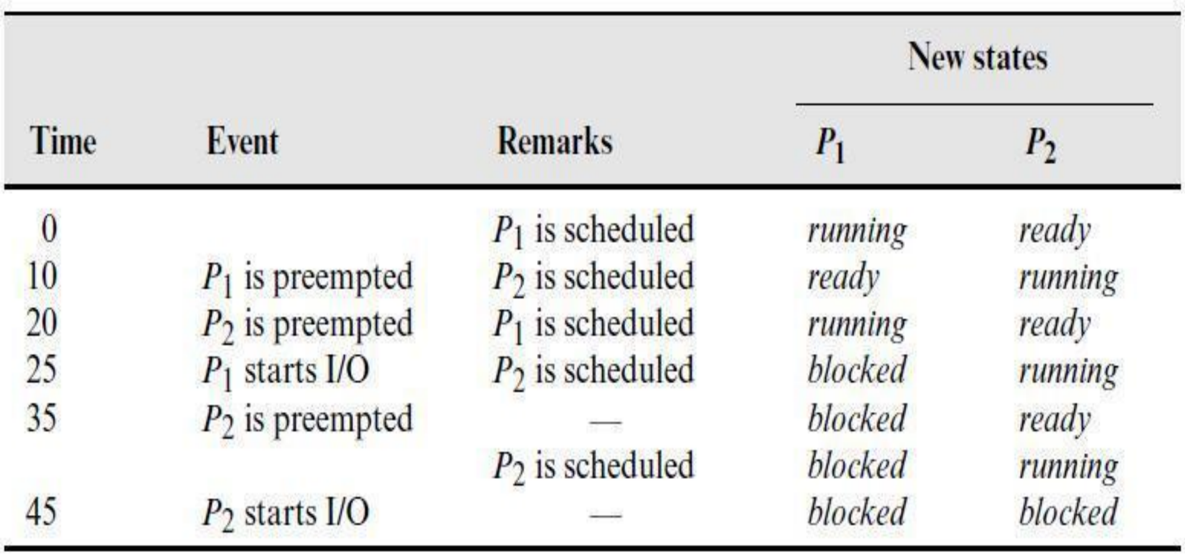


Figure 2.3 Fundamental state transitions for a process.



			New states	
Time	Event	Remarks	P_1	P_2
0		P_1 is scheduled	*running*	*ready*
10	P_1 is preempted	P_2 is scheduled	*ready*	*running*
20	P_2 is preempted	P_1 is scheduled	*running*	*ready*
25	P_1 starts I/O	P_2 is scheduled	*blocked*	*running*
35	P_2 is preempted	—	*blocked*	*ready*
		P_2 is scheduled	*blocked*	*running*
45	P_2 starts I/O	—	*blocked*	*blocked*

Table 2.3 Fundamental state transitions for a process.

2.2.1 Suspended Processes

A kernel needs additional states to describe the nature Of the activity of a process that is not in one of the four fundamental states described earlier.

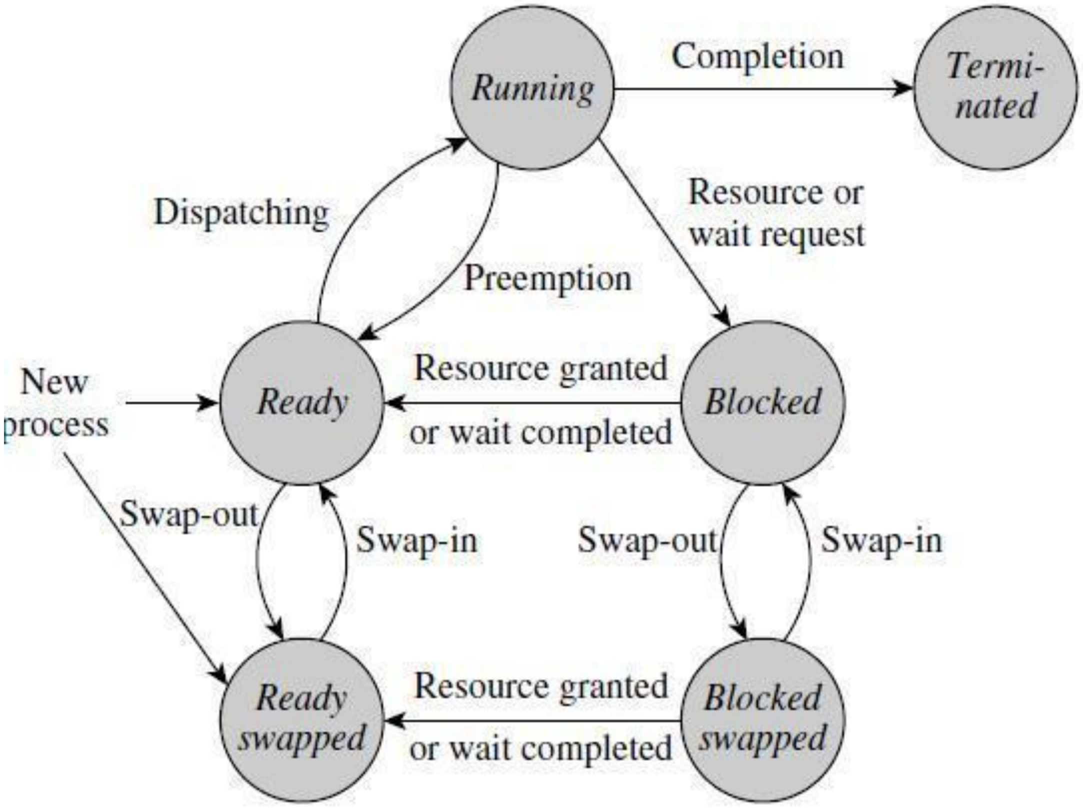


Figure 2.4 Process states and state transitions using two swapped states.

Consider a process that was in the ready or the blocked state when it got swapped out of memory. The process needs to be swapped back into memory before it can resume its activity. Hence it is no longer in the ready or blocked state; the kernel must define a new state for it. We call such a process a suspended process. If a user indicates that his process should not be considered for scheduling for a specific period of time, it, too, would become a suspended process. When a suspended process is to resume its old activity, it should go back to the state it was in when it was suspended. To facilitate this state transition, the kernel may define many suspend states and put a suspended process into the appropriate suspend state. We restrict the discussion of suspended processes to swapped processes and use two suspend states called ready swapped and blocked swapped. Accordingly, Figure 2.5 shows process states and state transitions. The transition ready → ready swapped or blocked → blocked swapped is caused by a swap-out action.

The reverse state transition takes place when the process is swapped back into memory. The blocked swapped → ready swapped transition takes place if the request for which the process was waiting is granted even while the process is in a suspended state, for example, if a resource for which it was blocked is granted to it. However, the process continues to be swapped out.

When it is swapped back into memory, its state changes to ready and it competes with other ready processes for the CPU. A new process is put either in the ready state or in the ready swapped state depending on availability of memory.

2.3 THREADS

Applications use concurrent processes to speed up their operation. However, switching between processes within an application incurs high process switching overhead because the size of the process state information is large, so operating system designers developed an alternative model of execution of a program, called a thread, that could provide concurrency within an application with less overhead.

To understand the notion of threads, let us analyze process switching overhead and see where a saving can be made. Process switching overhead has two components:

- Execution related overhead: The CPU state of the running process has to be saved and the CPU state of the new process has to be loaded in the CPU. This overhead is unavoidable.

- Resource-use related overhead: The process context also has to be switched. It involves switching of the information about resources allocated to the process, such as memory and files, and interaction of the process with other processes. The large size of this information adds to the process switching overhead.

Consider child processes P_i and P_j of the primary process of an application. These processes inherit the context of their parent process. If none of these processes have allocated any resources of their own, their context is identical; their state information differs only in their CPU states and contents of their stacks. Consequently, while switching between P_i andP_j, much of the saving and loading of process state information is redundant. Threads exploit this feature to reduce the switching overhead.

A process creates a thread through a system call. The thread does not have resources of its own, so it does not have a context; it operates by using the context of the process, and accesses the resources of the process through it. We use the phrases ―"thread(s) of a process" and ―"parent process of a thread" to describe the relationship between a thread and the process whose context it uses.

Figure 2.5 illustrates the relationship between threads and processes. In the abstract view of Figure 2.5(a), process P_i has three threads, which are represented by wavy linesinside the circle representing process P_i. Figure 2.5 (b) shows an implementation arrangement. Process P_i has a context and a PCB. Each thread of P_i is an execution of a program, so it has its own stack and a thread control block (TCB), which is analogous to the PCB and stores the following information:

1. Thread scheduling information—thread id, priority and state.
2. CPU state, i.e., contents of the PSW and GPRs.
3. Pointer to PCB of parent process.
4. TCB pointer, which is used to make lists of TCBs for scheduling.

Use of threads effectively splits the process state into two parts—the resource state remains with the process while an execution state, which is the CPU state, is associated with a thread. The cost of concurrency within the context of a process is now merely replication of the execution state for each thread. The execution states need to be switched during switching between threads. The resource state is neither replicated nor switched during switching between threads of the process.

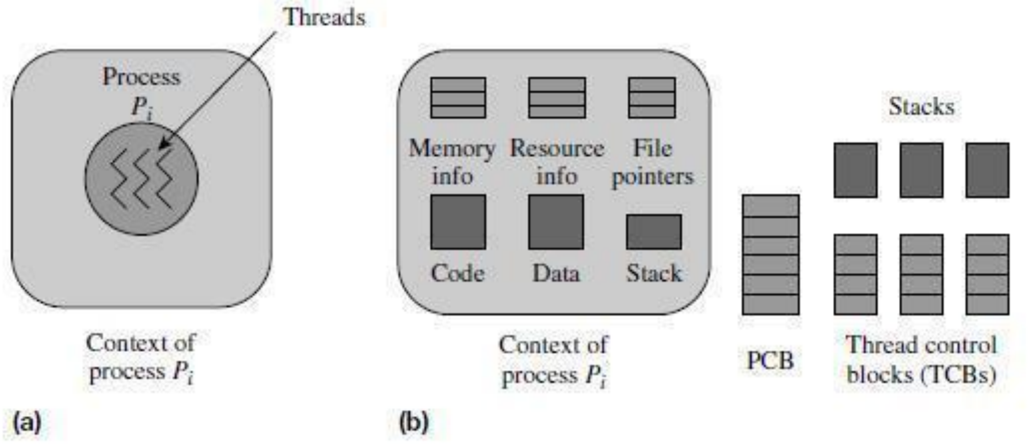


Figure 2.5 Threads in process P_i: (a) concept; (b) implementation.

2.2.1 Thread States and State Transitions

Barring the difference that threads do not have resources allocated to them, threads and processes are analogous. Hence thread states and thread state transitions are analogous to process states and process state transitions. When a thread is created, it is put in the ready state already has the necessary resources allocated to it. It enters the running state when it is dispatched. It does not enter the blocked state because of resource requests, because it does not make any resource requests; however, it can enter the blocked state because of process synchronization requirements.

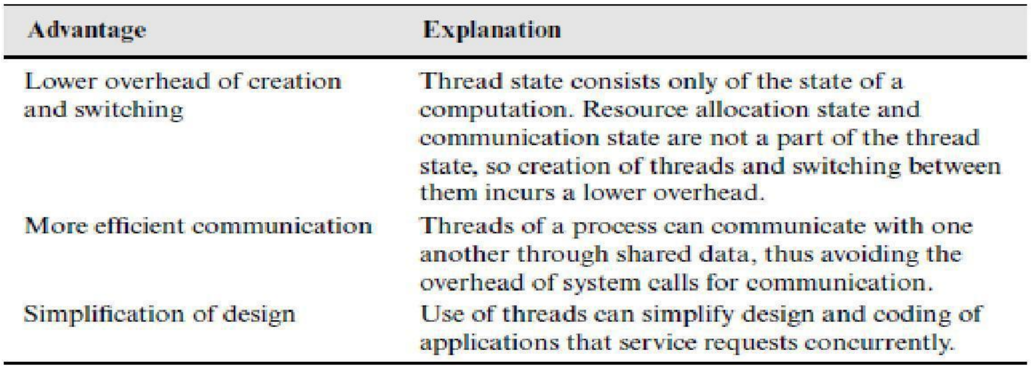
Advantages of Threads over Processes

Table 2.5 summarizes the advantages of threads over processes, of which we have already discussed the advantage of lower overhead of thread creation and switching. Unlike child processes, threads share the address space of the parent process, so they can communicate through shared data rather than through messages, thereby eliminating the overhead of system calls.

Applications that service requests received from users, such as airline reservation systems or banking systems, are called servers; their users are called clients. Performance of servers can be improved through concurrency or parallelism, i.e., either through interleaving of requests that involve I/O operations or through use of many CPUs to service different requests. Use of threads simplifies their design;

Figure 2.6(a) is a view of an airline reservation server. The server enters requests made by its clients in a queue and serves them one after another. If several requests are to be serviced concurrently, the server would have to employ advanced I/O techniques such as asynchronous I/O, and use complex logic to switch between the processing of requests. By contrast, a multithreaded server could create a new thread to service each new request it receives, and terminate the thread after servicing the request.

Table 2.5 Advantages of Threads over Processes



Advantage	Explanation
Lower overhead of creation and switching	Thread state consists only of the state of a computation. Resource allocation state and communication state are not a part of the thread state, so creation of threads and switching between them incurs a lower overhead.
More efficient communication	Threads of a process can communicate with one another through shared data, thus avoiding the overhead of system calls for communication.
Simplification of design	Use of threads can simplify design and coding of applications that service requests concurrently.

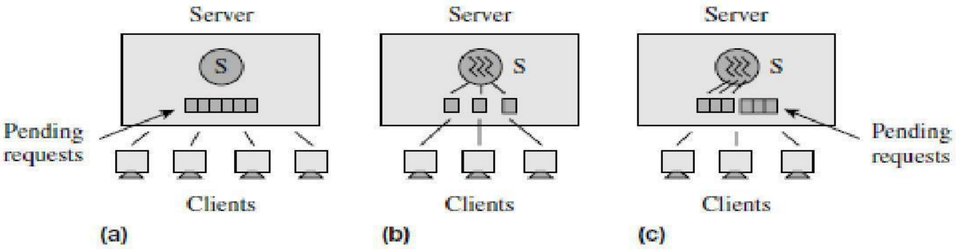


Figure 2.6 Use of threads in structuring a server: (a) server using sequential code; (b) multithreaded server; (c) server using a thread pool.

2.4 KERNEL-LEVEL AND USER-LEVEL

These two models of threads differ in the role of the process and the kernel in the creation and management of threads. This difference has a significant impact on the overhead of thread switching and the concurrency and parallelism within a process.

2.4.1 Kernel-Level Threads

A kernel-level thread is implemented by the kernel. Hence creation and termination of kernel-level threads, and checking of their status, is performed through system calls. Figure 2.7 shows a schematic of how the kernel handles kernel-level threads. When a process makes a create_thread system call, the kernel creates a thread, assigns an id to it, and allocates a thread control block (TCB). The TCB contains a pointer to the PCB of the parent process of the threads.

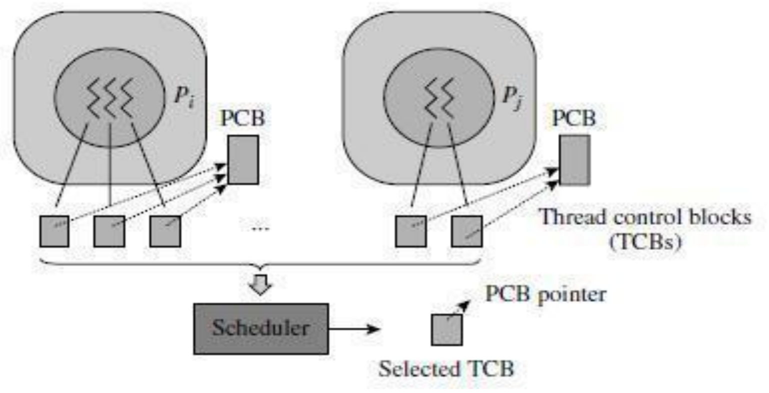


Figure 2.7 Scheduling of kernel-level threads.

TCB to check whether the selected thread belongs to a different process than the interrupted thread. If so, it saves the context of the process to which the interrupted thread belongs, and loads the context of the process to which the selected thread belongs. It then dispatches the selected thread. However, actions to save and load the process context are skipped if both threads belong to the same process. This feature reduces the switching overhead, hence switching between kernel-level threads of a process could be as much as an order of magnitude faster, i.e., 10 times faster, than switching between processes.

Advantages and Disadvantages of Kernel-Level Threads

A kernel-level thread is like a process except that it has a smaller amount of state information. This similarity is convenient for programmers—programming for threads is no different from programming for processes. In a multiprocessor system, kernel-level threads provide parallelism, as many threads belonging to a process can be scheduled simultaneously, which is not possible with the user-level threads described in the next section, so it provides better computation speedup than user-level threads.

However, handling threads like processes has its disadvantages too. Switching between threads is performed by the kernel as a result of event handling. Hence it incurs the overhead of event handling even if the interrupted thread and the selected thread belong to the same process. This feature limits the savings in the thread switching overhead.

2.4.2 User-Level Threads

User-level threads are implemented by a thread library, which is linked to the code of a process. The library sets up the thread implementation arrangement without involving the kernel, and itself interleaves operation of threads in the process. Thus, the kernel is not aware of presence of user-level threads in a process; it sees only the process. Most OSs implement the pthreads application program interface provided in the IEEE POSIX standard in this manner.

Scheduling of User-Level Threads

Figure 2.8 is a schematic diagram of scheduling of user-level threads. The thread library code is a part of each process. It performs "scheduling" to select a thread, and organizes its execution. We view this operation as "mapping" of the TCB of the selected thread into the PCB of the process.

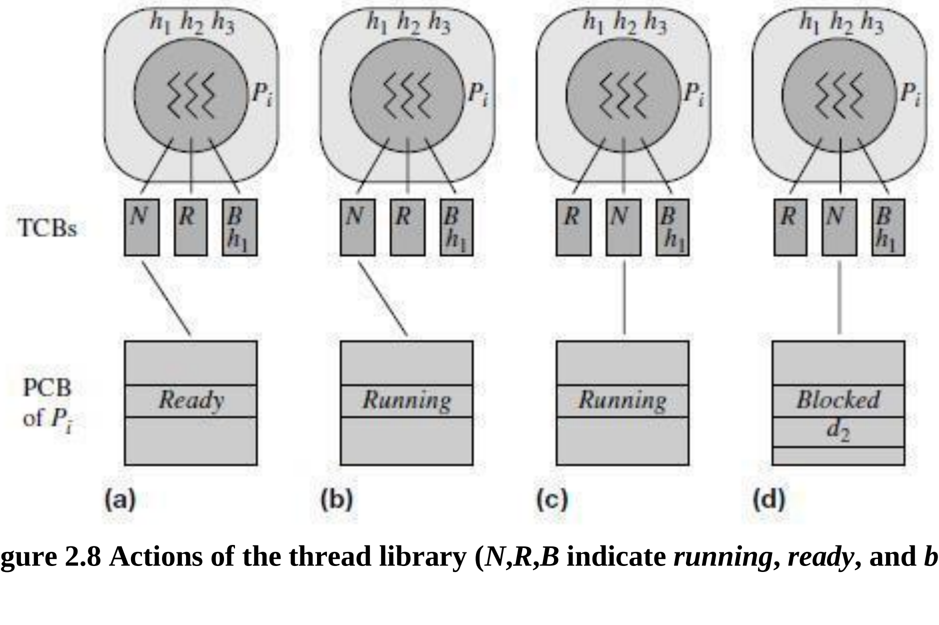


Figure 2.8 Actions of the thread library (*N*,*R*,*B* indicate *running*, *ready*, and *blocked*).

The thread library uses information in the TCBs to decide which thread should operate at any time. To ―dispatch‖ the thread, the CPU state of the thread should become the CPU state of the process, and the process stack pointer should point to the thread‗s stack. Since the thread library is a part of a process, the CPU is in the user mode. Hence a thread cannot be dispatched by loading new information into the PSW; the thread library has to use nonprivileged instructions to change PSW contents. Accordingly, it loads the address of the thread‗s stack into the stack address register, obtains the address contained in the program counter (PC) field of the thread‗s CPU state found in its TCB, and executes a branch instruction to transfer control to the instruction which has this address.

Advantages and Disadvantages of User-Level Threads

Thread synchronization and scheduling is implemented by the thread library. This arrangement avoids the overhead of a system call for synchronization between threads, so the thread switching overhead could be as much as an order of magnitude smaller than in kernel-level threads.

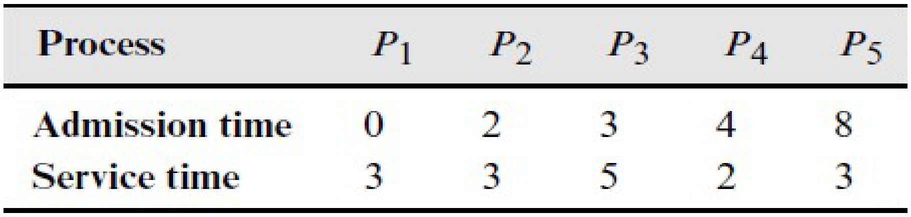
2.5 NON PRE EMPTIVE SCHEDULING

In **non pre emptive scheduling**, a server always services a scheduled request to completion. Thus, scheduling is performed only when servicing of a previously scheduled request is completed and so pre emption of a request as shown in Figure 7.1 never occurs. Non pre emptive scheduling is attractive because of its simplicity the scheduler does not have to distinguish between a un serviced request and a partially serviced one. Since a user service or system performance is reordering of requests. The three non pre emptive scheduling policies are:

- First-come, first-served (FCFS) scheduling
- Shortest request next (SRN) scheduling

FCFS Scheduling

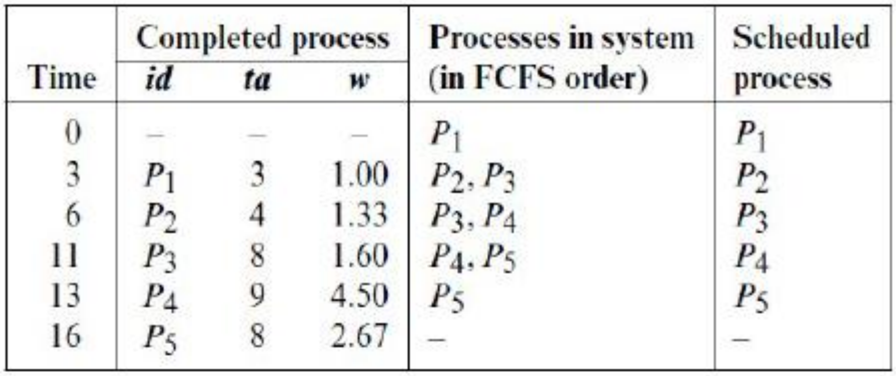
Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organized as a queue. The scheduler always schedules the first request in the list. An example of FCFS scheduling is a batch processing system in which jobs are ordered according to their arrival times (or arbitrarily, if they arrive at exactly the same time) and Results of a job are released to the user immediately on completion of the job. The following example illustrates operation of an FCFS scheduler.



Process	P_1	P_2	P_3	P_4	P_5
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

Table 2.6 Processes for Scheduling

Table 7.2 illustrates the scheduling decisions made by the FCFS scheduling policy for the processes of Table 2.6. Process P1 is scheduled at time 0. The pending list contains P2 andP3 when P1 completes at 3 seconds, so P2 is scheduled. The Completed column shows the id of the completed process and its turnaround time (ta) and weighted turnaround (w). The mean values of ta and w (i.e., ta and w) are shown below the table. The timing chart of Figure 7.2shows how the processes operated.



Time	Completed process			Processes in system (in FCFS order)	Scheduled process
	id	*ta*	*w*		
0	–	–	–	P_1	P_1
3	P_1	3	1.00	P_2, P_3	P_2
6	P_2	4	1.33	P_3, P_4	P_3
11	P_3	8	1.60	P_4, P_5	P_4
13	P_4	9	4.50	P_5	P_5
16	P_5	8	2.67	–	–

$\overline{ta}$ = 6.40 seconds

$\overline{w}$ = 2.22

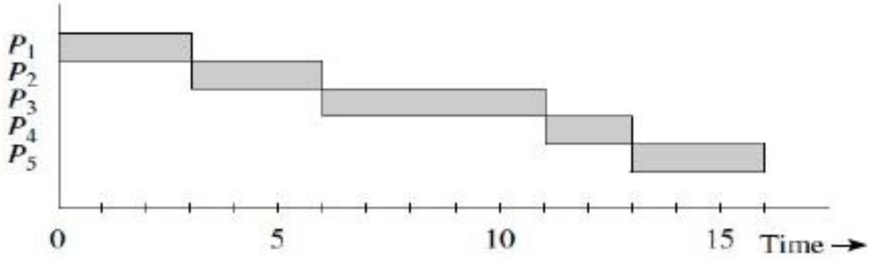


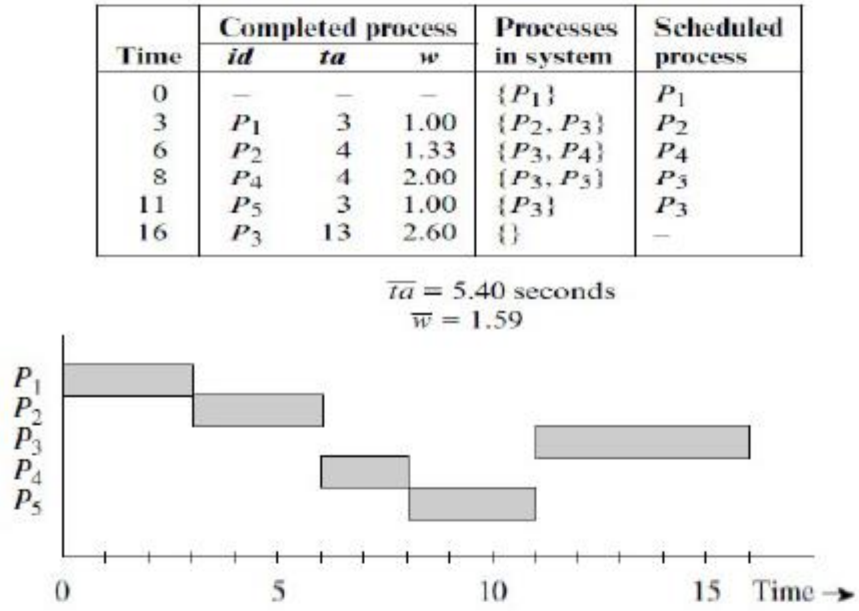
Figure 7.2 Scheduling using the FCFS policy.

Shortest Request Next (SRN) Scheduling

The SRN scheduler always schedules the request with the smallest service time. Thus, a request remains pending until all shorter requests have been serviced.

Figure 7.3 illustrates the scheduling decisions made by the SRN scheduling policy for the processes of Table 2.6, and the operation of the processes. At time 0, P1 is the only process in the system, so it is scheduled. It completes at time 3 seconds. At this time, processes P2 andP3 exist in the system, and P2 is shorter than P3. So P2 is scheduled, and so on. The mean turnaround time and the mean weighted turnaround are better than in FCFS scheduling because short requests tend to receive smaller turnaround times and weighted turnarounds than in FCFS scheduling. This feature degrades the service that long requests receive;

However, their weighted turnarounds do not increase much because their service times are large. The throughput is higher than in FCFS scheduling in the first 10 seconds of the schedule because short processes are being serviced; however, it is identical at the end of the schedule because the same processes have been serviced.



Time	Completed process *id*	*ta*	*w*	Processes in system	Scheduled process
0	–	–	–	$\{P_1\}$	P_1
3	P_1	3	1.00	$\{P_2, P_3\}$	P_2
6	P_2	4	1.33	$\{P_3, P_4\}$	P_4
8	P_4	4	2.00	$\{P_3, P_5\}$	P_5
11	P_5	3	1.00	$\{P_3\}$	P_3
16	P_3	13	2.60	$\{\}$	–

$\overline{ta} = 5.40$ seconds

$\overline{w} = 1.59$

Figure 7.3 Scheduling using the shortest request next (SRN) policy.

2.6 PRE EMPTIVE SCHEDULING POLICIES

In pre emptive scheduling, the server can be switched to the processing of a new request before completing the current request. The pre empted request is put back into the list of pending requests . Its servicing is resumed when it is scheduled again. Thus, a request might have to be scheduled many times before it completed. This feature causes a larger scheduling overhead than when non pre emptive scheduling is used.

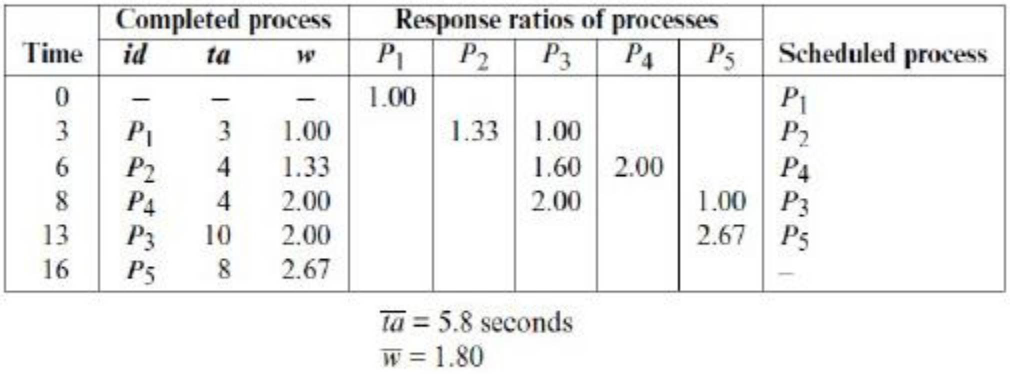
The three pre emptive scheduling policies are:

Round-robin scheduling with time-slicing (RR)

Least completed next (LCN) scheduling

Shortest time to go (STG) scheduling

The RR scheduling policy shares the CPU among admitted requests by servicing them in turn. The other two policies take into account the CPU time required by a request or the CPU



Time	Completed process			Response ratios of processes					Scheduled process
	id	*ta*	*w*	P_1	P_2	P_3	P_4	P_5	
0	–	–	–	1.00					P_1
3	P_1	3	1.00		1.33	1.00			P_2
6	P_2	4	1.33			1.60	2.00		P_4
8	P_4	4	2.00			2.00		1.00	P_3
13	P_3	10	2.00					2.67	P_5
16	P_5	8	2.67						–

$\overline{ta}$ = 5.8 seconds
$\overline{w}$ = 1.80

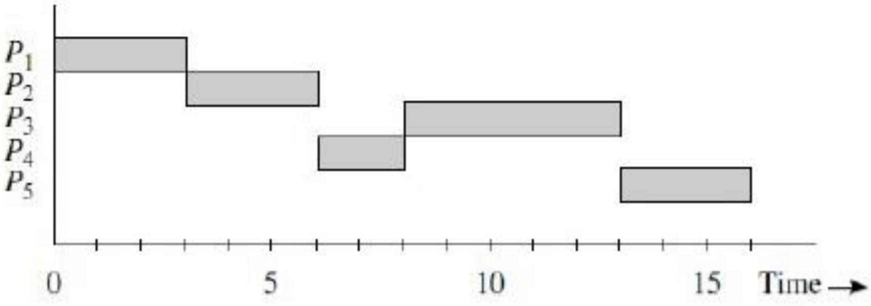


Figure 7.4 Operation of highest response ratio (HRN) policy.

Preemptive Scheduling Policies In preemptive scheduling, the server can be switched to the processing of a new request before completing the current request. The preempted request is put back into the list of pending requests (see Figure 7.1). Its servicing is resumed when it is scheduled again. Thus, a request might have to be scheduled many times before it completed. This feature causes a larger scheduling overhead than when non pre emptive scheduling is used.

The three preemptive scheduling policies are:

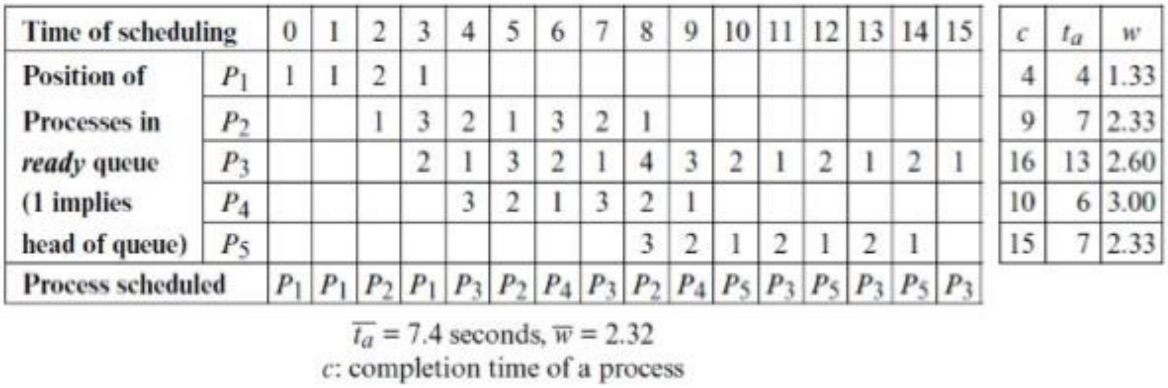
- Round-robin scheduling with time-slicing (RR)
- Least completed next (LCN) scheduling
- Shortest time to go (STG) scheduling

The RR scheduling policy shares the CPU among admitted requests by servicing them in turn. The other two policies take into account the CPU time required by a request or the CPU time consumed by it while making their scheduling decisions. 7.2.1 Round-Robin Scheduling with Time-Slicing (RR) The RR policy aims at providing good response times to all requests. The time slice, which is designated as , is the largest amount of CPU time a request may use when scheduled. A request is preempted at the end of a time slice. To facilitate this, the kernel arranges to raise a timer interrupt when the time slice elapses.

TheRRpolicy provides comparable service to all CPU-bound processes. This feature is reflected in approximately equal values of their weighted turnarounds. The actual value of the weighted turnaround of a process depends on the number of processes in the system.Weighted turnarounds provided to processes that perform I/O operations would depend on the durations of their I/O operations. The RR policy does not fare well on measures of system performance like throughput because it does not give a favored treatment to short processes. The following example illustrates the performance of RR scheduling.

Example 2.3 Round-Robin (RR) Scheduling

Around-robin scheduler maintains a queue of processes in the ready state and simply selects the first process in the queue. The running process is pre-empted when the time slice elapses and it is put at the end of the queue. It is assumed that a new process that was admitted into the system at the same instant a process was preempted will be entered into the queue before the pre-empted process. Figure 7.5 summarizes operation of the RR scheduler with = 1 second for the five processes shown in Table 2.6. The scheduler makes scheduling decisions every second. The time when a decision is made is shown in the first row of the table in the top half of Figure 7.5. The next five rows show positions of the five processes in the ready queue. A blank entry indicates that the process is not in the system at the designated time. The last row shows the process selected by the scheduler; it is the process occupying the first position in the ready queue. Consider the situation at 2 seconds. The scheduling queue contains P2 followed by P1. Hence P2 is scheduled. Process P3 arrives at 3 seconds, and is entered in the queue. P2 is also preempted at 3 seconds and it is entered in the queue. Hence the queue has process P1 followed by P3 and P2, so P1 is scheduled.



Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		c	t_a	w
Position of	P_1	1	1	2	1														4	4	1.33
Processes in	P_2			1	3	2	1	3	2	1									9	7	2.33
ready queue	P_3				2	1	3	2	1	4	3	2	1	2	1	2	1		16	13	2.60
(1 implies	P_4					3	2	1	3	2	1								10	6	3.00
head of queue)	P_5									3	2	1	2	1	2	1			15	7	2.33
Process scheduled		P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5	P_3				

$\overline{t_a} = 7.4$ seconds, $\overline{w} = 2.32$
c: completion time of a process

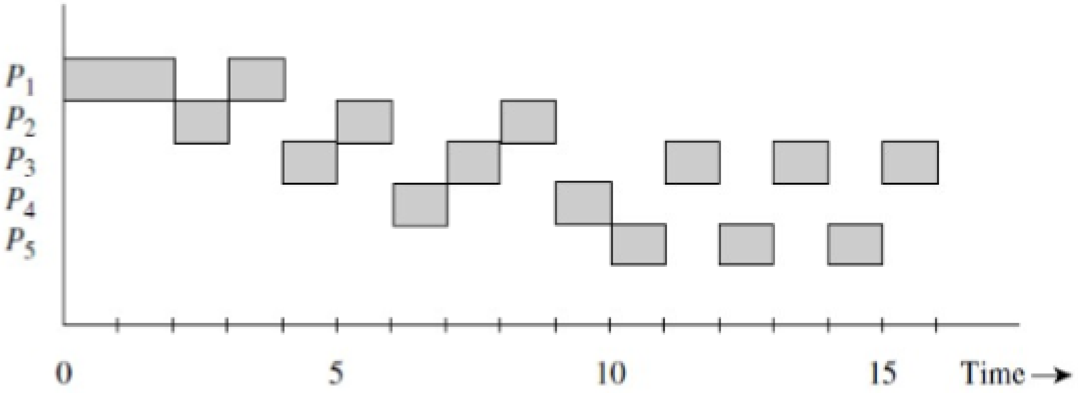
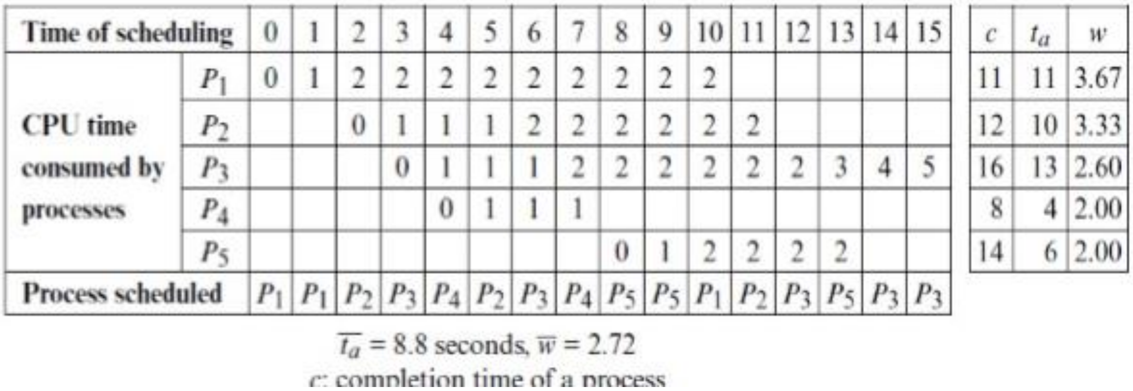


Figure 7.5 Scheduling using the round-robin policy with time-slicing (RR).

Least Completed Next (LCN) Scheduling: The LCN policy schedules the process that has so far consumed the least amount of CPU time. Thus, the nature of a process, whether CPU-bound or I/O-bound, and its CPU time requirement do not influence its progress in the system. Under the LCN policy, all processes will make approximately equal progress in terms of the CPU time consumed by them, so this policy guarantees that short processes will finish ahead of long processes. Ultimately, however, this policy has the familiar drawback of starving long processes of CPU attention. It also neglects existing processes if new processes keep arriving in the system. So even not-so long processes tend to suffer starvation or large turnaround times.

Example 7.6 Least Completed Next (LCN) Scheduling Implementation of the LCN scheduling policy for the five processes shown in Table 2.6 is summarized in Figure 7.6. The middle rows in the table in the upper half of the figure show the amount of CPU time already consumed by a process. The scheduler analyzes this information and selects the process that has consumed the least amount of CPU time. In case of a tie, it selects the process that has not been serviced for the longest period of time. The turnaround times and weighted turnarounds of the processes are shown in the right half of the table.



Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	c	t_a	w
CPU time consumed by processes	P_1	0	1	2	2	2	2	2	2	2	2	2						11	11	3.67
	P_2			0	1	1	1	2	2	2	2	2	2					12	10	3.33
	P_3				0	1	1	1	2	2	2	2	2	2	3	4	5	16	13	2.60
	P_4					0	1	1	1									8	4	2.00
	P_5									0	1	2	2	2	2			14	6	2.00
Process scheduled		P_1	P_1	P_2	P_3	P_4	P_2	P_3	P_4	P_5	P_5	P_1	P_2	P_3	P_5	P_3	P_3			

$\overline{t_a}$ = 8.8 seconds, $\overline{w}$ = 2.72

c: completion time of a process

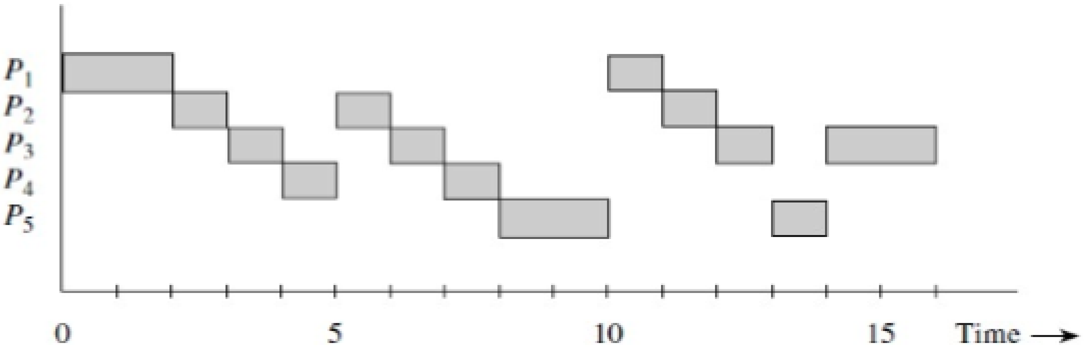


Figure 7.6 Scheduling using the least completed next (LCN) policy.

It can be seen that servicing of P1, P2, and P3 is delayed because new processes arrive and obtain CPU service before these processes can make further progress.

The LCN policy provides poorer turnaround times and weighted turnarounds than those provided by the RR policy (See Example 7.4) and the STG policy (to be discussed next) because it favours newly arriving processes over existing processes in the system until the new processes catch up in terms of CPU utilization; e.g., it favours P5 over P1, P2, and P3.

2.7 LONG TERM, MEDIUM TERM AND SHORT TERM SCHEDULING IN A TIME SHARING SYSTEM

Long-term scheduler: Decides when to admit an arrived process for scheduling, depending on its nature (whether CPU-bound or I/O-bound) and on availability of resources like kernel data structures and disk space for swapping.

Medium-term scheduler: Decides when to swap-out a process from memory and when to load it back, so that a sufficient number of ready processes would exist in memory.

Short-term scheduler: Decides which ready process to service next on the CPU and for how long. Thus, the short-term scheduler is the one that actually selects a process for operation. Hence it is also called the process scheduler, or simply the scheduler.

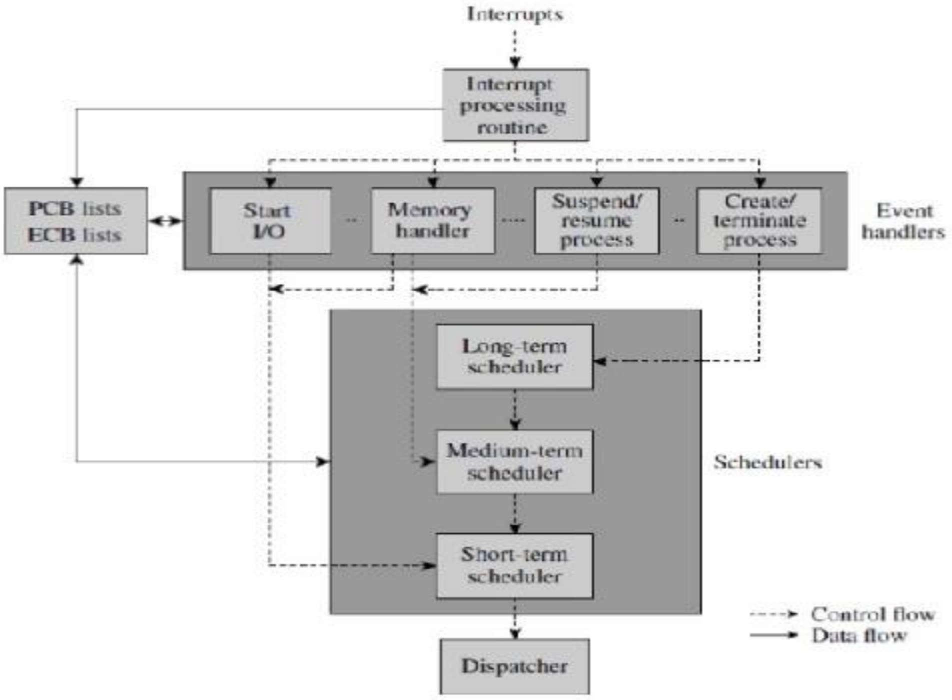


Figure 7.9 Event handling and scheduling.

Figure 7.8 shows an overview of scheduling and related actions. The operation of the kernel is interrupt-driven. Every event that requires the kernel.

Long-Term Scheduling The long-term scheduler may defer admission of a request for two reasons: it may not be able to allocate sufficient resources like kernel data structures or I/O devices to a request when it arrives, or it may find that admission of a request would affect system performance in some way; e.g., if the system currently contained a large number of CPU-bound requests, the scheduler might defer admission of a new CPU-bound request, but it might admit a new I/O-bound request right away.

Long-term scheduling was used in the 1960s and 1970s for job scheduling because computer systems had limited resources, so a long-term scheduler was required to decide whether a process could be initiated at the present time. It continues to be important in operating systems where resources are limited. It is also used in systems where requests have deadlines, or a set of requests are repeated with a known periodicity, to decide when a process should be initiated to meet response requirements of applications. Long-term scheduling is not relevant in other operating systems.

Medium-Term Scheduling Medium-term scheduling maps the large number of requests that have been admitted to the system into the smaller number of requests that can fit into the memory of the system at any time. Thus its focus is on making a sufficient number of ready processes available to the short-term scheduler by suspending or reactivating processes. The medium term scheduler decides when to swap out a process from memory and when to swap it back into memory, changes the state of the process appropriately, and enters its process control block (PCB) in the appropriate list of PCBs. The actual swapping-in and swapping out operations are performed by the memory manager.

The kernel can suspend a process when a user requests suspension, when the kernel runs out of free memory, or when it finds that the CPU is not likely to be allocated to the process in the near future. In time-sharing systems, processes in blocked or ready states are candidates for suspension. Short-Term Scheduling Short-term scheduling is concerned with effective use of the CPU. It selects one process from a list of ready processes and hands it to the dispatching mechanism. It may also decide how long the process should be allowed to use the CPU and instruct the kernel to produce a timer interrupt accordingly.

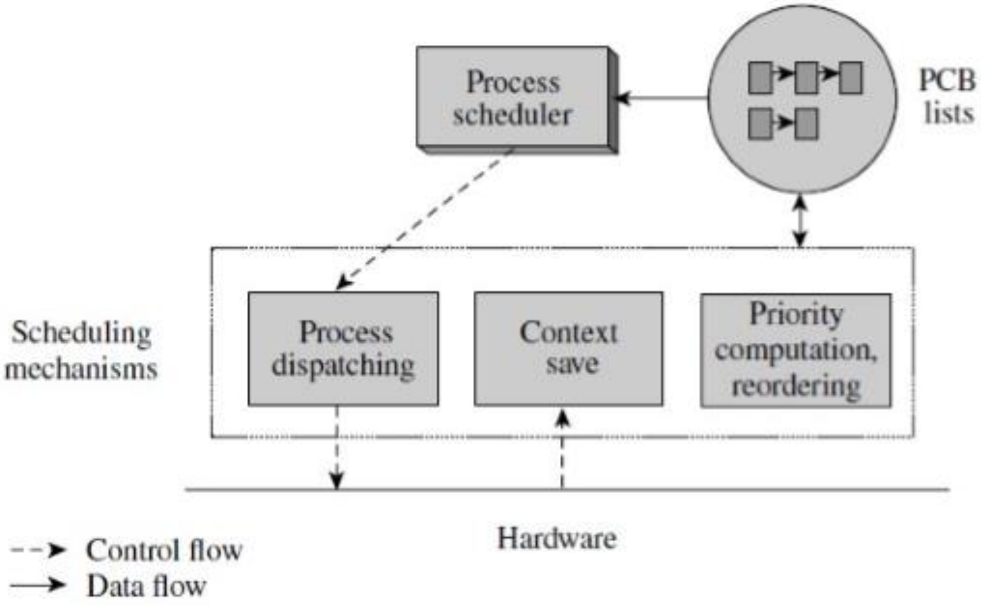


Figure 7.10 A schematic of the process scheduler.

Figure 7.10 is a schematic diagram of the process scheduler. It uses several lists of PCBs whose organization and use depends on the scheduling policy. The process scheduler selects one process and passes its id to the process dispatching mechanism. The process dispatching mechanism loads contents of two PCB fields the program status word (PSW) and general purpose registers (GPRs) fields into the CPU to resume operation of the selected process. Thus, the dispatching mechanism interfaces with the scheduler on one side and the hardware on the other side. The context save mechanism is a part of the interrupt processing routine. When an interrupt occurs, it is invoked to save the PSW and GPRs of the interrupted process. The priority computation and reordering mechanism re computes the priority of requests and reorders the PCB lists to reflect the new priorities.

2.8 SUMMARY

The process concept helps to explain, understand and organize execution of programs in an OS. A process is an execution of a program. The emphasis on 'an' implies that several processes may represent executions of the same program. A scheduling policy decides which process should be given the CPU at the present moment. This decision influences both system performance and user service. The scheduling policy in modern operating system must provide the best combination of user service and system performance to suit its computing environment.

2.9 QUESTIONS

1. Explain the contents of process control block.
2. Explain with a neat diagram, process state and transistion.
3. What is process? What are the components of a process? Explain .
4. Explain with neat diagrams a)User threads b)Kernel level threads.
5. Define a process . list the different fields of a process control block.
6. Explain the four fundamental states of a process with state transition diagram.
7. What are the advantages of threads over process? Explain kernel level threads.
8. Mention the three kinds of entities used for concurrency within a process in threads in solaris, along with a neat diagram.
9. With a state transition diagram and PCB structure , explain the function of the states , state transitions and the functions of a schedule.
10. Summarize the main approaches to real time scheduling.
11. Explain briefly the mechanism and policy modules of short term process scheduler with a neat block diagram.
12. Briefly explain the features of time sharing system. Also explain process state transitions in time sharing system.
13. Compare i) pre-emptive and non-preemptive scheduling ii) long term and short term schedulers

2.10 FURTHER READINGS

- www.studytonight.com/operating-system/cpu-scheduling
- https://www.youtube.com/watch?v=1fwxHAf1E88

MODULE 3
MEMORY MANAGEMENT

STRUCTURE

1. Contiguous and noncontiguous allocation to programs
2. Paging
3. Segmentation
4. Segmentation with paging
5. Virtual Memory Management
6. Demand Paging
7. Paging Hardware
8. VM handler
9. FIFO, LRU page replacement policies

OBJECTIVE:

On completion of this chapter student will be able to

- Discuss the concepts of memory management.
- Discuss the concepts of contiguous and non contiguous memory allocation.
- Explain how memory is allocated to kernel.
- Discuss the concepts of Segmentation , Paging and Segmentation with paging.

INTRODUCTION

The memory hierarchy comprises the cache, the memory management unit (MMU), random access memory (RAM), which is simply called memory in this chapter, and a disk. We discuss management of memory by the OS in two parts—this chapter discusses techniques for efficient use of memory, whereas the next chapter discusses management of virtual memory, which is part of the memory hierarchy consisting of the memory and the disk.

3.1 CONTIGUOUS MEMORY ALLOCATION

Contiguous memory allocation is the classical memory allocation model inwhich each process is allocated a single contiguous area in memory. Contiguous memory allocation faces the problem of memory fragmentation. In this section we focus on techniques to address this problem.

3.1.1Handling Memory Fragmentation

Internal fragmentation has no cure in contiguous memory allocation because the kernel has no means of estimating the memory requirement of a process accurately. Example 3.1 illustrates use of memory compaction.

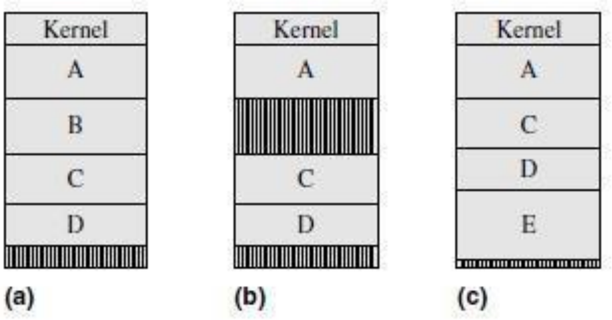


Figure 3.1 Memory compaction.

3.1.1 Contiguous memory

Allocation Example 3.1

Processes A, B, C, and D are in memory in Figure 3.1(a). Two free areas of memory exist after B terminates; however, neither of them is large enough to accommodate another process [see Figure 3.1(b)]. The kernel performs compaction to create a single free memory area and initiates process E in this area [see Figure 3.1(c)]. It involves moving processes C and D in memory during their execution.

Memory compaction involves dynamic relocation, which is not feasible without a relocation register. In computers not having a relocation register, the kernel must resort to reuse of free memory areas. However, this approach incurs delays in initiation of processes when large free memory areas do not exist, e.g., initiation of process E would be delayed in Example 4.8 even though the total free memory in the system exceeds the size of E.

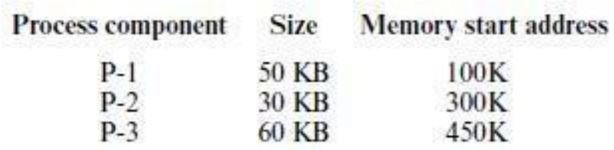
3.1.2 Swapping

The kernel swaps out a process that is not in the running state by writing out its code and data space to a swapping area on the disk. The swapped out process is brought back into memory before it is due for another burst of CPU time. A basic issue in swapping is whether a swapped-in process should be loaded back into the same memory area that it occupied before it was swapped out. If so, it‘s swapping in depends on swapping out of some other process that may have been allocated that memory area in the meanwhile. It would be useful to be able to place the swapped-in process elsewhere in memory; however, it would amount to dynamic relocation of the process to a new memory area. As mentioned earlier, only computer systems that provide a relocation register can achieve it.

3.2 NONCONTIGUOUS MEMORY ALLOCATION

Modern computer architectures provide the noncontiguous memory allocation model, in which a process can operate correctly even when portions of its address space are distributed among many areas of memory. This model of memory allocation permits the kernel to reuse free memory areas that are smaller than the size of a process, so it can reduce external fragmentation. Noncontiguous memory allocation using paging can even eliminate external fragmentation completely. Example 4.9 illustrates noncontiguous memory allocation. We use the term component for that portion of the process address space that is loaded in a single memory area.

Example 3.2 Noncontiguous Memory Allocation In Figure 3.2(a), four free memory areas starting at addresses 100K, 300K, 450K, and 600K, where K = 1024, with sizes of 50 KB, 30 KB, 80 KB and 40 KB, respectively, are present in memory. Process P, which has a size of 140 KB, is to be initiated [see Figure 3.2 (b)]. If process P consists of three components called P-1, P-2, and P-3, with sizes of 50 KB, 30 KB and 60 KB, respectively; these components can be loaded into three of the free memory areas as follows [see Figure 3.2(c)]:



Process component	Size	Memory start address
P-1	50 KB	100K
P-2	30 KB	300K
P-3	60 KB	450K

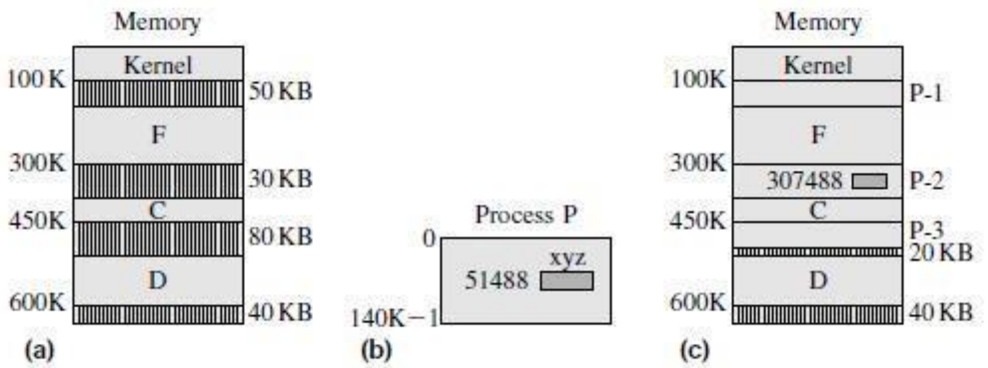


Figure 3.2 Noncontiguous memory allocation to process P.

3.2.1 Logical Addresses, Physical Addresses, and Address Translation

The abstract view of a system is called its logical view and the arrangement and relationship among its components is called the logical organization. On the other hand, the real view of the system is called its physical view and the arrangement depicted in it is called the physical organization. Accordingly, the views of process P shown in Figures 3.2(b) and Figures 3.2(c) constitute the logical and physical views of process P.

A logical address is the address of an instruction or data byte as used in a process; it may be obtained using index, base, or segment registers. The logical addresses in a process constitute the logical address space of the process. A physical address is the address in memory where an instruction or data byte exists. The set of physical addresses in the system constitutes the physical address space of the system.

The schematic diagram of Figure 3.3 shows how the CPU obtains the physical address that corresponds to a logical address. The kernel stores information about the memory areas allocated to process P in a table and makes it available to the memory management unit (MMU).

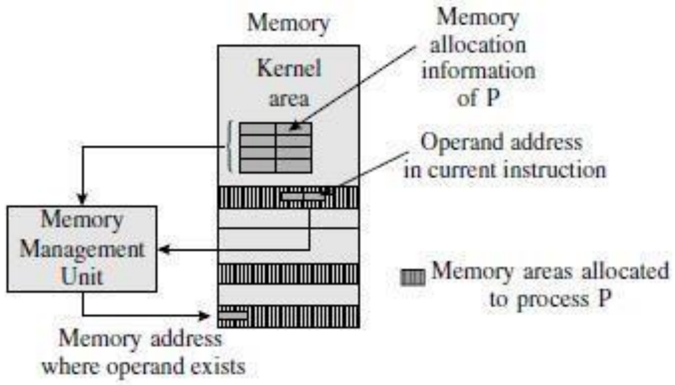
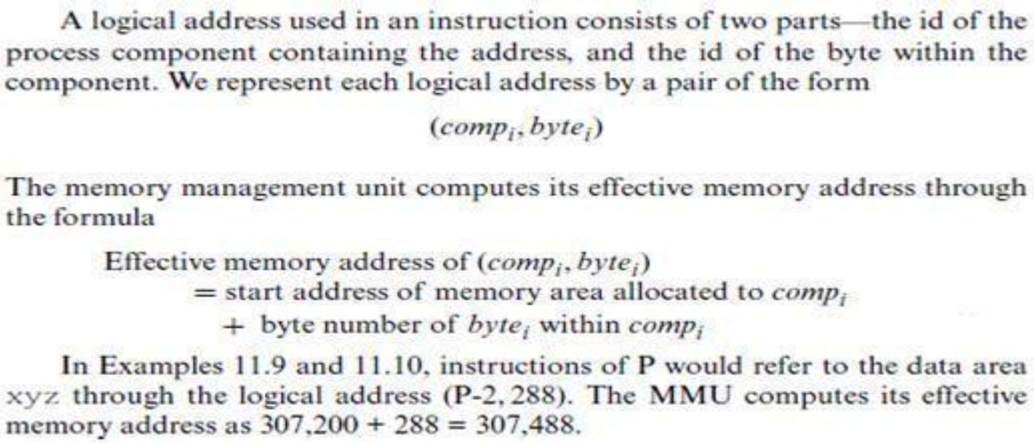

Memory
Kernel area
Memory allocation information of P
Operand address in current instruction
Memory Management Unit
Memory areas allocated to process P
Memory address where operand exists

Figure 3.3 A schematic of address translation in noncontiguous memory allocation.



A logical address used in an instruction consists of two parts—the id of the process component containing the address, and the id of the byte within the component. We represent each logical address by a pair of the form

$$(comp_i, byte_i)$$

The memory management unit computes its effective memory address through the formula

$$\begin{aligned}\text{Effective memory address of } (comp_i, byte_i) \\ = \text{start address of memory area allocated to } comp_i \\ + \text{ byte number of } byte_i \text{ within } comp_i\end{aligned}$$

In Examples 11.9 and 11.10, instructions of P would refer to the data area xyz through the logical address (P-2, 288). The MMU computes its effective memory address as 307,200 + 288 = 307,488.

Approaches to Noncontiguous Memory Allocation

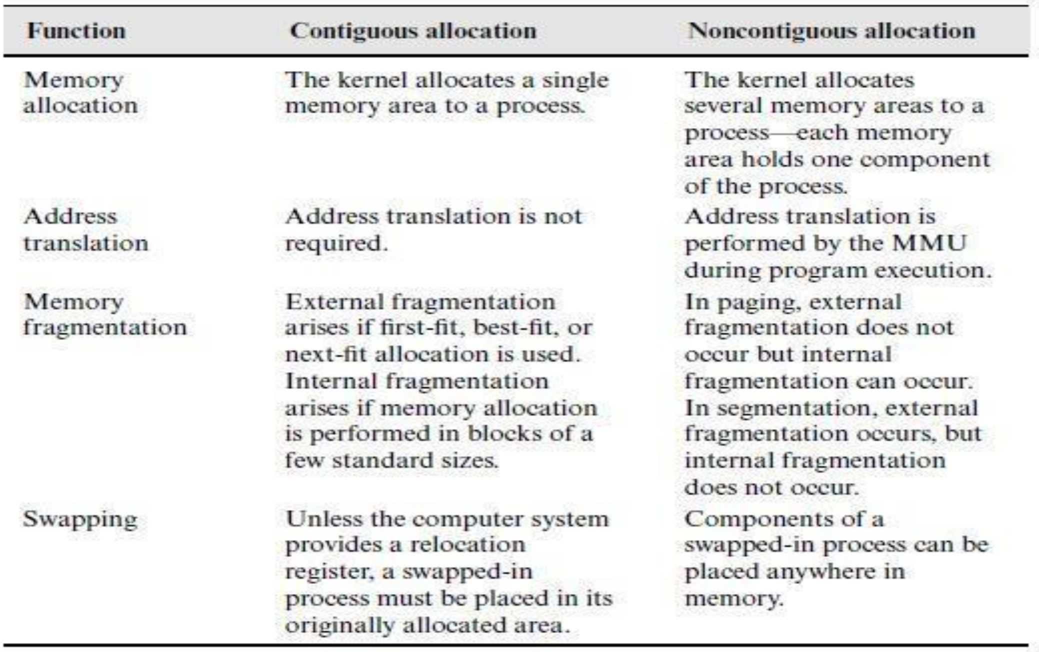
There are two fundamental approaches to implementing noncontiguous memory allocation:

- Paging
- Segmentation

In paging, each process consists of fixed-size components called pages. The size of a page is defined by the hardware of a computer, and demarcation of pages is implicit in it. The memory can accommodate an integral number of pages. It is partitioned into memory areas that have the same size as a page, and each of these memory areas is considered separately for allocation to a page. This way, any free memory area is exactly the same size as a page, so external fragmentation does not arise in the system. Internal fragmentation can arise because the last page of a process is allocated a page-size memory area even if it is smaller than a page in size.

In segmentation, a programmer identifies components called segments in a process. A segment is a logical entity in a program, e.g., a set of functions, data structures, or objects. Segmentation facilitates sharing of code, data, and program modules between processes. However, segments have different sizes, so the kernel has to use memory reuse techniques such as first-fit or best-fit allocation. Consequently, external fragmentation can arise. A hybrid approach called segmentation with paging combines the features of both segmentation and paging. It facilitates sharing of code, data, and program modules between processes without incurring external fragmentation; however, internal fragmentation occurs as in paging.

Table 3.1 **Comparison of Contiguous and Noncontiguous**



Function	Contiguous allocation	Noncontiguous allocation
Memory allocation	The kernel allocates a single memory area to a process.	The kernel allocates several memory areas to a process—each memory area holds one component of the process.
Address translation	Address translation is not required.	Address translation is performed by the MMU during program execution.
Memory fragmentation	External fragmentation arises if first-fit, best-fit, or next-fit allocation is used. Internal fragmentation arises if memory allocation is performed in blocks of a few standard sizes.	In paging, external fragmentation does not occur but internal fragmentation can occur. In segmentation, external fragmentation occurs, but internal fragmentation does not occur.
Swapping	Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area.	Components of a swapped-in process can be placed anywhere in memory.

Table 3.1 summarizes the advantages of noncontiguous memory allocation over contiguous memory allocation. Swapping is more effective in non-contiguous memory allocation because address translation enables the kernel to load components of a swapped-in process in any parts of memory.

3.3 PAGING

In the logical view, the address space of a process consists of a linear arrangement of pages. Each page has s bytes in it, where s is a power of 2. The value of s is specified in the architecture of the computer system. Processes use numeric logical addresses. The MMU decomposes a logical address into the pair (pi , bi), where pi is the page number and bi is the byte number within page pi . Pages in a program and bytes in a page are numbered from 0; so, in a logical address (pi, bi), pi ≥ 0 and 0 ≤ bi < s. In the physical view, pages of a process exist in nonadjacent areas of memory.

Consider two processes P and R in a system using a page size of 1 KB. The bytes in a page are numbered from 0 to 1023. Process P has the start address 0 and a size of 5500 bytes. Hence it has 6 pages numbered from 0 to 5. The last page contains only 380 bytes. If a data item sample had the address 5248, which is 5 × 1024 + 128, the MMU would view its address as the pair (5, 128). Process R has a size of 2500 bytes. Hence it has 3 pages, numbered from 0 to 2. Figure 3.4 shows the logical view of processes P and R. The hardware partitions memory into areas called page frames; page frames in memory are numbered from 0. Each page frame is the same size as a page. At any moment, some page frames are allocated to pages of processes, while others are free. The kernel maintains a list called the free frames list to note the frame numbers of free page frames. While loading a process for execution, the kernel consults the free frames list and allocates a free page frame to each page of the process. To facilitate address translation, the kernel constructs a page table (PT) for each process. The page table has an entry for each page of the process, which indicates the page frame allocated to the page. While performing address translation for a logical address (pi , bi), the MMU uses the page number pi to index the page table of the process, obtains the frame number of the page frame allocated to pi , and computes the effective memory address according to Equation above.

Figure 3.5 shows the physical view of execution of processes P and R. Each page frame is 1 KB in size. The computer has a memory of 10 KB, so page frames are numbered from 0 to 9. Six page frames are occupied by process P, and three page frames are occupied by process R. The pages contained in the page frames are shown as P-0, . . . , P-5 and R-0, . . . , R-2. Page frame 4 is free. Hence the free frames list contains only one entry. The page table of P indicates the page frame allocated to each page of P. As mentioned earlier, the variable sample process P has the logical address (5, 128).

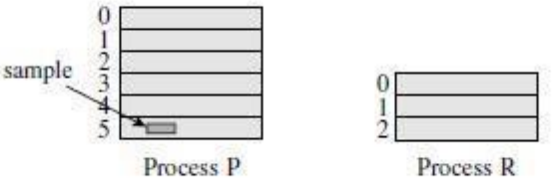


Figure 3.4 Logical view of processes in paging.

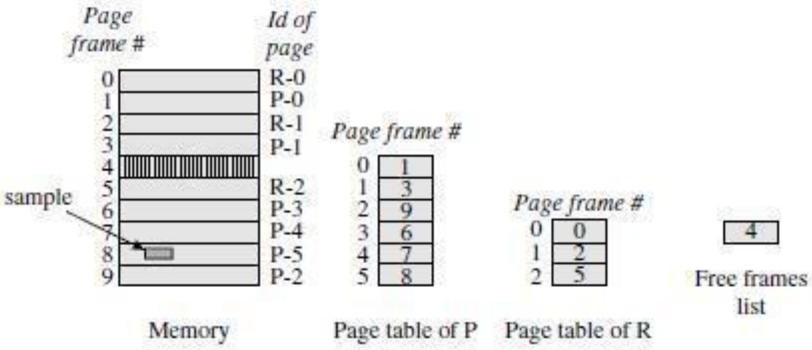


Figure 3.5 Physical organization in paging.

When process P uses this logical address during its execution, it will be translated into the effective memory address by using Eq. as follows:

Effective memory address of (5, 128)

= start address of page frame #8 + 128

=8 × 1024 + 128 =8320

We use the following notation to describe how address translation is actually performed: s Size of a page

ll Length of a logical address (i.e., number of bits in it) lp Length of a physical address

nb Number of bits used to represent the byte number in a logical address

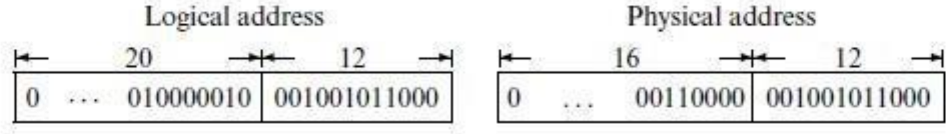
np Number of bits used to represent the page number in a logical address

nf Number of bits used to represent the frame number in a physical address

The size of a page, s, is a power of 2. nb is chosen such that s = 2nb . Hence the least significant nb bits in a logical address give us bi , the byte number within a page. The remaining bits in a logical address form pi , the page number.

Example 3.2 Address Translation in Paging

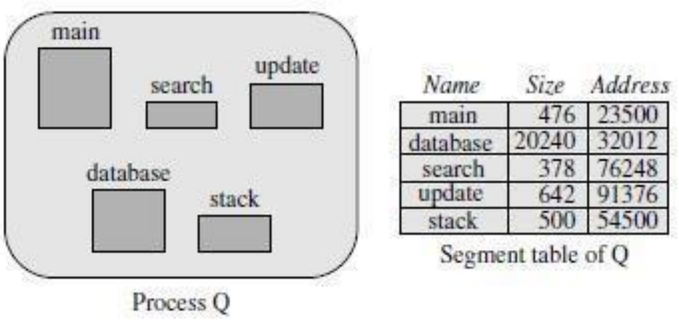
A hypothetical computer uses 32-bit logical addresses and a page size of 4KB. 12 bits are adequate to address the bytes in a page. Thus, the higher order 20 bits in a logical address represent pi and the 12 lower order bits represent bi . For a memory size of 256 MB, lp = 28. Thus, the higher-order 16 bits in a physical address represent qi . If page 130 exists in page frame 48, pi = 130, and qi = 48. If bi = 600, the logical and physical addresses look as follows: Logical address



During address translation, the MMU obtains pi and bi merely by grouping the bits of the logical address as shown above. The 130th entry of the page table is now accessed to obtain qi , which is 48. This number is concatenated with bi to form the physical address.

3.4 SEGMENTATION

A segment is a logical entity in a program, e.g., a function, a data structure, or an object. Hence it is meaningful to manage it as a unit—load it into memory for execution or share it with other programs. In the logical view, a process consists of a collection of segments. In the physical view, segments of a process exist in nonadjacent areas of memory. A process Q consists of five logical entities with the symbolic names main, database, search, update, and stack. While coding the program, the programmer declares these five as segments in Q. This information is used by the compiler or assembler to generate logical addresses while translating the program. Each logical address used in Q has the form (si , bi) where si and bi are the ids of a segment and a byte within a segment. For example, the instruction corresponding to a statement call get_sample, where get_sample is a procedure in segment update, may use the operand address (update, get_sample). Alternatively, it may use a numeric representation in which si and bi are the segment number and byte number within a segment, respectively



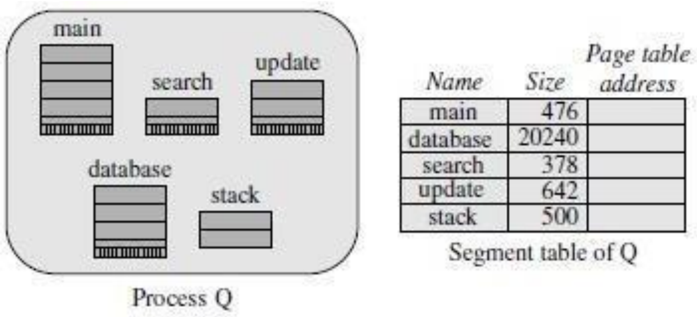
Name	*Size*	*Address*
main	476	23500
database	20240	32012
search	378	76248
update	642	91376
stack	500	54500

Figure 3.6 A process Q in segmentation.

3.4 SEGMENTATION WITH PAGING

In this approach, each segment in a program is paged separately. Accordingly, an integral number of pages is allocated to each segment. This approach simplifies memory allocation and speeds it up, and also avoids external fragmentation. A page table is constructed for each segment, and the address of the page table is kept in the segment‘s entry in the segment table.

Address translation for a logical address (si , bi) is now done in two stages. In the first stage, the entry of si is located in the segment table, and the address of its page table is obtained. The byte number bi is now split into a pair (psi , bpi), where psi is the page number in segment si , and bpi is the byte number in page pi . The effective address calculation is now completed as in paging, i.e., the frame number of psi is obtained and bpi is concatenated with it to obtain the effective address.



Name	Size	Page table address
main	476	
database	20240	
search	378	
update	642	
stack	500	

Figure 3.7 A process Q in segmentation with paging.

Figure 3.7 shows process Q of Figure 3.6 in a system using segmentation with paging. Each segment is paged independently, so internal fragmentation exists in the last page of each segment. Each segment table entry now contains the address of the page table of the segment. The size field in a segment‘s entry is used to facilitate a bound check for memory protection.

VIRTUAL MEMORY

Virtual memory is a part of the memory hierarchy that consists of memory and a disk. In accordance with the principle of memory hierarchies only some portions of the address space of a process—that is, of its code and data—exist in memory at any time; other portions of its address space reside on disk and are loaded into memory when needed during operation of the process. The kernel employs virtual memory to reduce the memory commitment to a process so that it can service a large number of processes concurrently, and to handle processes whose address space is larger than the size of memory.

Virtual memory is implemented through the noncontiguous memory allocation model and comprises both hardware components and a software component called a virtual memory manager. The hardware components speed up address translation and help the virtual memory manager perform its tasks more effectively. The virtual memory manager decides which portions of a process address space should be in memory at any time.

VIRTUAL MEMORY BASICS

Users always want more from a computer system—more resources and more services. The need for more resources is satisfied either by obtaining more efficient use of existing resources, or by creating an illusion that more resources exist in the system. A virtual memory is what its name indicates—it is an illusion of a memory that is larger than the real memory, i.e., RAM, of the computer system.

As we pointed out in Section 1.1, this illusion is a part of a user‘s abstract view of memory. A user or his application program sees only the virtual memory. The kernel implements the illusion through a combination of hardware and software means. We refer to real memory simply as memory. We refer to the software component of virtual memory as a virtual memory manager.

The illusion of memory larger than the system‘s memory crops up any time a process whose size exceeds the size of memory is initiated. The process is able to operate because it is kept in its entirety on a disk and only its required portions are loaded in memory at any time. The basis of virtual memory is the non-contiguous memory allocation model. The address space of each process is assumed to consist of portions called components. The portions can be loaded into nonadjacent areas of memory. The address of each operand or instruction in the code of a process is a logical address of the form (compi

, bytei).

The memory management unit (MMU) translates it into the address in memory where the operand or instruction actually resides.

Figure 5.1 shows a schematic diagram of a virtual memory. The logical address space of the process shown consists of five components. Three of these components are presently in memory. Information about the memory areas where these components exist is maintained in a data structure of the virtual memory manager. This information is used by the MMU during address translation. When an instruction in the process refers to a data item or instruction that is not in memory, the component containing it is loaded from the disk. Occasionally, the virtual memory manager removes some components from memory to make room for other components.

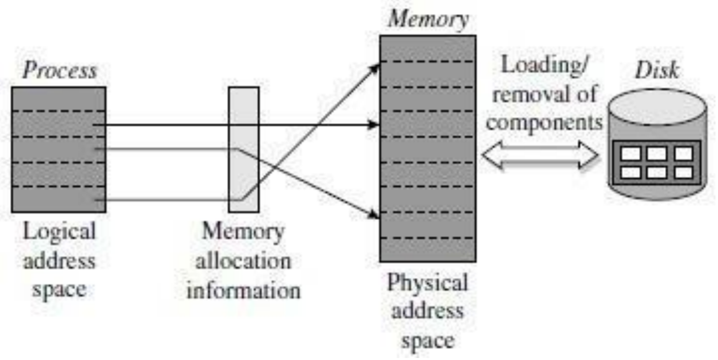


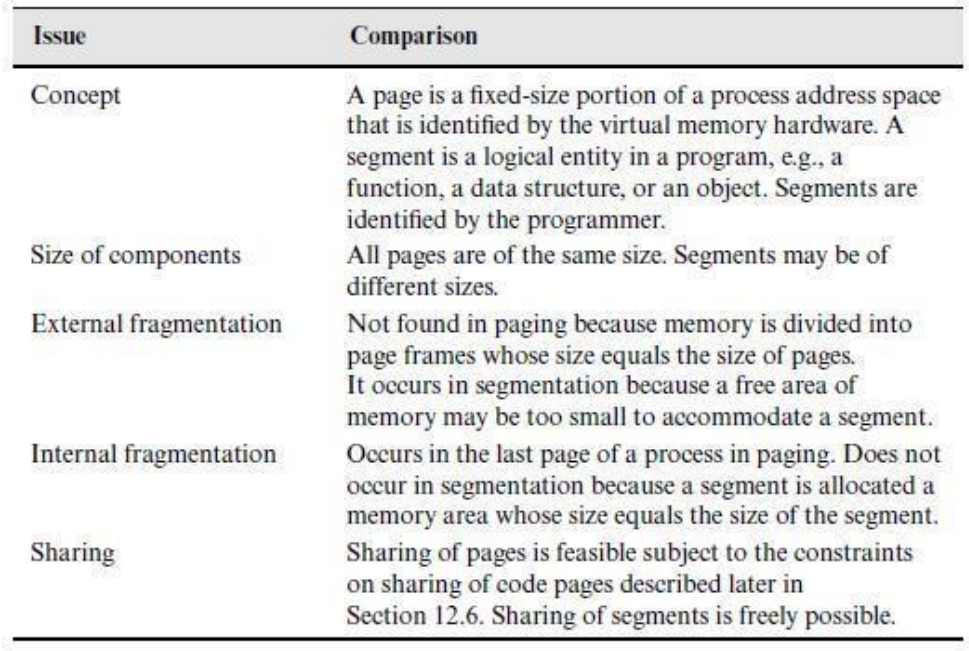
Figure 5.1 Overview of virtual memory.

Virtual Memory A memory hierarchy, consisting of a computer system‘s memory and a disk, that enables a process to operate with only some portions of its address space in memory.

Demand Loading of Process Components The virtual memory manager loads only one component of a process address space in memory to begin with—the component that contains the start address of the process, that is, address of the instruction with which its execution begins. It loads other components of the process only when they are needed. This technique is called demand loading. To keep the memory commitment to a process low, the virtual memory manager removes components of the process from memory from time to time. These components would be loaded back in memory when needed again.

Performance of a process in virtual memory depends on the rate at which its components have to be loaded into memory. The virtual memory manager exploits the law of locality of reference to achieve a low rate of loading of process components.

Table 5.1 **Comparison of Paging and Segmentation**



Issue	Comparison
Concept	A page is a fixed-size portion of a process address space that is identified by the virtual memory hardware. A segment is a logical entity in a program, e.g., a function, a data structure, or an object. Segments are identified by the programmer.
Size of components	All pages are of the same size. Segments may be of different sizes.
External fragmentation	Not found in paging because memory is divided into page frames whose size equals the size of pages. It occurs in segmentation because a free area of memory may be too small to accommodate a segment.
Internal fragmentation	Occurs in the last page of a process in paging. Does not occur in segmentation because a segment is allocated a memory area whose size equals the size of the segment.
Sharing	Sharing of pages is feasible subject to the constraints on sharing of code pages described later in Section 12.6. Sharing of segments is freely possible.

Paging and Segmentation The two approaches to implementation of virtual memory differ in the manner in which the boundaries and sizes of address space components are determined.

Table 5.1 compares the two approaches. In paging, each component of an address space is called a page. All pages have identical size, which is a power of two. Page size is defined by the computer hardware and demarcation of pages in the address space of a process is performed implicitly by it. In segmentation, each component of an address space is called a segment. A programmer declares some significant logical entities (e.g., data structures or objects) in a process as segments. Thus identification of components is performed by the programmer, and segments can have different sizes. This fundamental difference leads to different implications for efficient use of memory and for sharing of programs or data. Some systems use a hybrid segmentation-with-paging approach to obtain advantages of both
the approaches.

3.5 DEMAND PAGING

A process is considered to consist of pages, numbered from 0 onward. Each page is of size s bytes, where s is a power of 2. The memory of the computer system is considered to consist of page frames, where a page frame is a memory area that has the same size as a page. Page frames are numbered from 0 to #frames−1 where #frames is the number of page frames of memory. Accordingly, the physical address space consists of addresses from 0 to #frames × s

− 1. At any moment, a page frame may be free, or it may contain a page of some process. Each logical address used in a process is considered to be a pair (pi , bi), where pi is a page number and bi is the byte number in pi, 0 ≤ bi < s.

The effective memory address of a logical address (pi , bi) is computed as follows: Effective memory address of logical address (pi , bi)

= start address of the page frame containing page pi + bi **(5.1)**

The size of a page is a power of 2, and so calculation of the effective address is performed through bit concatenation, which is much faster than addition.

Figure 5.2 is a schematic diagram of a virtual memory using paging in which page size is assumed to be 1KB, where 1KB = 1024 bytes. Three processes P1, P2 and P3, have some of their pages in memory. The memory contains 8 page frames numbered from 0 to 7. Memory allocation information for a process is stored in a page table. Each entry in the page table contains memory allocation information for one page of a process. It contains the page frame number where a page resides. Process P2 has its pages 1 and 2 in memory. They occupy page frames 5 and 7 respectively. Process P1 has its pages 0 and 2 in page frames 4 and 1, while process P3 has its pages 1, 3 and 4 in page frames 0, 2 and 3, respectively. The free frames list contains a list of free page frames. Currently only page frame 6 is free.

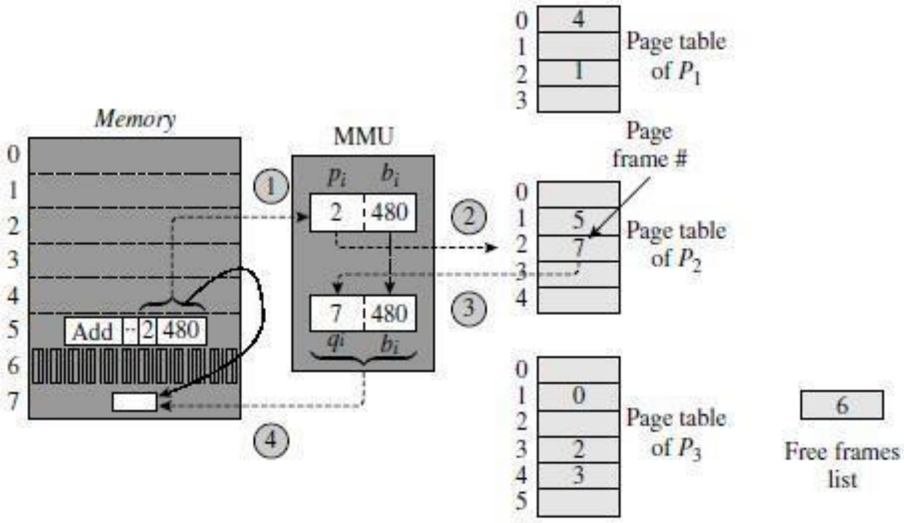


Figure 5.2 Address translation in virtual memory using paging.

Process P2 is currently executing the instruction ‗Add ·· 2528‘, so the MMU uses P2‘s page table for address translation. The MMU views the operand address 2528 as the pair (2, 480) because 2528=2×1024+480. It now accesses the entry for page 2 in P2‘s page table. This entry contains frame number 7, so the MMU forms the effective address 7×1024+480 according to Eq. (5.1), and uses it to make a memory access. In effect, byte 480 in page frame 7 is accessed.

3.5.1 Demand Paging Preliminaries

If an instruction of P2 in Figure 5.2 refers to a byte in page 3, the virtual memory manager will load page 3 in memory and put its frame number in entry 3 of P2‘s page table. These actions constitute demand loading of pages, or simply demand paging.

To implement demand paging, a copy of the entire logical address space of a process is maintained on a disk. The disk area used to store this copy is called the swap space of a process. While initiating a process, the virtual memory manager allocates the swap space for the process and copies its code and data into the swap space. During operation of the process, the virtual memory manager is alerted when the process wishes to use some data item or instruction that is located in a page that is not present in memory. It now loads the page from the swap space into memory. This operation is called a page-in operation. When the virtual memory manager decides to remove a page from memory, the page is copied back into the swap space of the process to which it belongs if the page was modified since the last time it was loaded in memory. This operation is called a page-out operation.

This way the swap space of a process contains an up-to-date copy of every page of the process that is not present in memory. A page replacement operation is one that loads a page into a page frame that previously contained another page.

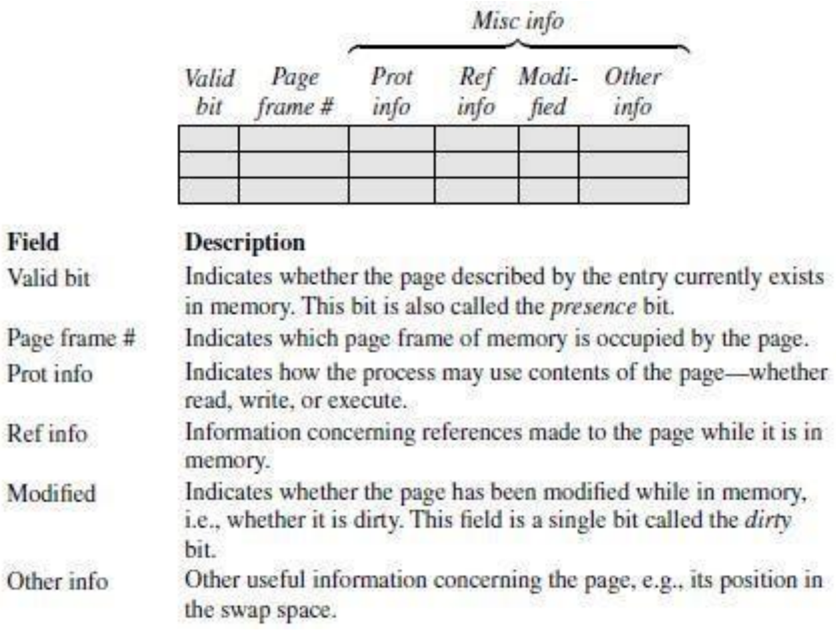
It may involve a page-out operation if the previous page was modified while it occupied the page frame, and involves a page-in operation to load the new page.

Page Table The page table for a process facilitates implementation of address translation, demand loading, and page replacement operations. Figure 5.3 shows the format of a page table entry. The valid bit field contains a Boolean value to indicate whether the page exists in memory. We use the convention that 1 indicates ―resident in memory‖ and 0 indicates ―not resident in memory.‖ The page frame# field, which was described earlier, facilitates address translation. The misc info field is divided into four subfields. Information in the prot info field is used for protecting contents of the page against interference. It indicates whether the process can read or write data in the page or execute instructions in it. ref info contains information concerning references made to the page while it is in memory.

The modified bit indicates whether the page has been modified, i.e., whether it is dirty. It is used to decide whether a page-out operation is needed while replacing the page. The other info field contains information such as the address of the disk block in the swap space where a copy of the page is maintained.

3.5.2 Page Faults and Demand Loading of Pages

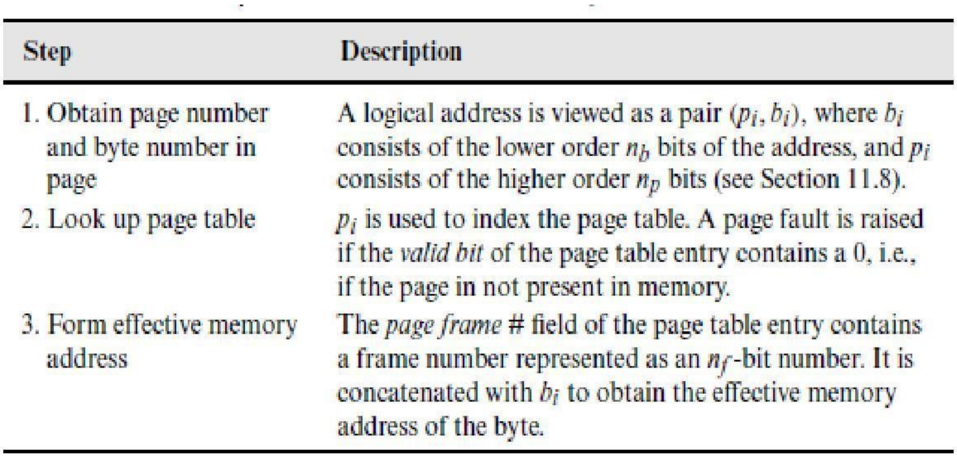
Table 5.2 summarizes steps in address translation by the MMU. While performing address translation for a logical address (pi , bi), the MMU checks the valid bit of the page table entry of pi



Field	Description
Valid bit	Indicates whether the page described by the entry currently exists in memory. This bit is also called the *presence* bit.
Page frame #	Indicates which page frame of memory is occupied by the page.
Prot info	Indicates how the process may use contents of the page—whether read, write, or execute.
Ref info	Information concerning references made to the page while it is in memory.
Modified	Indicates whether the page has been modified while in memory, i.e., whether it is dirty. This field is a single bit called the *dirty* bit.
Other info	Other useful information concerning the page, e.g., its position in the swap space.

Figure 5.3 Fields in a page table entry.

Table 5.2 **Steps in Address Translation by the MMU**



Step	Description
1. Obtain page number and byte number in page	A logical address is viewed as a pair (p_i, b_i), where b_i consists of the lower order n_b bits of the address, and p_i consists of the higher order n_p bits (see Section 11.8).
2. Look up page table	p_i is used to index the page table. A page fault is raised if the *valid bit* of the page table entry contains a 0, i.e., if the page in not present in memory.
3. Form effective memory address	The *page frame* # field of the page table entry contains a frame number represented as an n_f-bit number. It is concatenated with b_i to obtain the effective memory address of the byte.

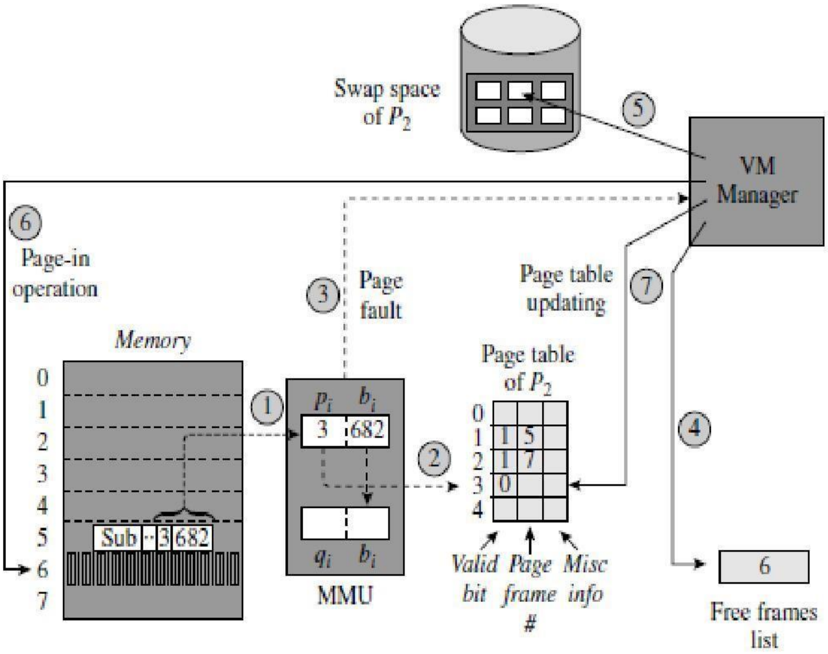


Figure 5.4 Demand loading of a page.

If the bit indicates that pi is not present in memory, the MMU raises an interrupt called a missing page interrupt or a page fault, which is a program interrupt (see Section 2.2.5).

The interrupt servicing routine for program interrupts finds that the interrupt was caused by a page fault, so it invokes the virtual memory manager with the page number that caused the page fault, i.e., pi , as a parameter. The virtual memory manager now loads page pi in memory and updates its page table entry. Thus, the MMU and the virtual memory manager interact to decide when a page of a process should be loaded in memory. Figure 5.4 is an overview of the virtual memory manager‘s actions in demand loading of a page. The broken arrows indicate actions of the MMU, whereas firm arrows indicate accesses to the data structures, memory, and the disk by the virtual memory manager when a page fault occurs. The numbers in circles indicate the steps in address translation, raising, and handling of the page fault—

Steps 1–3 were described earlier in Table 12.2. Process P2 of Figure 5.2 is in operation. While translating the logical address (3, 682), the MMU raises a page fault because the valid bit of page 3‘s entry is 0.

3.5.3 Page-in, Page-out, and Page Replacement Operations

Figure 5.4 showed how a page-in operation is performed for a required page when a page fault occurs in a process and a free page frame is available in memory. If no page frame is free, the virtual memory manager performs a page replacement operation to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows: The virtual memory manager uses a page replacement algorithm to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as ―not present‖ in memory, and initiates a page-out operation for it if the modified bit of its page table entry indicates that it is a dirty page.

In the next step, the virtual memory manager initiates a page-in operation to load the required page into the page frame that was occupied by the selected page. After the page-in operation completes, it updates the page table entry of the page to record the frame number of the page frame, marks the page as ―present,‖ and makes provision to resume operation of the process. The process now reexecutes its current instruction. This time, the address translation for the logical address in the current instruction completes without a page fault. The page-in and page-out operations required to implement demand paging constitute page I/O; we use the termpage traffic to describe movement of pages in and out of memory. Note that page I/O is distinct from I/O operations performed by processes,whichwe will call programI/O. The state of a process that encounters a page fault is changed to blocked until the required page is loaded in memory, and so its performance suffers because of a page fault. The kernel can switch theCPU to another process to safeguard system performance.

Effective Memory Access Time The effective memory access time for a process in demand paging is the average memory access time experienced by the process.

It depends on two factors: time consumed by the MMU in performing address translation, and the average time consumed by the virtual memory manager in handling a page fault. We use the following notation to compute the effective memory access time:

pr1 probability that a page exists in memory

tmem memory access time

tpfh time overhead of page fault handling

pr1 is called the memory hit ratio. tpfh is a few orders of magnitude larger than tmem because it involves disk I/O—one disk I/O operation is required if only a page-in operation is sufficient, and two disk I/O operations are required if a page replacement is necessary.

A process‘s page table exists in memory when the process is in operation. Hence, accessing an operand with the logical address (pi , bi) consumes two memory cycles if page pi exists in memory—one to access the page table entry of pi for address translation, and the other to access the operand in memory using the effective memory address of (pi , bi). If the page is not present in memory, a page fault is raised after referencing the page table entry of pi , i.e., after one memory cycle.Accordingly, the effective memory access time is as follows:

Effective memory access time = $pr1 \times 2 \times tmem + (1 - pr1) \times (tmem + tpfh + 2 \times tmem)$ **(5.2)**

The effective memory access time can be improved by reducing
the number of page faults.

3.5.3 Page Replacement

Page replacement becomes necessary when a page fault occurs and there are no free page frames in memory. However, another page fault would arise if the replaced page is referenced again. Hence it is important to replace a page that is not likely to be referenced in the immediate future. But how does the virtual memory manager know which page is not likely to be referenced in the immediate future?

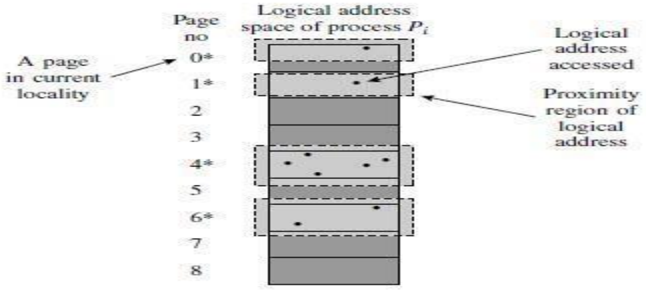
The empirical law of locality of reference states that logical addresses used by a process in any short interval of time during its operation tend to be bunched together in certain portions of its logical address space. Processes exhibit this behaviour for two reasons. Execution of instructions in a process is mostly sequential in nature, because only 10–20 percent of instructions executed by a process are branch instructions. Processes also tend to perform similar operations on several elements of nonscalar data such as arrays. Due to the combined effect of these two reasons, instruction and data references made by a process tend to be in close proximity to previous instruction and data references made by it. We define the current locality of a process as the set of pages referenced in its previous few instructions. Thus, the law of locality indicates that the logical address used in an instruction is likely to refer to a page that is in the current locality of the process.

The virtual memory manager can exploit the law of locality to achieve an analogous effect— fewer page faults would arise if it ensures that pages that are in the current locality of a process are present in memory.

Note that locality of reference does not imply an absence of page faults. Let the proximity region of a logical address ai contain all logical addresses that are in close proximity to ai . Page faults can occur for two reasons: First, the proximity region of a logical address may not fit into a page; in this case, the next address may lie in an adjoining page that is not included in the current locality of the process. Second, an instruction or data referenced by a process may not be in the proximity of previous references. We call this situation a shift in locality of a process. It typically occurs when a process makes a transition from one action in its logic to another. The next example illustrates the locality of a process.

The law of locality helps to decide which page should be replaced when a page fault occurs. Let us assume that the number of page frames allocated to a process Pi is a constant. Hence whenever a page fault occurs during operation of Pi , one of Pi ‘s own pages existing in memory must be replaced. Let t1 and t2 be the periods of time for which pages p1 and p2 have not been referenced during the operation of Pi . Let t1 > t2, implying that some byte of page p2 has been referenced or executed (as an instruction) more recently than any byte of page p1.

Hence page p2 is more likely to be a part of the current locality of the process than page p1; that is, a byte of page p2 is more likely to be referenced or executed than a byte of page p1. We use this argument to choose page p1 for replacement when a page fault occurs. If many pages of Pi exist in memory, we can rank them according to the times of their last references and replace the page that has been least recently referenced. This page replacement policy is called LRU page replacement.


Page no
Logical address space of process P_i
A page in current locality
0*
1*
2
3
4*
5
6*
7
8
Logical address accessed
Proximity region of logical address

. Figure 5.5 Proximity regions of previous references and current locality of a process

3.5.4 Memory Allocation to a Process

Figure 5.6 shows how the page fault rate of a process should vary with the amount of memory allocated to it. The page fault rate is large when a small amount of memory is allocated to the process; however, it drops when more memory is allocated to the process. This page fault characteristic of a process is desired because it enables the virtual memory manager to take corrective action when it finds that a process has a high page fault rate—it can bring about a reduction in the page fault rate by increasing the memory allocated to the process.

As we shall discuss in Section 5.4, the LRU page replacement policy possesses a page fault characteristic that is similar to the curve of Figure 12.6 because it replaces a page that is less likely to be in the current locality of the process than other pages of the process that are in memory.How much memory should the virtual memory manager allocate to a process? Two opposite factors influence this decision. From Figure 5.6, we see that an over commitment of memory to a process implies a low page fault rate for the process; hence it ensures good process performance. However, a smaller number of processes would fit in memory, which could cause CPU idling and poor system performance. An under commitment of memory to a process causes a high page fault rate, which would lead to poor performance of the process.

The desirable operating zone marked in Figure 5.6 avoids the regions of over commitment and under commitment of memory.

The main problem in deciding how much memory to allocate to a process is that the page fault characteristic, i.e., the slope of the curve and the page fault rate in Figure 5.6, varies among processes. Even for the same process, the page fault characteristic may be different when it operates with different data.

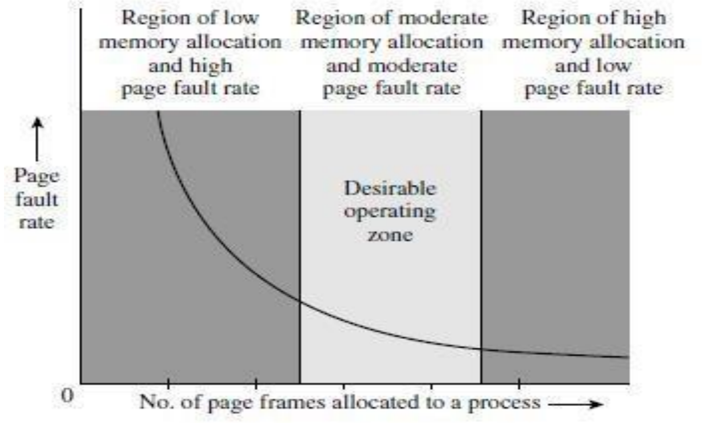


Figure 5.6 Desirable variation of page fault rate with memory allocation.

If all processes in the system operate in the region of high page fault rates, the CPU would be engaged in performing page traffic and process switching most of the time. CPU efficiency would be low and system performance, measured either in terms of average response time or throughput, would be poor. This situation is called thrashing.

Thrashing A condition in which high page traffic and low CPU efficiency coincide. Note that low CPU efficiency can occur because of other causes as well, e.g., if too few processes exist in memory or all processes in memory perform I/O operations frequently. The thrashing situation is different in that all processes make poor progress because of high page fault rates.

From Figure 5.6, we can infer that the cause of thrashing is an under commitment of memory to each process. The cure is to increase the memory allocation for each process. This may have to be achieved by removing some processes from memory—that is, by reducing the degree of multiprogramming.

A process may individually experience a high page fault rate without the system thrashing. The same analysis now applies to the process—it must suffer from an under commitment of memory, so the cure is to increase the amount of memory allocated to it.

3.5.4 Optimal Page Size

The size of a page is defined by computer hardware. It determines the number of bits required to represent the byte number in a page. Page size also determines

1. Memory wastage due to internal fragmentation
2. Size of the page table for a process
3. Page fault rates when a fixed amount of memory is allocated to a process

Consider a process Pi of size z bytes. A page size of s bytes implies that the process has n pages, where n = _z/s_ is the value of z/s rounded upward. Average internal fragmentation is s/2 bytes because the last page would be half empty on the average. The number of entries in the page table is n. Thus internal fragmentation varies directly with the page size, while page table size varies inversely with it.

Address Translation and Page Fault Generation

The MMU follows the steps of Table 5.2 to perform address translation. For a logical address (pi , bi), it accesses the page table entry of pi by using pi ×lPT_entry as an offset into the page table, where lPT_entry is the length of a page table entry.

lPT_entry is typically a power of 2, so pi ×lPT_entry can be computed efficiently by shifting the value of pi by a few bits.

Address Translation Buffers A reference to the page table during address translation consumes one memory cycle because the page table is stored in memory. The translation look-aside buffer (TLB) is a small and fast associative memory that is used to eliminate the reference to the page table, thus speeding up address translation. The TLB contains entries of the form (page #, page frame #, protection info) for a few recently accessed pages of a program that are in memory. During address translation of a logical address (pi , bi), the TLB hardware searches for an entry of page pi . If an entry is found, the page frame # from the entry is used to complete address translation for the logical address (pi , bi). Figure 5.8 illustrates operation of the TLB. The arrows marked 2_ and 3_ indicate TLB lookup. The TLB contains entries for pages 1 and 2 of process P2. If pi is either 1 or 2, the TLB lookup scores a hit, so the MMU takes the page frame number from the TLB and completes address translation. A TLB miss occurs if pi is some other page, hence theMMUaccesses the page table and completes the address translation if page pi is present in memory; otherwise, it generates a page fault, which activates the virtual memory manager to load pi in memory.

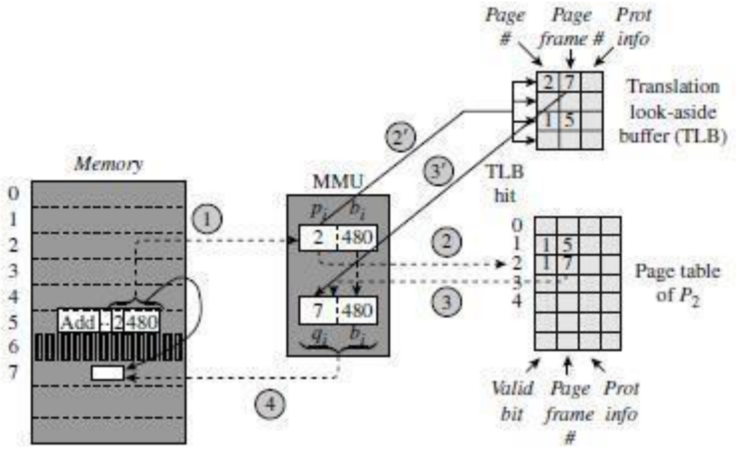


Figure 5.8 Address translation using the translation look-aside buffer and the page table

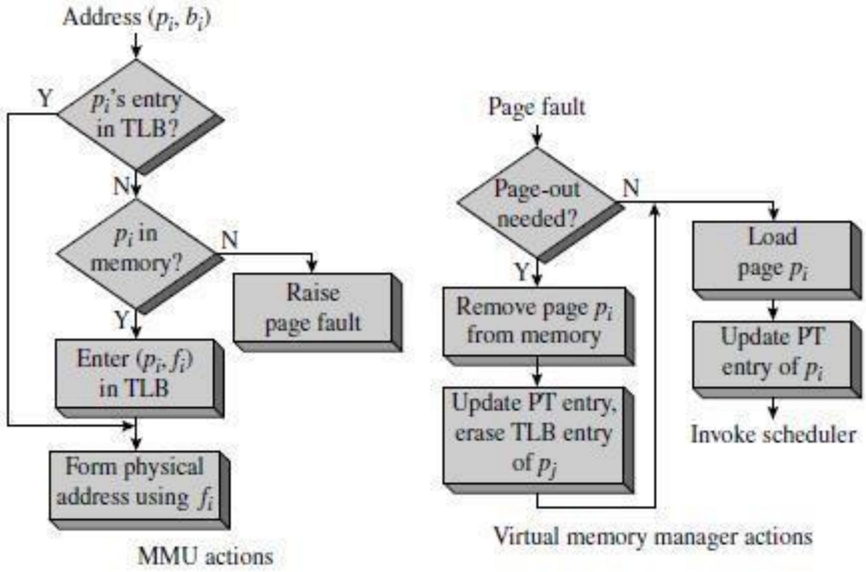


Figure 5.9 Summary of address translation of (pi , bi) (note: PT = page table).

Figure 5.9 summarizes the MMU and software actions in address translation and page fault handling for a logical address (pi , bi). MMU actions concerning use of the TLB and the page table are as described earlier. The virtual memory manager is activated by a page fault. If an empty page frame is not available to load page pi , it initiates a page-out operation for some page pj to free the page frame, say page frame fj , occupied by it. pj 's page table entry is updated to indicate that it is no longer present in memory. If pj has an entry in the TLB, the virtual memory manager erases it by executing an ―erase TLB entry‖ instruction. This action is essential for preventing incorrect address translation at pj 's next reference. A page-in operation is now performed to load pi in page frame fj , and pi 's page table entry is updated when the page-in operation is completed. Execution of the instruction that caused the page fault is repeated when the process is scheduled again. This time pi does not have an entry in the TLB but it exists in memory, and so the MMU uses information in the page table to complete the address translation. An entry for pi has to be made in the TLB at this time.

We use the following notation to compute the effective memory access time when a TLB is used:

pr1 probability that a page exists in memory pr2 probability that a page entry exists in TLB tmem memory access time

tTLB access time of TLB

tpfh time overhead of page fault handling pr1 is the memory hit ratio and tmem is a few orders of magnitude smaller than tpfh. Typically tTLB is at least an order of magnitude smaller than tmem. pr2 is called the TLB hit ratio.

When the TLB is not used, the effective memory access time is as given by Eq. (5.2). The page table is accessed only if the page being referenced does not have an entry in the TLB. Accordingly, a page reference consumes (tTLB +tmem) time if the page has an entry in the TLB, and (tTLB + 2 × tmem) time if it does not have a TLB entry but exists in memory. The probability of the latter situation is (pr1− pr2). When the TLB is used, pr2 is the probability that an entry for the required page exists in the TLB. The probability that a page table reference is both necessary and sufficient for address translation is (pr1− pr2). The time consumed by each such reference is (tTLB + 2 × tmem) since an unsuccessful TLB search would precede the page table lookup. The probability of a page fault is (1−pr1).

It occurs after the TLB and the page tables have been looked up, and it requires (tpfh + tTLB

+ 2 × tmem) time if we assume that the TLB entry is made for the page while the effective memory address is being calculated. Hence the effective memory access time is

$$\text{Effective memory access time} = pr2 \times (tTLB + tmem) + (pr1 - pr2) \times (tTLB + 2 \times tmem) + (1 - pr1) \times (tTLB + tmem + tpfh + tTLB + 2 \times tmem)$$

(5.3)

To provide efficient memory access during operation of the kernel, most computers provide wired TLB entries for kernel pages. These entries are never touched by replacement algorithms.

3.6 THE VIRTUAL MEMORY MANAGER

The virtual memory manager uses two data structures—the page table, whose entry format is shown in Figure 5.3, and the free frames list. The ref info and modified fields in a page table entry are typically set by the paging hardware. All other fields are set by the virtual memory manager itself. Table 5.4 summarizes the functions of the virtual memory manager.

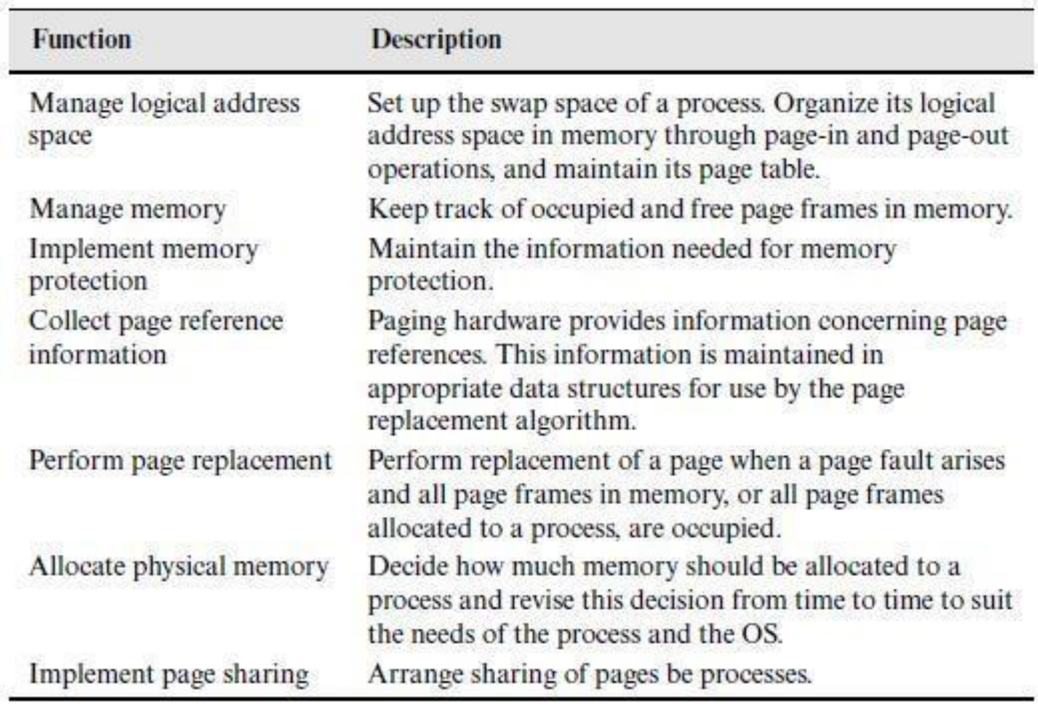
Management of the Logical Address Space of a Process The virtual memory manager manages the logical address space of a process through the following subfunctions:

1. Organize a copy of the instructions and data of the process in its swap space.
2. Maintain the page table.
3. Perform page-in and page-out operations.
4. Perform process initiation.

A copy of the entire logical address space of a process is maintained in the swap space of the process. When a reference to a page leads to a page fault, the page is loaded from the swap space by using a page-in operation. When a dirty page is to be removed from memory, a page-out operation is performed to copy it from memory into a disk block in the swap space. Thus the copy of a page in the swap space is current if that page is not in memory, or it is in memory but it has not been modified since it was last loaded.

For other pages the copy in the swap space is stale (i.e., outdated), whereas that in memory is current.

Table 5.4 **Functions of the Virtual Memory Manager**



Function	Description
Manage logical address space	Set up the swap space of a process. Organize its logical address space in memory through page-in and page-out operations, and maintain its page table.
Manage memory	Keep track of occupied and free page frames in memory.
Implement memory protection	Maintain the information needed for memory protection.
Collect page reference information	Paging hardware provides information concerning page references. This information is maintained in appropriate data structures for use by the page replacement algorithm.
Perform page replacement	Perform replacement of a page when a page fault arises and all page frames in memory, or all page frames allocated to a process, are occupied.
Allocate physical memory	Decide how much memory should be allocated to a process and revise this decision from time to time to suit the needs of the process and the OS.
Implement page sharing	Arrange sharing of pages be processes.

Management of Memory The free frames list is maintained at all times. A page frame is taken off the free list to load a new page, and a frame is added to it when a page-out operation is performed. All page frames allocated to a process are added to the free list when the process terminates.

Protection During process creation, the virtual memory manager constructs its page table and puts information concerning the start address of the page table and its size in the PCB of the process. The virtual memory manager records access privileges of the process for a page in the prot info field of its page table entry.

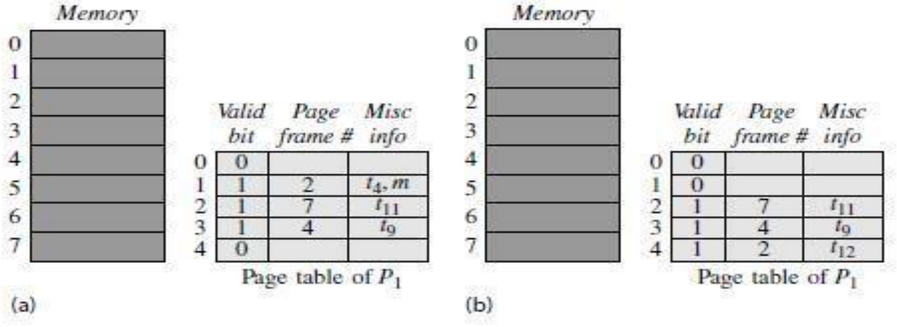
During dispatching of the process, the kernel loads the page-table start address of the process and its page-table size into registers of the MMU. During translation of a logical address (pi , bi), the MMU ensures that the entry of page pi exists in the page table and contains appropriate access privileges in the prot info field.

Collection of Information for Page Replacement The ref info field of the page table entry of a page indicates when the page was last referenced, and the modified field indicates whether it has been modified since it was last loaded in memory.

Page reference information is useful only so long as a page remains in memory; it is reinitialized the next time a page-in operation is performed for the page. Most computers provide a single bit in the ref info field to collect page reference information. This information is not adequate to select the best candidate for page replacement. Hence the virtual memory manager may periodically reset the bit used to store this information.

Example: Page Replacement

The memory of a computer consists of eight page frames. A process P1 consists of five pages numbered 0 to 4. Only pages 1, 2, and 3 are in memory at the moment; they occupy page frames 2, 7, and 4, respectively. Remaining page frames have been allocated to other processes and no free page frames are left in the system.



	Valid bit	Page frame #	Misc info
0	0		
1	1	2	t_4, m
2	1	7	t_{11}
3	1	4	t_9
4	0		

Page table of P_1

	Valid bit	Page frame #	Misc info
0	0		
1	0		
2	1	7	t_{11}
3	1	4	t_9
4	1	2	t_{12}

Page table of P_1

Figure 5.13 Data structures of the virtual memory manager: (a) before and (b) after a page replacemen

Figure 4.13(a) illustrates the situation in the system at time instant t+11, i.e., a little after t11. Only the page table of P1 is shown in the figure since process P1 has been scheduled. Contents of the ref info and modified fields are shown in the misc info field. Pages 1, 2, and 3 were last referenced at time instants t4, t11, and t9, respectively. Page 1 was modified sometime after it was last loaded. Hence the misc info field of its page table entry contains the information t4,m.

At time instant t12, process P1 gives rise to a page fault for page 4. Since all page frames in memory are occupied, the virtual memory manager decides to replace page 1 of the process. The mark m in the misc info field of page 1‘s page table entry indicates that it was modified since it was last loaded, so a page-out\ operation is necessary. The page frame # field of the page table entry of page 1 indicates that the page exists in page frame 2. The virtual memory manager performs a page-out operation to write the contents of page frame 2 into the swap area reserved for page 1 of P1, and modifies the valid bit in the page table entry of page 1 to indicate that it is not present in memory. A page-in operation is now initiated for page 4 of P1. At the end of the operation, the page table entry of page 4 is modified to indicate that it exists in memory in page frame 2.

Execution of P1 is resumed. It now makes a reference to page 4, and so the page reference information of page 4 indicates that it was last referenced at t12. Figure 5.13(b) indicates the page table of P1 at time instant t+12.

Overview of Operation of the Virtual Memory Manager

The virtual memory manager makes two important decisions during its operation:

- When a page fault occurs during operation of some process proci , it decides which page should be replaced.

- Periodically it decides how much memory, i.e., howmany page frames, should be allocated to each process.

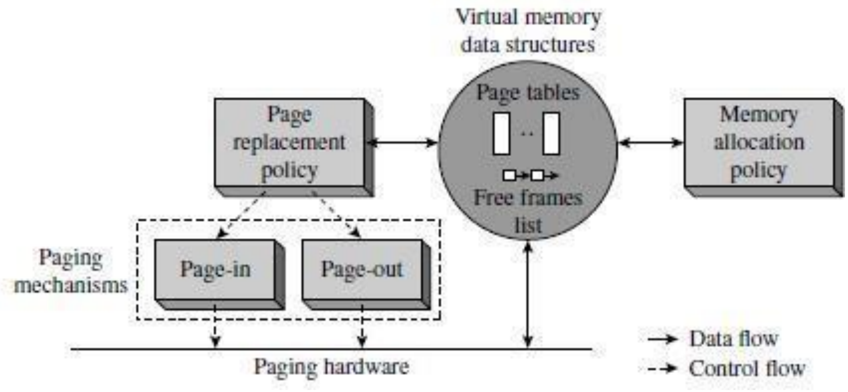


Figure 5.14 Modules of the virtual memory manager.

3.7 PAGE REPLACEMENT POLICIES

A page replacement policy should replace a page that is not likely to be referenced in the immediate future. We evaluate the following three page replacement policies to see how well they fulfill this requirement.

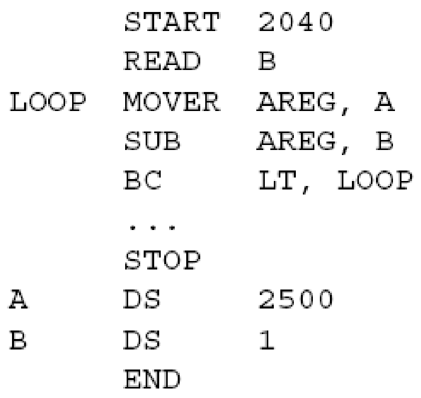
- Optimal page replacement policy
- First-in, first-out (FIFO) page replacement policy
- Least recently used (LRU) page replacement policy

For the analysis of these page replacement policies, we rely on the concept of page reference strings. A page reference string of a process is a trace of the pages accessed by the process during its operation. It can be constructed by monitoring the operation of a process, and forming a sequence of page numbers that appear in logical addresses generated by it. The page reference string of a process depends on the data input to it, so use of different data would lead to a different page reference string for a process.

For convenience we associate a reference time string t1, t2, t3, . . . with each page reference string. This way, the kth page reference in a page reference string is assumed to have occurred at time instant tk. (In effect, we assume a logical clock that runs only when a process is in the running state and gets advanced only when the process refers to a logical address.) Example 5.5 illustrates the page reference string and the associated reference time string for a process.

Page Reference String Example 5.5

A computer supports instructions that are 4 bytes in length, and uses a page size of 1KB. It executes the following nonsense program in which the symbols A and B are in pages 2 and 5, respectively:



```
        START  2040
        READ   B
LOOP    MOVER  AREG, A
        SUB    AREG, B
        BC     LT, LOOP
        ...
        STOP
A       DS     2500
B       DS     1
        END
```

The page reference string and the reference time string for the process
are as follows:

Page reference string 1, 5, 1, 2, 2, 5, 2, 1, . . .
Reference time string t1, t2, t3, t4, t5, t6, t7, t8, . . .

The logical address of the first instruction is 2040, and so it lies in page 1. The first page reference in the string is therefore 1. It occurs at time instant t1. B, the operand of the instruction is situated in page 5, and so the second page reference in the string is 5, at time t2. The next instruction is located in page 1 and refers to A, which is located in page 2, and thus the next two page references are to pages 1 and 2. The next two instructions are located in page 2, and the instruction with the label LOOP is located in page 1. Therefore, if the value of B input to the READ statement is greater than the value of A, the next four page references would be to pages 2, 5, 2 and 1, respectively; otherwise, the next four page references would be to pages 2, 5, 2 and 2, respectively.

Optimal Page Replacement Optimal page replacement means making page replacement decisions in such a manner that the total number of page faults during operation of a process is the minimum possible; i.e., no other sequence of page replacement decisions can lead to a smaller number of page faults. To achieve optimal page replacement, at each page fault, the page replacement policy would have to consider all alternative page replacement decisions, analyze their implications for future page faults, and select the best alternative. Of course, such a policy is infeasible in reality: the virtual memory manager does not have knowledge of the future behaviour of a process. As an analytical tool, however, this policy provides a useful comparison in hindsight for the performance of other page replacement policies.

Although optimal page replacement might seem to require excessive analysis, Belady (1966) showed that it is equivalent to the following simple rule: At a page fault, replace the page whose next reference is farthest in the page reference string.

FIFO Page Replacement At every page fault, the FIFO page replacement policy replaces the page that was loaded into memory earlier than any other page of the process. To facilitate FIFO page replacement, the virtual memory manager records the time of loading of a page in the ref info field of its page table entry.

When a page fault occurs, this information is used to determine pearliest, the page that was loaded earlier than any other page of the process. This is the page that will be replaced with the page whose reference led to the page fault.

LRU Page Replacement The LRU policy uses the law of locality of reference as the basis for its replacement decisions. Its operation can be described as follows: At every page fault the least recently used (LRU) page is replaced by the required page. The page table entry of a page records the time when the page was last referenced. This information is initialized when a page is loaded, and it is updated every time the page is referenced. When a page fault occurs, this information is used to locate the page pLRU whose last reference is earlier than that of every other page. This page is replaced with the page whose reference led to the page fault.

Analysis of Page Replacement Policies Example 5.6 illustrates operation
of the optimal, FIFO, and LRU page replacement policies.

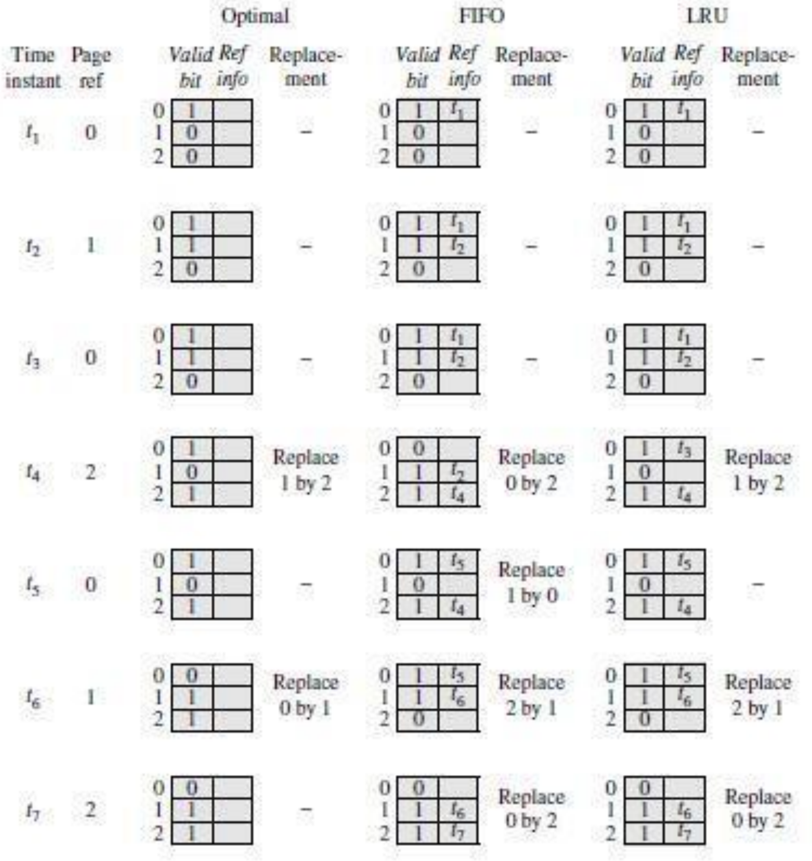
Example 5.6 Operation of Page Replacement Policies

A page reference string and the reference time string for a process P are as follows:

Page reference string 0, 1, 0, 2, 0, 1, 2, . . . **(5.4)**

Reference time string t1, t2, t3, t4, t5, t6, t7, . . . **(5.5)**

Figure 5.15 illustrates operation of the optimal, FIFO and LRU page replacement policies for this page reference string with alloc = 2. For convenience, we show only two fields of the page table, valid bit and ref info. In the interval t0 to t3 (inclusive), only two distinct pages are referenced: pages 0 and 1. They can both be accommodated in memory at the same time because alloc = 2. t4 is the first time instant when a page fault leads to page replacement.



Time instant	Page ref		Optimal Valid bit	Optimal Ref info	Optimal Replacement	FIFO Valid bit	FIFO Ref info	FIFO Replacement	LRU Valid bit	LRU Ref info	LRU Replacement
t_1	0	0	1		–	1	t_1	–	1	t_1	–
		1	0			0			0		
		2	0			0			0		
t_2	1	0	1		–	1	t_1	–	1	t_1	–
		1	1			1	t_2		1	t_2	
		2	0			0			0		
t_3	0	0	1		–	1	t_1	–	1	t_1	–
		1	1			1	t_2		1	t_2	
		2	0			0			0		
t_4	2	0	1		Replace 1 by 2	0		Replace 0 by 2	1	t_3	Replace 1 by 2
		1	0			1	t_2		0		
		2	1			1	t_4		1	t_4	
t_5	0	0	1		–	1	t_5	Replace 1 by 0	1	t_5	–
		1	0			0			0		
		2	1			1	t_4		1	t_4	
t_6	1	0	0		Replace 0 by 1	1	t_5	Replace 2 by 1	1	t_5	Replace 2 by 1
		1	1			1	t_6		1	t_6	
		2	1			0			0		
t_7	2	0	0		–	0		Replace 0 by 2	0		Replace 0 by 2
		1	1			1	t_6		1	t_6	
		2	1			1	t_7		1	t_7	

Figure 5.15 Comparison of page replacement policies with alloc = 2.

The left column shows the results for optimal page replacement. Page reference information is not shown in the page table since information concerning past references is not needed for optimal page replacement. When the page fault occurs at time instant t4, page 1 is replaced because its next reference is farther in the page reference string than that of page 0. At time t6 page 1 replaces page 0 because page 0‘s next reference is farther than that of page 2.

The middle column of Figure 5.15 shows the results for the FIFO replacement policy. When the page fault occurs at time t4, the ref info field shows that page 0 was loaded earlier than page 1, and so page 0 is replaced by page 2.

The last column of Figure 5.15 shows the results for the LRU replacement policy. The ref info field of the page table indicates when a page was last referenced. At time t4, page 1 is replaced by page 2 because the last reference of page 1 is earlier than the last reference of page 0.

The total number of page faults occurring under the optimal, FIFO, and LRU policies are

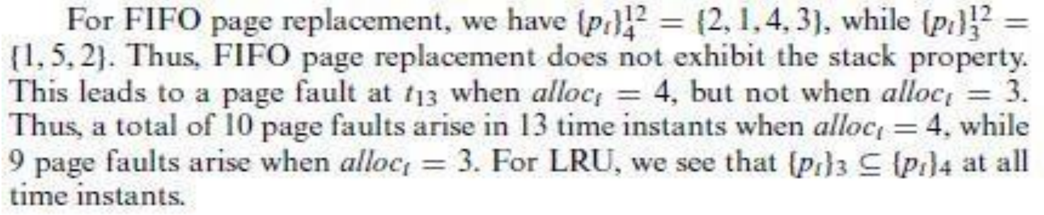
4, 6, and 5, respectively. By definition, no other policy has fewer page faults than the optimal page replacement policy.

Problems in FIFO Page
Replacement Example
5.7

Consider the following page reference and reference time strings for a process:

Page reference string 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5, . . . **(5**
.6) Reference time string t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, . . . **(5**
.7)

Figure 5.17 shows operation of the FIFO and LRU page replacement policies for this page reference string. Page references that cause page faults and result in page replacement are marked with a * mark. A column of boxes is associated with each time instant. Each box is a page frame; the number contained in it indicates which page occupies it after execution of the memory reference marked under the column.



For FIFO page replacement, we have $\{p_i\}_4^{12} = \{2, 1, 4, 3\}$, while $\{p_i\}_3^{12} = \{1, 5, 2\}$. Thus, FIFO page replacement does not exhibit the stack property. This leads to a page fault at t_{13} when $alloc_i = 4$, but not when $alloc_i = 3$. Thus, a total of 10 page faults arise in 13 time instants when $alloc_i = 4$, while 9 page faults arise when $alloc_i = 3$. For LRU, we see that $\{p_i\}_3 \subseteq \{p_i\}_4$ at all time instants.

•

Figure 5.18 illustrates the page fault characteristic of FIFO and LRU page replacement for page reference string (12.6). For simplicity, the vertical axis shows the total number of page faults rather than the page fault frequency.

Figure 5.18(a) illustrates an anomaly in behavior of FIFO page replacement— the number of page faults increases when memory allocation for the process is increased. This anomalous behavior was first reported by Belady and is therefore known as Belady‘s anomaly.

Figure 5.18 (a) Belady's anomaly in FIFO page replacement; (b) page fault characteristic for LRU page replacement.

The virtual memory manager cannot use FIFO page replacement because increasing the allocation to a process may increase the page fault frequency of the process. This feature would make it difficult to combat thrashing in the system.

However, when LRU page replacement is used, the number of page faults is a nonincreasing function of alloc. Hence it is possible to combat thrashing by increasing the value of alloc for each process.

QUESTION BANK

1. Explain the important concepts in the operation of demand paging.
2. Write a note on page replacement policies.
3. How can virtual memory be implemented?
4. Explain FIFO and LRU page replacement policy.
5. What are the functions performed by a virtual memory manager? Explain.
6. For the following page reference string, calculate the number of page faults with FIFO and LRU page replacement policies when i) number of page frames=3 ii) number of page frames=4.
   Page reference string: 5 4 3 2 1 4 3 5 4 3 2 1 5. Reference time string: t1,t2,……,t13
7. Describe the address translation using TU and TLB in demand paged allocation with a block diagram.

MODULE 4

FILE SYSTEMS

Structure

Objectives

4.1 File system and IOCS

4.2 Files operation

4.3 File Organization

4.4 Directory and structures

4.5 File protection

4.6 Interface between file systems and IOCS

4.7 Allocation of Disk Space

4.8 Implementing file access

4.9 Questions

4.10 Further Readings

OBJECTIVES

Upon Completion of this chapter, the student should be able to:

- Discuss the concepts of file systems and IOCS.
- Discuss the concepts of file organization.
- Explain the interface between file systems and IOCS.
- Explain the concepts of allocations in disk.

4.1 FILE SYSTEMS AND IOCS

- A file system views a file as a collection of data that is owned by a user, can be shared by a set of authorized users, and has to be reliably stored over an extended period of time. A files System gives users freedom in naming their files, as an aspect of ownership, so that a user can give a desired name to a file without worrying whether it conflicts with names of files; and it provides privacy by protecting against interference by other users.

- The IOCS, on the other hand, views a file as a repository of data that need to be accessed speedily and are stored on an I/O device that needs to be used efficiently. Table 6.1 summarizes the facilities provided by the file system and the IOCS.

Facilities provided by the file system and the IOCS are

File system:

- Directory structures for convenient grouping of files
- Protection of files against illegal accesses
- File sharing semantics
- Reliable storage of files

IOCS:

- Efficient operation of I/O devices
- Efficient access to records in a file

4.2 Files and file operation

File Types A file system houses and organizes different types of files, e.g., data files, executable programs, object modules, textual information, documents, spreadsheets, photos, and video clips. Each of these file types has its own format for recording the data. These file types can be grouped into two classes:

- Structured files
- Byte stream files

A **structured file** is a collection of records, where a record is a meaningful unit for processing of data. A *record* is a collection of fields, and a *field* contains a single data item. Each record in a file is assumed to contain a *key* field. The value in the key field of a record is unique in a file; i.e., no two records contain an identical key.

Many file types mentioned earlier are structured files. File types used by standard system software like compilers and linkers have a structure determined by the OS designer, while file types of user files depend on the applications or programs that create them.

A **byte stream file** is flat. There are no records and fields in it. It is looked upon as a Sequence of bytes by the processes that use it. The next example illustrates structured and byte stream files. Structured and Byte Stream Files Example 4.1 (a) shows a structured file named employee info. Each record in the file contains information about one employee. A record contains four fields: employee id, name, designation, and age. The field containing the employee id is the key field. Figure 4.1(b) shows a byte stream file report.

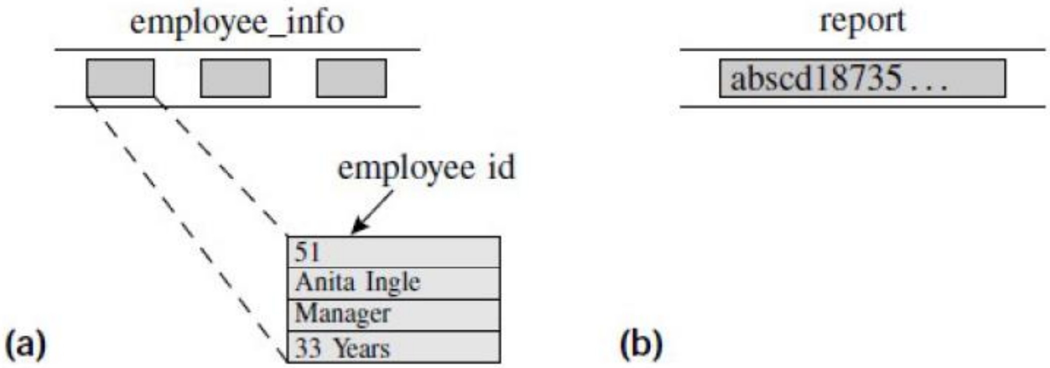


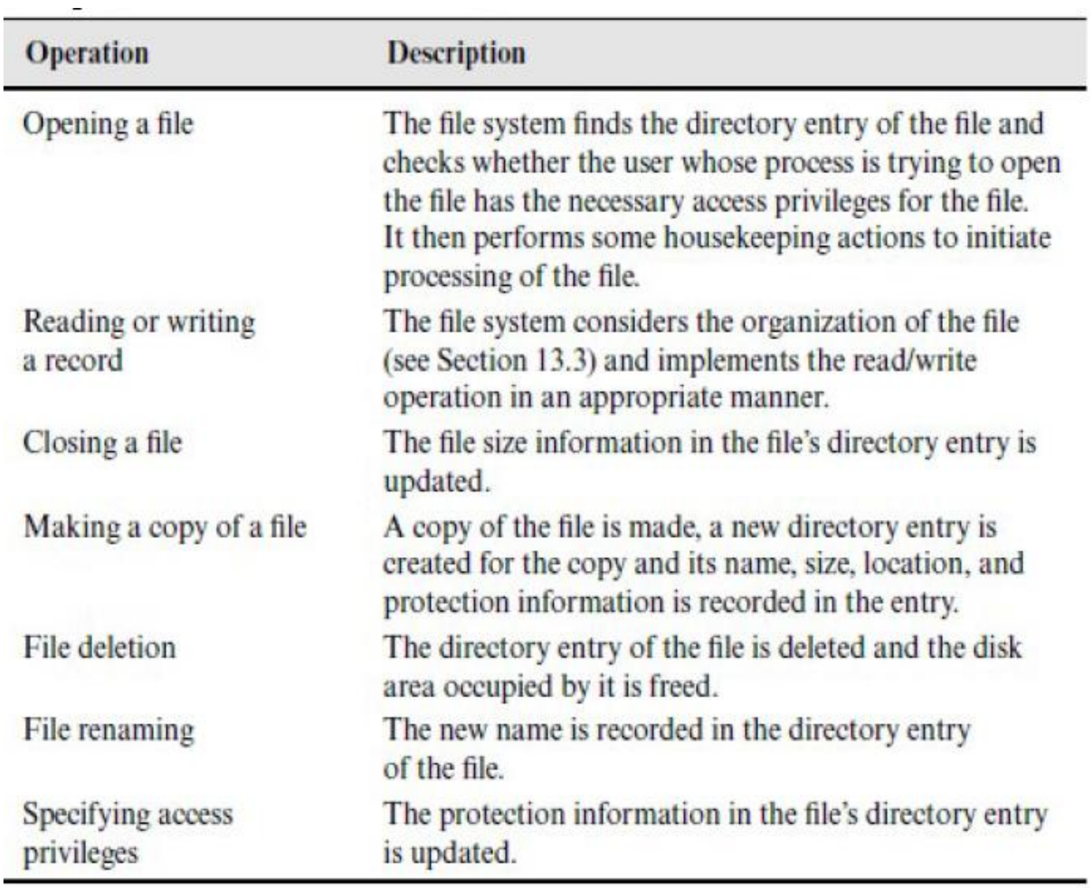
Fig 4.1 (a) logical views of structured file (b) stream file support

File Attributes: A file attribute is a characteristic of a file that is important either to its users or to the file system, or both. Commonly used attributes of a file are: type, organization, size, location on disk, access control information, which indicates the manner in which different users can access the file; owner name, time of creation, and time of last use. The file system stores the attributes of a file in its directory entry. During a file processing activity, the file system uses the attributes of a file to locate it, and to ensure that each operation being performed on it is consistent with its attributes.

File operation

- Table 4.1 Facilities the different operations of file system

Table 4.1 Operation of file



Operation	Description
Opening a file	The file system finds the directory entry of the file and checks whether the user whose process is trying to open the file has the necessary access privileges for the file. It then performs some housekeeping actions to initiate processing of the file.
Reading or writing a record	The file system considers the organization of the file (see Section 13.3) and implements the read/write operation in an appropriate manner.
Closing a file	The file size information in the file's directory entry is updated.
Making a copy of a file	A copy of the file is made, a new directory entry is created for the copy and its name, size, location, and protection information is recorded in the entry.
File deletion	The directory entry of the file is deleted and the disk area occupied by it is freed.
File renaming	The new name is recorded in the directory entry of the file.
Specifying access privileges	The protection information in the file's directory entry is updated.

4.3 FUNDAMENTAL FILE ORGANISATIONS

- A **file organization** is a combination of two features a method of arranging records in a file and a procedure for accessing them. A file organization is designed to exploit the characteristics of an I/O device for providing efficient record access for a specific record access pattern. A file system supports several file organizations so that a process can employ the one that best suits its file processing requirements and the I/O device in use. This section describes three fundamental file organizations sequential file organization, direct file organization and index sequential file organization. Other file organizations used in practice are either variants of these fundamental ones or are special-purpose organizations that exploit less commonly used I/O devices. Accesses to files governed by a specific file organization are implemented by an IOCS module called an *access method*. An access method is a policy module of the IOCS.

- In **sequential file organization,** records are stored in an ascending or descending sequence according to the key field; the record access pattern of an application is expected to follow suit. Hence sequential file organization supports two kinds of operations: read the next (or previous) record, and skip the next (or previous) record. A sequential-access file is used in an application if its data can be conveniently pre-sorted into an ascending or descending order. The sequential file organization is also used for byte stream files.
- The **direct file organization** provides convenience and efficiency of file processing when records are accessed in a random order. To access a record, a read/write command needs to mention the value in its key field. We refer to such files as *direct access files*. A direct-access file is implemented as follows: When a process provides the key value of a record to be accessed, the access method module for the direct file organization applies a transformation to the key value that generates the address of the record in the storage medium. If the file is Organized on a disk, the transformation generates a (*track no*, *record no*) address. The disk heads are now positioned on the track *track no* before a read or write command is issued on the record *record no*.
- The **index sequential file organization** is a hybrid organization that combines elements of the indexed and the sequential file organizations. To locate a desired record, the access method module for this organization searches an index to identify a section of the disk that *may* contain the record, and searches the records in this section of the disk sequentially to find the record. The search succeeds if the record is present in the file; otherwise, it results in a failure. This arrangement requires a much smaller index than does a pure indexed file because the index contains entries for only some of the key values. It also provides better access efficiency than the sequential file organization while ensuring comparably efficient use of I/O media.

4.4 Directory Structure

A directory contains information about a group of files. Each entry in a directory contains the attributes of one file, such as its type, organization, size, location, and the manner in which it may be accessed by various users in the system. Figure 4.2 shows the fields of a typical directory entry. The *open count* and *lock* fields are used when several processes open a file concurrently.

The *open count* indicates the number of such processes. As long as this count is non zero, the file system keeps some of the metadata concerning the file in memory to speedup accesses to the data in the file. The *lock* field is used when a process desires exclusive access to a file.

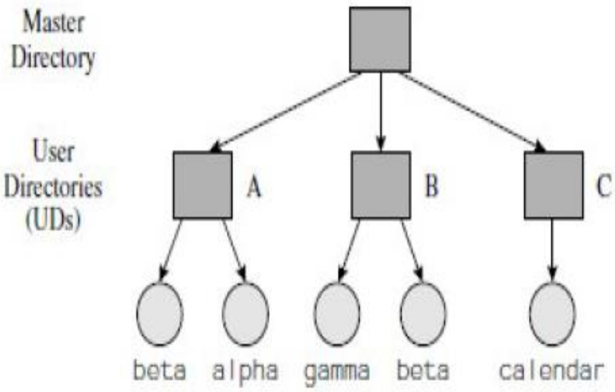
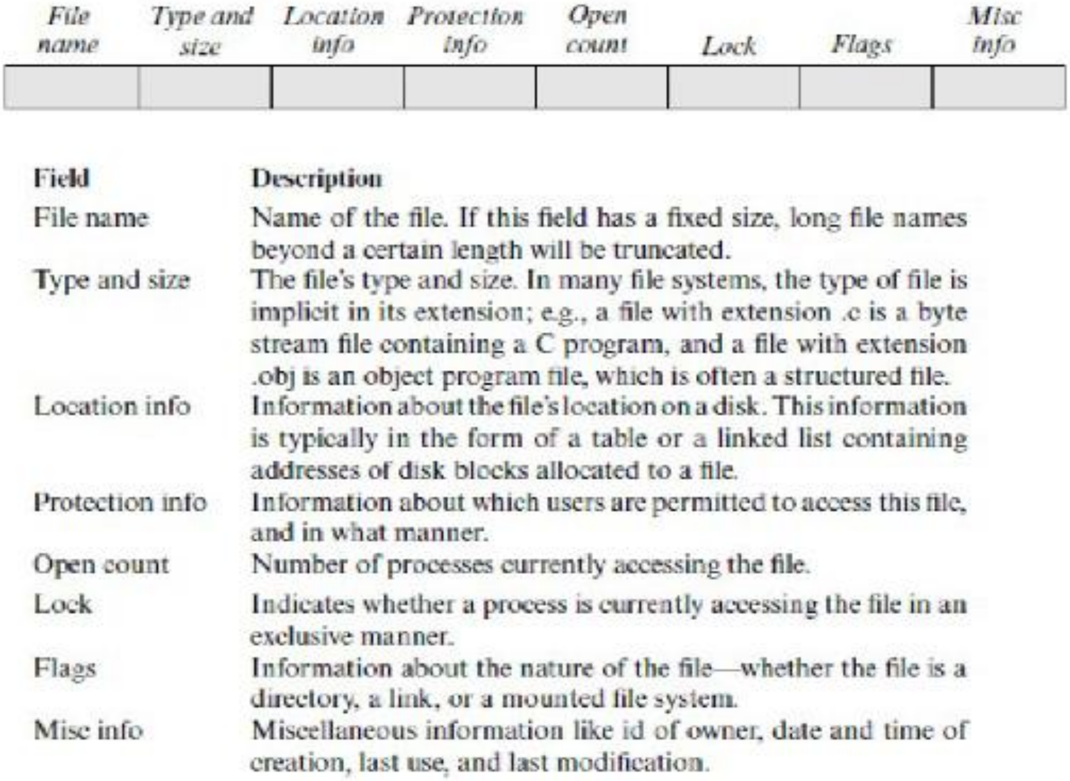


Fig4.2 Directory structure composed of files and directories


File name
Type and size
Location info
Protection info
Open count
Lock
Flags
Misc info

Field	Description
File name	Name of the file. If this field has a fixed size, long file names beyond a certain length will be truncated.
Type and size	The file's type and size. In many file systems, the type of file is implicit in its extension; e.g., a file with extension .c is a byte stream file containing a C program, and a file with extension .obj is an object program file, which is often a structured file.
Location info	Information about the file's location on a disk. This information is typically in the form of a table or a linked list containing addresses of disk blocks allocated to a file.
Protection info	Information about which users are permitted to access this file, and in what manner.
Open count	Number of processes currently accessing the file.
Lock	Indicates whether a process is currently accessing the file in an exclusive manner.
Flags	Information about the nature of the file—whether the file is a directory, a link, or a mounted file system.
Misc info	Miscellaneous information like id of owner, date and time of creation, last use, and last modification.

Fig 4.3 Fields in a typical directory entry

Directory Trees

The MULTICS file system of the 1960s contained features that allowed the user to create a new directory, give it a name of his choice, and create files and other directories in it up to any desired level. The resulting directory structure is a *tree*, it is called the *directory tree*. After MULTICS, most file systems have provided directory trees.

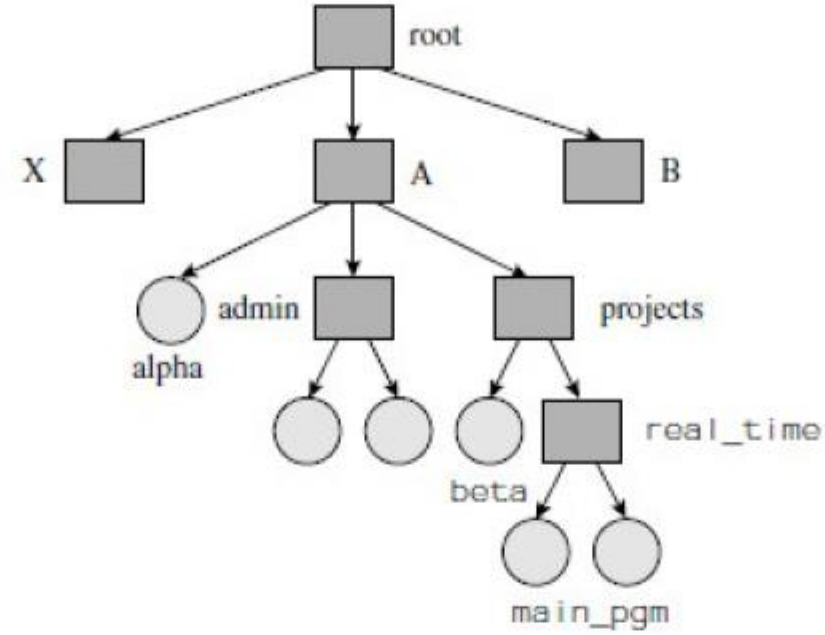


Fig 4.4 Directory trees of file system and of user A

A user can create a file to hold data or to act as a directory. When a distinction between the two is important, we will call these files respectively *data files* and *directory files*, or simply *directories*.

In a **directory tree**, each file except the root directory has exactly one parent directory. This provides total separation of different users files and complete file naming freedom. However, it makes file sharing rather cumbersome

Links: A *link* is a directed connection between two existing files in the directory structure. It can be written as a triple (<*from_ file_name*>, <*to_file_name*>, <*link_name*>), where<*from_ file_name*>is a directory and<*to_ file_name*>can be a directory or a file. Once a link is established, <*to_ file_name*>can be accessed as if it were a file named<*link_name*> in the directory <*from_ file_name*>.

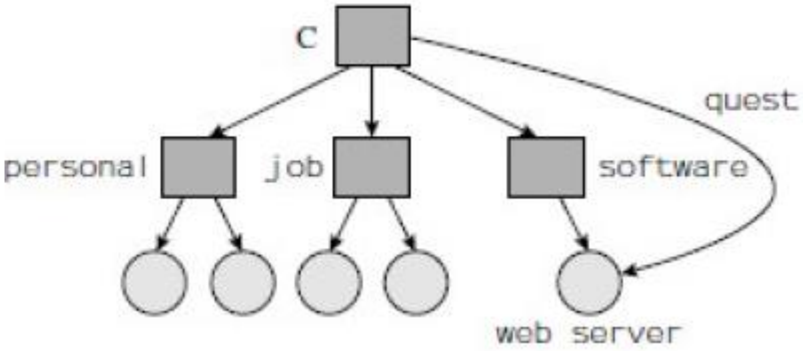


Fig 4.5 A link in the directory structure

4.5 FILE PROTECTION

A user would like to share a file with collaborators, but not with others. We call this requirement *controlled sharing* of files. To implement it, the owner of a file specifies which users can access the file in what manner. The file system stores this information in the *protection info field of the file*s directory entry, and uses it to control access to the file.

4.6 INTERFACES BETWEEN FILE SYSTEM AND IOCS

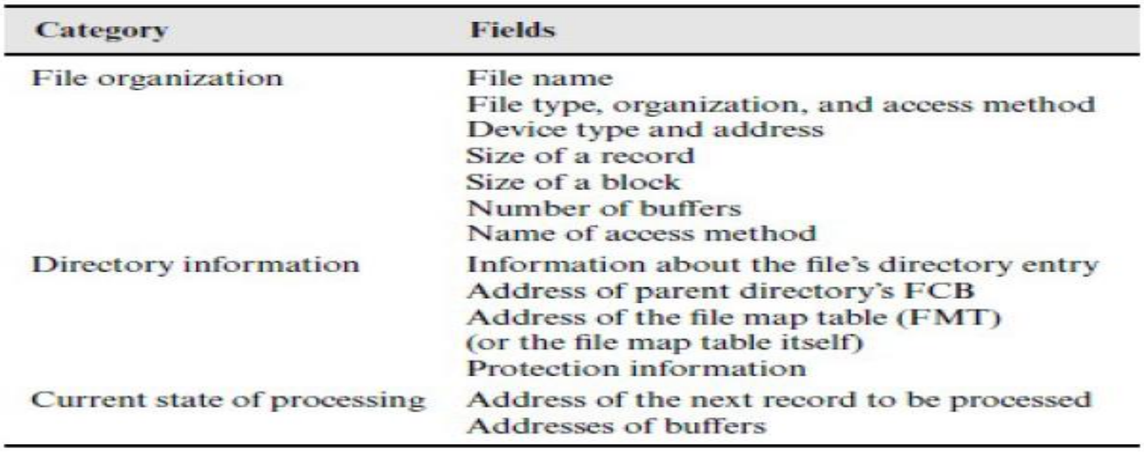
The file system uses the IOCS to perform I/O operations and the IOCS implements them rough kernel calls. The interface between the file system and the IOCS consists of three data structures the *file map table* (FMT), the *file control block* (FCB), and the *open files table* (OFT) and functions that perform I/O operations. Use of these data structures avoids repeated processing of file attributes by the file system, and provides a convenient method of tracking the status of ongoing file processing activities. The file system allocates disk space to a file and stores information about the allocated disk space in the *file map table* (FMT). The FMT is typically held in memory during the processing of a file.

A **file control block (FCB)** contains all information concerning an ongoing file processing activity.

- This information can be classified into the three categories shown in Table 6.2.Information in the file organization category is either simply extracted from the file declaration statement in an application program, or inferred from it by the compiler, e.g., information such as the size of a record and number of buffers is extracted from a file declaration, while the name of the access method is inferred from the type and organization of a file.

- The compiler puts this information as parameters in the open call.
- When the call is made during execution of the program, the file system puts this information in the FCB.
- Directory information is copied into the FCB through joint actions of the file system and the IOCS when a new file is created.
- Information concerning the current state of processing is written into the FCB by the IOCS. This information is continually updated during the processing of a file. The *open files table* (OFT) holds the FCBs of all open files.
- The OFT resides in the kernel address space so that user processes cannot tamper with it. When a file is opened, the file system stores its FCB in a new entry of the OFT. The offset of this entry in the OFT is called the *internal id* of the file.
- The internal id is passed back to the process, which uses it as a parameter in all future file system calls.
- Figure 6.7 shows the arrangement set up when a file alpha is opened. The file system copies *fmt* alpha in memory; creates *fcb* alpha, which is an FCB for alpha, in the OFT; initializes its fields appropriately; and passes back its offset in OFT, which in this case is 6, to the process as *internal_id* alpha.

Table 4.2 Fields in FCB



Category	Fields
File organization	File name
	File type, organization, and access method
	Device type and address
	Size of a record
	Size of a block
	Number of buffers
	Name of access method
Directory information	Information about the file's directory entry
	Address of parent directory's FCB
	Address of the file map table (FMT) (or the file map table itself)
	Protection information
Current state of processing	Address of the next record to be processed
	Addresses of buffers

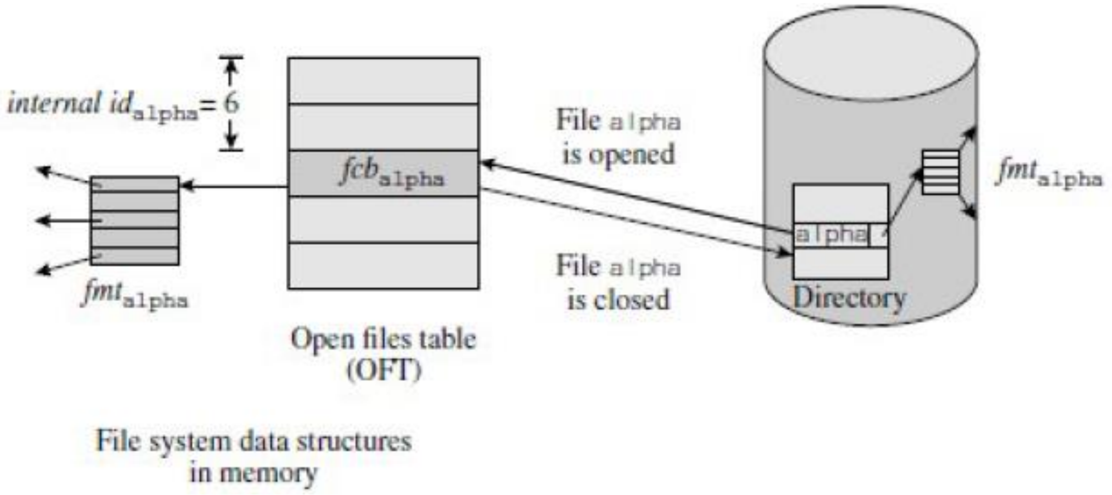


Fig 4.6 Interface between file system and IOCS OFT, FCB and FMT.

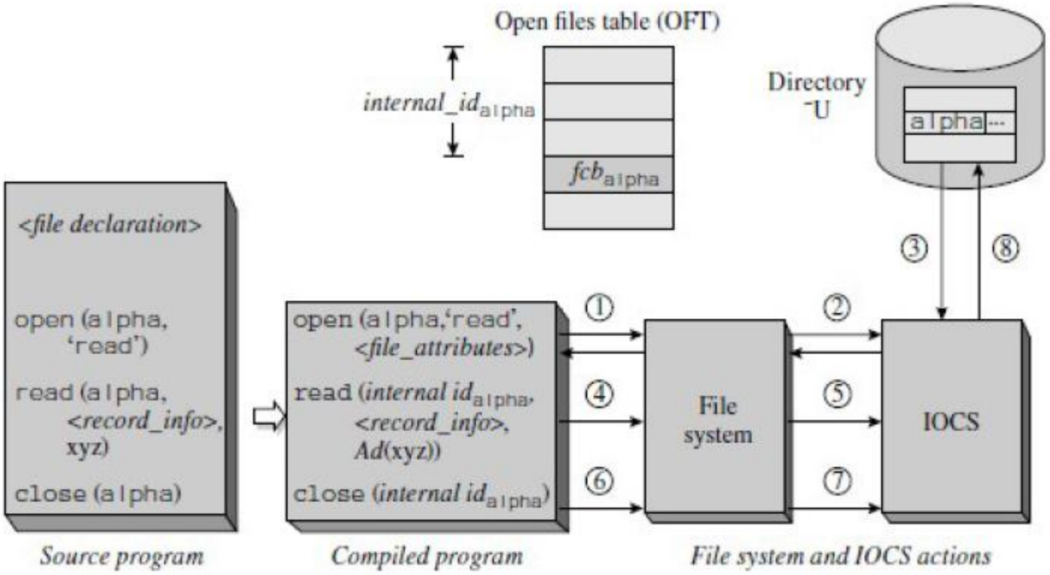

Open files table (OFT)
internal_id alpha
fcb alpha
Directory U
alpha
<file declaration>
open (alpha, 'read')
read (alpha, <record_info>, xyz)
close (alpha)
open (alpha,'read', <file_attributes>)
read (internal id alpha, <record_info>, Ad(xyz))
close (internal id alpha)
①
②
③
④
⑤
⑥
⑦
⑧
File system
IOCS
Source program
Compiled program
File system and IOCS actions

Fig 4.7 Files processing operations

Figure 4.7 is a schematic diagram of the processing of an existing file alpha in a process executed by some user U. The compiler replaces the statements open, read, and close in the source program with calls on the file system operations open, read, and close, respectively. The following are the significant steps in file processing involving the file system and the IOCS, shown by numbered arrows in Figure 4.7:

- The file executes the call open (alpha, read**,**<*file attributes*>). The call returns with *internal_id* with protection information of the file. The process saves *internal_id*alpha for use while performing operations on file alpha.
- The file system creates a new FCB in the open files table. It resolves the path name alpha, locates the directory entry of alpha, and stores the information about it in the new FCB for use while closing the file. Thus, the new FCB becomes *fcb* alpha. The file system now makes a call IOCS-open with *internal_id* alpha and the address of the directory entry of alpha as parameters.
- The IOCS accesses the directory entry of alpha, and copies the file size and address of the FMT, or the FMT itself, from the directory entry into *fcb* alpha.
- When the process wishes to read a record of alpha into area xyz, it invokes the read operation of the file system with *internal_id*alpha, <*record_info*>, and *Ad* (xyz) as parameters.
- Information about the location of alpha is now available in *fcb* alpha. Hence the read/write operations merely invoke IOCS-read/write operations.
- The process invokes the close operation with *internal_id*alpha as a parameter.
- The file system makes a call IOCS-close with *internal_id*alpha.
- The IOCS obtains information about the directory entry of alpha from *fcb* alpha and copies the file size and FMT address, or the FMT itself, from *fcb* alpha into the directory entry of alpha.

4.7 ALLOCATION OF DISK SPACE

A disk may contain many file systems, each in its own partition of the disk. The file system knows which partition a file belongs to, but the IOCS does not. Hence disk space allocation is performed by the file system.

Early file systems adapted the contiguous memory allocation model by allocating a single contiguous disk area to a file when it was created. This model was simple to implement. It also provided data access efficiency by reducing disk head movement during sequential access to data in a file. However, contiguous allocation of disk space led to external fragmentation.

Interestingly, it also suffered from internal fragmentation because the file system found it prudent to allocate some extra disk space to allow for expansion of a file.

Contiguity of disk space also necessitated complicated arrangements to avoid use of bad disk blocks: The file system identified bad disk blocks while formatting the disk and noted their addresses. It then allocated substitute disk blocks for the bad ones and built a table showing addresses of bad blocks and their substitutes. During a read/write operation, the IOCS checked whether the disk block to be accessed was a bad block. If it was, it obtained the address of the substitute disk block and accessed it. Modern file systems adapt the non contiguous memory allocation model to disk space allocation. In this approach, a chunk of disk space is allocated on demand, i.e., when the file is created or when its size grows because of an update operation. The file system has to address three issues for implementing this approach:

Managing free disk space*:* Keep track of free disk space and allocate from it when a file requires a new disk block.

Avoiding excessive disk head movement*:* Ensure that data in a file is not dispersed to different parts of a disk, as it would cause excessive movement of the disk heads during file processing.

Accessing file data*:* Maintain information about the disk space allocated to a file and use it
To find the disk block that contains required data.

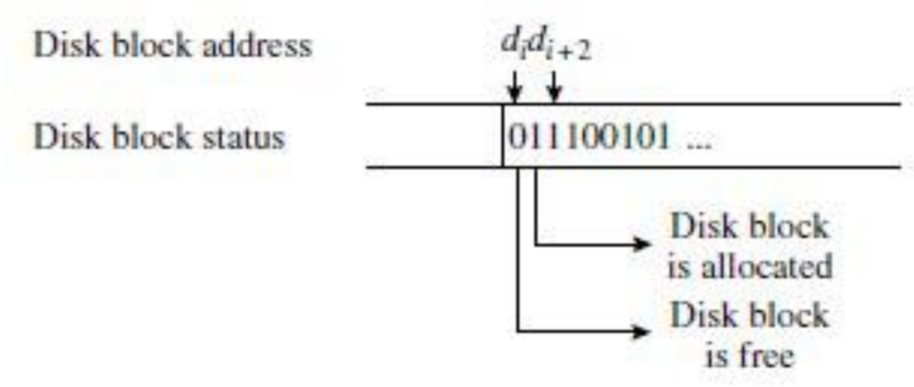


Fig 4.9 Disk Status Map

Use of a disk status map, rather than a free list, has the advantage that it allows the file system to readily pick disk blocks from an extent or cylinder group.

The two fundamental approaches to non contiguous disk space allocation. They differ in the manner they maintain information about disk space allocated to a file.

Linked Allocation

A file is represented by a linked list of disk blocks. Each disk block has two fields in it *data* and *metadata*. The *data* field contains the data written into the file, while the *metadata* field is the link field, which contains the address of the next disk block allocated to the file. Figure4.10 illustrates linked allocation. The *location info* field of the directory entry of file alpha points to the first disk block of the file. Other blocks are accessed by following the pointers in the list of disk blocks. The last disk block contains null information in its metadata field. Thus, file alpha consists of disk blocks 3 and 2, while file beta consists of blocks 4, 5,and 7.

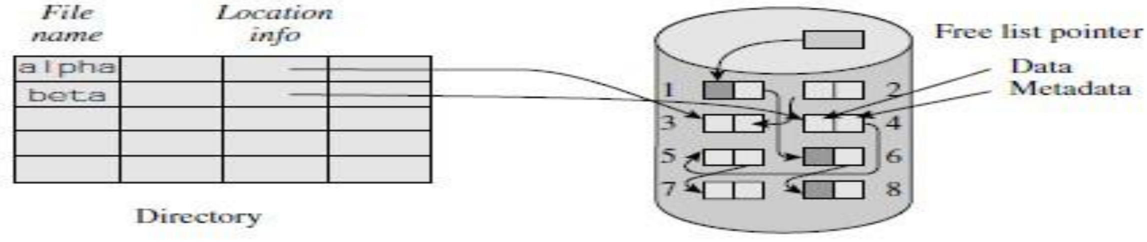


Fig 4.10 linked allocation of disk space

File Allocation Table (FAT)

MS-DOS uses a variant of linked allocation that stores the metadata separately from the file data. A **file allocation table (FAT)** of a disk is an array that has one element corresponding to every disk block in the disk. For a disk block that is allocated to a file, the corresponding FAT element contains the address of the next disk block. Thus the disk block and its FAT element together form a pair that contains the same information as the disk block in a classical linked allocation scheme.

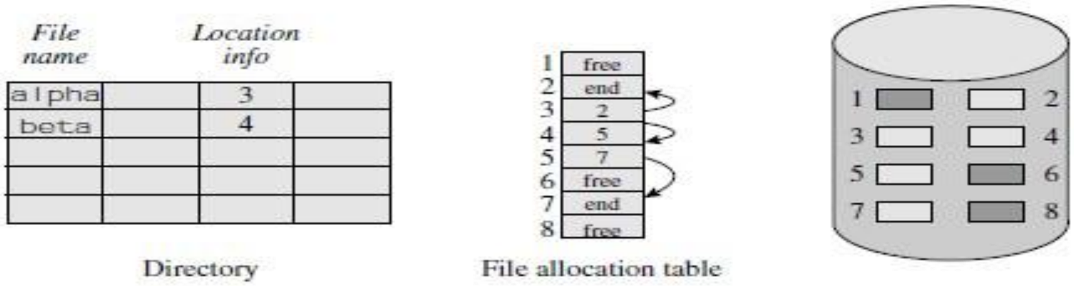


Fig 4.11 File Allocation Table

Indexed Allocation

In indexed allocation, an index called the *file map table* (FMT) is maintained to note the addresses of disk blocks allocated to a file. In its simplest form, an FMT can be an array containing disk block addresses. Each disk block contains a single field the data field. The field of a file's directory entry points *location info* to the FMT for the file

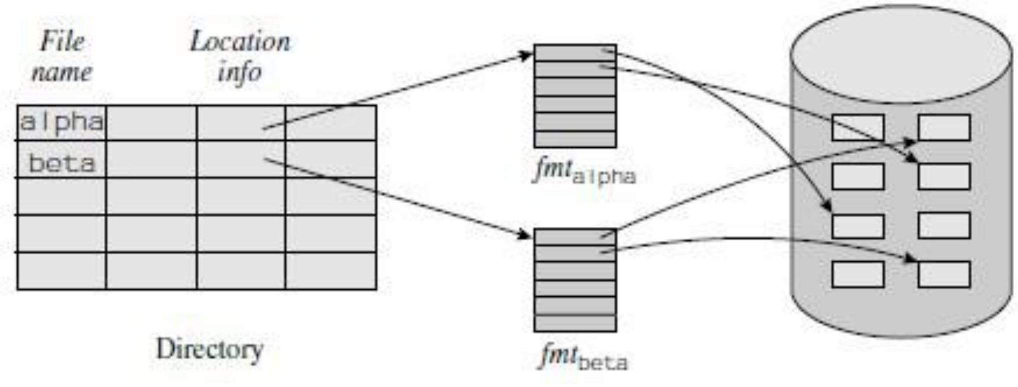


Figure 4.12 Indexed allocation of disk space.

4.8 Implementing file access

Implementing file access includes

(i) File system action at open

(ii) File system actions at file operation

(iii) File system actions at close

File system action at open

A call open (<pathname>…) is to set up the processing of the file. Open performs the following actions

(i) An FCB for the file <filename> is created in the AFT

(ii) Internal id of the file <filename> is passed back to the process for use in file processing

(iii) If the file <filename> is being created or updated, provision is made to update its directory entry when a close call is made by the process.

File system at file operation

After opening a file <filename>, a process initiated by user U performs some read or write operations on it. Each such operation is translated into a call

<opn> (internal id, record id, <IO_area addr>);

The file system performs the following actions to process this call

1. Locate the FCB of <filename> in the AFT using internal id.
2. Search the access control list of <filename> for the pair (U, ….). Give an error if <opn> does not exist in the list of access privileges for U.
3. Make a call on iocs-read or iocs-write with the parameters internal id, record id and <IO_area addr>.

File system actions at close

The file system performs the following actions when a process executes the statement close statement

1. If the file has been newly created or updated
   - If a file is newly created use directory FCB pointer to locate the FCB of the directory in which the file is to exist.
   - If the file has been updated and its size has changed, the directory entry of the file is updated using directory FCB pointer
2. The FCB of the file and FCB's of its parent and ancestor directories are erased from the AFT.

4.9 QUESTIONS

1. Describe the different operations performed on files.

2. Explain the file system and IOCS in detail.

3. Discuss the methods of allocation of disk space with block representation.

4. Explain briefly the file control block.

5. Explain the index sequential file organization with an example.

6. What is a link? With an example, illustrate the use of a link in an acyclic graph structure directory.

7. Compare sequential and direct file organization.

8. Describe the interface between file system and IOCS.

9. Explain the file system actions when a file is opened and a file is closed.

4.10 FURTHER READINGS

1. https://en.wikipedia.org/wiki/**File_system**
2. www.tldp.org/LDP/sag/html/filesystems
3. https://en.wikipedia.org/wiki/**File_system**

MODULE 5

MESSAGE PASSING

Structure

5.1 Objective

5.2 Overview of Message Passing

5.3 Implementing message passing

5.4 Mailboxes

5.5 Deadlocks

5.6 Deadlocks in resource allocation

5.7 Resource state modeling

5.8 Deadlock Prevention

OBJECTIVES

- Discuss the issues of message passing.
- Explain the direct naming and indirect naming.
- Explain the implementation in message passing.
- Discuss about mailboxes.
- Discuss about deadlocks in resource allocation , deadlocks detection and prevention.

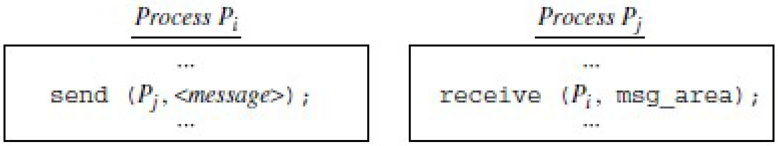
5.1 OVERVIEW OF MESSAGE PASSING

Message passing suits diverse situations where exchange of information between processes plays a key role. One of its prominent uses is in the client server paradigm, wherein a server process offers a service, and other processes, called its clients, send messages to it to use its service. This paradigm is used widely a microkernel- based OS structures functionalities such as scheduling in the form of servers, a conventional OS offer services such as printing through servers, and, on the Internet, a variety of services are offered by Web servers.

Another prominent use of message passing is in higher-level protocols for exchange of electronic mails and communication between tasks in parallel or distributed programs. Here, message passing is used to exchange information, while other parts of the protocol are employed to ensure reliability.

The key issues in message passing are how the processes that send and receive messages identify each other, and how the kernel performs various actions related to delivery of messages how it stores and delivers messages and whether it blocks a process that sends a message until its message is delivered. These features are operating system specific. We describe different message passing arrangements employed in operating systems and discuss their significance for user processes and for the kernel. We also describe message passing in UNIX and in Windows operating systems.

The four ways in which processes interact with one another data sharing, message passing, synchronization, and signals. Data sharing provides means to access values of shared data ina mutually exclusive manner. Process synchronization is performed by blocking a process until other processes have performed certain specific actions. Capabilities of message passing overlap those of data sharing and synchronization; however, each form of process interaction has its own niche application area..Figure 8.1 shows an example of message passing. Process Pi sends a message to process Pj by executing the statement send (Pj, <message>). The compiled code of the send statement invokes the library module send. Send makes a system call send, with Pj and the message as parameters. Execution of the statement receives (Pi, msg_area), where msg_area is an area in Pj space, results in a system call receive.



```
...
send (Pj, <message>);
...
```

```
...
receive (Pi, msg_area);
...
```

Figure 5.1 Message passing.

The semantics of message passing are as follows: At a send call by Pi , the kernel checks whether process Pj is blocked on a receive call for receiving a message from process Pi . If so, it copies the message into msg_area and activates Pj. If process Pj has not already made a receive call, the kernel arranges to deliver the message to it when Pj eventually makes a receive call. When process Pj receives the message, it interprets the message and takes an appropriate action. Messages may be passed between processes that exist in the same

computer or in different computers connected to a network. Also, the processes participating in message passing may decide on what a specific message means and what actions the receiver process should perform on receiving it. Because of this flexibility, message passing

is used in the following applications:

Message passing is employed in the client server paradigm, which is used to communicate between components of a microkernel-based operating system and user processes, to provide services such as the print service to processes within an OS, or to provide Web-based services to client processes located in other computers.

Message passing is used as the backbone of higher-level protocols employed for communicating between computers or for providing the electronic mail facility.

Message passing is used to implement communication between tasks in a parallel or distributed program.

In principle, message passing can be performed by using shared variables. For example, msg_area in Figure 5.1 could be a shared variable. Pi could deposit a value or a message in it and Pj could collect it from there. However, this approach is cumbersome because the processes would have to create a shared variable with the correct size and share its name.

They would also have to use synchronization analogous to the producer’s consumers problem to ensure that a receiver process accessed a message in a shared variable only after a sender process had deposited it there. Message passing is far simpler in this situation. It is also more general, because it can be used in a distributed system environment, where the shared the producers consumers problem with a single buffer, a single producer process, and a

single consumer process can be implemented by message passing as shown in Figure 5.2. The solution does not use any shared variables. Instead, process Pi, which is the producer process, has a variable called buffer and process Pj , which is the consumer process, has a variable called message area. The producer process produces in buffer and sends the contents of buffer in a message to the consumer.

```
begin
Parbegin
var buffer : . . . ;                    var message_area ;
repeat                                 repeat
{ Produce in buffer }              receive (Pi , message_area);
send (Pj , buffer);            { Consume from message_area }
{ Remainder of the cycle }       { Remainder of the cycle }
forever;                               forever;
Parend;
end.

Process Pi                              Process Pj
```

Figure 5.2 Producers consumers solution using message passing

The consumer receives the message in message area and consumes it from there. The send system call blocks the producer process until the message is delivered to the consumer, and the receive system call blocks the consumer until a message is sent to it.

Issues in Message Passing

Two important issues in message passing are:

Naming of processes: Whether names of sender and receiver processes are explicitly indicated in send and receive statements, or whether their identities are deduced by the kernel in some other manner.

Delivery of messages: Whether a sender process is blocked until the message sent by it is delivered, what the order is in which messages are delivered to the receiver process, and how exceptional conditions are handled.

These issues dictate implementation arrangements and also influence the generality of message passing. For example, if a sender process is required to know the identity of a receiver process, the scope of message passing would be limited to processes in the same application. Relaxing this requirement would extend message passing to processes indifferent applications and processes operating in different computer systems. Similarly, providing FCFS message delivery may be rather restrictive; processes may wish to receive
messages in some other order.

Provide good system performance, as in multiprogramming system; or provide favoured treatment to important functions, as in a real time system.

Direct and indirect naming

In direct naming example, the send and receive statements might have the following syntax:

Send (<destination_process>,<message_length>,<message_address>);

Receive (<source_ process>,<message_area>);

Direct naming can be used in two ways: In symmetric naming, both sender and receiver which process to receive a message from. However, it has to know the name of every process that wishes to send it a message, which is difficult when processes of different applications wish to communicate, or when a server wishes to receive a request from any one of a set of clients. In asymmetric naming, the receiver does not name the process from which it wishes to receive a message; the kernel gives it a message sent to it by some process.

In indirect naming processes do not mention the name in send and receive statements

Blocking and Non blocking Sends

A blocking send blocks a sender process until the message to be sent is delivered to the destination process. This method of message passing is called synchronous message passing.

A non blocking send call permits a sender to continue its operation after making a send call, irrespective of whether the message is delivered immediately; such message passing is called asynchronous message passing. In both cases, the receive primitive is typically blocking. Synchronous message passing provides some nice properties for user processes and simplifies actions of the kernel. A sender process has a guarantee that the message sent by it is delivered before it continues its operation. This feature simplifies the design of concurrent processes. The kernel delivers the message immediately if the destination process has already made a receive call for receiving a message; otherwise, it blocks the sender process until the destination process makes a receive call.

Asynchronous message passing enhances concurrency between the sender and receiver processes by letting the sender process continue its operation. However, it also causes a synchronization problem because the sender should not alter contents of the memory area which contains text of the message until the message is delivered. To overcome this problem, the kernel performs message buffering when a process makes a send call, the kernel allocates a buffer in the system area and copies the message into the buffer. This way, the sender process is free to access the memory area that contained text of the message.

Exceptional Conditions in Message Passing

To facilitate handling of exceptional conditions, the send and receive calls take two additional parameters. The first parameter is a set of flags indicating how the process wants exceptional conditions to be handled; we will call this parameter flags. The second parameter is the address of a memory area in which the kernel provides a condition code describing the outcome of the send or receives call; we will call this area status_area. When a process makes a send or receive call, the kernel deposits a condition code in

status_area. It then checks flags to decide whether it should handle any exceptional conditions and performs the necessary actions. It then returns control to the process. The process checks the condition code provided by the kernel and handles any exceptional conditions it wished to handle itself.

Some exceptional conditions and their handling actions are as follows:

1. The destination process mentioned in a send call does not exist.

2. In symmetric naming, the source process mentioned in a receive call does not exist.

3. A send call cannot be processed because the kernel has run out of buffer memory.

4. No message exists for a process when it makes a receive call.

5. A set of processes becomes deadlocked when a process is blocked on a receive call.

In cases 1 and 2, the kernel may abort the process that made the send or receive call and set its termination code to describe the exceptional condition. In case 3, the sender process may be blocked until some buffer space becomes available. Case 4 is really not an exception if receives are blocking (they generally are!), but it may be treated as an exception so that the receiving process has an opportunity to handle the condition if it so desires. A process may prefer the standard action, which is that the kernel should block the process until a message arrives for it, or it may prefer an action of its own choice, like waiting for specified amount of time before giving up.

More severe exceptions belong to the realm of OS policies. The deadlock situation of case 5is an example. Most operating systems do not handle this particular exception because it incurs the overhead of deadlock detection. Difficult-to handle situations, such as a process waiting a long time on a receive call, also belong to the realm of OS policies

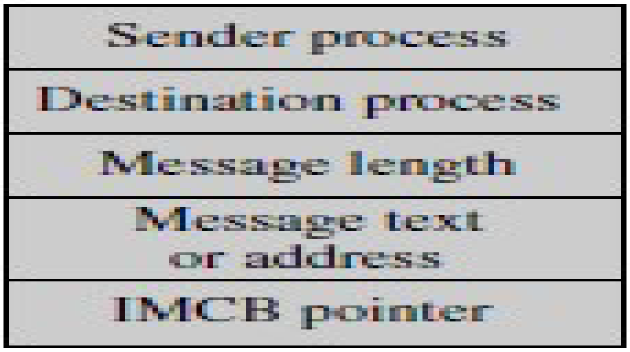
5.2 IMPLEMENTING MESSAGE PASSING

Buffering of Inter process Messages

When a process Pi sends a message to some process Pj by using a non blocking send, the kernel builds an inter process message control block (IMCB) to store all information needed To deliver the message (see Figure 8.3). The control block contains names of the sender and Destination processes, the length of the message, and the text of the message. The control block is allocated a buffer in the kernel area. When process Pj makes a receive call, the kernel copies the message from the appropriate IMCB into the message area provided by Pj .The pointer fields of IMCBs are used to form IMCB lists to simplify message delivery. Figure 9.4 shows the organization of IMCB lists when blocking sends and FCFS message

delivery are used. In symmetric naming, a separate list is used for every pair of communicating processes. When a process Pi performs a receive call to receive a message from process Pj , the IMCB list for the pair Pi Pj is used to deliver the message. In asymmetric naming, a single IMCB list can be maintained per recipient process. When a process performs a receive, the first IMCB in its list is processed to deliver a message.

If blocking sends are used, at most one message sent by a process can be undelivered at any point in time. The process is blocked until the message is delivered. Hence it is not necessary to copy the message into an IMCB.



Sender process

Destination process

Message length

Message text
or address

IMCB pointer

Figure 5.3 Inter process message control block (IMCB).

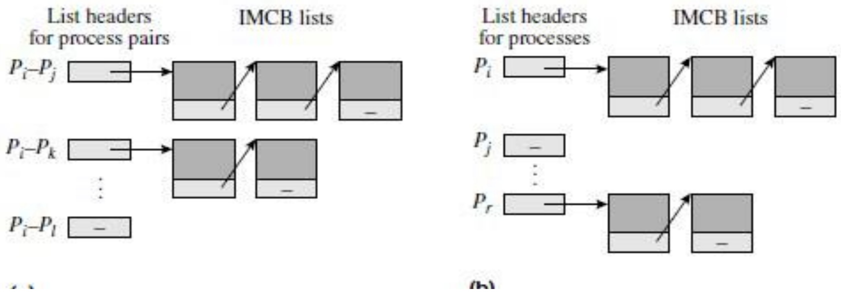


Figure 5.4 Lists of IMCBs for blocking sends in (a) symmetric naming; (b) asymmetric naming

The kernel notes the address of the message text in senders memory area, and use this information while delivering the message. This arrangement saves one copy operation on the message.

Delivery of Inter process Messages

When a process Pi sends a message to process Pj, the kernel delivers the message to Pj immediately if Pj is currently blocked on a receive call for a message from Pi , or from any process. After delivering the message, the kernel must also change the state of Pj to ready. Ifprocess Pj has not already performed a receive call, the kernel must arrange to deliver themessage when Pj performs a receive call later. Thus, message delivery actions occur at both send and receive calls. The kernel uses an event control block (ECB) to note actions that should be performed when an anticipated event occurs. The ECB contains three fields:

- Description of the anticipated event
- Id of the process that awaits the event
- An ECB pointer for forming ECB lists

Figure 5.5 shows use of ECBs to implement message passing with symmetric naming and blocking sends. When Pi makes a send call, the kernel checks whether an ECB exists for the send call by Pi , i.e., whether Pj had made a receive call and was waiting for Pi to send a message. If it is not the case, the kernel knows that the receive call would occur sometime in future, so it creates an ECB for the event Pi by Pj Pi as the process that will be affected by the event. Process Pi is put into the blocked state and the address of the ECB is put in the event info field of its PCB [see Figure 8.5(a)].

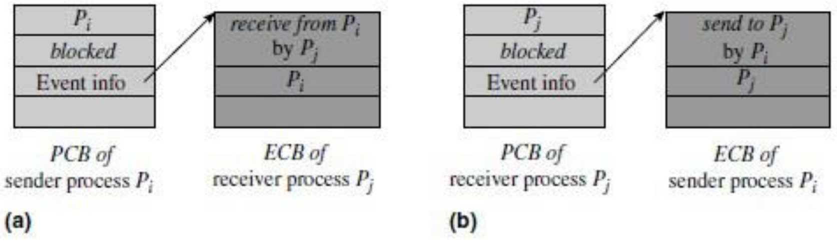

P_i
blocked
Event info
receive from P_i by P_j
P_i
PCB of sender process P_i
ECB of receiver process P_j
(a)
P_j
blocked
Event info
send to P_j by P_i
P_j
PCB of receiver process P_j
ECB of sender process P_i
(b)

Figure 5.5 ECBs to implement symmetric naming and blocking sends (a) at send; (b) at receive

Figure 5.5(b) illustrates the case when process Pj makes a receive call before Pi makes a send call. An ECB for a send to Pj by Pi event is now created. The id of Pj is put in the ECB to indicate that the state of Pj will be affected when the send event occurs

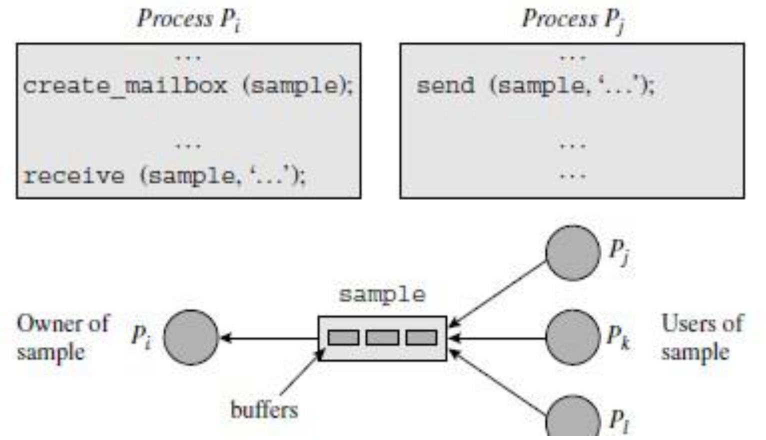
5.3 MAILBOXES

A mailbox is a repository for inter process messages. It has a unique name. The owner of a mailbox is typically the process that created it. Only the owner process can receive messages from a mailbox. Any process that knows the name of a mailbox can send messages to it. Thus, sender and receiver processes use the name of a mailbox, rather than each other's names, in send and receive statements; it is an instance of indirect naming.

Figure 8.6llustrates message passing using a mailbox named sample. Process Pi creates the mailbox, using the statement create_mailbox. Process Pj sends a message to the mailbox, using the mailbox name in its send statement. If Pi has not already executed a receive statement, the kernel would store the message in a buffer. The kernel may associate a fixed set of buffers with each mailbox, or it may allocate buffers from a common pool of buffers when a message is sent. Both create_mailbox and send statements return with condition codes.

The kernel may provide a fixed set of mailbox names, or it may permit user processes to assign mailbox names of their choice. In the former case, confidentiality of communication between a pair of processes cannot be guaranteed because any process can use a mailbox.

Confidentiality greatly improves when processes can assign mailbox names of their own choice. To exercise control over creation and destruction of mailboxes, the kernel may require a process for explicitly connect to a mailbox before using it and to disconnect when it finishes using it.



```
...
create_mailbox (sample);
...
receive (sample, '...');
```

```
...
send (sample, '...');
...
...
```

Figure 5.6 Creation and use of mailbox sample.

Use of a mailbox has following advantages:

Anonymity of receiver: A process sending a message to request a service may have no interest in the identity of the receiver process, as long as the receiver process can perform the needed function. A mailbox relieves the sender process of the need to know the identity of the receiver. Additionally, if the OS permits the ownership of a mailbox to be changed dynamically, one process can readily take over the service of another.

Classification of messages: A process may create several mailboxes, and use each mailbox to receive messages of a specific kind. This arrangement permits easy classification of messages.

DEADLOCKS

A deadlock is a situation concerning a set of processes in which each process in the set waits for an event that must be caused by another process in the set.

5.4 DEADLOCKS IN RESOURCE ALLOCATION

Resource allocation in a system entails three kinds of events- request for resource, actual allocation of the resource and release of the resource.

A process may utilize the resources in only the following sequences:

- Request:-If the request is not granted immediately then the requesting process must wait it can acquire the resources.
- Use:-The process can operate on the resource.
- Release:-The process releases the resource after using it.

5.4.1 CONDITIONS FOR RESOURCE ALLOCATION

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

- Mutual Exclusion:Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released

- Hold and Wait:-A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
- No Preemption:-Resources can't be preempted i.e., only the process holding the resources must release it after the process has completed its task.
- Circular Wait:-A set {P0,P1……..Pn} of waiting process must exist such that P0 is waiting for a resource i.e., held by P1, P1 is waiting for a resource i.e., held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1. All the four conditions must hold for a deadlock to occur.

5.5 MODELLING RESOURCE ALLOCATION STATE

Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e). The set of vertices (v) can be described into two different types of nodes P={P1,P2……..Pn} i.e., set consisting of all active processes and R={R1,R2……….Rn}i.e., set consisting of all resource types in the system. A directed edge from process Pi to resource type Rj denoted by Pi->Ri indicates that Pi requested an instance of resource Rj and is waiting.This edge is called Request edge. A directed edge Ri-> Pj signifies that resource Rj is held by process Pi. This is called Assignment edge

If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist. If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.

R1 R3

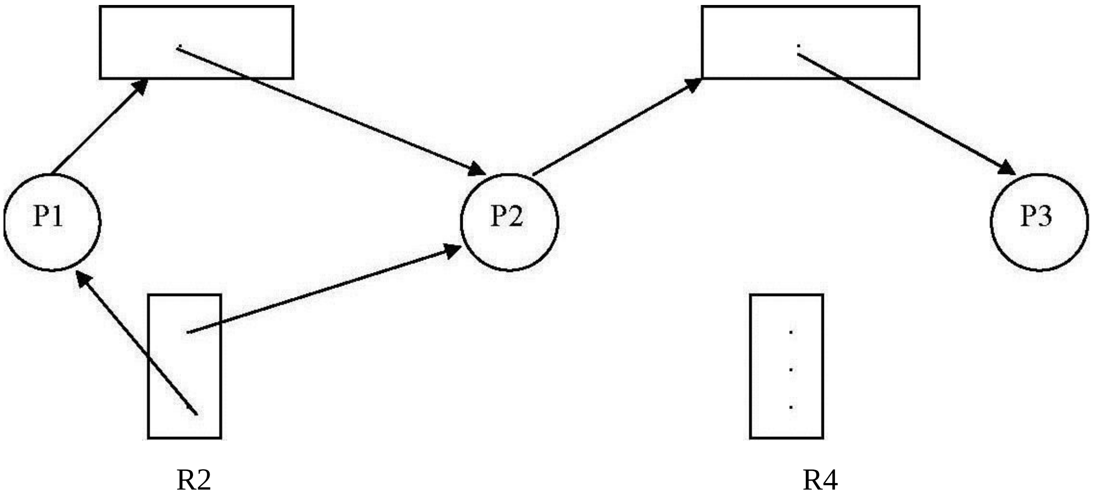


Figure 5.6 Resource State Modelling

5.6 DEADLOCK PREVENTION

For a deadlock to occur each of the four necessary conditions must hold. If at least one of the there condition does not hold then we can prevent occurrence of deadlock.

- Non – Shareable Rescource : This holds for non-sharable resources. Eg:-A printer can be used by only one process at a time. Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources
- Hold and Wait:This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available.x One protocol can be used is that each process is allocated with all of its resources before its start execution. Eg:-consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because

it requires printer at the last and is allocated with it from the beginning so that no other process can use it. x Another protocol that can be used is to allow a process to request a resource when the process has none. i.e., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

- No Preemption:To ensure that this condition never occurs the resources must be preempted. The following protocol can be used. x If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting. x When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait.
- Circular Wait:-The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

  Let R={R1,R2,………Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering. Eg:-we can define a one to one function

  F:R□N as follows :-F(disk drive)=5 F(printer)=12 F(tape drive)=1

Deadlock can be prevented by using the following protocol:x Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri). x Alternatively when a process requests an instance of resource type Rj, it has released any resource Ri such that F(Ri) >= F(Rj). If these two protocol are used then the circular wait can’t hold.