Definition of operating system?

The low-level software that supports a computer’s basic functions, such as scheduling tasks and controlling peripherals.

The operating system is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs and applications. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.
The fundamental goals of an operating system are:

- Efficient use
- User convenience
- Non-interference

**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.

**User Convenience**

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.

**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.

**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.

### Common Tasks Performed by Operating Systems are

<table>
<thead>
<tr>
<th>Task</th>
<th>When performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a list of resources</td>
<td>during booting</td>
</tr>
<tr>
<td>Maintain information for security</td>
<td>while registering new users</td>
</tr>
<tr>
<td>Verify identity of a user</td>
<td>at login time</td>
</tr>
<tr>
<td>Initiate execution of programs</td>
<td>at user commands</td>
</tr>
<tr>
<td>Maintain authorization information</td>
<td>when a user specifies which</td>
</tr>
<tr>
<td></td>
<td>Collaborators can access what programs or data.</td>
</tr>
<tr>
<td>Perform resource allocation</td>
<td>when requested by users or programs</td>
</tr>
<tr>
<td>Maintain current status of resources</td>
<td>during resource allocation/deallocation</td>
</tr>
<tr>
<td>Maintain current status of programs</td>
<td>continually during OS operation</td>
</tr>
<tr>
<td>and perform scheduling</td>
<td></td>
</tr>
</tbody>
</table>

### Resource allocation and virtual resources

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. It is constructed by the boot procedure by sensing the presence.

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.
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 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.

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

The five classes of operating system are:

1) 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.

2) 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.
3) **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.

4) **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.

5) **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.

**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. Figure 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 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 below 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.

FIGURE: CONTROL STATEMENTS

// 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

MULTIPROGRAMMING SYSTEMS

Multiprogramming operating systems were developed to provide efficient resource utilization in a noninteractive environment.
A computer must possess the features summarized in Table above 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 a 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.

**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 \( t_f \) is \( n/(t_f - t_0) \). 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 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.

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.

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 below 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.
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 below): 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 \( \delta \) 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 \( \delta \) CPU seconds, and let \( \sigma \) 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 \( (\eta) \) are given by

\[
rt = n \times (\delta + \sigma) \quad (1.1)
\]

\[
\eta = \frac{\delta}{\delta + \sigma} \quad (1.2)
\]

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 \( \delta \) CPU seconds to produce a response. Hence the relationship of \( rt \) and \( \eta \) with \( \delta \) is more complex than shown in Eqs (1.1) and (1.2).

**Swapping of Programs**

Throughput of subrequests is the appropriate measure of performance of a timesharing operating system. The time-sharing OS of Example 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.

<table>
<thead>
<tr>
<th>Time</th>
<th>Scheduling list</th>
<th>Scheduled program</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( P_1, P_2 )</td>
<td>( P_1 )</td>
<td>( P_1 ) is preempted at 10 ms</td>
</tr>
<tr>
<td>10</td>
<td>( P_2, P_1 )</td>
<td>( P_2 )</td>
<td>( P_2 ) is preempted at 20 ms</td>
</tr>
<tr>
<td>20</td>
<td>( P_1, P_2 )</td>
<td>( P_1 )</td>
<td>( P_1 ) starts I/O at 25 ms</td>
</tr>
<tr>
<td>25</td>
<td>( P_2 )</td>
<td>( P_2 )</td>
<td>( P_2 ) is preempted at 35 ms</td>
</tr>
<tr>
<td>35</td>
<td>( P_2 )</td>
<td>( P_2 )</td>
<td>( P_2 ) starts I/O at 45 ms</td>
</tr>
<tr>
<td>45</td>
<td>–</td>
<td>–</td>
<td>CPU is idle</td>
</tr>
</tbody>
</table>

Operation of processes \( P_1 \) and \( P_2 \) in a time-sharing system.
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 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 \textit{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.

The kernel performs a \textit{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 \textit{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 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.

\section*{REAL-TIME OPERATING SYSTEMS}

In a class of applications called \textit{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 \textit{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.
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 in 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.

Features of a Real-Time Operating System

<table>
<thead>
<tr>
<th>Feature</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency within an application</td>
<td>A programmer can indicate that some parts of an application should be executed concurrently with one another. The OS considers execution of each such part as a process.</td>
</tr>
<tr>
<td>Process priorities</td>
<td>A programmer can assign priorities to processes.</td>
</tr>
<tr>
<td>Scheduling</td>
<td>The OS uses priority-based or deadline-aware scheduling.</td>
</tr>
<tr>
<td>Domain-specific events, interrupts</td>
<td>A programmer can define special situations within the external system as events, associate interrupts with them, and specify event handling actions for them.</td>
</tr>
<tr>
<td>Predictability</td>
<td>Policies and overhead of the OS should be predictable.</td>
</tr>
<tr>
<td>Reliability</td>
<td>The OS ensures that an application can continue to function even when faults occur in the computer.</td>
</tr>
</tbody>
</table>

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.

**Benefits of Distributed Operating Systems**

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

**Key Concepts and Techniques Used in a Distributed OS**

<table>
<thead>
<tr>
<th>Concept/Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed control</td>
<td>A control function is performed through participation of several nodes, possibly all nodes, in a distributed system.</td>
</tr>
<tr>
<td>Transparency</td>
<td>A resource or service can be accessed without having to know its location in the distributed system.</td>
</tr>
<tr>
<td>Remote procedure call (RPC)</td>
<td>A process calls a procedure that is located in a different computer system. The RPC is analogous to a procedure or function call in a programming language, except that the OS passes parameters to the remote procedure over the network and returns its results over the network.</td>
</tr>
</tbody>
</table>

**Questions**

1. Explain the goals of an operating system, its operations and resource allocation of OS.
2. List the common tasks performed by an OS. Explain briefly.
3. Explain the features and special techniques of distributed OS.
4. Discuss the spooling technique with a block representation.
5. Explain briefly the key features of different classes of OS.
6. Explain the concepts of memory compaction and virtual memory with respect to memory management.
7. Define an OS. What are the different facets of user convenience?
8. Explain partition and pool based resource allocation strategies.
9. Explain time sharing OS with respect to i) scheduling and ii) memory management.
10. Describe the batch processing system and functions of scheduling and memory management for the same.
When a computer is switched on, the *boot procedure* analyzes its configuration, CPU type, memory size, I/O devices, and details of other hardware connected to the computer. It then loads a part of the OS in memory, initializes its data structures with this information, and hands over control of the computer system to it.

Figure below is a schematic diagram of OS operation. An event like I/O completion or end of a time slice causes an interrupt. When a process makes a *system call*, e.g., to request resources or start an I/O operation, it too leads to an interrupt called a *software interrupt*.

### Functions of an OS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process management</td>
<td>Initiation and termination of processes, scheduling</td>
</tr>
<tr>
<td>Memory management</td>
<td>Allocation and deallocation of memory, swapping, virtual memory management</td>
</tr>
<tr>
<td>I/O management</td>
<td>I/O interrupt servicing, initiation of I/O operations, optimization of I/O device performance</td>
</tr>
<tr>
<td>File management</td>
<td>Creation, storage and access of files</td>
</tr>
<tr>
<td>Security and protection</td>
<td>Preventing interference with processes and resources</td>
</tr>
<tr>
<td>Network management</td>
<td>Sending and receiving of data over the network</td>
</tr>
</tbody>
</table>
The interrupt action switches the CPU to an interrupt servicing routine. The interrupt servicing routine performs a context save action to save information about the interrupted program and activates an event handler, which takes appropriate actions to handle the event. The scheduler then selects a process and switches the CPU to it. CPU switching occurs twice during the processing of an event—first to the kernel to perform event handling and then to the process selected by the scheduler.

**STRUCTURE OF AN OPERATING SYSTEM**

**Policies and Mechanisms**

In determining how an operating system is to perform one of its functions, the OS designer needs to think at two distinct levels:

- **Policy**: A policy is the guiding principle under which the operating system will perform the function.

- **Mechanism**: A mechanism is a specific action needed to implement a policy.

A policy decides what should be done, while a mechanism determines how something should be done and actually does it. A policy is implemented as a decision-making module that decides which mechanism modules to call under what conditions. A mechanism is implemented as a module that performs a specific action. The following example identifies policies and mechanisms in round-robin scheduling.

**Example Policies and Mechanisms in Round-Robin Scheduling**

In scheduling, we would consider the round-robin technique to be a policy. The following mechanisms would be needed to implement the round-robin scheduling policy:

- Maintain a queue of ready processes
- Switch the CPU to execution of the selected process (this action is called dispatching).

The priority-based scheduling policy, which is used in multiprogramming systems, would also require a mechanism for maintaining information about ready processes; however, it would be different from the mechanism used in round-robin scheduling because it would organize information according to process priority. The dispatching mechanism, however, would be common to all scheduling policies.

Apart from mechanisms for implementing specific process or resource management policies, the OS also has mechanisms for performing housekeeping actions. The context save action mentioned in is implemented as a mechanism.

**Portability and Extensibility of Operating Systems**

The design and implementation of operating systems involves huge financial investments. To protect these investments, an operating system design should have a lifetime of more than a
decade. Since several changes will take place in computer architecture, I/O device technology, and application environments during this time, it should be possible to adapt an OS to these changes. Two features are important in this context—portability and extensibility.

**Porting** is the act of adapting software for use in a new computer system. **Portability** refers to the ease with which a software program can be ported—it is inversely proportional to the porting effort. **Extensibility** refers to the ease with which new functionalities can be added to a software system.

Porting of an OS implies changing parts of its code that are architecture dependent so that the OS can work with new hardware. Some examples of architecture-dependent data and instructions in an OS are:

- An interrupt vector contains information that should be loaded in various fields of the PSW to switch the CPU to an interrupt servicing routine. This information is architecture-specific.

- Information concerning memory protection and information to be provided to the memory management unit (MMU) is architecture-specific.

- I/O instructions used to perform an I/O operation are architecture-specific.

The architecture-dependent part of an operating system’s code is typically associated with mechanisms rather than with policies. An OS would have high portability if its architecture-dependent code is small in size, and its complete code is structured such that the porting effort is determined by the size of the architecture dependent code, rather than by the size of its complete code. Hence the issue of OS portability is addressed by separating the architecture-dependent and architecture-independent parts of an OS and providing well-defined interfaces between the two parts.

Extensibility of an OS is needed for two purposes: for incorporating new hardware in a computer system—typically new I/O devices or network adapters—and for providing new functionalities in response to new user expectations. Early operating systems did not provide either kind of extensibility. Hence even addition of a new I/O device required modifications to the OS. Later operating systems solved this problem by adding a functionality to the boot procedure. It would check for hardware that was not present when the OS was last booted, and either prompt the user to select appropriate software to handle the new hardware, typically a set of routines called a **device driver** that handled the new device, or itself select such software. The new software was then loaded and integrated with the kernel so that it would be invoked and used appropriately.
OPERATING SYSTEMS WITH MONOLITHIC STRUCTURE

An OS is a complex software that has a large number of functionalities and may contain millions of instructions. It is designed to consist of a set of software modules, where each module has a well-defined interface that must be used to access any of its functions or data. Such a design has the property that a module cannot — see inner details of functioning of other modules. This property simplifies design, coding, and testing of an OS.

Early operating systems had a monolithic structure, whereby the OS formed a single software layer between the user and the bare machine, i.e., the computer system’s hardware (figure below). The user interface was provided by a command interpreter. The command interpreter organized creation of user processes. Both the command interpreter and user processes invoked OS functionalities and services through system calls.

Two kinds of problems with the monolithic structure were realized over a period of time. The sole OS layer had an interface with the bare machine. Hence architecture-dependent code was spread throughout the OS, and so there was poor portability. It also made testing and debugging difficult, leading to high costs of maintenance and enhancement. These problems led to the search for alternative ways to structure an OS.

- **Layered structure:** The layered structure attacks the complexity and cost of developing and maintaining an OS by structuring it into a number of layers.
  The multiprogramming system of the 1960s is a well known example of a layered OS.

- **Kernel-based structure:** The kernel-based structure confines architecture dependence to a small section of the OS code that constitutes the kernel, so that portability is increased. The Unix OS has a kernel-based structure.

- **Microkernel-based OS structure:** The microkernel provides a minimal set of facilities and services for implementing an OS. Its use provides portability. It also provides extensibility because changes can be made to the OS without requiring changes in the microkernel.

![Figure: Monolithic OS](image)
LAYERED DESIGN OF OPERATING SYSTEMS

The monolithic OS structure suffered from the problem that all OS components had to be able to work with the bare machine. This feature increased the cost and effort in developing an OS because of the large semantic gap between the operating system and the bare machine.

The semantic gap can be illustrated as follows: A machine instruction implements a machine-level primitive operation like arithmetic or logical manipulation of operands. An OS module may contain an algorithm, say, that uses OS-level primitive operations like saving the context of a process and initiating an I/O operation. These operations are more complex than the machine-level primitive operations. This difference leads to a large semantic gap, which has to be bridged through programming. Each operation desired by the OS now becomes a sequence of instructions, possibly a routine (see Figure below). It leads to high programming costs.

The semantic gap between an OS and the machine on which it operates can be reduced by either using a more capable machine—a machine that provides instructions to perform some (or all) operations that operating systems have to perform—or by simulating a more capable machine in the software. The former approach is expensive.

![Figure: Semantic gap](image)

In the latter approach, however, the simulator, which is a program, executes on the bare machine and mimics a more powerful machine that has many features desired by the OS. This new—machine is called an extended machine, and its simulator is called the extended machine software.

Figure below illustrates a two-layered OS. The extended machine provides operations like context save, dispatching, swapping, and I/O initiation. The operating system layer is located on top of the extended machine layer. This arrangement considerably simplifies the coding and testing of OS modules by separating the algorithm of a function from the implementation of its primitive operations. It is now easier to test, debug, and modify an OS module than in a monolithic OS. We say that the lower layer provides an abstraction that is the extended machine. We call the operating system layer the top layer of the OS.

The layered structures of operating systems have been evolved in various ways—using different abstractions and a different number of layers.
The layered approach to OS design suffers from three problems. The operation of a system may be slowed down by the layered structure. Each layer can interact only with adjoining layers. It implies that a request for OS service made by a user process must move down from the highest numbered layer to the lowest numbered layer before the required action is performed by the bare machine. This feature leads to high overhead.

The second problem concerns difficulties in developing a layered design. Since a layer can access only the immediately lower layer, all features and facilities needed by it must be available in lower layers. This requirement poses a problem in the ordering of layers that require each other’s services. This problem is often solved by splitting a layer into two and putting other layers between the two halves.

VIRTUAL MACHINE OPERATING SYSTEMS

Different classes of users need different kinds of user service. Hence running a single OS on a computer system can disappoint many users. Operating the computer under different OSs during different periods is not a satisfactory solution because it would make accessible services offered under only one of the operating systems at any time. This problem is solved by using a virtual machine operating system.

(VM OS) to control the computer system. The VM OS creates several virtual machines. Each virtual machine is allocated to one user, who can use any OS of his own choice on the virtual machine and run his programs under this OS. This way user of the computer system can use different operating systems at the same time.

Each of these operating systems a guest OS and call the virtual machine OS the host OS. The computer used by the VM OS is called the host machine. A virtual machine is a virtual resource. Let us consider a virtual machine that has the same architecture as the host machine; i.e., it has a virtual CPU capable of executing the same instructions, and similar memory and I/O devices. It may, however, differ from the host machine in terms of some elements of its configuration like memory size and I/O devices. Because of the identical architectures of the virtual and host
machines, no semantic gap exists between them, so operation of a virtual machine does not introduce any performance, software intervention is also not needed to run a guest OS on a virtual machine.

The VM OS achieves concurrent operation of guest operating systems through an action that resembles process scheduling—it selects a virtual machine and arranges to let the guest OS running on it execute its instructions on the CPU. The guest OS in operation enjoys complete control over the host machine’s environment, including interrupt servicing. The absence of a software layer between the host machine and guest OS ensures efficient use of the host machine.

A guest OS remains in control of the host machine until the VM OS decides to switch to another virtual machine, which typically happens in response to an interrupt. The VM OS can employ the timer to implement time-slicing and round-robin scheduling of guest OSs.

A somewhat complex arrangement is needed to handle interrupts that arise when a guest OS is in operation. Some of the interrupts would arise in its own domain, e.g., an I/O interrupt from a device included in its own virtual machine, while others would arise in the domains of other guest OSs. The VM OS can arrange to get control when an interrupt occurs, find the guest OS whose domain the interrupt belongs to, and —schedule it — that guest OS to handle it. However, this arrangement incurs high overhead because of two context switch operations—the first context switch passes control to the VM OS, and the second passes control to the correct guest OS. Hence the VM OS may use an arrangement in which the guest OS in operation would be invoked directly by interrupts arising in its own domain. It is implemented as follows: While passing control to a guest operating system, the VMOS replaces its own interrupt vectors by those defined in the guest OS. This action ensures that an interrupt would switch the CPU to an interrupt servicing routine of the guest OS. If the guest OS finds that the interrupt did not occur in its own domain, it passes control to the VM OS by making a special system call —invoke VM OS. The VM OS now arranges to pass the interrupt to the appropriate guest OS. When a large number of virtual machines exists, interrupt processing can cause excessive shuffling between virtual machines, hence the VMOS may not immediately activate the guest OS in whose domain an interrupt occurred—it may simply note occurrence of interrupts that occurred in the domain of a guest OS and provide this information to the guest OS the next time it is —scheduled.

Virtual machines are employed for diverse purposes:

- To use an existing server for a new application that requires use of a different operating system. This is called workload consolidation; it reduces the hardware and operational cost of computing by reducing the number of servers needed in an organization.

- To provide security and reliability for applications that use the same host and the same OS. This benefit arises from the fact that virtual machines of different applications cannot access each other’s resources.
• To test a modified OS (or a new version of application code) on a server concurrently with production runs of that OS.

• To provide disaster management capabilities by transferring a virtual machine from a server that has to shut down because of an emergency to another server available on the network.

A VM OS is large, complex and expensive. To make the benefits of virtual machines available widely at a lower cost, virtual machines are also used without a VM OS. Two such arrangements are described in the following.

**Virtual Machine Monitors (VMMs)** A VMM, also called a hypervisor, is a software layer that operates on top of a host OS. It virtualizes the resources of the host computer and supports concurrent operation of many virtual machines. When a guest OS is run in each virtual machine provided by a VMM, the host OS and the VMM together provide a capability that is equivalent of a VM OS.

VMware and XEN are two VMMs that aim at implementing hundreds of guest OSs on a host computer while ensuring that a guest OS suffers only a marginal performance degradation when compared to its implementation on a bare machine.

**Programming Language Virtual Machines** Programming languages have used virtual machines to obtain some of the benefits discussed earlier. In the 1970s, the Pascal programming language employed a virtual machine to provide portability. The virtual machine had instructions called P-code instructions that were wellsuited to execution of Pascal programs. It was implemented in the software in the form of an interpreter for P-code instructions. A compiler converted a Pascal program into a sequence of P-code instructions, and these could be executed on any computer that had a P-code interpreter. The virtual machine had a small number of instructions, so the interpreter was compact and easily portable. This feature facilitated widespread use of Pascal in the 1970s.

**KERNEL-BASED OPERATING SYSTEMS**

Figure below is an abstract view of a kernel-based OS. The kernel is the core of the OS; it provides a set of functions and services to support various OS functionalities. The rest of the OS is organized as a set of nonkernel routines, which implement operations on processes and resources that are of interest to users, and a user interface.
A system call may be made by the user interface to implement a user command, by a process to invoke a service in the kernel, or by a nonkernel routine to invoke a function of the kernel. For example, when a user issues a command to execute the program stored in some file, say file alpha, the user interface makes a system call, and the interrupt servicing routine invokes a nonkernel routine to set up execution of the program. The nonkernel routine would make system calls to allocate memory for the program’s execution, open file alpha, and load its contents into the allocated memory area, followed by another system call to initiate operation of the process that represents execution of the program. If a process wishes to create a child process to execute the program in file alpha, it, too, would make a system call and identical actions would follow.

The historical motivations for the kernel-based OS structure were portability of the OS and convenience in the design and coding of nonkernel routines.

Portability of the OS is achieved by putting architecture-dependent parts of OS code—which typically consist of mechanisms—in the kernel and keeping architecture-independent parts of code outside it, so that the porting effort is limited only to porting of the kernel. The kernel is typically monolithic to ensure efficiency; the nonkernel part of an OS may be monolithic, or it may be further structured into layers.

Kernel-based operating systems have poor extensibility because addition of a new functionality to the OS may require changes in the functions and services offered by the kernel.

**Evolution of Kernel-Based Structure of Operating Systems**

The structure of kernel-based operating systems evolved to offset some of its drawbacks. Two steps in this evolution were dynamically loadable kernel modules and user-level device drivers.

To provide *dynamically loadable kernel modules*, the kernel is designed as a set of modules that interact among themselves through well-specified interfaces. A *base kernel* consisting of a core set of modules is loaded when the system is booted. Other modules, which conform to interfaces of the base kernel, are loaded when their functionalities are needed, and are removed from memory when they are no longer needed. Use of loadable modules conserves memory during OS operation because only required modules of the kernel are in memory at any time. It also provides extensibility, as kernel modules can be modified.
separately and new modules can be added to the kernel easily. Use of loadable kernel modules has a few drawbacks too. Loading and removal of modules fragments memory, so the kernel has to perform memory management actions to reduce its memory requirement.

**MICROKERNEL BASED OPERATING SYSTEMS**

Putting all architecture-dependent code of the OS into the kernel provides good portability. However, in practice, kernels also include some architecture independent code. This feature leads to several problems. It leads to a large kernel size, which detracts from the goal of portability. It may also necessitate kernel modification to incorporate new features, which causes low extensibility.

A large kernel supports a large number of system calls. Some of these calls may be used rarely, and so their implementations across different versions of the kernel may not be tested thoroughly. This compromises reliability of the OS.

The microkernel was developed in the early 1990s to overcome the problems concerning portability, extensibility, and reliability of kernels. A microkernel is an essential core of OS code, thus it contains only a subset of the mechanisms typically included in a kernel and supports only a small number of system calls, which are heavily tested and used.

![Structure of microkernel-based operating systems.](image)

This feature enhances portability and reliability of the microkernel. Less essential parts of OS code are outside the microkernel and use its services, hence these parts could be modified without affecting the kernel; in principle, these modifications could be made without having to reboot the OS! The services provided in a microkernel are not biased toward any specific features or policies in an OS, so new functionalities and features could be added to the OS to suit specific operating environments.

Figure above illustrates the structure of a microkernel-based OS. The microkernel includes mechanisms for process scheduling and memory management, etc., but does not include a scheduler or memory handler. These functions are implemented as servers, which are simply processes that never terminate. The servers and user processes operate on top of the microkernel, which merely performs interrupt handling and provides communication between
the servers and user processes.

The small size and extensibility of microkernels are valuable properties for the embedded systems environment, because operating systems need to be both small and fine-tuned to the requirements of an embedded application. Extensibility of microkernels also conjures the vision of using the same microkernel for a wide spectrum of computer systems, from palm-held systems to large parallel and distributed systems. This vision has been realized to some extent.

Questions
1. What are the problems associated with monolithic structure OS?
2. Explain i) layered structure of OS and ii) kernel based OS.
3. Describe the functions of an OS.
4. Discuss OS with monolithic structure and multi-programming system.
5. Explain with block diagram the structures of micro kernel based, time sharing and real time OS classes.
6. Define process, process states and state transitions.
7. Explain VMOS. What are the advantages of virtual machines?
8. Explain system generation operations.
10. How is layered OS structure superior to monolithic structure?
11. In a batch processing system, the results of 1000 students are to be printed. Reading a card or printing a result takes 100 ms whereas read/write operation in a disk needs only 20 ms. Processing a record needs only 10 ms of CPU time. Compute the program elapsed time and CPU idle time with and without spooling.
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.

What Is a Process?
To understand what a process is, let us discuss how the OS executes a program. Program P shown in Figure below 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.

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

**Figure: A program and an abstract view of its execution.**
During execution, instructions of this program use values in its data area and the stack to perform the intended calculations. Figure 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.

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.

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 above. 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.

**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.

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.

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.

**Programmer view of processes**

**Child Processes in a Real-Time Application**

Example 3.1

The real-time data logging application of Section 3.7 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 below. Note that a process is depicted by a circle and a parent–child relationship is depicted by an arrow. As shown in below, *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
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 3.1, 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.

### Concurrency and Parallelism

*Parallelism* is the quality of occurring at the same time. Two events are parallel if they occur at the same time, and two tasks are parallel if they are performed at the same time. *Concurrency* is an illusion of parallelism. Thus, two tasks are concurrent if there is an illusion that they are being performed in parallel, whereas, in reality, only one of them may be performed at any time.

In an OS, concurrency is obtained by interleaving operation of processes on the CPU, which creates the illusion that these processes are operating at the same time. Parallelism is obtained by using multiple CPUs, as in a multiprocessor system, and operating different processes on
these CPUs.

Computation speedup of an application through concurrency and parallelism would depend on several factors:

- **Inherent parallelism within the application**: Does the application have activities that can progress independently of one another.

- **Overhead of concurrency and parallelism**: The overhead of setting up and managing concurrency should not predominate over the benefits of performing activities concurrently, e.g., if the chunks of work sought to be performed concurrently are too small, the overhead of concurrency may swamp its contributions to computation speedup.

- **Model of concurrency and parallelism supported by the OS**: How much overhead does the model incur, and how much of the inherent parallelism within an application can be exploited through it.

**IMPLEMENTING PROCESSES**

In the operating system’s view, a process is a unit of computational work. Hence the kernel’s primary task is to control operation of processes to provide effective utilization of the computer system. Accordingly, the kernel allocates resources to a process, protects the process and its resources from interference by other processes, and ensures that the process gets to use the CPU until it completes its operation.

![Figure: Fundamental functions of the kernel for controlling processes](image)

The kernel is activated when an *event*, which is a situation that requires the kernel’s attention, leads to either a hardware interrupt or a system call. The kernel now performs four fundamental functions to control operation of processes (see Figure above):
1. **Context save:** Saving CPU state and information concerning resources of the process whose operation is interrupted.

2. **Event handling:** Analyzing the condition that led to an interrupt, or the request by a process that led to a system call, and taking appropriate actions.

3. **Scheduling:** Selecting the process to be executed next on the CPU.

4. **Dispatching:** Setting up access to resources of the scheduled process and loading its saved CPU state in the CPU to begin or resume its operation.

The kernel performs the context save function to save information concerning the interrupted process. It is followed by execution of an appropriate event handling routine, which may inhibit further operation of the interrupted process, e.g., if this process has made a system call to start an I/O operation, or may enable operation of some other process, e.g., if the interrupt was caused by completion of its I/O operation. The kernel now performs the scheduling function to select a process and the dispatching function to begin or resume its operation.

**Process States and State Transitions**

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 3.3. The kernel considers a process to be in the **blocked** 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 $P_i$ 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 $P_i$ 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, $P_i$’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.

Figure below 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. Eventually it enters the *terminated* state.
<table>
<thead>
<tr>
<th>State transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready → running</td>
<td>The process is dispatched. The CPU begins or resumes execution of its instructions.</td>
</tr>
<tr>
<td>blocked → ready</td>
<td>A request made by the process is granted or an event for which it was waiting occurs.</td>
</tr>
<tr>
<td>running → ready</td>
<td>The process is preempted because the kernel decides to schedule some other process. This transition occurs either because a higher-priority process becomes ready, or because the time slice of the process elapses.</td>
</tr>
<tr>
<td>running → blocked</td>
<td>The process in operation makes a system call to indicate that it wishes to wait until some resource request made by it is granted, or until a specific event occurs in the system. Five major causes of blocking are:</td>
</tr>
<tr>
<td></td>
<td>• Process requests an I/O operation</td>
</tr>
<tr>
<td></td>
<td>• Process requests a resource</td>
</tr>
<tr>
<td></td>
<td>• Process wishes to wait for a specified interval of time</td>
</tr>
<tr>
<td></td>
<td>• Process waits for a message from another process</td>
</tr>
<tr>
<td></td>
<td>• Process waits for some action by another process</td>
</tr>
<tr>
<td>running → terminated</td>
<td>Execution of the program is completed. Five primary reasons for process termination are:</td>
</tr>
<tr>
<td></td>
<td>• Self-termination: The process in operation either completes its task or realizes that it cannot operate meaningfully and makes a “terminate me” system call. Examples of the latter condition are incorrect or inconsistent data, or inability to access data in a desired manner, e.g., incorrect file access privileges.</td>
</tr>
<tr>
<td></td>
<td>• Termination by a parent: A process makes a “terminate P_i” system call to terminate a child process P_i, when it finds that execution of the child process is no longer necessary or meaningful.</td>
</tr>
<tr>
<td></td>
<td>• Exceeding resource utilization: An OS may limit the resources that a process may consume. A process exceeding a resource limit would be aborted by the kernel.</td>
</tr>
<tr>
<td></td>
<td>• Abnormal conditions during operation: The kernel aborts a process if an abnormal condition arises due to the instruction being executed, e.g., execution of an invalid instruction, execution of a privileged instruction, arithmetic conditions like overflow, or memory protection violation.</td>
</tr>
<tr>
<td></td>
<td>• Incorrect interaction with other processes: The kernel may abort a process if it gets involved in a deadlock.</td>
</tr>
</tbody>
</table>
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.

![Diagram of process state transitions](image-url)

**Figure: Fundamental state transitions for a process.**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Remarks</th>
<th>New states</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$P_1$ is preempted</td>
<td>$P_1$ is scheduled</td>
<td>running</td>
</tr>
<tr>
<td>20</td>
<td>$P_2$ is preempted</td>
<td>$P_2$ is scheduled</td>
<td>ready</td>
</tr>
<tr>
<td>25</td>
<td>$P_1$ starts I/O</td>
<td>$P_2$ is scheduled</td>
<td>blocked</td>
</tr>
<tr>
<td>35</td>
<td>$P_2$ is preempted</td>
<td>$P_2$ is scheduled</td>
<td>blocked</td>
</tr>
<tr>
<td>45</td>
<td>$P_2$ starts I/O</td>
<td></td>
<td>blocked</td>
</tr>
</tbody>
</table>

**Table: Process State Transitions in a Time-Sharing System**

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 above shows process states and state
transitions. The transition \textit{ready} \rightarrow \textit{ready swapped} or \textit{blocked} \rightarrow \textit{blocked swapped} is caused by a swap-out action.

The reverse state transition takes place when the process is swapped back into memory. The \textit{blocked swapped} \rightarrow \textit{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.

\textbf{Figure} : Process states and state transitions using two swapped states.

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

\textbf{Process Context and the Process Control Block}

The kernel allocates resources to a process and schedules it for use of the CPU. Accordingly, the kernel's view of a process consists of two parts:

\begin{itemize}
  \item Code, data, and stack of the process, and information concerning memory and other resources, such as files, allocated to it.
  \item Information concerning execution of a program, such as the process state, the CPU state including the stack pointer, and some other items of information described later in this section.
\end{itemize}

These two parts of the kernel's view are contained in the \textit{process context} and the \textit{process control block} (PCB), respectively (see Figure below). This arrangement enables different OS modules to
access relevant process-related information conveniently and efficiently.

**Process Context** The process context consists of the following:

1. **Address space of the process:** The code, data, and stack components of the Process.

2. **Memory allocation information:** Information concerning memory areas allocated to a process. This information is used by the memory management unit (MMU) during operation of the process.

3. **Status of file processing activities:** Information about files being used, such as current positions in the files.

![Figure: Kernel's view of a process.](image)

4. **Process interaction information:** Information necessary to control interaction of the process with other processes, e.g., ids of parent and child processes, and interprocess messages sent to it that have not yet been delivered to it.

5. **Resource information:** Information concerning resources allocated to the process.

6. **Miscellaneous information:** Miscellaneous information needed for operation of a process. The OS creates a process context by allocating memory to the process, loading the process code in the allocated memory and setting up its data space. Information concerning resources allocated to the process and its interaction with other processes is maintained in the process context throughout the life of the process.

This information changes as a result of actions like file open and close and creation and destruction of data by the process during its operation.
**Process Control Block (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 5.6 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 $P_i$ that is blocked on an I/O operation on device $d$. The event information field in $P_i$’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 $P_i$.

**Context Save, Scheduling, and Dispatching**

The context save function performs housekeeping whenever an event occurs. It saves the CPU state of the interrupted process in its PCB, and saves information concerning its context. The interrupted process would have been in the running state before the event occurred. The context save function changes its state to ready. The event handler may later change the interrupted process’s state to blocked, e.g., if the current event was a request for I/O initiation by the interrupted processes itself.

The scheduling function uses the process state information from PCBs to select a ready process for execution and passes its id to the dispatching function.

The dispatching function sets up the context of the selected process, changes its state to running, and loads the saved CPU state from its PCB into the CPU.

To prevent loss of protection, it flushes the address translation buffers used by the memory management unit (MMU). Example illustrates the context save, scheduling, and dispatching functions in an OS using priority-based scheduling.

**Process Switching**

Switching between processes also occurs when a running process becomes blocked as a result of
a request or gets pre-empted at the end of a time slice. An event does not lead to switching between processes if occurrence of the event either (1) causes a state transition only in a process whose priority is lower than that of the process whose operation is interrupted by the event or (2) does not cause any state transition, e.g., if the event is caused by a request that is immediately satisfied. In the former case, the scheduling function selects the interrupted process itself for dispatching. In the latter case, scheduling need not be performed at all; the dispatching function could simply change the state of the interrupted process back to running and dispatch it.

Switching between processes involves more than saving the CPU state of one process and loading the CPU state of another process. The process context needs to be switched as well. We use the term state information of a process to refer to all the information that needs to be saved and restored during process switching. Process switching overhead depends on the size of the state information of a process. Some computer systems provide special instructions to reduce the process switching overhead, e.g., instructions that save or load the PSW and all general-purpose registers, or flush the address translation buffers used by the memory management unit (MMU).

Process switching has some indirect overhead as well. The newly scheduled process may not have any part of its address space in the cache, and so it may perform poorly until it builds sufficient information in the cache. Virtual memory operation is also poorer initially because address translation buffers in the MMU do not contain any information relevant to the newly scheduled process.

**Event Handling**

The following events occur during the operation of an OS:

1. **Process creation event:** A new process is created.
2. **Process termination event:** A process completes its operation.
3. **Timer event:** The timer interrupt occurs.
4. **Resource request event:** Process makes a resource request.
5. **Resource release event:** A process releases a resource.
6. **I/O initiation request event:** Process wishes to initiate an I/O operation.
7. **I/O completion event:** An I/O operation completes.
8. **Message send event:** A message is sent by one process to another.
9. **Message receive event:** A message is received by a process.
10. **Signal send event:** A signal is sent by one process to another.
11. **Signal receive event:** A signal is received by a process.
12. **A program interrupt:** The current instruction in the running process malfunctions.
13. **A hardware malfunction event:** A unit in the computer’s hardware malfunctions.

The timer, I/O completion, and hardware malfunction events are caused by situations that are external to the running process. All other events are caused by actions in the running process. The kernel performs a standard action like aborting the running process when events 12 or 13 occur.
Events Pertaining to Process Creation, Termination, and Preemption

When a user issues a command to execute a program, the command interpreter of the user interface makes a `create_process` system call with the name of the program as a parameter. When a process wishes to create a child process to execute a program, it itself makes a `create_process` system call with the name of the program as a parameter.

The event handling routine for the `create_process` system call creates a PCB for the new process, assigns a unique process id and a priority to it, and puts this information and id of the parent process into relevant fields of the PCB. It now determines the amount of memory required to accommodate the address space of the process, i.e., the code and data of the program to be executed and its stack, and arranges to allocate this much memory to the process (memory allocation techniques are discussed later in Chapters 11 and 12). In most operating systems, some standard resources are associated with each process, e.g., a keyboard, and standard input and output files; the kernel allocates these standard resources to the process at this time. It now enters information about allocated memory and resources into the context of the new process. After completing these chores, it sets the state of the process to `ready` in its PCB and enters this process in an appropriate PCB list.

When a process makes a system call to terminate itself or terminate a child process, the kernel delays termination until the I/O operations that were initiated by the process are completed. It now releases the memory and resources allocated to it. This function is performed by using the information in appropriate fields of the process context. The kernel now changes the state of the process to `terminated`.

The parent of the process may wish to check its status sometime in future, so the PCB of the terminated process is not destroyed now; it will be done sometime after the parent process has checked its status or has itself terminated. If the parent of the process is already waiting for its termination, the kernel must activate the parent process. To perform this action, the kernel takes the id of the parent process from the PCB of the terminated process, and checks the `event information` field of the parent process’s PCB to find whether the parent process is waiting for termination of the child process.

The process in the `running` state should be preempted if its time slice elapses. The context save function would have already changed the state of the running process to `ready` before invoking the event handler for timer interrupts, so the event handler simply moves the PCB of the process to an appropriate scheduling list. Preemption should also occur when a higher-priority process becomes `ready`, but that is realized implicitly when the higher-priority process is scheduled so an event handler need not perform any explicit action for it.

Events Pertaining to Resource Utilization When a process requests a resource through a
system call, the kernel may be able to allocate the resource immediately, in which case event handling does not cause any process state transitions, so the kernel can skip scheduling and directly invoke the dispatching function to resume operation of the interrupted process. If the resource cannot be allocated, the event handler changes the state of the interrupted process to blocked and notes the id of the required resource in the event information field of the PCB. When a process releases a resource through a system call, the event handler need not change the state of the process that made the system call. However, it should check whether any other processes were blocked because they needed the resource, and, if so, it should allocate the resource to one of the blocked processes and change its state to ready. This action requires a special arrangement that we will discuss shortly.

A system call to request initiation of an I/O operation and an interrupt signaling end of the I/O operation lead to analogous event handling actions.

The state of the process is changed to blocked when the I/O operation is initiated and the cause of blocking is noted in the event information field of its PCB; its state is changed back to ready when the I/O operation completes. A request to receive a message from another process and a request to send a message to another process also lead to analogous actions.

**Event Control Block (ECB)** When an event occurs, the kernel must find the process whose state is affected by it. For example, when an I/O completion interrupt occurs, the kernel must identify the process awaiting its completion.

It can achieve this by searching the event information field of the PCBs of all processes.

<table>
<thead>
<tr>
<th>Event description</th>
<th>Process id</th>
<th>ECB pointer</th>
</tr>
</thead>
</table>

**Figure : Event control block (ECB).**

This search is expensive, so operating systems use various schemes to speed it up. We discuss a scheme that uses event control blocks (ECBs). As shown in Figure, an ECB contains three fields. The event description field describes an event, and the process id field contains the id of the process awaiting the event. When a process \( P_i \) gets blocked for occurrence of an event \( e_i \), the kernel forms an ECB and puts relevant information concerning \( e_i \) and \( P_i \) into it. The kernel can maintain a separate ECB list for each class of events like interprocess messages or I/O operations, so the ECB pointer field is used to enter the newly created ECB into an appropriate list of ECBs.

**Summary of Event Handling** Figure below illustrates event handling actions of the kernel.
described earlier. The block action always changes the state of the process that made a system call from ready to blocked. The unblock action finds a process whose request can be fulfilled now and changes its state from blocked to ready. A system call for requesting a resource leads to a block action if the resource cannot be allocated to the requesting process. This action is followed by scheduling and dispatching because another process has to be selected for use of the CPU. The block action is not performed if the resource can be allocated straightaway. In this case, the interrupted process is simply dispatched again.

![Diagram: PCB-ECB interrelationship.](image)

**Figure**: PCB-ECB interrelationship.
When a process releases a resource, an *unblock* action is performed if some other process is waiting for the released resource, followed by scheduling and dispatching because the unblocked process may have a higher priority than the process that released the resource. Again, scheduling is skipped if no process is unblocked because of the event.

**Sharing, Communication, and Synchronization between Processes**

Processes of an application need to interact with one another because they work toward a common goal. Table describes four kinds of process interaction. We summarize their important features in the following.

**Data Sharing** A shared variable may get inconsistent values if many processes update it concurrently. For example, if two processes concurrently execute the statement `a := a + 1`, where `a` is a shared variable, the result may depend on the way the kernel interleaves their execution—the value of `a` may be incremented by only 1. To avoid this problem, only one process should
access shared data at any time, so a data access in one process may have to be delayed if another process is accessing the data. This is called mutual exclusion. Thus, data sharing by concurrent processes incurs the overhead of mutual exclusion.

**Message Passing** A process may send some information to another process in the form of a message. The other process can copy the information into its own data structures and use it. Both the sender and the receiver process must anticipate the information exchange, i.e., a process must know when it is expected to send or receive a message, so the information exchange becomes a part of the convention or protocol between processes.

**Synchronization** The logic of a program may require that an action $a_i$ should be performed only after some action $a_j$ has been performed. Synchronization between processes is required if these actions are performed in different processes—the process that wishes to perform action $a_i$ is made to wait until another process performs action $a_j$.

**Signals** A signal is used to convey an exceptional situation to a process so that it may handle the situation through appropriate actions. The code that a process wishes to execute on receiving a signal is called a signal handler. The signal mechanism is modeled along the lines of interrupts. Thus, when a signal is sent to a process, the kernel interrupts operation of the process and executes a signal handler, if one has been specified by the process; otherwise, it may perform a default action. Operating systems differ in the way they resume a process after executing a signal handler.

**Signals**

A signal is used to notify an exceptional situation to a process and enable it to attend to it immediately. A list of exceptional situations and associated signal names or signal numbers are defined in an OS, e.g., CPU conditions like overflows, and conditions related to child processes, resource utilization, or emergency communications from a user to a process. The kernel sends a signal to a process when the corresponding exceptional situation occurs. Some kinds of signals may also be sent by processes. A signal sent to a process because of a condition in its own activity, such as an overflow condition in the CPU, is said to be a synchronous signal, whereas that sent because of some other condition is said to be an asynchronous signal.

To utilize signals, a process makes a register handler system call specifying a routine that should be executed when a specific signal is sent to it; this routine is called a signal handler. If a process does not specify a signal handler for a signal, the kernel executes a default handler that performs some standard actions like dumping the address space of the process and aborting it.

**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 \( P_i \) and \( P_j \), 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 illustrates the relationship between threads and processes. In the abstract view of Figure, process \( P_i \) has three threads, which are represented by wavy lines inside the circle representing process \( P_i \). Figure 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.

**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 because its parent process 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 3.8 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.

**Coding for Use of Threads**

Threads should ensure correctness of data sharing and synchronization. Action 5.3.1 describes features in the POSIX threads standard that can be used for this purpose. Correctness of data
sharing also has another facet. Functions or subroutines that use static or global data to carry values across their successive activations may produce incorrect results when invoked concurrently, because the invocations effectively share the global or static data concurrently without mutual exclusion. Such routines are said to be thread unsafe. An application that uses threads must be coded in a thread safe manner and must invoke routines only from a thread safe library.

Signal handling requires special attention in a multithreaded application. When several threads are created in a process, which thread should handle a signal? There are several possibilities. The kernel may select one of the threads for signal handling. This choice can be made either statically, e.g., either the first or the last thread created in the process, or dynamically, e.g., the highest priority thread. Alternatively, the kernel may permit an application to specify which thread should handle signals at any time.

A synchronous signal arises as a result of the activity of a thread, so it is best that the thread itself handles it. Ideally, each thread should be able to specify which synchronous signals it is interested in handling.

**Kernel-Level, User-Level, and Hybrid Threads**

These three 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.

### 3.3.2.1 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 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 thread.

![Scheduling of kernel-level threads.](image)

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.

### 3.3.2.2 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 shown in Figure 5.11(b) 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 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.

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.

Figure: Scheduling of user-level threads.

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.

Hybrid Thread Models

A hybrid thread model has both user-level threads and kernel-level threads and a method of associating user-level threads with kernel-level threads. Different methods of associating user- and kernel-level threads provide different combinations of the low switching overhead of user-level threads and the high concurrency and parallelism of kernel-level threads.

Figure illustrates three methods of associating user-level threads with kernel-level threads. The thread library creates user-level threads in a process and associates a thread control block (TCB) with each user-level thread. The kernel creates kernel-level threads in a process and associates a kernel thread control block (KTCB) with each kernel-level thread. In the many-to-one association method, a single kernel-level thread is created in a process by the kernel and all user-level threads created in a process by the thread library are associated with this kernel-level thread. This method of association provides an effect similar to mere user-level threads: User-level threads can be concurrent without being parallel, thread switching incurs low overhead, and blocking of a user-level thread leads to blocking of all threads in the process.

In the one-to-one method of association, each user-level thread is permanently mapped into a kernel-level thread. This association provides an effect similar to mere kernel-level threads: Threads can operate in parallel on different CPUs of a multiprocessor system; however, switching between threads is performed at the kernel level and incurs high overhead. Blocking
of a user-level thread does not block other user-level threads of the process because they are mapped into different kernel-level threads.

The many-to-many association method permits a user-level thread to be mapped into different kernel-level threads at different times.

It provides parallelism between user-level threads that are mapped into different kernel-level threads at the same time, and provides low overhead of switching between user-level threads that are scheduled on the same kernel-level thread by the thread library. However, the many-to-many association method requires a complex implementation.

![Diagram](image)

Figure: (a) Many-to-one; (b) one-to-one; (c) many-to-many associations in hybrid threads.

**Processes in Unix**

**Data Structures** Unix uses two data structures to hold control data about processes:

- **proc structure**: Contains process id, process state, priority, information about relationships with other processes, a descriptor of the event for which a blocked process is waiting, signal handling mask, and memory management information.

- **uarea** (stands for —user areal): Contains a process control block, which stores the CPU state for a blocked process; pointer to proc structure, user and group ids, and information concerning the following: signal handlers, open files and the current directory, terminal attached to the process, and CPU usage by the process.

These data structures together hold information analogous to the PCB data structure. The proc structure mainly holds scheduling related data while the u area contains data related to resource allocation and signal handling. The proc structure of a process is always held in memory. The u area needs to be in memory only when the process is in operation.

**Types of Processes** Two types of processes exist in Unix—user processes and kernel processes.
A user process executes a user computation. It is associated with the user's terminal. When a user initiates a program, the kernel creates the primary process for it, which can create child processes. A daemon process is one that is detached from the user's terminal. It runs in the background and typically performs functions on a system wide basis, e.g., print spooling and network management. Once created, daemon processes can exist throughout the lifetime of the OS. Kernel processes execute code of the kernel.

They are concerned with background activities of the kernel like swapping. They are created automatically when the system is booted and they can invoke kernel functionalities or refer to kernel data structures without having to perform a system call.

**Process Creation and Termination** The system call *fork* creates a child process and sets up its context (called the user-level context in Unix literature). It allocates a proc structure for the newly created process and marks its state as ready, and also allocates a u area for the process. The kernel keeps track of the parent–child relationships using the proc structure. *fork* returns the id of the child process.

The user-level context of the child process is a copy of the parent's user level context. Hence the child executes the same code as the parent. At creation, the program counter of the child process is set to contain the address of the instruction at which the *fork* call returns. The fork call returns a 0 in the child process, which is the only difference between parent and child processes. A child process can execute the same program as its parent, or it can use a system call from the exec family of system calls to load some other program for execution. Although this arrangement is cumbersome, it gives the child process an option of executing the parent's code in the parent's context or choosing its own program for execution.

**Process States and State Transitions**

A process in the running state is put in the ready state the moment its execution is interrupted. A system process then handles the event that caused the interrupt. If the running process had itself caused a software interrupt by executing an <SI_instrn>, its state may further change to blocked if its request cannot be granted immediately. In this model a user process executes only user code; it does not need any special privileges. A system process may have to use privileged instructions like I/O initiation and setting of memory protection information, so the system process executes with the CPU in the kernel mode. Processes behave differently in the Unix model. When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call. To ensure that it has the necessary privileges, it needs to execute with the CPU in the kernel mode. A mode change is thus necessary every time a system call is made. The opposite mode change is necessary after processing a system call. Similar mode changes are needed when a process starts executing the interrupt servicing code in the kernel because of an interrupt, and when it returns after servicing an interrupt.
The Unix kernel code is made reentrant so that many processes can execute it concurrently. This feature takes care of the situation where a process gets blocked while executing kernel code, e.g., when it makes a system call to initiate an I/O operation, or makes a request that cannot be granted immediately. To ensure reentrancy of code, every process executing the kernel code must use its own kernel stack. This stack contains the history of function invocations since the time the process entered the kernel code. If another process also enters the kernel code, the history of its function invocations will be maintained on its own kernel stack. Thus, their operation would not interfere. In principle, the kernel stack of a process need not be distinct from its user stack; however, distinct stacks are used in practice because most computer architectures use different stacks when the CPU is in the kernel and user modes.

Unix uses two distinct running states. These states are called user running and kernel running states. A user process executes user code while in the user running state, and kernel code while in the kernel running state. It makes the transition from user running to kernel running when it makes a system call, or when an interrupt occurs. It may get blocked while in the kernel running state because of an I/O operation or nonavailability of a resource.

![Process state transitions in Unix](image)

**Figure: Process state transitions in Unix.**

**Recommended Questions**

1. List the different types of process interaction and explain them in brief.
2. What is a process? Describe the components of the process environment.
3. List the events that occur during the operation of an OS.
4. Explain in detail the programmer view of processes.
5. What is a process state? Explain the various states of a process giving state transition diagram.
6. Explain event handling pertaining to a process.
7. Explain arrangement and working of threads in SOLARIS with a neat block diagram.
8. Explain with neat diagram i) user threads ii) kernel level threads.
9. With a neat diagram explain different states of a process and state transitions in the UNIX OS.

10. Mention the three kinds of entities used for concurrency within a process in threads in SOLARIS along with a diagram.

11. With a state transition diagram and PCB structure, explain the function of the states, state transitions and functions of a schedule.

12. Explain the race condition in airline reservation system with an algorithm.
UNIT-4

Memory Management

MANAGING THE MEMORY HIERARCHY

A memory hierarchy comprises cache memories like the L1 and L3 caches, the memory management unit (MMU), memory, and a disk. Its purpose is to create an illusion of a fast and large memory at a low cost.

![Diagram of memory hierarchy]

Figure: Managing the memory hierarchy. CPU refers to the fastest memory, the cache, when it needs to access an instruction or data. If the required instruction or data is not available in the cache, it is fetched from the next lower level in the memory hierarchy, which could be a slower cache or the random access memory (RAM), simply called memory in this book. If the required instruction or data is also not available in the next lower level memory, it is fetched there from a still lower level, and so on. Performance of a process depends on the hit ratios in various levels of the memory hierarchy, where the hit ratio in a level indicates what fraction of instructions or data bytes that were looked for in that level were actually present in it.

STATIC AND DYNAMIC MEMORY ALLOCATION

Memory allocation is an aspect of a more general action in software operation known as binding. Two other actions related to a program—its linking and loading—are also aspects of binding. Any entity in a program, e.g., a function or a variable, has a set of attributes, and each attribute has a value. Binding is the act of specifying the value of an attribute. For example, a variable in a program has attributes such as name, type, dimensionality, scope, and memory address. A name binding specifies the variable’s name and a type binding specifies its type. Memory binding is the act of specifying the variable’s memory address; it constitutes memory allocation for the variable. Memory allocation to a process is the act of specifying memory addresses of its instructions and data. A binding for an attribute of an entity such as a function or
a variable can be performed any time before the attribute is used. Different binding methods perform the binding at different times. The exact time at which binding is performed may determine the efficiency and flexibility with which the entity can be used. Broadly speaking, we can differentiate between early binding and late binding. Late binding is useful in cases where the OS or run-time library may have more information about an entity at a later time, using which it may be able to perform a better quality binding. For example, it may be able to achieve more efficient use of resources such as memory. Early and late binding are represented by the two fundamental binding methods of static and dynamic binding, respectively.

**Static Binding** A binding performed before the execution of a program (or operation of a software system) is set in motion.

**Dynamic Binding** A binding performed during the execution of a program (or operation of a software system).

*Static memory allocation* can be performed by a compiler, linker, or loader while a program is being readied for execution. *Dynamic memory allocation* is performed in a —lazyl manner during the execution of a program; memory is allocated to a function or a variable just before it is used for the first time. Static memory allocation to a process is possible only if sizes of its data structures are known before its execution begins. If sizes are not known, they have to be guessed; wrong estimates can lead to wastage of memory and lack of flexibility. For example, consider an array whose size is not known during compilation. Memory is wasted if we overestimate the array's size, whereas the process may not be able to operate correctly if we underestimate its size. Dynamic memory allocation can avoid both these problems by allocating a memory area whose size matches the actual size of the array, which would be known by the time the allocation is performed. It can even permit the array size to vary during operation of the process. However, dynamic memory allocation incurs the overhead of memory allocation actions performed during operation of a process. Operating systems choose static and dynamic memory allocation under different circumstances to obtain the best combination of execution efficiency and memory efficiency. When sufficient information about memory requirements is available a priori, the kernel or the run-time library makes memory allocation decisions statically, which provides execution efficiency. When little information is available a priori, the memory allocation decisions are made dynamically, which incurs higher overhead but ensures efficient use of memory. In other situations, the available information is used to make some decisions concerning memory allocation statically, so that the overhead of dynamic memory allocation can be reduced.

**EXECUTION OF PROGRAMS**

A program P written in a language $L$ has to be transformed before it can be executed. Several of these transformations perform memory binding—each one binds the instructions and data of the program to a new set of addresses. Figure below is a schematic diagram of three transformations performed on program P before it can be loaded in memory for execution.

- **Compilation or assembly**: A compiler or an assembler is generically called a *translator*. It translates program P into an equivalent program in the *object module* form. This program contains instructions in the machine language of the computer. While invoking the translator,
the user specifies the *origin* of the program, which is the address of its first instruction or byte; otherwise, the translator assumes a default address, typically 0. The translator accordingly assigns addresses to other instructions and data in the program and uses these addresses as operand addresses in its instructions. The *execution start address* or simply the *start address* of a program is the address of the instruction with which its execution is to begin. It can be the same as the origin of the program, or it can be different. The addresses assigned by the translator are called *translated addresses*. Thus, the translator binds instructions and data in program P to translated addresses. An object module indicates the translated origin of the program, its translated start address, and size.

**Linking**: Program P may call other programs during its execution, e.g., functions from mathematical libraries. These functions should be included in the program, and their start addresses should be used in the function call instructions in P. This procedure is called *linking*. It is achieved by selecting object modules for the called functions from one or more libraries and merging them with program P.

![Figure: Schematic diagram of transformation and execution of a program.](image)

**Relocation**: Some object module(s) merged with program P may have conflicting translated time addresses. This conflict is resolved by changing the memory binding of the object module(s); this action is called *relocation* of object modules. It involves changing addresses of operands used in their instructions.

The relocation and linking functions are performed by a program called a *linker*. The addresses assigned by it are called *linked addresses*. The user may specify the linked origin for the program; otherwise, the linker assumes the linked origin to be the same as the translated origin. In accordance with the linked origin and the relocation necessary to avoid address conflicts, the linker binds instructions and data of the program to a set of linked addresses. The resulting program, which is in a ready-to-execute program form called a *binary program*, is stored in a library. The directory of the library stores its name, linked origin, size, and the linked start address. A binary program has to be loaded in memory for execution. This function is performed by the *loader*. If the start address of the memory area where a program is to be loaded, which is called its *load origin*, differs from the linked origin of program, the loader has to change its memory binding yet again. A loader possessing this capability is called a *relocating loader*, whereas a loader without this capability is called an *absolute loader*.

**Static and Dynamic Relocation of Programs**
When a program is to be executed, the kernel allocates it a memory area that is large enough to accommodate it, and invokes the loader with the name of the program and the load origin as parameters. The loader loads the program in the memory allocated to it, relocates it using the scheme illustrated in Example 4.1 if the linked origin is different from the load origin, and passes it control for execution. This relocation is static relocation as it is performed before execution of the program begins. Sometime after the program’s execution has begun, the kernel may wish to change the memory area allocated to it so that other programs can be accommodated in memory. This time, the relocation has to be performed during execution of the program, hence it constitutes dynamic relocation. Dynamic relocation can be performed by suspending a program’s execution, carrying out the relocation procedure described earlier, and then resuming its execution. However, it would require information concerning the translated origin and address-sensitive instructions to be available during the program’s execution. It would also incur the memory and processing costs described earlier. Some computer architectures provide a relocation register to simplify dynamic relocation. The relocation register is a special register in the CPU whose contents are added to every memory address used during execution of a program. The result is another memory address, which is actually used to make a memory reference. Thus,

\[
\text{Effective memory address} = \text{memory address used in the current instruction} + \text{contents of relocation register}
\]

The following example illustrates how dynamic relocation of a program is achieved by using the relocation register.

**Example Dynamic Relocation through Relocation Register**

A program has the linked origin of 50000, and it has also been loaded in the memory area that has the start address of 50000. During its execution, it is to be shifted to the memory area having the start address of 70000, so it has to be relocated to execute in this memory area. This relocation is achieved simply by loading an appropriate value in the relocation register, which is computed as follows:

Value to be loaded in relocation register

\[
= \text{start address of allocated memory area} - \text{linked origin of program}
\]

\[
70000 - 50000 = 20000
\]

![Diagram of program relocation using a relocation register](image)

Figure: Program relocation using a relocation register: (a) program; (b) its execution.

Consider execution of the Add instruction in the program shown in Figure 4.4(a). This
instruction has the linked address 55000 in the program and uses an operand whose linked address is 65784. As a result of relocation, the program exists in the memory area starting with the address 70000. Figure 11.4(b) shows the load addresses of its instructions and data; the corresponding linked addresses are shown in parenthesis for easy reference. The Add instruction exists in the location with address 75000. The address of its operand is 65784 and the relocation register contains 20000, so during execution of the instruction, the effective address of its operand is 65784 + 20000 = 85784. Hence the actual memory access is performed at the address 85784.

**Linking**

An ENTRY statement in an assembly program indicates symbols that are defined in the assembly program and may be referenced in some other assembly programs. Such symbols are called *entry points*. An EXTRN statement in an assembly program indicates symbols that are used in the assembly program but are defined in some other assembly program. These symbols are called external symbols and uses of these symbols in the assembly program are called *external references*. The assembler puts information about the ENTRY and EXTRN statements in an object module for use by the linker.

*Linking* is the process of binding an external reference to the correct linked address. The linker first scans all object modules being linked together to collect the names of all entry points and their linked addresses. It stores this information in a table for its own use. It then considers each external reference, obtains the linked address of the external symbol being referenced from its table, and puts this address in the instruction containing the external reference. This action is called resolution of an external reference. The next example illustrates the steps in linking.

**MEMORY ALLOCATION TO A PROCESS**

**Stacks and Heaps**

The compiler of a programming language generates code for a program and allocates its static data. It creates an object module for the program. The linker links the program with library functions and the run-time support of the programming language, prepares a ready-to-execute form of the program, and stores it in a file. The program size information is recorded in the directory entry of the file. The run-time support allocates two kinds of data during execution of the program. The first kind of data includes variables whose scope is associated with functions, procedures, or blocks, in a program and parameters of function or procedure calls. This data is allocated when a function, procedure or block is entered and is deallocated when it is exited. Because of the last-in, first-out nature of the allocation/deallocation, the data is allocated on the stack. The second kind of data is dynamically created by a program through language features like the new statement of Pascal, C++, or Java, or the malloc, calloc statements of C. We refer to such data as *program-controlled dynamic data* (PCD data). The PCD data is allocated by using a data structure called a heap.

**Stack** In a stack, allocations and deallocations are performed in a last-in, first-out (LIFO) manner in response to *push* and *pop* operations, respectively. We assume each entry in the stack to be of some standard size, say, $l$ bytes. Only the last entry of the stack is accessible at any time. A contiguous area of memory is reserved for the stack. A pointer called the *stack base* (SB) points to the first entry of the stack, while a pointer called the *top of stack* (TOS) points to the last entry allocated in the stack. We will use the convention that a stack grows toward the lower end of memory; we depict it as upward growth in the figures. During execution of a program, a stack is
used to support function calls. The group of stack entries that pertain to one function call is called a stack frame; it is also called an activation record in compiler terminology. A stack frame is pushed on the stack when a function is called. To start with, the stack frame contains either addresses or values of the function’s parameters, and the return address, i.e., the address of the instruction to which control should be returned after completing the function’s execution. During execution of the function, the run-time support of the programming language in which the program is coded creates local data of the function within the stack frame. At the end of the function’s execution, the entire stack frame is popped off the stack and the return address contained in it is used to pass control back to the calling program. Two provisions are made to facilitate use of stack frames: The first entry in a stack frame is a pointer to the previous stack frame on the stack. This entry facilitates popping off of a stack frame. A pointer called the frame base (FB) is used to point to the start of the topmost stack frame in the stack. It helps in accessing various stack entries in the stack frame. Example illustrates how the stack is used to implement function calls.

Figure: Stack after (a) main calls sample; (b) sample calls calc.

Heap A heap permits allocation and deallocation of memory in a random order. An allocation request by a process returns with a pointer to the allocated memory area in the heap, and the process accesses the allocated memory area through this pointer. A deallocation request must present a pointer to the memory area to be deallocated. The next example illustrates use of a heap to manage the PCD data of a process. As illustrated there, —holes— develop in the memory allocation as data structures are created and freed. The heap allocator has to reuse such free memory areas while meeting future demands for memory.

HEAP MANAGEMENT

Reuse of Memory
The speed of memory allocation and efficient use of memory are the two fundamental concerns in the design of a memory allocator. Stack-based allocation addresses both these concerns effectively since memory allocation and deallocation is very fast—the allocator modifies only the SB, FB, and TOS pointers to manage the free and allocated memory and released memory is
reused automatically when fresh allocations are made. However, stack based allocation cannot be used for data that are allocated and released in an unordered manner. Hence heap allocators are used by run-time support of programming languages to manage PCD data, and by the kernel to manage its own memory requirements. In a heap, reuse of memory is not automatic; the heap allocator must try to reuse a free memory area while making fresh allocations. However, the size of a memory request rarely matches the size of a previously used memory area, so some memory area is left over when a fresh allocation is made. This memory area will be wasted if it is too small to satisfy a memory request, so the allocator must carefully select the memory area that is to be allocated to the request. This requirement slows down the allocator. Because of the combined effect of unusably small memory areas and memory used by the allocator for its own data structures, a heap allocator may not be able to ensure a high efficiency of memory utilization.

**Maintaining a Free List**

The kernel needs to maintain two items of control information for each memory area in the free list: the size of the memory area and pointers used for forming the list. To avoid incurring a memory overhead for this control information, the kernel stores it in the first few bytes of a free memory area itself. Figure 4.10(a) shows a *singly linked free list* in a heap that contains five areas marked a–e in active use and three free areas x–z. Each memory area in the free list contains its size and a pointer to the next memory area in the list. This organization is simple; however, it requires a lot of work when a memory area is to be inserted into the list or deleted from it. For example, deletion of a memory area from the list requires a change in the pointer stored in the previous memory area in the list. Insertion of a memory area at a specific place in the list also involves a similar operation.

**Performing Fresh Allocations by Using a Free List**

Three techniques can be used to perform memory allocation by using a free list:
- First-fit technique
- Best-fit technique
- Next-fit technique

**CONTIGUOUS MEMORY ALLOCATION**

Contiguous memory allocation is the classical memory allocation model in which each process is allocated a single contiguous area in memory. Thus the kernel allocates a large enough memory area to accommodate the code, data, stack, and PCD data of a process as shown in Figure 11.9. Contiguous memory allocation faces the problem of memory fragmentation. In this section we focus on techniques to address this problem.

**Handling 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. The techniques of memory compaction and reuse of memory discussed earlier in Section 11.5 can be applied to overcome the problem of external fragmentation. Example 11.8 illustrates use of memory compaction.
Contiguous Memory Allocation
Example
Processes A, B, C, and D are in memory in Figure. Two free areas of memory exist after B terminates; however, neither of them is large enough to accommodate another process [see Figure]. The kernel performs compaction to create a single free memory area and initiates process E in this area [see Figure]. 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.

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.

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 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 Noncontiguous Memory Allocation In Figure, 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 ]. 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 ]:

![Noncontiguous memory allocation to process P.](image)

**4.8 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 \((pi, bi)\), \( pi \geq 0 \) and \( 0 \leq bi < s \). 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 \( 5 \times 1024 + 128 \), 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 11.19 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 Eq above.

![Figure shows the physical view of execution of processes P and R.](image)
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).

\[
\text{Effective memory address of (5, 128)} = \text{start address of page frame #8 + 128} = 8 \times 1024 + 128 = 8320
\]

We use the following notation to describe how address translation is actually performed: 

\begin{align*}
&\text{s} \quad \text{Size of a page} \\
&\text{ll} \quad \text{Length of a logical address (i.e., number of bits in it) } \text{lp} \\
&\text{Length of a physical address} \\
&\text{nb} \quad \text{Number of bits used to represent the byte number in a logical address } \text{np} \\
&\text{Number of bits used to represent the page number in a logical address } \text{nf} \quad \text{Number of bits used to represent the frame number in a physical address}
\end{align*}

The size of a page, s, is a power of 2. nb is chosen such that \( s = 2^{nb} \). 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.

**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.

![Process Q in segmentation.](image)

**KERNEL MEMORY ALLOCATION**

The kernel creates and destroys data structures at a high rate during its operation. These are mostly *control blocks* that control the allocation and use of resources in the system. Some familiar control blocks are the process control block (PCB) created for every process and the event control block (ECB) created whenever the occurrence of an event is anticipated. The I/O control block (IOCB) created for an I/O operation and the file control block (FCB) created for every open file.) The sizes of control blocks are known in the design stage of an OS. This prior knowledge helps make kernel memory allocation simple and efficient—memory that is released when one control block is destroyed can be reused when a similar control block is created. To realize this benefit, a separate free list can be maintained for each type of control block. Kernels of modern operating systems use noncontiguous memory allocation with paging to satisfy their own memory requirements, and make special efforts to use each page effectively. Three of the leading memory allocators are:

- McKusick–Karels allocator
- Lazy buddy allocator
- Slab allocator

**McKusick--Karels Allocator:**

This is a modified power-of-2 allocator; it is used in Unix 4.4 BSD. The allocator has an integral number of pages at its disposal at any time, and asks the paging system for more pages when it runs out of memory to allocate. The basic operating principle of the allocator is to divide each page into blocks of equal size and record two items of information—the block size, and a free
list pointer—under the logical address of the page. This way, the address of the page in which a block is located will be sufficient for finding the size of the block and the free list to which the block should be added when it is freed. Hence, it is not necessary to have a header containing this information in each allocated block as in a conventional power-of-2 allocator. With the elimination of the header element, the entire memory in a block can be used for the intended purpose.

Consequently, the McKusick–Karels allocator is superior to the power-of-2 allocator when a memory request is for an area whose size is an exact power of 2. A block of identical size can be allocated to satisfy the request, whereas the conventional power-of-2 allocator would have allocated a block whose size is the next higher power of 2. The allocator seeks a free page among those in its possession when it does not find a block of the size it is looking for. It then divides this page into blocks of the desired size. It allocates one of these blocks to satisfy the current request, and enters the remaining blocks in the appropriate free list. If no free page is held by the allocator, it asks the paging system for a new page to be allocated to it. To ensure that it does not consume a larger number of pages than necessary, the allocator marks any page in its possession as free when all blocks in it become free. However, it lacks a feature to return free pages to the paging system. Thus, the total number of pages allocated to the allocator at any given moment is the largest number of pages it has held at any time. This burden may reduce the memory utilization factor.

Lazy Buddy Allocator: The buddy system in its basic form may perform one or more splits at every allocation and one or more coalescing actions at every release. Some of these actions are wasteful because a coalesced block may need to be split again later. The basic design principle of the lazy buddy allocator is to delay coalescing actions if a data structure requiring the same amount of memory as a released block is likely to be created. Under the correct set of conditions, this principle avoids the overhead of both coalescing and splitting. The lazy buddy allocator used in Unix 5.4 works as follows: Blocks with the same size are considered to constitute a class of blocks. Coalescing decisions for a class are made on the basis of the rates at which data structures of the class are created and destroyed. Accordingly, the allocator characterizes the behaviour of the OS with respect to a class of blocks into three states called lazy, reclaiming, and accelerated. For simplicity we refer to these as states of a class of blocks.

In the lazy state, allocations and releases of blocks of a class occur at matching rates. Consequently, there is a steady and potentially wasteful cycle of splitting and coalescing. As a remedy, excessive coalescing and splitting can both be avoided by delaying coalescing. In the reclaiming state, releases occur at a faster rate than allocations so it is a good idea to coalesce at every release. In the accelerated state, releases occur much faster than allocations, and so it is desirable to coalesce at an even faster rate; the allocator should attempt to coalesce a block being released, and, additionally, it should also try to coalesce some other blocks that were released but not coalesced in the past. The lazy buddy allocator maintains the free list as a doubly linked list. This way both the start and end of the list can be accessed equally easily. A bit map is maintained to indicate the allocation status of blocks. In the lazy state, a block being released is simply added to the head of the free list. No effort is made to coalesce it with its buddy. It is also not marked free in the bit map. This way the block will not be coalesced even if its buddy is released in future. Such a block is said to be locally free. Being at the head of the list, this block will be allocated before any other block in the list. Its allocation is efficient and fast because the bit map does not need to be updated—it still says that the block is allocated. In the reclaiming and accelerated states a block is both added to the free list and marked free in the bit map. Such
a block is said to be *globally free*. Globally free blocks are added to the end of the free list. In the reclaiming state the allocator tries to coalesce a new globally free block transitively with its buddy. Eventually a block is added to some free list—either to a free list to which the block being released would have belonged, or to a free list containing larger-size blocks. Note that the block being added to a free list could be a locally free block or a globally free block according to the state of that class of blocks. In the accelerated state the allocator tries to coalesce the block being released, just as in the reclaiming state, and additionally tries to coalesce one other locally free block—the block found at the start of the free list—with its buddy. The state of a class of blocks is characterized as follows: Let $A$, $L$, and $G$ be the number of allocated, locally free, and globally free blocks of a class, respectively. The total number of blocks of a class is given by $N = A + L + G$. A parameter called *slack* is computed as follows:

$$ slack = N - 2 \times L - G $$

A class is said to be in the lazy, reclaiming, or accelerated state if the value of *slack* is $\geq 2$, $1$, or $0$, respectively. (The allocator ensures that slack is never $< 0$.)

The coalescing overhead is different in these three states. There is no overhead in the lazy state. Hence release and allocation of blocks is fast. In the reclaiming state the overhead would be comparable with that in the buddy system, whereas in the accelerated state the overhead would be heavier than in the buddy system. It has been shown that the average delays with the lazy buddy allocator are 10 to 32 percent lower than average delays in the case of a buddy allocator. The implementation of the lazy buddy allocator in Unix 5.4 uses two kinds of blocks. Small blocks vary in size between 8 and 256 bytes. Large blocks vary in size between 512 and 16 KB. The allocator obtains memory from the paging system in 4 KB areas. In each area, it creates a pool of blocks and a bit map to keep track of the allocation status of the blocks. When all blocks in the pool are free, it returns the area to the paging system. This action overcomes the problem of nonreturnable blocks seen in the McKusick–Karels allocator.

**Slab Allocator:**

The slab allocator was first used in the Solaris 2.4 operating system; it has been used in Linux since version 2.2. A *slab* consists of many *slots*, where each slot can hold an active object that is a kernel data structure, or it may be empty. The allocator obtains standard-size memory areas from the paging system and organizes a slab in each memory area. It obtains an additional memory area from the paging system and constructs a slab in it when it runs out of memory to allocate, and it returns a memory area to the paging system when all slots in its slab are unused. All kernel objects of the same class form a pool. For small objects, a pool consists of many slabs and each slab contains many slots.

The slabs of a pool are entered in a doubly linked list to facilitate addition and deletion of slabs. A slab may be full, partially empty, or empty, depending on the number of active objects existing in it. To facilitate searches for an empty slab, the doubly linked list contains the slabs of a pool is sorted according to the slab’s status—all full slabs are at the start of the list, partially empty slabs are in the middle, and empty slabs are at the end of the list. Each slab contains a free list from which free slots can be allocated. Each pool contains a pointer to the first slab that contains a free slot. This arrangement makes allocation very efficient.
Figure 4.23 shows the format of a slab. When the allocator obtains a memory area from the paging system, it formats the memory area into a slab by creating an integral number of slots, a free list containing all slots, and a descriptor field at the end of the slab that contains both the count of active objects in it and the free list header. Each slot in the slab is then initialized; this action involves initializing the various fields in it with object-specific information like fixed strings of constant values. When allocated, the slot can be used as an object straightaway.

![Figure: Format of a slab.](image)

At deallocation time, the object is brought back to its allocation time status, and the slot is added to the free list. Since some fields of the objects never change, or change in such a manner that their values at deallocation time are the same as their values at allocation time, this approach eliminates the repetitive overhead of object initialization suffered in most other allocators. However, use of initialized objects has some implications for the memory utilization factor. If a free slot were simply free memory, a part of this memory itself could be used as the free list pointer; but a slot is an initialized object, and so the pointer field must be located outside the object’s area even when the slot is free.

**Questions**

1. Describe the features of static and dynamic memory allocation. What are the four program components for which memory is to be allocated?
2. Compare contiguous and non-contiguous memory allocation. Enumerate the practical issues associated with them.
3. Explain the slab allocator of Solaris 2.4 system.
4. With a neat diagram mention the components of a memory allocation to a program during its execution. Also describe the memory allocation preliminaries.
5. Explain the working of a buddy system allocator for program controlled data. How does it differ from process-of-two allocator?
6. What is memory fragmentation? Discuss the method of memory compaction and reuse of memory concepts to overcome the problem of memory fragmentation. Give examples.
7. Explain the techniques used to perform memory allocation using a free list.
8. Explain internal and external fragmentation with examples.

Describe i) best fit technique for free space allocation ii) variable partitioned allocation with their merits and demerits
UNIT-5

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.

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 (comp, byte).

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

Figure below 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.
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.

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. 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.

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 $\#frames \times s - 1$. 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 $pi$, $0 \leq bi < s$.

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

Effective memory address of logical address $(pi, bi) = start address of the page frame containing page pi + bi$

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 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 $P_2$ has its pages 1 and 2 in memory. They occupy page frames 5 and 7 respectively. Process $P_1$ has its pages 0 and 2 in page frames 4 and 1, while process $P_3$ 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.

![Diagram showing memory and page frames]

Figure: Address translation in virtual memory using paging.

Process $P_2$ is currently executing the instruction ‘Add 2528’, so the MMU uses $P_2$’s page table for address translation. The MMU views the operand address 2528 as the pair (2, 480) because $2528 = 2 \times 1024 + 480$. It now accesses the entry for page 2 in $P_2$’s page table. This entry contains frame number 7, so the MMU forms the effective address $7 \times 1024 + 480$ according to Eq. , and uses it to make a memory access. In effect, byte 480 in page frame 7 is accessed.

**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. The *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.

**Page Faults and Demand Loading of Pages**

Table 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 $pi$.
Figure above 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 term page 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, which we will call program I/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 the CPU 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:

\[ \text{Effective memory access time} = pr_1 \times 2 \times t_{\text{mem}} + (1 - pr_1) \times (t_{\text{mem}} + t_{\text{pfh}} + 2 \times t_{\text{mem}}) \]

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

**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 $p1$.

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.

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.

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 \(t_1, t_2, t_3, \ldots\) with each page reference string. This way, the \(k\)th page reference in a page reference string is assumed to have occurred at time instant \(t_k\). (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
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 $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, ...$

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 $t_1$. B, the operand of the instruction is situated in page 5, and so the second page reference in the string is 5, at time $t_2$. 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 illustrates operation of the optimal, FIFO, and LRU page replacement policies.

**Example  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, ...
Reference time string t₁, t₂, t₃, t₄, t₅, t₆, t₇, ...

Figure 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.

<table>
<thead>
<tr>
<th>Time instant</th>
<th>Page ref</th>
<th>Optimal Valid</th>
<th>Optimal Ref info</th>
<th>FIFO Valid</th>
<th>FIFO Ref info</th>
<th>LRU Valid</th>
<th>LRU Ref info</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t₂</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t₃</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t₄</td>
<td>2</td>
<td>Replace 1 by 2</td>
<td>Replace 1 by 2</td>
<td>Replace 1 by 2</td>
<td>Replace 1 by 2</td>
<td>Replace 1 by 2</td>
<td></td>
</tr>
<tr>
<td>t₅</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>t₆</td>
<td>1</td>
<td>Replace 0 by 1</td>
<td>Replace 2 by 1</td>
<td>Replace 2 by 1</td>
<td>Replace 2 by 1</td>
<td>Replace 2 by 1</td>
<td></td>
</tr>
<tr>
<td>t₇</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure**: Comparison of page replacement policies with alloc = 2.
Problems in FIFO Page Replacement

Example
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, . . .
Reference time string t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, . . .

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 \( p_{1}^{12} = \{2, 1, 4, 3\} \), while \( p_{1}^{12} = \{1, 5, 2\} \). Thus, FIFO page replacement does not exhibit the stack property. This leads to a page fault at \( t_{13} \) when \( alloc = 4 \), but not when \( alloc = 3 \). Thus, a total of 10 page faults arise in 13 time instants when \( alloc = 4 \), while 9 page faults arise when \( alloc = 3 \). For LRU, we see that \( p_{1}^{13} \subseteq p_{1}^{14} \) at all time instants.

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

Figure 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.
SHARED PAGES

**Static sharing** results from static binding performed by a linker or loader before execution of a program begins. Figure below shows the logical address space of program C. The Add (4,12) instruction in page 1 has its operand in page 4. With static binding, if two processes A and B statically share program C, then C is included in the code of both A and B. Let the 0th page of C become page $i$ of process A [see Figure]. The instruction Add (4,12) in page 1 of program would be relocated to use the address $(i+4,12)$. If the 0th page of C becomes page $j$ in process B, the Add instruction would be relocated to become Add $(j+4, 12)$ Thus, each page of program C has two copies in the address spaces of A and B. These copies may exist in memory at the same time if processes A and B are in operation simultaneously.

**Dynamic binding** can be used to conserve memory by binding the same copy of a program or data to several processes. In this case, the program or data to be shared would retain its identity [see Figure ]. It is achieved as follows: The virtual memory manager maintains a *shared pages table* to hold information about shared pages in memory. Process A makes a system call to bind
program C as a shared program starting at a specific page, say, page $i$, in its logical address space. The kernel invokes the virtual memory manager, which creates entries in the page table of A for pages of program C, and sets an $s$ flag in each of these entries to indicate that it pertains to a shared page. It now checks whether the pages of program C have entries in the shared pages table. If not, it makes such entries now, sets up the swap space for program C, and invokes the dynamic linker, which dynamically binds program C to the code of process A. During this binding, it relocates the address-sensitive instructions of C. Thus, the Add instruction in page 1 of program C is modified to read Add $(i + 4, 12)$ [see Figure 5.22(c)]. When a reference to an address in program C page faults, the virtual memory manager finds that it is a shared page, so it checks the shared pages table to check whether the required page is already in memory, which would happen if another process had used it recently. If so, it copies the page frame number of the page from the shared pages table into the entry of that page in A’s page table; otherwise, it loads the page in memory and updates its entry in A’s page table and in the shared pages table.

Figure: Sharing of program C by processes A and B: (a) program C; (b) static binding of C to the codes of processes A and B; and (c) dynamic binding of C.

Unix terms of virtual memory

**Page Replacement** The system permits a process to fix a certain fraction of its pages in memory to reduce its own page fault rate and improve its own performance. These pages cannot be removed from memory until they are unfixed by the process. Interestingly, there is no I/O fixing of pages in Unix since I/O operations take place between a disk block and a block in the buffer cache rather than between a disk block and the address space of a process.

Unix page replacement is analogous to the schematic of Figure , including the use of a clock algorithm. To facilitate fast page-in operations, Unix virtual memory manager maintain a list of free page frames and try to keep at least 5 percent of total page frames on this list at all times. A daemon called the pageout daemon (which is labeled process 2 in the system) is created for this purpose. It is activated any time the total number of free page frames falls below 5 percent. It tries to add pages to the free list and puts itself to sleep when the free list contains more than 5 percent free page frames. Some versions of Unix use two thresholds—a high threshold and a low threshold—instead of a single threshold at 5 percent. The daemon goes to sleep when it finds that the number of pages in the free list exceeds the high threshold. It is activated when this number falls below the low threshold. This arrangement avoids frequent activation and deactivation of the daemon.
The virtual memory manager divides pages that are not fixed in memory into active pages, i.e., pages that are actively in use by a process, and inactive pages, i.e., pages that have not been referenced in the recent past. The virtual memory manager maintains two lists, the active list and the inactive list. Both lists are treated as queues. A page is added to the active list when it becomes active, and to the inactive list when it is deemed to have become inactive. Thus the least recently activated page is at the head of the active list and the oldest inactive page is at the head of the inactive list. A page is moved from the inactive list to the active list when it is referenced. The pageout daemon tries to maintain a certain number of pages, computed as a fraction of total resident pages, in the inactive list. If it reaches the end of the inactive list while adding page frames to the free list, it checks whether the total number of pages in the inactive list is smaller than the expected number. If so, it transfers a sufficient number of pages from the active list to the inactive list.

**Swapping** The Unix virtual memory manager does not use a working set memory allocator because of the high overhead of such an allocator. Instead it focuses on maintaining needed pages in memory. A process is swapped out if all its required pages cannot be maintained in memory and conditions resembling thrashing exist in the system. An inactive process, i.e., a process that is blocked for a long time, may also be swapped out in order to maintain a sufficient number of free page frames. When this situation arises and a swap-out becomes necessary, the pageout daemon activates the swapper, which is always process 0 in the system.

The swapper finds and swaps out inactive processes. If that does not free sufficient memory, it is activated again by the pageout daemon. This time it swaps out the process that has been resident the longest amount of time. When swapped out processes exist in the system, the swapper periodically checks whether sufficient free memory exists to swap some of them back in. A swap-in priority—which is a function of when the process was swapped out, when it was last active, its size and its nice value—is used for this purpose (see Section 7.6.1 for details of the nice value). This function ensures that no process remains swapped out indefinitely.

In Unix 4.3BSD, a process was swapped-in only if it could be allocated as much memory as it held when it was swapped out. In Unix 4.4BSD this requirement was relaxed; a process is brought in if enough memory to accommodate its user structure and kernel stack can be allocated to it.
Questions
1. Explain the important concepts in the operation of demand paging.
2. Write a note on page replacement policies and page sharing.
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. Explain —page-out daemon— for handling virtual memory in UNIX OS.
7. Describe the address translation using TU and TLB in demand paged allocation with a block diagram.
8. 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
UNIT-6

FILE SYSTEMS

This chapter deals with the design of the file system. After discussing the basics of file organizations, directory structures and disk space management, we describe the file sharing semantics that govern concurrent sharing of files and file system reliability.

File System and the 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 file 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 other users' 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 summarizes the facilities provided by the file system and the IOCS. The file system provides directory structures that enable users to organize their data into logical groups of files, e.g., one group of files for each professional activity. The file system provides protection against illegal file accesses and ensures correct operation when processes access and update a file concurrently. It also ensures that data is reliably stored, i.e., data is not lost when system crashes occur. Facilities of the IOCS are as described earlier.

The file system and the IOCS form a hierarchy. Each of them has policies and provides mechanisms to implement the policies.

Table: Facilities Provided by the File System and the Input-Output Control System

<table>
<thead>
<tr>
<th>File System</th>
<th>Input-Output Control System (IOCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Directory structures for convenient grouping of files</td>
<td>• Efficient operation of I/O devices</td>
</tr>
<tr>
<td>• Protection of files against illegal accesses</td>
<td>• Efficient access to data in a file</td>
</tr>
<tr>
<td>• File sharing semantics for concurrent accesses to a file</td>
<td></td>
</tr>
<tr>
<td>• Reliable storage of files</td>
<td></td>
</tr>
</tbody>
</table>

Data and Metadata

A file system houses two kinds of data—data contained within files, and data used to access files. We call the data within files file data, or simply data. The data used to access files is called control data, or metadata. In the logical view shown in Figure 6.1, data contained in the directory structure is metadata.
FILES AND FILE OPERATIONS

**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
Figure below 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  shows a byte stream file report.

![Logical views of a structured file employee_info; a byte stream file report.](image)

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. At the end of the file processing activity, the file system stores changed values of the file’s attributes, if any, in the file’s directory entry.

**FUNDAMENTAL FILE ORGANIZATIONS AND ACCESS METHODS**

The term—record access pattern—to describe the order in which records in a file are accessed by a process. The two fundamental record access patterns are *sequential access*, in which records are accessed in the order in which they fall in a file (or in the reverse of that order), and *random access*, in which records may be accessed in any order. The file processing actions of a process will execute efficiently only if the process’s record access pattern can be implemented efficiently in the file system. The characteristics of an I/O device make it suitable for a specific record access pattern. For example, a tape drive can access only the record that is placed immediately before or after the current position of its read/write head.

Hence it is suitable for sequential access to records. A disk drive can directly access any record given its address. Hence it can efficiently implement both the sequential and random record access patterns.

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. While compiling a program, the compiler infers the file organization governing a file from the file’s declaration statement (or from the rules for default, if the program does not contain a file declaration statement), and identifies the correct access method to invoke for operations on the file.

**4 Access Methods**

An *access method* is a module of the IOCS that implements accesses to a class of files using a specific file organization. The procedure to be used for accessing records in a file, whether by a sequential search or by address calculation, is determined by the file organization. The access method module uses this procedure to access records. It may also use some advanced techniques in I/O programming to make file processing more efficient. Two such techniques are *buffering* and *blocking* of records.

**Buffering of Records** The access method reads records of an input file ahead of the time when they are needed by a process and holds them temporarily in memory areas called *buffers* until they are actually used by the process. The purpose of buffering is to reduce or eliminate the wait
for an I/O operation to complete; the process faces a wait only when the required record does
not already exist in a buffer. The converse actions are performed for an output file. When the
process performs a write operation, the data to be written into the file is copied into a buffer and
the process is allowed to continue its operation. The data is written on the I/O device sometime
later and the buffer is released for reuse. The process faces a wait only if a buffer is not available
when it performs a write operation.

**Blocking of Records** The access method always reads or writes a large block of data, which
contains several file records, from or to the I/O medium. This feature reduces the total number
of I/O operations required for processing a file, thereby improving the file processing efficiency
of a process. Blocking also improves utilization of an I/O medium and throughput of a device.

**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 noncontiguous 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.

The file system can maintain a *free list* of disk space and allocate from it when a file requires a
new disk block. Alternatively, it can use a table called the *disk status map* (DSM) to indicate the
status of disk blocks. The DSM has one entry for each disk block, which indicates whether the disk block is free or has been allocated to a file. This information can be maintained in a single bit, and so a DSM is also called a bit map. Figure 13.12 illustrates a DSM. A 1 in an entry indicates that the corresponding disk block is allocated. The DSM is consulted every time a new disk block has to be allocated to a file. To avoid dispersing file data to different parts of a disk, file systems confine the disk space allocation for a file either to consecutive disk blocks, which form an extent, also called a cluster, or consecutive cylinders in a disk, which form cylinder groups.

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 noncontiguous disk space allocation. They differ in the manner they maintain information about disk space allocated to a file.

**INTERFACE BETWEEN FILE SYSTEM AND IOCS**

The file system uses the IOCS to perform I/O operations and the IOCS implements them through 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 6.3. 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 6.15 shows the arrangement set up when a file alpha is opened. The file system copies fmtalpha in memory; creates fcbalpha, 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_idalpha.

Table: **Fields in the File Control Block (FCB)**
Figure: Interface between file system and IOCS—OFT, FCB and FMT.
The file system supports the following operations:

- **open** (<file_name>, <processing_mode>, <file_attributes>)
- **close** (<internal_id_of_file>)
- **read/write** (<internal_id_of_file>, <record_info>, <I/O_area_addr>)

<file_name> is an absolute or relative path name of the file to be opened. <processing_mode> indicates what kind of operations will be performed on the file—the values —input, —create, —append of it
have obvious meanings, while —update‖ indicates that the process intends to update existing data in place.

<file_attributes> is a list of file attributes, such as the file's organization, record size, and protection information. It is relevant only when a new file is being created—attributes from the list are copied into the directory entry of the file at this time. <record_info> indicates the identity of the record to be read or written if the file is being processed in a nonsequential mode. <I/O_area_addr> indicates the address of the memory area where data from the record should be read, or the memory area that contains the data to be written into the record.

The IOCS interface supports the following operations:
- iocs-open (<internal_id_of_file>, <directory_entry_address>)
- iocs-close (<internal_id_of_file>, <directory_entry_address>)
- iocs-read/write (<internal_id_of_file>, <record_info>, <I/O_area_addr>)
  - Each of these operations is a generic operation for the various file organizations supported by the file system. It works in two parts: It performs some actions that are common to all file organizations, and invokes a module of the access method mentioned in the FCB of the file for performing special actions required for specific file organizations.
  - The iocs-open and iocs-close operations are specialized read and write operations that copy information into the FCB from the directory entry or from the FCB into the directory entry. The iocs-read/write operations access the FCB to obtain information concerning the current state of the file processing activity, such as the address of the next record to be processed. When a write operation requires more disk space, iocs-write invokes a function of the file system to perform disk space allocation.

**Unix File System**

The design of the Unix file system is greatly influenced by the MULTICS file system. In this section we describe important features common to most versions of Unix, in the context of the generic description of file processing.

**Inodes, File Descriptors, and File Structures** The information that constituted the directory entry of a file in Figure below is split in Unix between the directory entry and the inode of the file. The directory entry contains only the file name and the inode number; the bulk of the information concerning a file is contained in its inode. Files are considered to be streams of characters and are accessed sequentially. The system administrator can specify a disk quota for each user. It prevents a user from occupying too much disk space. The inode data structure is maintained on disk. Some of its fields contain the following information:
- File type, e.g., whether directory, link, or special file
- Number of links to the file
- File size
- Id of the device on which the file is stored
- Inode serial number
- User and group ids of the owner
- Access permissions
• Allocation information

The splitting of the conventional directory entry into the directory entry and the inode facilitates creation and deletion of links. A file can be deleted when its number of links drops to zero. Note the similarity between fields of the inode and those of the FCB (see Table 6.3).

Figure 6.17 illustrates the arrangement in memory during the processing of a file. It consists of inodes, file structures, and file descriptors. A file structure contains two fields—the current position in an open file, which is in the form of an offset from the start of the file; and a pointer to the inode for the file. Thus an inode and a file structure together contain all the information necessary to access the file. A file descriptor points to a file structure. File descriptors are stored in a per-process table. This table resembles the open files table (OFT).

When a process opens a file alpha, the directory entry for alpha is located. A directory lookup cache is employed to speed up this operation. Once the entry of alpha is located, its inode is copied into memory, unless memory already contains such a copy. The arrangement shown in Figure is now set up and the index of the file descriptor in the file descriptors table, which is an integer, is passed back to the process that opened the file.

![Diagram of Unix file system data structures](image)

The process can use it in a manner that resembles use of the internal id of a file in the generic arrangement. When a process creates a child process, a table of descriptors is created for the child process, and the file descriptors of the parent process are copied into it. Thus more than one file descriptor may point to the same file structure. Processes owning these file descriptors share the offset into the file. A read or write by one process will modify the offset for the other processes as well.

**File Sharing Semantics** Several processes may independently open the same file. In that case, the arrangement of Figure is set up for each process. Thus, two or more file structures may point to the same inode. Processes using these file structures have their own offsets into the file, so a read or write by one process does not modify the offset used by other processes.

Unix provides single-imagemutable file semantics for concurrent file sharing. As shown in Figure, every process that opens a file points to the copy of its inode through the file descriptor and file structure. Thus, all processes sharing a file use the same copy of the file; changes made by one process are immediately visible to other processes sharing the file. Implementation of
these semantics is aided by the fact that Unix uses a disk cache called *buffer cache* rather than buffers for individual file processing activities.

To avoid race conditions while the inode of a shared file is accessed, a lock field is provided in the memory copy of an inode. A process trying to access an inode must sleep if the lock is set by some other process. Processes concurrently using a file must make their own arrangements to avoid race conditions on data contained in the file.

**Disk Space Allocation** Unix uses indexed disk space allocation, with a disk block size of 4 KB. Each file has a *file allocation table* analogous to an FMT, which is maintained in its inode. The allocation table contains 15 entries.

Twelve of these entries directly point to data blocks of the file. The next entry in the allocation table points to an indirect block, i.e., a block that itself contains pointers to data blocks. The next two entries point to double and triple indirect blocks, respectively. In this manner, the total file size can be as large as 242 bytes.

![Unix file allocation table](image)

However, the file size information is stored in a 32-bit word of the inode. Hence file size is limited to $2^{32} - 1$ bytes, for which the direct, single, and double indirect blocks of the allocation table are adequate.

For file sizes smaller than 48 KB, this arrangement is as efficient as the flat FMT arrangement discussed in Section 13.7. Such files also have a small allocation table that can fit into the inode itself. The indirect blocks permit files to grow to large sizes, although their access involves traversing the indirection in the file allocation table. A survey of Unix file sizes conducted in 1996 reported that the average file size in Unix was 22 KB, and over 93 percent of files had sizes smaller than 32 KB. Thus the Unix file allocation table is as efficient as the flat FMT for most files.
Unix maintains a free list of disk blocks. Each entry in the list is similar to an indirect block in an FMT—it contains addresses of free disk blocks, and the id of the next disk block in the free list. This arrangement minimizes the overhead of adding disk blocks to the free list when a file is deleted; only marginal processing is required for files that contain only direct and single indirect blocks. A lock field is associated with the free list to avoid race conditions when disk blocks are added and deleted from it. A file system program named mkfs is used to form the free list when a new file system is created. mkfs lists the free blocks in ascending order by block number while forming the free list. However, this ordering is lost as disk blocks are added to and deleted from the free list during file system operation.

The file system makes no effort to restore this order. Thus blocks allocated to a file may be dispersed throughout a disk, which reduces the access efficiency of a file. BSD Unix uses cylinder groups to address this issue.

Multiple File Systems The root of a file system is called the superblock. It contains the size of the file system, the free list, and the size of the inode list. In the interest of efficiency, Unix maintains the superblock in memory but copies it onto the disk periodically. This arrangement implies that some part of file system state is lost in the event of a system crash. The file system can reconstruct some of this information, e.g., the free list, by analyzing the disk status. This is done as a part of the system booting procedure.

There can be many file systems in a Unix system. Each file system has to be kept on a single logical disk device; hence files cannot span different logical disks. A physical disk can be partitioned into many logical disks and a file system can be constructed on each of them. Such partitioning provides some protection across file systems, and also prevents a file system from occupying too much disk space. A file system has to be mounted before being accessed. Only a user with the root password, typically a system administrator, can mount a file system.

Mounting and unmounting of file systems works as follows: A logical disk containing a file system is given a device special file name. This name is indicated as FS_name in a mount command. When a file system is mounted, the superblock of the mounted file system is loaded in memory. Disk block allocation for a file in the mounted file system is performed within the logical disk device of the mounted file system. Files in a mounted file system are Accessed.

Questions

1. Describe the different operations performed on files.
2. Explain the UNIX file system.
3. Explain the file system and IOCS in detail.
4. Discuss the methods of allocation of disk space with block representation.
5. Explain briefly the file control block.
6. Explain the index sequential file organization with an example.
7. What is a link? With an example, illustrate the use of a link in an acyclic graph structure directory.
8. Compare sequential and direct file organization.
9. Describe the interface between file system and IOCS.
10. Explain the file system actions when a file is opened and a file is closed.
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 a modern operating system must provide the best combination of user service and system performance to suit its computing environment.

A scheduling policy employs three fundamental techniques to achieve the desired combination of user service and system performance.

Assignment of priorities to processes can provide good system performance, as in a multiprogramming system; or provide favoured treatment to important functions, as in a real-time system.

Variation of time slice permits the scheduler to adapt the time slice to the nature of a process so that it can provide an appropriate response time to the process, and also control its own overhead.

Reordering of processes can improve both system performance, measured as throughput, and user service, measured as turnaround times or response times of processes. We discuss the use of these techniques and a set of scheduling heuristics in modern operating systems.

Nonpreemptive Scheduling Policies

In nonpreemptive 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 preemption of a request as shown in Figure 7.1 never occurs. Nonpreemptive scheduling is attractive because of its simplicity—the scheduler does not have to distinguish between an unserviced request and a partially serviced one. Since a request is never preempted, the scheduler's only function in improving user service or system performance is reordering of requests. The three nonpreemptive scheduling policies are:

- First-come, first-served (FCFS) scheduling
- Shortest request next (SRN) scheduling
- Highest response ratio next (HRN) scheduling

For simplicity we assume that these processes do not perform I/O operations.

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.

<table>
<thead>
<tr>
<th>Process</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admission time</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Service time</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table: Processes for Scheduling

Example: FCFS Scheduling

Figure below illustrates the scheduling decisions made by the FCFS scheduling policy for the processes of Table. Process $P_1$ is scheduled at time 0. The pending list contains $P_2$ and $P_3$ when $P_1$ completes at 3 seconds, so $P_2$ 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 7.2 shows how the processes operated.
<table>
<thead>
<tr>
<th>Time</th>
<th>Completed process</th>
<th>Processes in system (in FCFS order)</th>
<th>Scheduled process</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>3</td>
<td>$P_1$</td>
<td>$P_2, P_3$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>6</td>
<td>$P_2$</td>
<td>$P_3, P_4$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>11</td>
<td>$P_3$</td>
<td>$P_4, P_5$</td>
<td>$P_4$</td>
</tr>
<tr>
<td>13</td>
<td>$P_4$</td>
<td>$P_5$</td>
<td>$P_5$</td>
</tr>
<tr>
<td>16</td>
<td>$P_5$</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

$\overline{ta} = 6.40$ seconds

$\overline{w} = 2.22$

Figure: 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.

Example Shortest Request Next (SRN) Scheduling

Figure illustrates the scheduling decisions made by the SRN scheduling policy for the processes of Table, and the operation of the processes. At time 0, $P_1$ is the only process in the system, so it is scheduled. It completes at time 3 seconds. At this time, processes $P_2$ and $P_3$ exist in the system, and $P_2$ is shorter than $P_3$. So $P_2$ 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.

<table>
<thead>
<tr>
<th>Time</th>
<th>Completed process</th>
<th>Processes in system</th>
<th>Scheduled process</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>{P_1}</td>
<td>P_1</td>
</tr>
<tr>
<td>3</td>
<td>P_1 3</td>
<td>{P_2, P_3}</td>
<td>P_2</td>
</tr>
<tr>
<td>6</td>
<td>P_2 4</td>
<td>{P_3, P_4}</td>
<td>P_4</td>
</tr>
<tr>
<td>8</td>
<td>P_4 4</td>
<td>{P_3, P_5}</td>
<td>P_5</td>
</tr>
<tr>
<td>11</td>
<td>P_5 3</td>
<td>{P_3}</td>
<td>P_3</td>
</tr>
<tr>
<td>16</td>
<td>P_3 13</td>
<td>{}</td>
<td>-</td>
</tr>
</tbody>
</table>

$t\bar{a} = 5.40$ seconds
$\bar{w} = 1.59$

Figure: Scheduling using the shortest request next (SRN) policy

Use of the SRN policy faces several difficulties in practice. Service times of processes are not
known to the operating system a priori, hence the OS may expect users to provide estimates of service times of processes. However, scheduling performance would be erratic if users do not possess sufficient experience in estimating service times, or they manipulate the system to obtain better service by giving low service time estimates for their processes. The SRN policy offers poor service to long processes, because a steady stream of short processes arriving in the system can starve a long process.

**Highest Response Ratio Next (HRN) Scheduling**

The HRN policy computes the response ratios of all processes in the system according to Eq. and selects the process with the highest response ratio.

\[
\text{Response ratio} = \frac{\text{time since arrival} + \text{service time of the process}}{\text{service time of the process}}
\]

The response ratio of a newly arrived process is 1. It keeps increasing at the rate \(\frac{1}{\text{service time}}\) as it waits to be serviced. The response ratio of a short process increases more rapidly than that of a long process, so shorter processes are favored for scheduling. However, the response ratio of a long process eventually becomes large enough for the process to get scheduled. This feature provides an effect similar to the technique of *aging*, so long processes do not starve. The next example illustrates this property.

**Example : Highest Response Ratio Next (HRN) Scheduling**

Operation of the HRN scheduling policy for the five processes shown in Table is summarized in Figure. By the time process \(P_1\) completes, processes \(P_2\) and \(P_3\) have arrived. \(P_2\) has a higher response ratio than \(P_3\), so it is scheduled next. When it completes, \(P_3\) has a higher response ratio than before; however, \(P_4\), which arrived after \(P_3\), has an even higher response ratio because it is a shorter process, so \(P_4\) is scheduled. When \(P_4\) completes, \(P_3\) has a higher response ratio than the shorter process \(P_5\) because it has spent a lot of time waiting, whereas \(P_5\) has just arrived. Hence \(P_3\) is scheduled now. This action results in a smaller weighted turnaround for \(P_3\) than in SRN scheduling (see Figure 7.3). Thus, after a long wait, a long process gets scheduled ahead of a shorter one.

**Figure : 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-preemptive 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.
**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 $\delta$, 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.

The RR policy 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: 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 pre-empted will be entered into the queue before the pre-empted process.

Figure below summarizes operation of the RR scheduler with $\delta = 1$ second for the five processes shown in Table. 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. 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 $P_2$ followed by $P_1$. Hence $P_2$ is scheduled. Process $P_3$ arrives at 3 seconds, and is entered in the queue. $P_2$ is also pre-empted at 3 seconds and it is entered in the queue. Hence the queue has process $P_1$ followed by $P_3$ and $P_2$, so $P_1$ is scheduled.

### Table

<table>
<thead>
<tr>
<th>Time of scheduling</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position of Processes in ready queue (1 implies head of queue)</td>
<td>$P_1$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process scheduled</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
<td>$P_2$</td>
<td>$P_4$</td>
<td>$P_3$</td>
<td>$P_2$</td>
<td>$P_4$</td>
<td>$P_5$</td>
<td>$P_3$</td>
<td>$P_5$</td>
<td>$P_3$</td>
<td>$P_5$</td>
<td>$P_3$</td>
<td></td>
</tr>
</tbody>
</table>

$\overline{t_a} = 7.4$ seconds, $\overline{w} = 2.32$

c: completion time of a process
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  Least Completed Next (LCN) Scheduling
Implementation of the LCN scheduling policy for the five processes shown in Table is summarized in Figure. 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.

$$\bar{t_a} = 8.8 \text{ seconds, } \bar{w} = 2.72$$
$$c: \text{ completion time of a process}$$
It can be seen that servicing of $P_1$, $P_2$, and $P_3$ 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) 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 $P_5$ over $P_1$, $P_2$, and $P_3$.

**Shortest Time to Go (STG) Scheduling**

The shortest time to go policy schedules a process whose remaining CPU time requirements are the smallest in the system. It is a preemptive version of the shortest request next (SRN) policy. So it favours short processes over long ones and provides good throughput. Additionally, the STG policy also favours a process that is nearing completion over short processes entering the system. This feature helps to improve the turnaround times and weighted turnarounds of processes. Since it is analogous to the SRN policy, long processes might face starvation.
Example : Shortest Time to Go (STG) Scheduling

Figure above summarizes performance of the STG scheduling policy for the five processes shown in Table above. The scheduling information used by the policy is the CPU time needed by each process for completion. In case of a tie, the scheduler selects whatever process has not been serviced for the longest period of time. Execution of \( P_3 \) is delayed because \( P_2, P_4, \) and \( P_5 \) require lesser CPU time than it.

**Scheduling in practice**

**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.

Scheduling in UNIX UNIX is a pure time-sharing operating system. It uses a multilevel adaptive scheduling policy in which process priorities are varied to ensure good system performance and also to provide good user service. Processes are allocated numerical priorities, where a larger numerical value implies a lower effective priority.

In Unix 4.3 BSD, the priorities are in the range 0 to 127. Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49. When a process is blocked in a system call, its priority is changed to a value in the range 0–49, depending on the cause of blocking.

When it becomes active again, it executes the remainder of the system call with this priority. This arrangement ensures that the process would be scheduled as soon as possible, complete the task it was performing in the kernel mode and release kernel resources. When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50–127.

Unix uses the following formula to vary the priority of a process:

\[
\text{Process priority} = \text{base priority for user processes} + f(\text{CPU time used recently}) + \text{nice value} \quad (7.5)
\]

It is implemented as follows: The scheduler maintains the CPU time used by a process in its process table entry. This field is initialized to 0. The real-time clock raises an interrupt 60 times a second, and the clock handler increments the count in the CPU usage field of the running process. The scheduler recomputes process priorities every second in a loop. For each process, it divides the value in the CPU usage field by 2, stores it back, and also uses it as the value of \( f \). Recall that a large numerical value implies a lower effective priority, so the second factor in Eq. (7.5) lowers the priority of a process. The division by 2 ensures that the effect of CPU time used by a process decays; i.e., it wears off over a period of time, to avoid the problem of starvation.
A process can vary its own priority through the last factor in Eq. (7.5). The system call
\texttt{nice(<priority value>);} sets the \textit{nice value} of a user process. It takes a zero or positive value as its argument. Thus, a process can only decrease its effective priority to be nice to other processes. It would typically do this when it enters a CPU-bound phase.

**Recommended Questions**

1. With a neat block diagram, explain event handling and scheduling.
2. Explain scheduling in UNIX.
3. Summarize the main approaches to real time scheduling.
4. What is scheduling? What are the events related to scheduling?
5. Explain briefly the mechanism and policy modules of short term process scheduler with a neat block diagram.
6. Briefly explain the features of time sharing system. Also explain process state transitions in time sharing system.
7. Compare i) pre-emptive and non-preemptive scheduling
   ii) long term and short term schedulers
8. Describe SRN and HRN scheduling policies and determine the average turn around time and weighted turn-around time for the following set of processes-

<table>
<thead>
<tr>
<th>Processes</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival time</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Service time</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

9. What are the functions of medium and short term schedulers?
UNIT-8

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.

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 receive (Pi, msg_area), where msg_area is an area in Pj ‘s address space, results in a system call receive.

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 8.1 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 producers–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 variable approach is not feasible.

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 in different 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.

Direct and Indirect Naming
In direct naming, sender and receiver processes mention each other’s name. For example, the send and receive statements might have the following syntax:

send (<destination_process>,<message_length>,<message_address>);
receive (<source_process>,<message_area>);

where <destination_process> and <source_process> are process names (typically, they are
process ids assigned by the kernel), `<message_address>` is the address of the memory area in the sender process’s address space that contains the textual form of the message to be sent, and `<message_area>` is a memory area in the receiver’s address space where the message is to be delivered. The processes of Figure 8.2 used direct naming.

Direct naming can be used in two ways: In symmetric naming, both sender and receiver processes specify each other’s name. Thus, a process can decide 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 each other’s name in send and receive statements.

**Blocking and Nonblocking 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 nonblocking `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. The kernel can simply let the message remain in the sender's memory area until it is delivered. However, use of blocking `sends` has one drawback—it may unnecessarily delay a sender process in some situations, for example, while communicating with a heavily loaded print server.

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.

**IMPLEMENTING MESSAGE PASSING**

1 Buffering of Interprocess Messages

When a process `Pi` sends a message to some process `Pj` by using a nonblocking `send`, the kernel builds an interprocess message control block (IMCB) to store all information needed to deliver the message (see Figure ). 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 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.

MAILBOXES

A mailbox is a repository for interprocess 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 illustrates 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 \( P_i \) 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.

![Diagram of mailbox use](image)

**Figure**: Creation and use of mailbox sample.

This way it can destroy a mailbox if no process is connected to it. Alternatively, it may permit the owner of a mailbox to destroy it. In that case, it has the responsibility of informing all processes that have —connected— to the mailbox. The kernel may permit the owner of a mailbox to transfer the ownership to another process.

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 (see Example below).

**Example Use of Mailboxes**

An airline reservation system consists of a centralized database and a set of booking processes; each process represents one booking agent. Figure 8.8 shows a pseudocode for the reservation server. It uses three mailboxes named \( enquire \), \( book \), and \( cancel \), and expects a booking process to send enquiry, booking, and cancellation messages to these mailboxes, respectively. Values of flags in the \( receive \) calls are chosen such that a \( receive \) call returns with an error code if no
message exists. For improved effectiveness, the server processes all pending cancellation messages before processing a booking request or an enquiry, and performs bookings before enquiries.

Repeat
while receive (book, flags1, msg_area1) returns a message
    while receive (cancel, flags2, msg_area2) returns a message
        process the cancellation;
    process the booking;
if receive (enquire, flags3, msg_area3) returns a message then while
    receive (cancel, flags2, msg_area2) returns a message
        process the cancellation;
    process the enquiry;
forever

Figure : Airline reservation server using three mailboxes: enquire, book, and cancel.

HIGHER-LEVEL PROTOCOLS USING MESSAGE PASSING

In this section, we discuss three protocols that use the message passing paradigm to provide diverse services. The simple mail transfer protocol (SMTP) delivers electronic mail. The remote procedure call (RPC) is a programming language facility for distributed computing; it is used to invoke a part of a program that is located in a different computer. Parallel virtual machine (PVM) and message passing interface (MPI) are message passing standards for parallel programming.

The Simple Mail Transfer Protocol (SMTP)
SMTP is used to deliver electronic mail to one or more users reliably and efficiently.
It uses asymmetric naming. A mail would be delivered to a user's terminal if the user is currently active; otherwise, it would be deposited in the user's mailbox. The SMTP protocol can deliver mail across a number of interprocess communication environments (IPCEs), where an IPCE may cover a part of a network, a complete network, or several networks. SMTP is an applications layer protocol. It uses the TCP as a transport protocol and IP as a routing protocol.

SMTP consists of several simple commands. The relevant ones for our purposes are as follows: The MAIL command indicates who is sending a mail.
It contains a reverse path in the network, which is an optional list of hosts and the name of the sender mailbox. The RCPT command indicates who is to receive the mail. It contains a forward path that is an optional list of hosts and a destination mailbox. One or more RCPT commands can follow a MAIL command.

The DATA command contains the actual data to be sent to its destinations. After processing the DATA command, the sender host starts processing of the MAIL command to send the data to the destination(s). When a host accepts the data for relaying or for delivery to the destination mailbox, the protocol generates a timestamp that indicates when the data was delivered to the host and inserts it at the start of the data. When the data reaches the host containing the destination mailbox, a line containing the reverse path mentioned in the MAIL command is
inserted at the start of the data. The protocol provides other commands to deliver a mail to the user's terminal, to both the user's terminal and the user's mailbox, and either to the user's terminal or the user's mailbox. SMTP does not provide a mailbox facility in the receiver, hence it is typically used with either the Internet Message Access Protocol (IMAP) or the Post Office Protocol (POP); these protocols allow users to save messages in mailboxes.

**Remote Procedure Calls**

Parts of a *distributed program* are executed in different computers. The *remote procedure call* (RPC) is a programming language feature that is used to invoke such parts. Its semantics resemble those of a conventional procedure call. Its typical syntax is

\[
\text{call } \langle \text{proc}_\text{id} \rangle \ (\langle \text{message} \rangle);
\]

where \(\text{proc}_\text{id}\) is the id of a remote procedure and \(\text{message}\) is a list of parameters.

The call results in sending \(\text{message}\) to remote procedure \(\langle \text{proc}_\text{id} \rangle\). The result of the call is modeled as the reply returned by procedure \(\langle \text{proc}_\text{id} \rangle\). RPC is implemented by using a blocking protocol. We can view the caller–callee relationship as a client–server relationship. Thus, the remote procedure is the server and a process calling it is a client. We will call the computers where the client and the server processes operate as the *client node* and *server node*, respectively.

Parameters may be passed by value or by reference. If the architecture of the server node is different from that of the client node, the RPC mechanism performs appropriate conversion of value parameters. For reference parameters, the caller must construct system wide capabilities for the parameters. These capabilities would be transmitted to the remote procedure in the message. Type checks on parameters can be performed at compilation time if the caller and the callee are integrated during compilation; otherwise, type checks have to be performed dynamically when a remote procedure call is made.

The schematic diagram of Figure depicts the arrangement used to implement a remote procedure call. The server procedure is the remote procedure that is to be invoked. The client process calls the *client stub* procedure, which exists in the same node. The client stub marshals the parameters—collects the parameters, converts them into a machine-independent format, and prepares a message containing this representation of parameters. It now calls the *server stub*, which exists in the node that contains the remote procedure. The server stub converts the parameters into a machine-specific form and invokes the remote procedure.

Results of the procedure call are passed back to the client process through the server stub and the client stub.

![Remote Procedure Call Diagram](image)

**Figure**: Overview of a remote procedure call (RPC). Two standards for remote procedure calls—Sun RPC and OSF/DCE—have emerged and are in use widely. Their use simplifies making of RPCs, and makes programs using RPCs portable.
across computers and their operating systems.

These standards specify an external representation of data for passing parameters and results between the client and the server, and an interface compiler that handles the drudgery of marshaling of parameters.

**Message Passing Standards for Parallel Programming**

A *parallel program* consists of a set of tasks that can be performed in parallel. Such programs can be executed on a heterogeneous collection of computers or on a *massively parallel processor* (MPP). Parallel programs use message passing libraries that enable parallel activities to communicate through messages.

*Parallel virtual machine* (PVM) and *message passing interface* (MPI) are the two standards that are used in coding message passing libraries. Both standards provide the following facilities:

- Point-to-point communication between two processes, using both symmetric and asymmetric naming, and collective communication among processes, which includes an ability to broadcast a message to a collection of processes.

- *Barrier synchronization* between a collection of processes wherein a process invoking the barrier synchronization function is blocked until *all* processes in that collection of processes have invoked the barrier synchronization function.

- Global operations for scattering disjoint portions of data in a message to different processes, gathering data from different processes, and performing global reduction operations on the received data.

In the PVM standard, a collection of heterogeneous networked computers operates as a parallel virtual machine, which is a single large parallel computer. The individual systems can be workstations, multiprocessors, or vector supercomputers.

Hence message passing faces the issue of heterogeneous representation of data in different computers forming the parallel virtual machine. After a message is received, a sequence of calls can be made to library routines that unpack and convert the data to a suitable form for consumption by the receiving process.

PVM also provides signals that can be used to notify tasks of specific events. MPI is a standard for a massively parallel processor. It provides a nonblocking send, which is implemented as follows: The message to be sent, which is some data, is copied into a buffer, and the process issuing the send is permitted to continue its operation. However, the process must not reuse the buffer before the previous send on the buffer has been completed. To facilitate it, a *request handle* is associated with every nonblocking send, and library calls are provided for checking the completion of a send operation by testing its request handle and for blocking until a specific send operation, or one of many send operations, is completed.
Message Passing in Unix

Unix supports three interprocess communication facilities called pipes, message queues, and sockets. A pipe is a data transfer facility, while message queues and sockets are used for message passing. These facilities have one common feature—processes can communicate without knowing each other's identities. The three facilities are different in scope. Unnamed pipes can be used only by processes that belong to the same process tree, while named pipes can be used by other processes as well. Message queues can be used only by processes existing within the —Unix system domain,— which is the domain of Unix operating on one computer system. Sockets can be used by processes within the Unix system domain and within certain Internet domains. Figure illustrates the concepts of pipes, message queues, and sockets.

Pipes A pipe is a first-in, first-out (FIFO) mechanism for data transfer between processes called reader processes and writer processes. A pipe is implemented in the file system in many versions of Unix; however, it differs from a file in one important respect—the data put into a pipe can be read only once. It is removed from the pipe when it is read by a process. Unix provides two kinds of pipes, called named and unnamed pipes. Both kinds of pipes are created through the system call pipe. Their semantics are identical except for the following differences:

A named pipe has an entry in a directory and can thus be used by any process, subject to file permissions, through the system call open. It is retained in the system until it is removed by an unlink system call.

![Diagram](Diagram)

Figure: Interprocess communication in Unix: (a) pipe; (b) message queue; (c) socket.

An unnamed pipe does not have an entry in a directory; it can be used only by its creator and its descendants in the process tree. The kernel deletes an unnamed pipe when readers or writers no longer exist for it.

A pipe is implemented like a file, except for two differences. The size of a pipe is limited so that data in a pipe is located in the direct blocks of the inode. The kernel treats a pipe as a queue by maintaining two offsets—one offset is used for writing data into the pipe and the other for reading data from the pipe [see Figure 8.10(a)]. The read and write offsets are maintained in the inode instead of in the file structure. This arrangement forbids a process from changing the offset of a pipe through any means other than reading or writing of data. When data is written, it is entered into the pipe by using the write offset, and the write offset is incremented by the number of bytes written. Data written by multiple writers gets mixed up if their writes are interleaved. If a pipe is full, a process wishing to write data into it would be put to sleep. A read operation is performed by using the read offset, and the read offset is incremented by the
number of bytes read. A process reading data from a pipe would be put to sleep if the pipe is empty.

**Message Queues** A message queue in Unix is analogous to a mailbox. It is created and owned by one process. Other processes can send or receive messages to or from a queue in accordance with access permissions specified by the creator of the message queue [see Figure]. These permissions are specified by using the same conventions as file permissions in Unix (see Section 15.6.3). The size of a message queue, in terms of the number of bytes that it can buffer, is specified at the time of its creation.

A message queue is created by a system call `msgget (key, flag)` where `key` specifies the name of the message queue and `flag` indicates some options. The kernel maintains an array of message queues and their keys. The position of a message queue in this array is used as the message queue id; it is returned by the `msgget` call, and the process issuing the call uses it for sending or receiving messages. The naming issue is tackled as follows: If a process makes a `msgget` call with a key that matches the name of an existing message queue, the kernel simply returns its message queue id. This way, a message queue can be used by any process in the system. If the key in a `msgget` call does not match the name of an existing message queue, the kernel creates a new message queue, sets the key as its name, and returns its message queue id. The process making the call becomes the owner of the message queue.

Each message consists of a message type, in the form of an integer, and a message text. The kernel copies each message into a buffer and builds a message header for it indicating the size of the message, its type, and a pointer to the memory area where the message text is stored. It also maintains a list of message headers for each message queue to represent messages that were sent to the message queue but have not yet been received.

Messages are sent and received by using following system calls: `msgsnd (msgqid, msg_struct_ptr, count, flag)` `msgrcv (msgqid, msg_struct_ptr, maxcount, type, flag)`

The `count` and `flag` parameters of a `msgsnd` call specify the number of bytes in a message and the actions to be taken if sufficient space is not available in the message queue, e.g., whether to block the sender, or return with an error code. `msg_struct_ptr` is the address of a structure that contains the type of a message, which is an integer, and the text of the message; `maxcount` is the maximum length of the message; and `type` indicates the type of the message to be received.

When a process makes a `msgrcv` call, the type parameter, which is an integer, indicates the type of message it wishes to receive. When the type parameter has a positive value, the call returns the first message in the queue with a matching type. If the type value is negative, it returns the lowest numbered message whose type is smaller than the absolute value of the type. If the type value is zero, it returns with the first message in the message queue, irrespective of its type. The process becomes blocked if the message queue does not contain any message that can be delivered to it.

When a process makes a `msgsnd` call, it becomes blocked if the message queue does not contain sufficient free space to accommodate the message. The kernel activates it when some process receives a message from the message queue, and the process repeats the check to find whether its message can be accommodated in the message queue. If the check fails, the process becomes
blocked once again. When it eventually inserts its message into the message queue, the kernel
activates all processes blocked on a receive on the message queue. When scheduled, each of
these processes checks whether a message of the type desired by it is available in the message
queue. If the check fails, it becomes blocked once again.

**Sockets** A socket is simply one end of a communication path. Sockets can be used for
interprocess communication within the Unix system domain and in the Internet domain; we limit
this discussion to the Unix system domain. A communication path between a client and the
server is set up as follows: The client and server processes create a socket each. These
two sockets are then connected together to set up a communication path for sending and receiving
messages [see Figure]. The server can set up communication paths with many clients
simultaneously.

The naming issue is tackled as follows: The server binds its socket to an address that is valid in
the domain in which the socket will be used. The address is now widely advertised in the
domain. A client process uses the address to perform a `connect` between its socket and that of
the server. This method avoids the use of process ids in communication; it is an instance of
indirect naming.

A server creates a socket \( s \) using the system call \( s = \text{socket} (\text{domain}, \text{type}, \text{protocol}) \) where \text{type}
and \text{protocol} are irrelevant in the Unix system domain. The \text{socket} call returns a socket identifier
to the process. The server process now makes a call \( \text{bind} (s, addr, \ldots) \), where \( s \) is the socket
identifier returned by the \text{socket} call and \text{addr} is the address for the socket. This call binds the
socket to the address \text{addr}; \text{addr} now becomes the ‘name’ of the socket, which is widely
advertised in the domain for use by clients. The server performs the system call \text{listen} \( (s, \ldots) \) to
indicate that it is interested in considering some connect calls to its socket \( s \).

A client creates a socket by means of a \text{socket} call, e.g., \( cs = \text{socket} (\ldots) \), and attempts to
connect it to a server’s socket using the system call \text{connect} \( (cs, \text{server_socket_addr},
\text{server_socket_addrlen}) \)

The server is activated when a client tries to connect to its socket. It now makes the call \text{new_soc} = \text{accept} \( (s, \text{client_addr}, \text{client_addrlen}) \). The kernel creates a new socket, connects it to the
socket mentioned in a client’s \text{connect} call, and returns the id of this new socket. The server uses
this socket to implement the client–server communication. The socket mentioned by the server
in its \text{listen} call is used merely to set up connections. Typically, after the \text{connect} call the server
forks a new process to handle the new connection. This method leaves the original socket
created by the server process free to accept more connections through \text{listen} and \text{connect} calls.
Communication between a client and a server is implemented through \text{read} and \text{write} or \text{send}
and \text{receive} calls. A \text{send} call has the format \text{count} = \text{send} \( (s, \text{message}, \text{message_length}, \text{flags}) \) It
returns the count of bytes actually sent. A socket connection is closed by using the call \text{close} \( (s) \)
or \text{shutdown} \( (s, \text{mode}) \).

**Questions**
1. Write short notes on i) Buffering of interprocess messages
   ii) Processes and threads
   iii) Process control block
2. Explain interprocess communication in UNIX in detail.
3. What is a mailbox? With an example, explain its features and advantages.
4. Explain implementation of message passing in detail.
5. Explain i) direct and indirect naming ii) blocking and non-blocking sends.
6. Explain the primitives used for the transmission and reception of messages in an OS.
7. Describe message delivery protocols and the exceptional conditions during message delivery with an example.