

UNIT 3. PROCESS MANAGEMENT

3.1 PROCESS

A process can be defined in several ways:

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

Two essential elements of a process are:

- **program code:** which may be shared with other processes that are executing the same program
- **Set of data:** associated with that code.

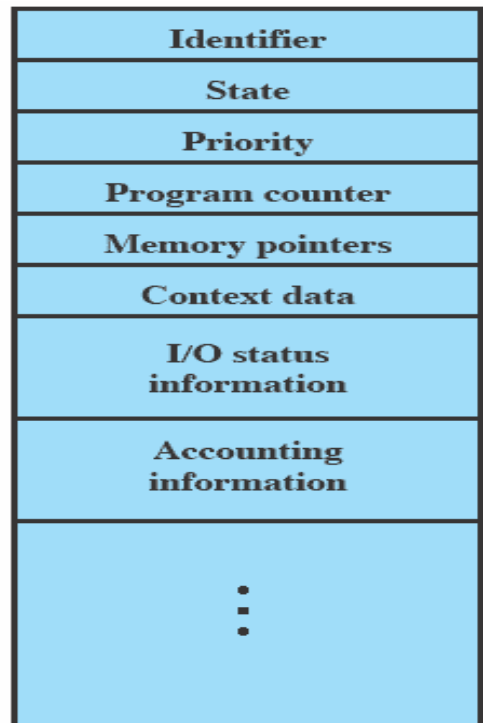


Figure 3.1 Process control block

At any given point in time, *while the program is executing*, this process can be uniquely characterized by a number of elements, including the following:

- **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.

- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

The above information is stored in a data structure known as **process control block** as shown in Figure 3.1. It is created and managed by OS.

3.2 PROCESS STATES

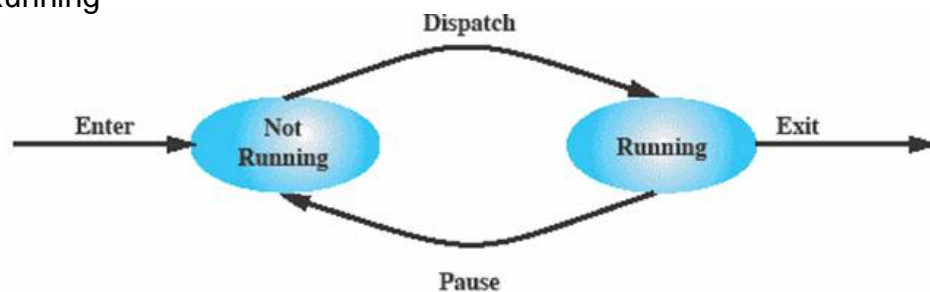
Execution of individual program involves a sequence of instructions within that program. This list of instructions is known as **trace** of the process. These instructions characterize the behavior of an individual process. **Dispatcher** is a small program which switches the processor from one process to another.

Major responsibility of an OS is to control the execution of processes. This task involves determining the inner pattern for execution and allocating resources to processes. Here we will consider two models of behavior of processes. These are known as process states.

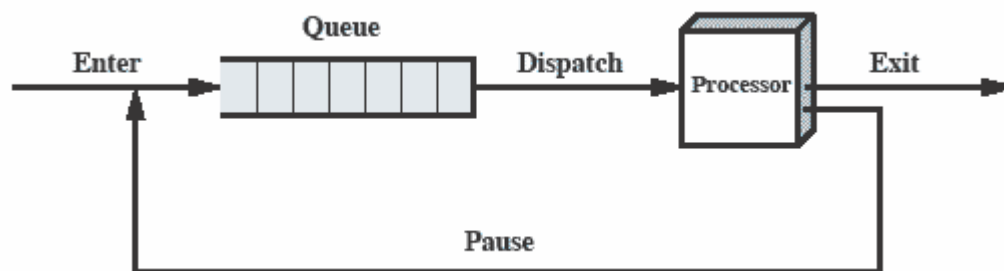
3.2.1 A Two – State Process Model

This is a simplest model and assumes that any process can be in any two states at any given point of time:

- Running
- Not Running



(a) State Transition Diagram



(b) Queuing Diagram

Figure 3.2 Two – State Process Model

When a job (or process) enters a job pool, it will be in a *not running* state. Once it is ready to execute and gets a processor, it will be into *running* state. When it has to wait for any resources or for I/O, it will be paused and sent into *not running* state again. During this time, another process may start *running*. Once the previous process is ready again, it will be switched from *not running* to *running*. This task is shown in Figure 3.2(a).

In this model, there must be some information relating to each process like current state, location in the memory etc. This will help OS to keep track of the process and is known as process control block. Processes that are not running must be kept in a queue, waiting for their turn to execute as shown in Figure 3.2(b). In this single queue, each entry is a pointer to the process control block of a particular process. In other words, each block represents one process here.

3.2.2 The Creation and Termination of Processes

The life of a process is bounded by its creation and termination. Hence, here we will discuss these aspects.

Process Creation: When a new process has to be added to the job pool, the OS builds the data structures for managing this process and allocates address space in main memory. These actions form the creation of a new process. The situations that will lead to the process creation are listed in Table 3.1.

Table 3.1 Reasons for Process Creation

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

Table 3.2 Reasons for Process Termination

- Normal Completion
- Bounds violation
- Time overrun
- Privileged instruction
- Parent termination
- Time Limit exceeded
- Protection Error
- I/O failure
- Data misuse
- Parent request
- Memory unavailable
- Arithmetic error
- Invalid instruction
- Operator/OS intervention

Most of the times, the OS creates the processes depending on various situations. But, sometimes, it creates a process based on the explicit request of another process. This is called as **process spawning**. When one process spawns another, the first process is called as **parent process** and the spawned process is called as **child process**.

Process Termination: Every process that is created will be terminated at some moment of time. There are several reasons for a process to get terminated as listed in Table 3.2.

3.2.3 A Five-State Model

If all the processes are always ready to execute, then the queue system showed in Figure 3.2(b) will be effective. This queue may work on first-in-first-out basis or round-robin basis. But, we cannot say that all the processes in *not running* state are ready to run. Because some of them may be waiting for some I/O, blocked by other process etc. So, the dispatcher needs to look for a process which is really ready to execute.

To avoid this problem, it is better to split *not running* state into two parts viz. *ready* and *blocked*. This model, known as five-state model is shown in Figure 3.3. The five states of this model are explained below:

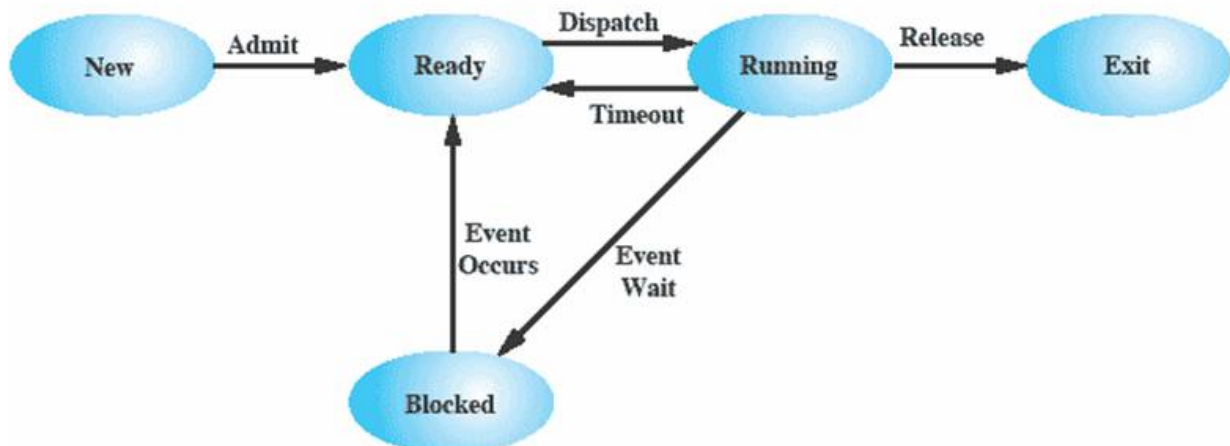
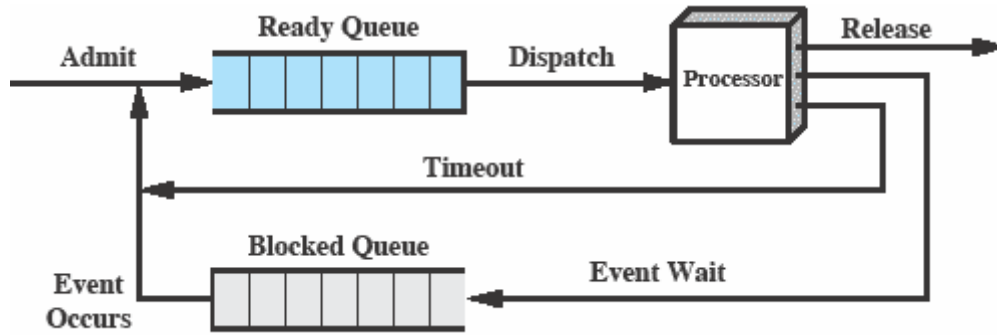
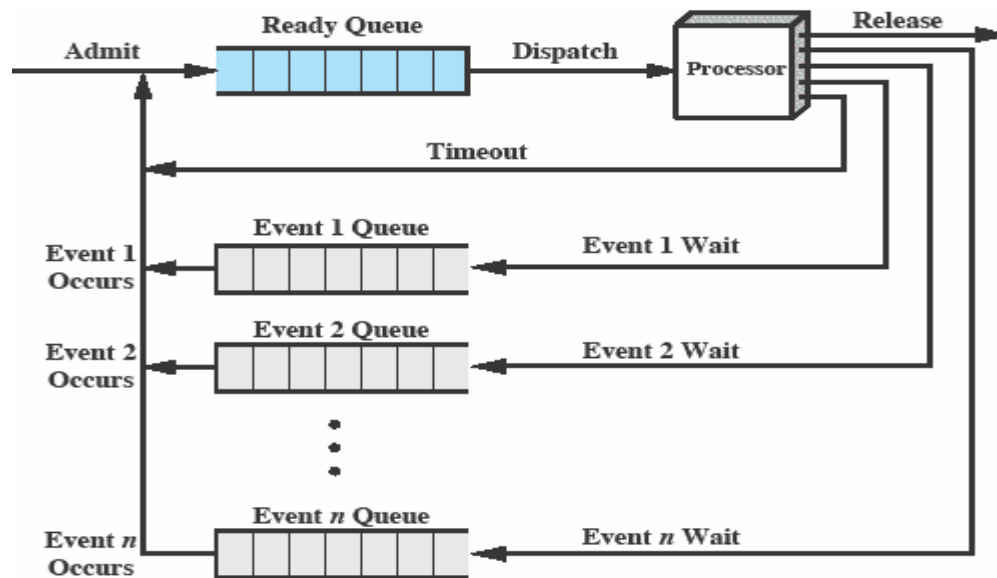


Figure 3.3 Five-state Process Model

- **Running:** The process that is currently being executed.
- **Ready:** A process that is ready to execute when given an opportunity.
- **Blocked/Waiting:** A process that cannot execute until some event occurs like I/O operation.
- **New:** A process that has been created just now and not yet been put into job pool by the OS. That is, the process which has not yet been loaded into main memory.
- **Exit:** A process that has been released from the job pool by OS, either because of successful termination or due to abnormal termination.



(a) Single Blocked Queue

(b) Multiple Blocked queues
Figure 3.4 Queuing model

Any process has to undergo transition between these states. The possible transitions are listed below:

- **Null to New:** A new process is created for execution. This event may occur due to any reasons listed in Table 3.1.
- **New to Ready:** The OS will move a process from New state to Ready when it is ready to take additional process. Generally, there will be a limit for a number of processes that can be kept on a queue so as to maintain the efficiency.
- **Ready to Running:** A scheduler or dispatcher selects one ready process for execution.
- **Running to Exit:** Currently running process is terminated by OS. (Refer Table 3.2 for reasons for termination).
- **Running to Ready:** Most of the OS have a limit for a time duration for which a process can run continuously. After that, even if it is not completed, it has to leave the processor for other waiting processes and has to enter a job queue.

- **Running to Blocked:** If a process requests for some resources, then it will be kept in a blocked state.
- **Blocked to Ready:** Once a process gets the requested resources and completes the task using those resources, it can enter into ready state.
- **Ready to Exit:** Though this state is not shown in the diagram, it may occur. Sometimes, a child process may be in ready state, and its parent may exit. Then, the child process also will be exited. Or, the parent itself may terminate the child.
- **Blocked to Exit:** A process may not get the resources that it requested or it exceeds the waiting time.

Figure 3.4 shows the queuing discipline maintained by processes with respect to the states mentioned in Figure 3.3.

3.2.4 Suspended Processes

In the whole course of process execution, OS may suspend a process sometimes. The reason is as explained below.

Assume an OS without virtual memory. Then all the processes to be executed must be loaded into main memory. Most of the processes will require I/O and hence may get blocked. As, I/O speed is much slower than that of processor, the processor would stay idle again. Hence, when none of the processes in the main memory is in Ready state, the OS can swap (move) one of the blocked processes into the disk. This will form a suspend queue. Now, any other process can be loaded into the main memory and start execution. As disk I/O is faster, the swapping will increase the performance. With the usage of swapping, one more state will be added to the process behavior model as shown in Figure 3.5 (a).

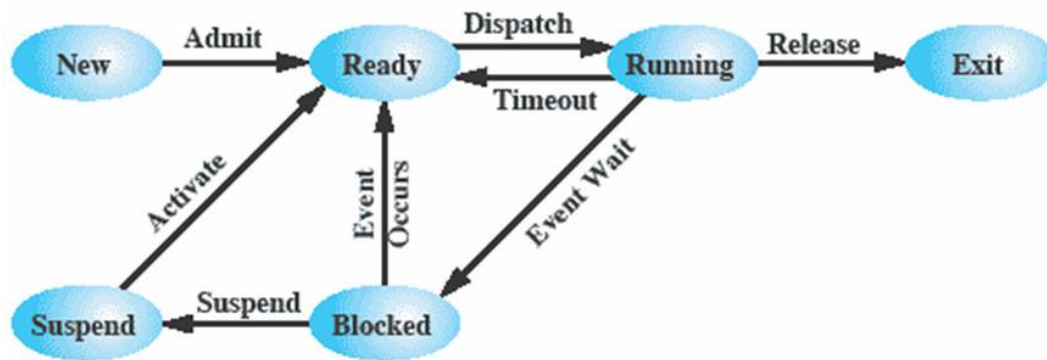
When OS performs a swapping-out operation, it can load either a newly created process into main memory or it can load a previously suspended process. But, a suspended process might have been actually a blocked process (waiting for I/O). Hence, the OS has to determine: Whether a process in suspended queue is

- a process waiting on an event – that is, blocked or not
- a process has been swapped out of main memory – that is suspended or not.

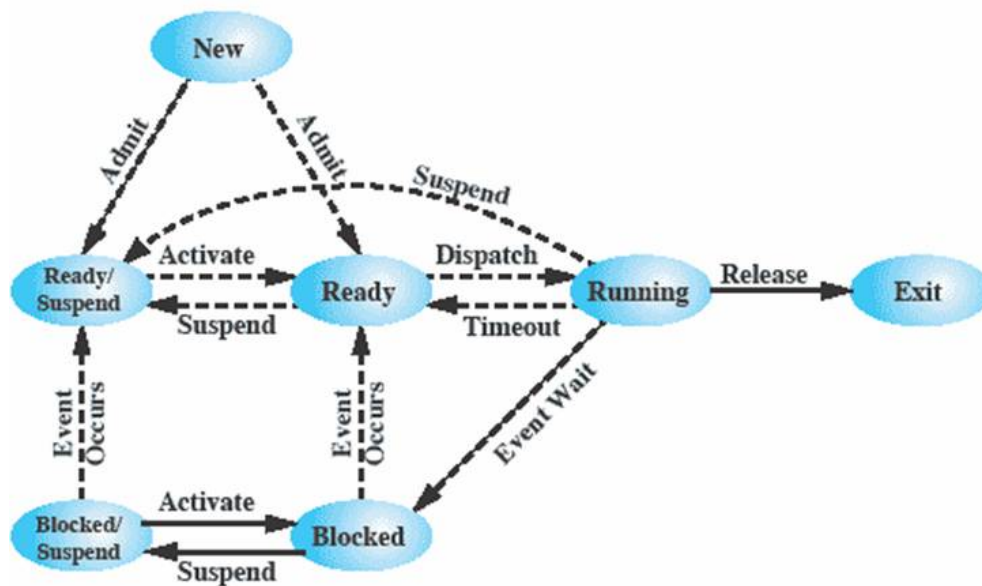
Keeping above two conditions, we will have four states:

- **Ready:** The process is in main memory and ready for execution
- **Blocked:** The process is in main memory and waiting for an event
- **Blocked/Suspend:** The process is on disk (secondary memory) and waiting for an event
- **Ready/Suspend:** The process is on disc, but ready for execution as soon as it gets loaded into main memory

These states can be included in the state diagram as shown in Figure 3.5(b).



(a) With one suspend state



(b) With two suspend states

Figure 3.5 Process State Transition Diagram with Suspend States

3.2.5 Other Uses of Suspension

A suspended process has following characteristics:

- The process is not immediately available for execution
- The process may or may not be waiting for an event
- The process was kept in suspended queue by an agent –the process itself, or parent or OS.
- The process may not be removed from this state until the agent explicitly orders to remove.

Some of the reason for process suspension is listed in Table 3.3. It also describes the usages by suspending the process.

Table 3.3 Reasons for Process Suspension

Reason	Description
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	User may wish to suspend a process to debug or in connection with the use of a resource
Timing	A process may be executed periodically and may be suspended while waiting for the next time slot.
Parent Process Request	A parent process may wish to suspend execution of a child to examine or modify the suspended process, or to coordinate the activity of various children.

3.3 PROCESS DESCRIPTION

OS is an entity which manages the usage of system resources by the processes as shown in Figure 3. 6.

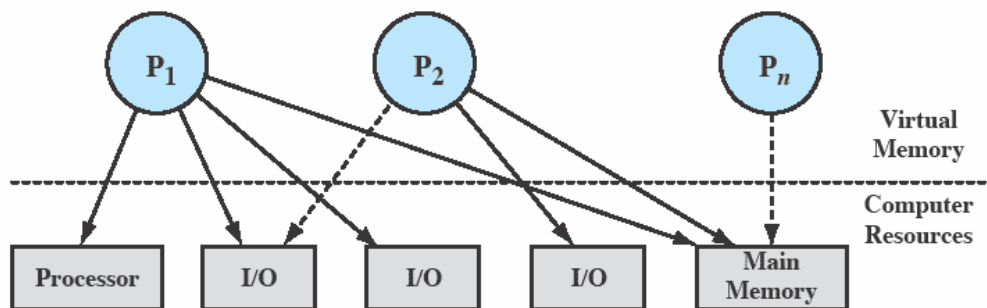


Figure 3.6 Processes and Resources (Resource allocation at one moment of time)

In a multiprogramming environment, there are a number of processes (P_1, \dots, P_n) that have been created and exist in virtual memory. Each process needs to access system resources during its execution. In Figure 3.6, process P_1 is running; at least part of the process is in main memory, and it has control of two I/O devices. Process P_2 is also in main memory but is blocked waiting for an I/O device allocated to P_1 . Process P_n has been swapped out and is therefore suspended.

3.3.1 OS Control Structures

To manage the processes, the OS should have the information about the current status of each process and resource. For this purpose OS constructs and maintains tables for each entity as shown in Figure 3.7. The four types of tables maintained by OS are explained here.

- **Memory Table:** Used to keep track of both main and secondary memory. They must include the following information:
 - Allocation of main memory to processes
 - Allocation of secondary memory to processes
 - Protection attributes for access to shared memory regions
 - Information needed to manage virtual memory

- **I/O Table:** Used by OS to manage the I/O devices and the channels. At any given moment of time, the OS needs to know
 - Whether the I/O device is available or assigned
 - The status of I/O operation
 - The location in main memory being used as the source or destination of the I/O transfer
- **File Table:** Most of the times, these tables are maintained by file management system. These tables provide information about:
 - Existence of files
 - Location on secondary memory
 - Current Status
 - Any other relevant attributes
- **Process Table:** To manage processes the OS needs to know following details of the processes
 - Current state
 - Process ID
 - Location in memory

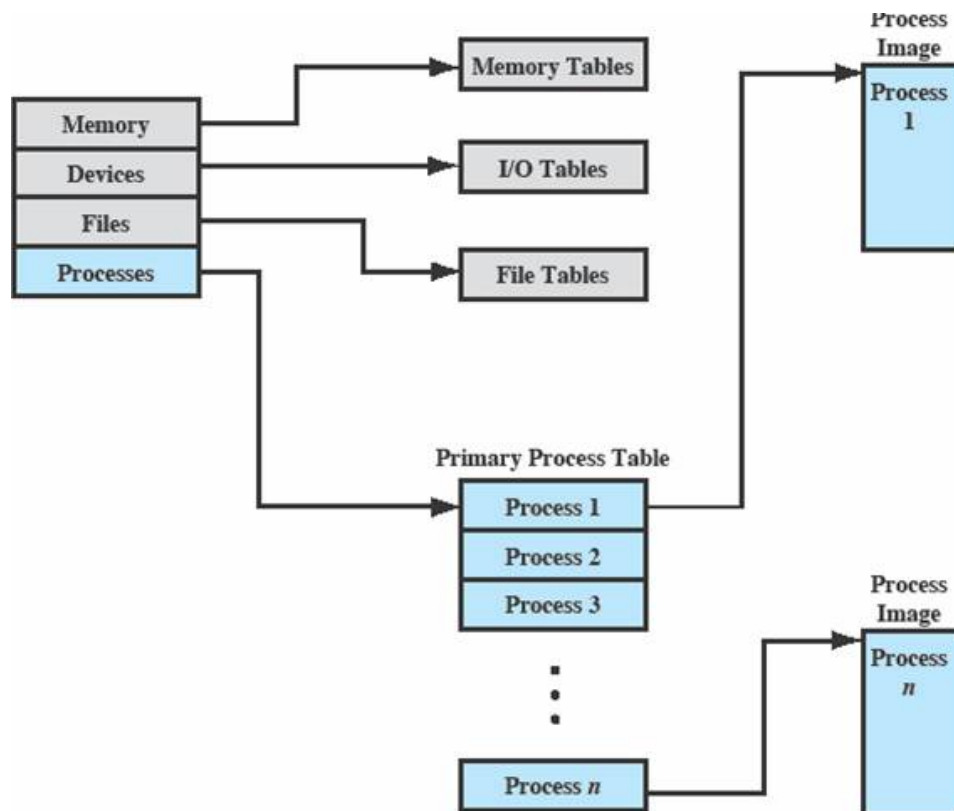


Figure 3.7 General Structure of OS Control Tables

3.3.2 Process Control Structures

To control and manage a process, OS must know where the process is located and the process attributes.

Process Location: Before checking where the process is located, we should understand the process image. **Process image** is a collection of user program, data, stack and process control block as shown in Table 3.4.

Table 3.4 Elements of Process Image

Elements	Description
User Data	A modifiable user program, user stack area, and relevant data
User program	The program to be executed
Stack	Each process has a stack associated with it. It is used for storing parameters and calling addresses
Process control block	Collection of attributes needed by OS to control the process

The process image is usually maintained as a contiguous block of memory and is stored on the disk. As the entire process image has to be loaded into main memory for process execution, the OS must know the exact location of process image on the disk.

The structure of location information can be understood using Figure 3.7. Here, the primary process table has one entry for each process. Each entry has one pointer to a process image.

Process Attributes: The information required by OS about a process can be categorized as follows:

- **Process Identification:** It is a unique number assigned every process. It includes
 - Identifier of this process
 - Identifier of parent process
 - User identifier
- **Processor state information:** Consists of processor registers like –
 - User visible registers
 - Control and status registers
 - Stack pointers
- **Process control information:** Includes the information like –
 - Scheduling and state information
 - Data structuring
 - Interprocess communication
 - Process privileges
 - Memory management
 - Resource ownership and utilization

Figure 3.8 shows the structure of process images in virtual memory.

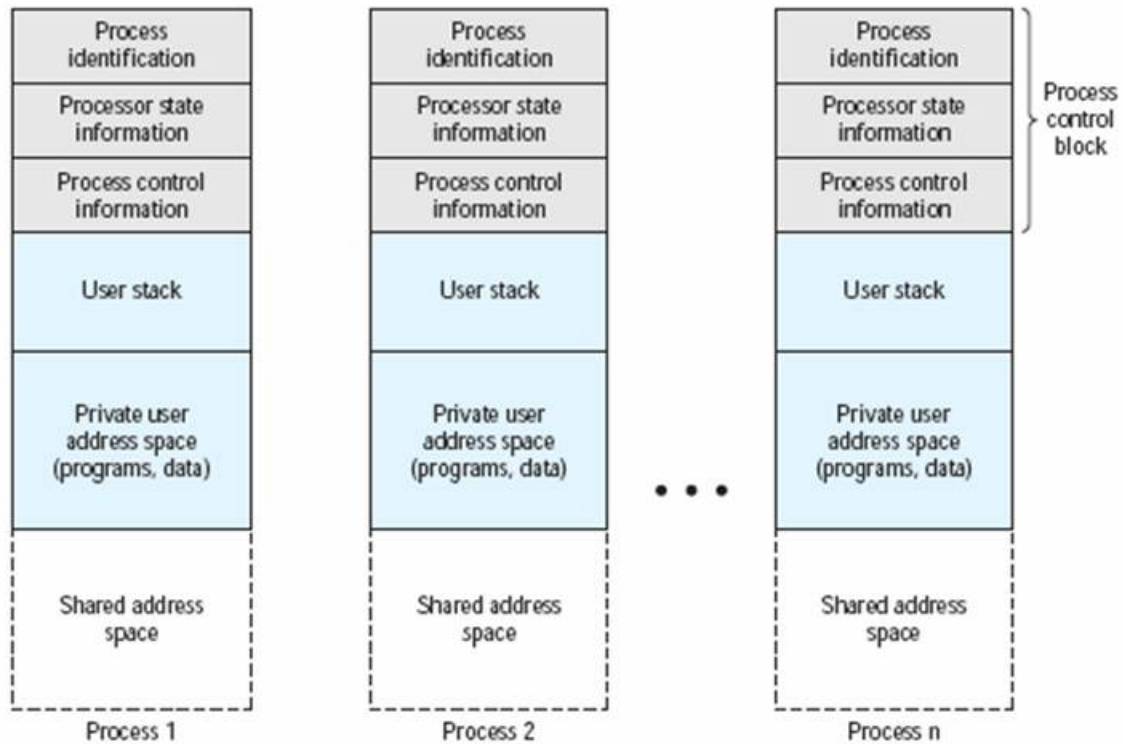


Figure 3.8 User Processes in virtual memory

3.4 PROCESS CONTROL

In the following sections, we will discuss how OS controls the processes.

3.4.1 Modes of Execution

Most of the processors support two modes of execution:

- **User Mode:** user programs are executed in this mode and it is a less-privileged mode.
- **System Mode:** It is also known as **control mode** or **kernel mode**. It is a more privileged mode.

The OS and the important tables, process control blocks etc. must be protected from interference of user program. Hence, two modes must be provided separately. Typical functions of OS kernel are:

- Process management
- Memory management
- I/O management
- Support functions (like interrupt handling, accounting etc)

3.4.2 Process Creation

When OS decides to create a new process, it does the following:

- Assign a unique process identifier to the new process
- Allocate space for the process

- Initialize the process control block
- Set the appropriate linkages
- Create or expand other data structures

3.4.3 Process Switching

Sometimes, the OS will interrupt a running process and assigns another process to running state. That is, OS switches the process. Table 3.5 indicates few situations when process switch occurs.

Table 3.5 Mechanisms for interrupting the execution of a process

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

3.4.4 Change of Process State

The steps involved in the change of process state are as below:

- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently in the Running state
- Move process control block to appropriate queue – ready; blocked; ready/suspend
- Select another process for execution
- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process

3.5 EXECUTION OF THE OPERATING SYSTEM

We normally say that OS is also software. Hence the question is: If the OS is just a collection of programs and if it is executed by the processor just like any other program, is the OS a process? If so, how is it controlled? Who (what) controls it? These aspects are discussed here.

3.5.1 Nonprocess Kernel

The traditional approach of OS execution is to execute the kernel of the OS outside any process. Here, the OS has its own memory region and own system stack. Hence, the concept of process is applied only to user programs and OS code is executed as a separate entity that operates in privileged mode. This procedure is shown in Figure 3.9 (a).

3.5.2 Execution within User Processes

Another approach of OS execution is to execute the OS within the user process as shown in Figure 3.9(b). Here, there is no need of process switch to run an OS routine. This approach is common in small computers like PC and workstations.

3.5.3 Process-based OS

OS can be treated as collection of system processes as shown in Figure 3.9(c). As in the other options, the software that is part of the kernel executes in a kernel mode. But, major kernel functions are organized as separate processes.

This approach has several advantages:

- It imposes a modular OS with minimal, clean interfaces between the modules.
- Some noncritical OS functions are conveniently implemented as separate processes.
- Implementing the OS as a set of processes is useful in a multiprocessor or multicomputer environment, in which some of the OS services can be shipped out to dedicated processors, improving performance.

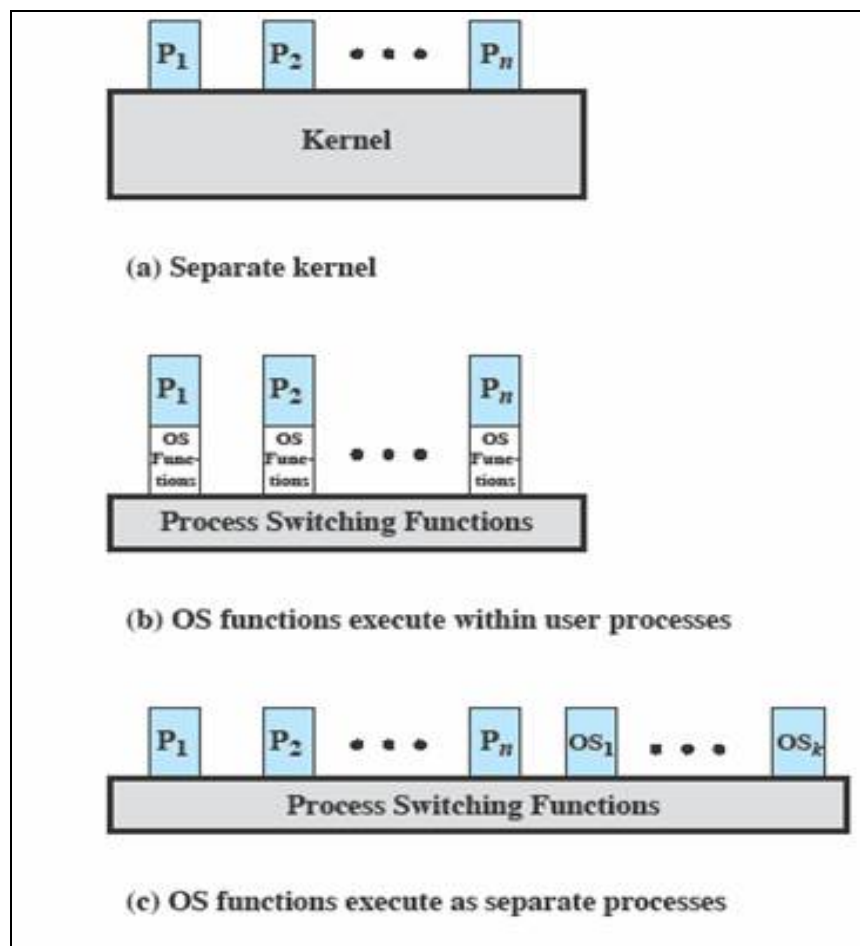


Figure 3.9 Relationship between OS and User Processes

3.6 SECURITY ISSUES

An OS associates a set of privileges with each process. These privileges indicate what resources the process may access, including regions of memory, files, privileged system instructions, and so on.

Typically the highest level of privilege is referred to as administrator, supervisor, or root, access. Root access provides access to all the functions and services of the operating system.

A key security issue in the design of any OS is to prevent anything (user or process) from gaining unauthorized privileges on the system – especially from gaining root access. Different types of threats to the OS and the respective countermeasures are discussed below.

3.6.1 System Access Threats

System access threats fall into two categories:

- **Intruders:** It is also called as hacker or cracker. There are three classes of intruders:
 - **Masquerader:** An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account
 - **Misfeasor:** A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges
 - **Clandestine user:** An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

- **Malicious Software:** These are very sophisticated types of threats that are presented by programs and exploit vulnerabilities in the computing system.

3.6.2 Countermeasures

Following are the different ways of facing the security threats:

- **Intrusion Detection:** Intrusion detection systems (IDS) are typically designed to detect human intruder and malicious software behaviour. They may be host or network based. Intrusion detection systems (IDS) typically comprise
 - Sensors
 - Analyzers
 - User Interface
- **Authentication:** User authentication is a basic building block for defense consisting of two steps:
 - Identification
 - Verification

There are four ways of authenticating a user's identity:

- Something the individual **knows** (like password)
 - Something the individual **possesses** (like ID Card)
 - Something the individual **is** (static biometrics like finger print, iris etc.)
 - Something the individual **does** (dynamic biometrics like voice, gait etc.)
- **Access Control:** It implements a security policy to indicate who/what can access each resource. A security administrator maintains an authorization database. The access control function consults this to determine whether to grant access. An

auditing function monitors and keeps a record of user accesses to system resources.

- **Firewalls:** Traditionally, a firewall is a dedicated computer that:
 - interfaces with computers outside a network
 - has special security precautions built into it to protect sensitive files on computers within the network.

3.7 PROCESSES AND THREADS

Processes have two characteristics:

- **Resource Ownership:** On regular intervals, a process will be allocated the control or ownership of resources like main memory, I/O devices, files etc. The OS performs a protection function to prevent unwanted interference between processes with respect to resources.
- **Scheduling/Execution:** The execution of a process follows an execution path through one or more programs. Then, OS will dispatch the process after completion.

To distinguish the two characteristics, the unit of dispatching is usually referred to as a **thread** or **lightweight process**. The unit of resource ownership is referred as *process* or *task*.

3.7.1 Multithreading

The ability of OS to support multiple, concurrent paths of execution within a single process is called as **multithreading**. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach. MS-DOS is an example of OS which supports single process and single thread. Some versions of UNIX support multiple processes, but single thread per process. Java runtime environment supports multiple threads per single process. These are depicted in Figure 3.10.

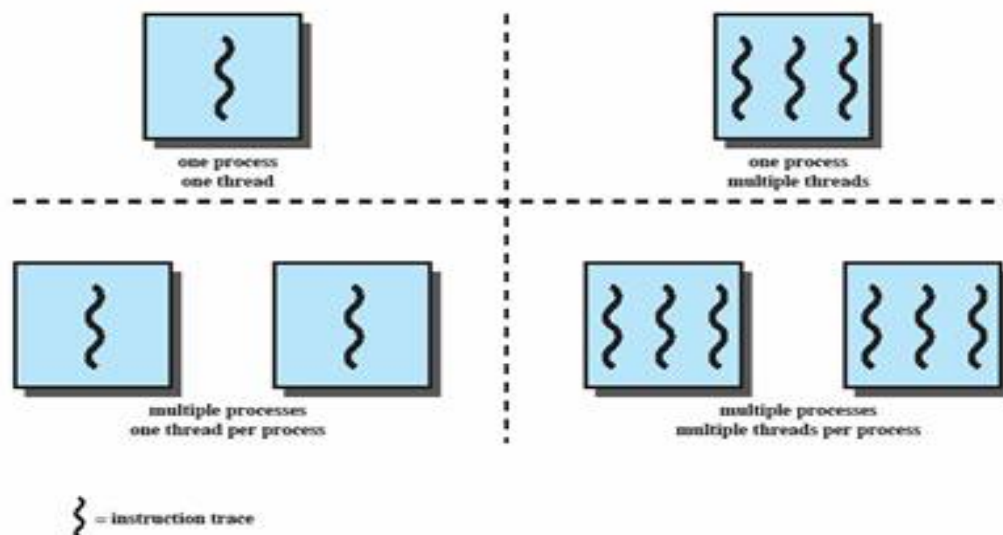


Figure 3.10 Threads and Processes

In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. Every process is associated with –

- A virtual address space that holds the process image
- Protected access to processors, other processes, files and I/O resources

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

- An execution state (running, ready, etc.)
- Saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process (all threads of a process share this)

The difference between threads and processes from the view of process management is shown in Figure 3.11. A single threaded process means there is no thread. In this model, the process includes its control block, user address space and user/kernel stacks. In multithreaded model, there will be a separate control block for each thread. And, all threads of one process share the state and resources of that process.

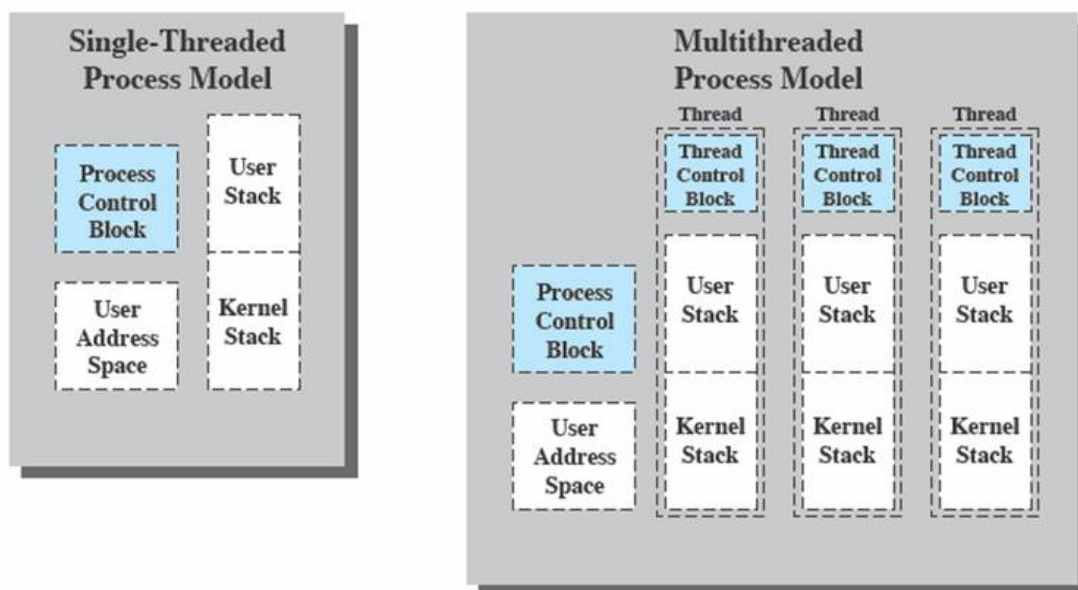


Figure 3.11 Single threaded and multithreaded process models

The benefits of threads are:

- Thread takes less time to create compared to a process
- It takes less time to terminate compared to a process
- Switching between two threads takes less time than switching processes
- Threads can communicate with each other without invoking the kernel

Thus, if any application has to be implemented as a set of related units of execution, it is better to have them as a collection of threads rather than collection of processes. The usage of threads in a single-user multiprocessing system are listed below –

- **Foreground and background work:** For example, in a spread sheet program, one thread may display menus and read user input; whereas another thread may enter user commands and update the spreadsheet.
- **Asynchronous Processing:** A thread may be created to save the word document in a periodic interval where as other thread may be writing the data into the document.
- **Speed of Execution:** A multithreaded process can compute one batch of data while reading the next batch from a device.
- **Modular program structure:** Programs that involve a variety of activities like input and output will be easier to design and implement using threads.

3.7.2 Thread Functionality

Similar to processes, threads also can have execution states and they may synchronize with one another.

Thread States: The key states of threads are Running, Ready and Blocked. But, since all the threads share the same address space of the process, they will be swapped out when a process is swapped out. Hence, suspending a single thread makes no sense.

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

- **Spawn:** When a new process is spawned, a thread for that process is also spawned. Also, a thread within a process may spawn another thread within the same process.
- **Block:** When thread needs to wait for an event, it will be blocked.
- **Unblock:** When an event for which a thread was blocked occurs, the thread is unblocked and moved into ready queue.
- **Finish:** When a thread completes, its registers and stacks are deallocated.

Thread Synchronization: All the threads of a process share the same address space and other resources like files. So, any alteration of a resource by one thread affects the behavior of other threads in the same process. Hence, it is necessary to synchronize the activities of the various threads so that they do not interfere with each other.

3.7.3 User – level and Kernel – level Threads

A thread can be implemented as either a user – level thread (ULT) or kernel – level thread (KLT). The KLT is also known as *kernel – supported threads* or *lightweight processes*.

User – level Threads: In ULT, all work of thread management is done by the application and the kernel is not aware of the existence of threads. It is shown in Figure 3.12 (a). Any application can be programmed to be multithreaded by using a threads library, which a package of routines for ULT management.

Usually, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. The application may spawn a new thread within the same process during its execution. But, kernel is not aware of this activity.

The advantages of ULT compared to KLT are given below:

- Thread switching doesn't require kernel mode privileges. Hence, the overhead of two switches (user to kernel and kernel back to user) is saved.
- Scheduling can be application specific. So, OS scheduling need not be disturbed.
- ULTs can run on any OS. So, no change in kernel design is required to support ULTs.

There are certain disadvantages of ULTs compared to KLTs:

- Usually, in OS many system calls are blocking. So, when a ULT executes a system call, all the threads within the process are blocked.
- In a pure ULT, a multithreaded application cannot take advantage of multiprocessing.

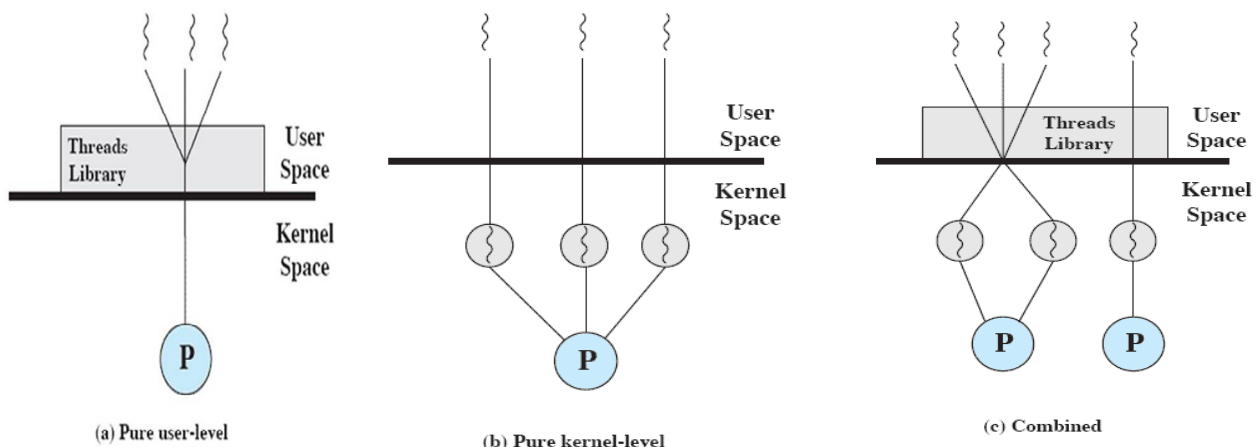


Figure 3.12 ULT and KLT

Kernel – level Threads: In pure KLT model, all work of thread management is done by the kernel. Thread management code will not be in the application level. This model is shown in Figure 3.12(b). The kernel maintains context information for the process as a whole and for individual threads within the process. So, there are certain advantages of KLT :

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

But, there is a disadvantage as well: The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combined Approach: Some OS provide a combination of ULT and KLT as shown in Figure 3.12 (c). In this model, thread creation is done completely in user space. The multiple ULTs from a single application are mapped onto number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best results.

3.7.4 Relationship between Threads and Processes

The interrelationship between threads and processes has been listed here:

Threads : Processes	Description
1:1	Each thread of execution is a unique process with its own address space and resources
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems
M:N	Combines attributes of M:1 and 1:M cases

3.8 SYMMETRIC MULTIPROCESSING (SMP)

Traditionally, the computer has been viewed as a sequential machine. That is, a processor executes instructions one at a time in a sequence and each instruction is a sequence of operations. But, as computer technology has evolved, parallel processing got importance. One of the popular approaches for providing parallelism is symmetric multiprocessors (SMPs), where processors are replicated.

3.8.1 SMP Architecture

Before understanding how SMP works, it is better to observe all parallel processors which are depicted in Figure 3.13.

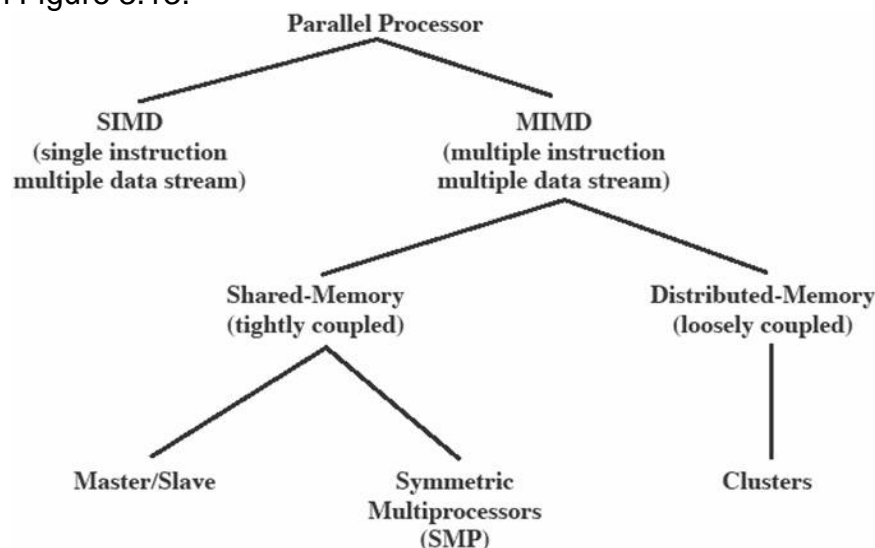


Figure 3.13 Parallel Processor Architectures

- **Single instruction single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory.
- **Single instruction multiple data (SIMD) stream:** Each instruction is executed on a different set of data by the different processors.
- **Multiple instruction single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of execute a different instruction sequence.
- **Multiple instruction multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets.

Based on the communication among processors, MIMD can be further divided. If every processor has a dedicated memory, then each processing element is a self-contained computer. Communication among the computers is either via fixed path or via some network. Such a system is known as a **cluster**. If the processors share a common memory, it is known as **shared-memory multiprocessor**. This again can be further divided into **master/slave** architecture and **SMP**. The master/slave architecture has disadvantages:

- A failure of the master brings down the whole system
- As master has to do all scheduling and process management, the performance may slow down.

But, in SMP, the kernel can execute on any processor and it allows portions of kernel to execute in parallel. Here, each processor does self-scheduling from the pool of available process or threads.

3.8.2 SMP Organization

In SMP, there are multiple processors, each of which contains its own control unit, arithmetic-logic unit and registers as shown in Figure 3.14. Each processor has access to a shared main memory and the I/O devices through a shared bus.

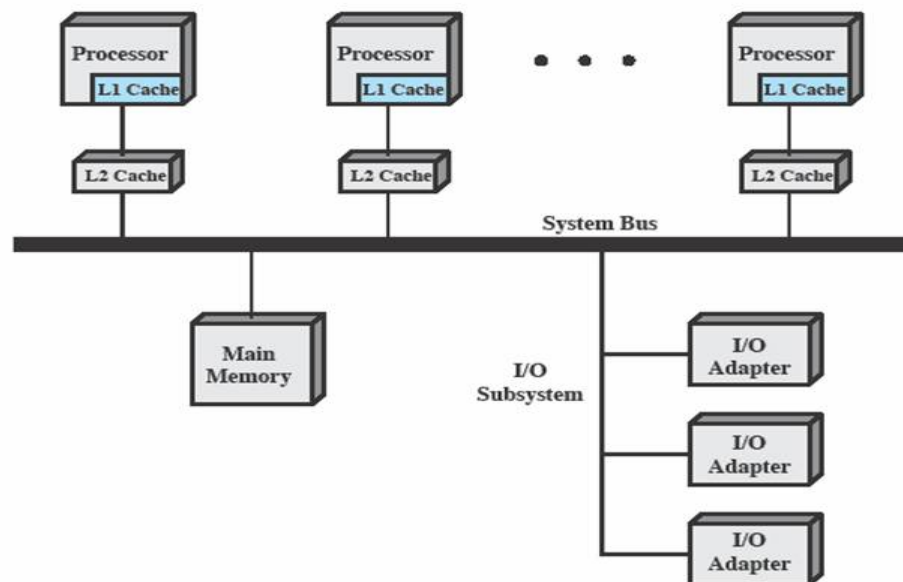


Figure 3.14 SMP Organization

3.8.3 Multiprocessor OS Design Considerations

An SMP OS manages processor and other computer resources so that the user may view the system as if it is a multiprogramming uni-processor system. Hence, the design issues of SMP OS are as below:

- **Simultaneous concurrent processes or threads:** Kernel routines should allow several processors to execute the same kernel code simultaneously.
- **Scheduling:** Scheduling may be performed by any processor to avoid conflicts.
- **Synchronization:** As multiple active processes can access shared address spaces and shared I/O resources, proper synchronization between the processes must be achieved.
- **Memory Management:** All the issues of memory management found in a uni-processor system have to be addressed here as well.
- **Reliability and Fault Tolerance:** OS should provide graceful degradation in case of processor failure.

3.9 MICRO KERNELS

A microkernel is a small OS core that provides the foundation for modular extensions. The question is how small must a kernel be to qualify as a microkernel? *Must* drivers be in user space? In theory, this approach provides a high degree of flexibility and modularity.

3.9.1 Microkernel Architecture

In the early days, OS were designed without a proper structure. Problems caused by mutual dependence and interaction were underestimated. This kind of **monolithic operating systems**, one procedure couldn't call another procedure. To solve these problems, **layered operating systems** got evolved as shown in Figure 3. 15(a). Here, functions are organized hierarchically and interaction takes between only adjacent layers.

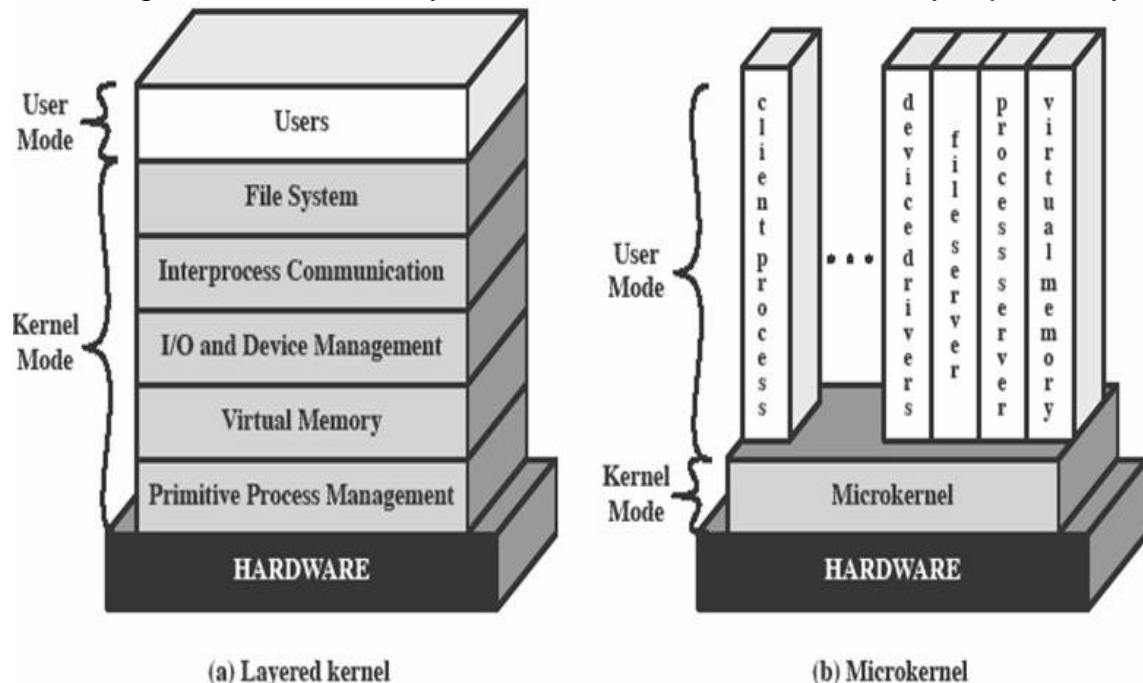


Figure 3.15 Kernel Architecture

But, the layered approach had problems: Change in one layer could affect the working of adjacent layers. To solve such problems, the concept of **microkernels** has been proposed. Here, only essential core OS functions will be in kernel as shown in Figure 3.15(b). And, the less important functions/services/applications are built on the microkernel and execute in user mode.

3.9.2 Benefits of Microkernel Organization

Advantages of using microkernels have been listed here:

- Uniform Interfaces
- Extensibility
- Flexibility
- Portability
- Reliability
- Distributed System Support
- Support for Object Oriented Operating Systems

3.9.3 Microkernel Performance

There are certain disadvantages of microkernels with respect to their performance:

- It takes longer to build and send a message
- Longer time to accept and decode the reply
- Performance depends on size and functionality of the microkernel.

3.9.4 Microkernel Design

There is no set of rules to indicate what functions must be provided by the microkernel and what structure has to be implemented. But, in general, it should include following functions:

- **Low – level Memory management:** The microkernel has to control the hardware concept of address space to make it possible to implement protection at the process level. Also, it should map each virtual page to a physical page frame as shown in Figure 3.16.

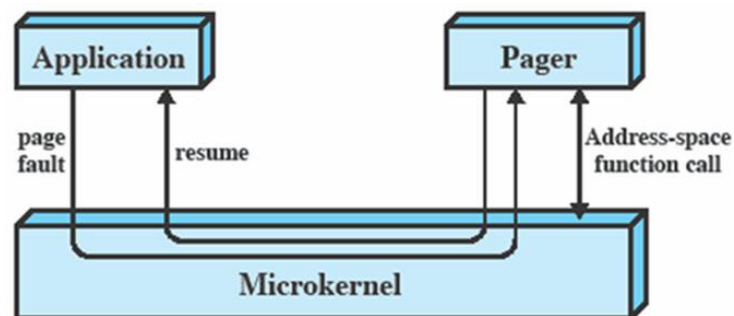


Figure 3.16 Page Fault Processing

- **Inter-process Communication:** Communication between processes or threads in a microkernel OS is via messages. A message includes:
 - A header that identifies the sending and receiving process and
 - A body that contains direct data, a pointer to a block of data, or some control information about the process.

- **I/O and Interrupt Management:** Within a microkernel it is possible to handle hardware interrupts as messages and to include I/O ports in address spaces. A particular user-level process is assigned to the interrupt and the kernel maintains the mapping.

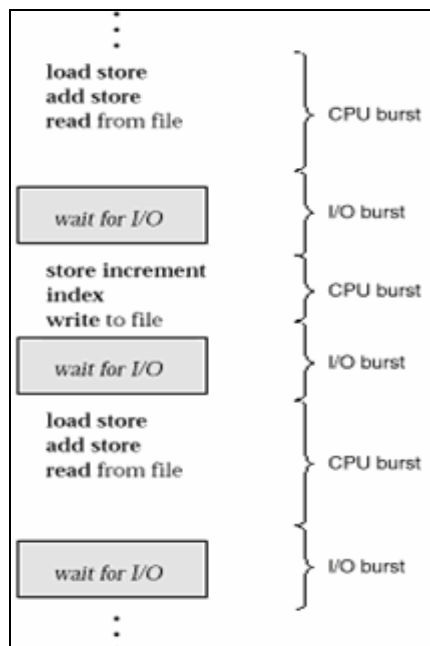
3.10 CPU SCHEDULER

CPU scheduling is a basis of multiprogrammed OS. By switching CPU among processes, the OS makes the computer more productive. In multiprogrammed OS, some process has to keep running all the time in CPU without keeping it idle. This will lead to maximum CPU utilization.

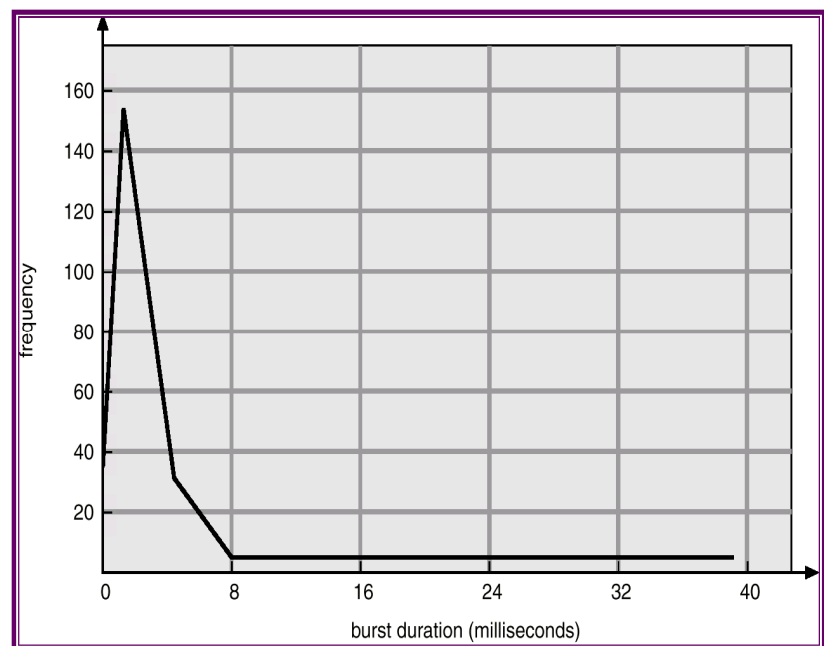
Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or **CPU scheduler**). The processes picked from ready queue need not be first-come-first-out queue. There are various types like shortest job first, round robin etc.

3.10.1 CPU – I/O Burst Cycle

The success of CPU scheduling depends on the following property of processes: Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst. This is illustrated in Figure 3.17(a). The duration of CPU bursts vary by the process and by the computer. Still, they have exponential/hyper exponential frequency curve with many short CPU bursts and few long CPU bursts as shown in Figure 3.17(b).



(a)



(b)

Figure 3.17(a) Alternating sequence of CPU & I/O bursts (b) Histogram of CPU burst times

3.10.2 Preemptive Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state
2. When a process switches from the running state to the ready state
3. When a process switches from the waiting state to the ready state
4. When a process terminates

In the situations (1) and (4) above, there is no choice for scheduling. A new process has to be taken from ready queue. This is called as **non-preemptive** scheduling. But, in the situations (2) and (3), there will be **preemptive** scheduling.

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state.

3.10.3 Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, because it will be invoked during every process switch.

3.10.4 Scheduling Criteria

Different scheduling algorithms have different properties to support different types of processes. There are many criteria to compare CPU scheduling algorithms as given below:

- **CPU Utilization:** CPU must be as busy as possible. CPU utilization may range from 0% to 100%. The real time systems have CPU utilization as 40% to 90%.
- **Throughput:** The number of processes completed per time unit is called as throughput.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting Time:** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response Time:** The time duration from the submission of a request till the first response received is known as response time.

The optimization criteria for CPU scheduling will be –

- Maximum CPU utilization
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time

3.11 SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. We will discuss various algorithms used for CPU scheduling.

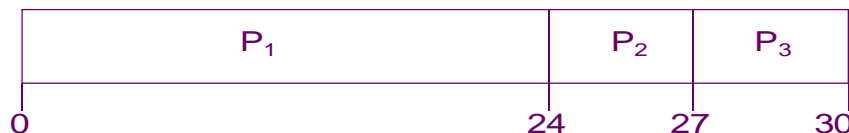
3.11.1 First Come, First Serve Scheduling

FCFS is the simplest algorithm which is managed by a simple FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. Usually, the average waiting time for FCFS will be more.

Example 1: The three processes P_1 , P_2 , and P_3 arrive at a time 0 with the CPU burst time given as below. Calculate average waiting time and average turnaround time.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Solution: Suppose, the processes arrive in the order P_1 , P_2 , P_3 , then the Gantt Chart for the schedule is –



We can observe that,

Waiting time for $P_1 = 0$

Waiting time for $P_2 = 24$

Waiting time for $P_3 = 27$

Thus, **Average waiting time = $(0 + 24 + 27)/3 = 17$ milliseconds**

Turnaround time is the duration from submission of the process till its completion. Hence, turnaround time for $P_1 = 24$

Turnaround time for $P_2 = 27$

Turnaround time for $P_3 = 30$

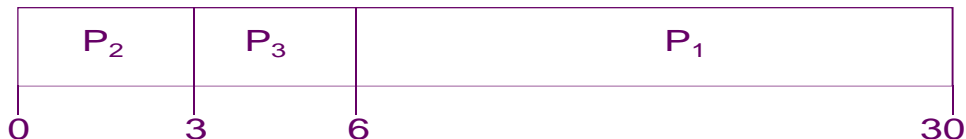
Average turnaround time = $(24+27+30)/3 = 27$ milliseconds

Throughput is total number of processes completed per one unit of time. Here, 30 time units were for completing 3 processes. So, for 1 time unit, the number of processes that can be completed is $3/30$. That is,

$$\text{Throughput} = 3/30 = 0.1$$

Example 2:

Assume, in the above example, the processes arrive in the order P_2 , P_3 , P_1 then the Gantt Chart for the schedule is –



Now,

$$\text{Waiting time for } P_1 = 6$$

$$\text{Waiting time for } P_2 = 0$$

$$\text{Waiting time for } P_3 = 3$$

Thus, **Average waiting time** = $(6 + 0 + 3)/3 = 3$ milliseconds

Average turnaround time = $(30 + 3 + 6)/3 = 13$ milliseconds

Here also, **throughput** = $3/30=0.1$

NOTE: We can observe that average waiting time in FCFS vary substantially if there is a much variation in CPU burst time of the processes.

Disadvantages:

- FCFS is non-preemptive, hence the average waiting time can be more.
- Troublesome in time-sharing systems, where each user needs to get a share of CPU. But, FCFS scheme will keep CPU for a longer duration.

3.11.2 Shortest-Job-First Scheduling

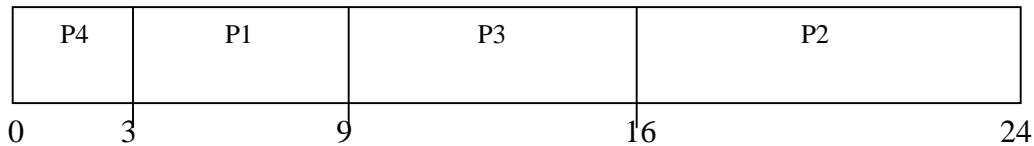
As the name suggests, in SJF scheduling, the process with a shortest CPU-burst is allocated the CPU first. If two processes have equal CPU-burst value, then FCFS is used to break a tie. There are two schemes in SJF viz. non-preemptive and preemptive.

- **Non-preemptive SJF Scheduling:** Here, once the CPU is given to the process, it cannot be preempted until completes its CPU burst.

Example 1: There are four processes P_1 to P_4 all arrived at the time 0 and burst time is as given below. Compute average waiting time and average turnaround time.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

The Gantt chart would be –



Waiting time for P1 = 3

Waiting time for P2 = 16

Waiting time for P3 = 9

Waiting time for P4 = 0

Average waiting time = $(3+16+9+0)/4 = 7$ milliseconds

Turnaround time for P1 = 9

Turnaround time for P2 = 24

Turnaround time for P3 = 16

Turnaround time for P4 = 3

Average turnaround time = $(9 + 24 + 16 + 3)/4 = 13$ milliseconds

Throughput = $4/24 = 0.1667$

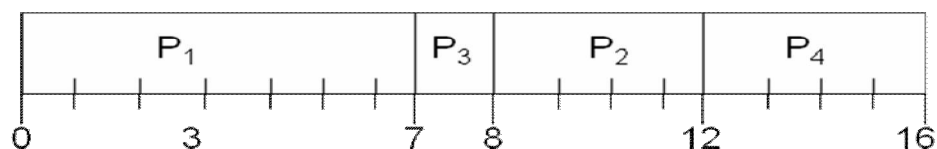
Note that, if we would have used FCFS here, the average waiting time would have been 10.25 milliseconds.

Example 2: There are four processes P1 to P4 which arrived at different times as given below. Compute average waiting time and average turnaround time.

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Solution:

In SJF – non-preemptive algorithm when arrival time of processes is different, one should be careful while drawing Gantt chart. Here, at the time 0, only one process P1 is in a ready queue. Hence, it will take a CPU. At the time 2, the process P2 arrives, whose burst time is smaller than P1. But, as it is non-preemptive, P1 will continue to execute. When P1 completes the execution and leaves the CPU at the time 7, observe that all three processes P2, P3 and P4 are arrived. Now, take the shorter one, that is P3. After completion of P3, there is a tie between P2 and P4 as both have same burst time. Now, resolve the tie using FCFS. Hence, the Gantt chart would be –



Waiting time for P1 = 0

Waiting time for P2 = 8 – 2 (arrival time) = 6

Waiting time for P3 = 7 – 4 (arrival time) = 3

Waiting time for P4 = 12 – 5 (arrival time) = 7

Average waiting time = (0+6+3+7)/4 = 4ms

Turnaround time for P1 = 7 (completion time) – 0 (arrival time) = 7

Turnaround time for P2 = 12 (completion time) – 2 (arrival time) = 10

Turnaround time for P3 = 8 (completion time) – 4 (arrival time) = 4

Turnaround time for P4 = 16 (completion time) – 5 (arrival time) = 11

Average turnaround time = (7+10+4+11)/4 = 8ms

Throughput = 4/16 = 0.25

SJF algorithms usually give minimum average-waiting time and hence an optimal algorithm. But, the problem is – in a dynamic situation, knowing the length of next CPU burst will be difficult.

- **Preemptive SJF Scheduling:** When a new process enters a ready queue while another process is still executing, there will be a choice. If a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is also known as the **Shortest-Remaining-Time-First (SRTF)**. In this scheme, the arrival time of every process plays a key role.

Example: There are four processes P1 to P4 arrived at different time as given below. Compute average waiting time and average turnaround time.

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

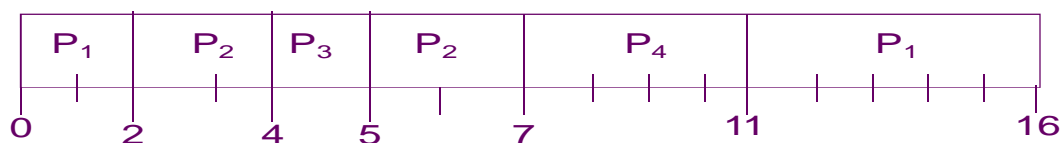
Solution: Here, at the time 0, only process P1 is in a ready queue and hence will start executing. But, after 2 milliseconds, the process P2 arrives with a burst time as 4. The remaining time for P1 is 5, which is greater than time required for P2. Hence, P1 is preempted and P2 gets CPU. After 2 more milliseconds (that is, at the time 4), P3 arrives. The burst time of P3 (1) is lesser than the remaining time of P2(2). Hence, P2 is preempted and P3 is given a CPU. When P3 completes, the remaining time of other three processes are –

P1 -- 5

P2 – 2

P4 – 4

Now, the shortest job will be getting the CPU. Hence, the Gantt chart would be –



Now, the waiting time for each process is calculated as below:

Waiting time for P1: P1 arrived at the time 0 and had been given CPU. After 2ms, it has been preempted. Later, at the time 11, it was given a CPU again. Note that, it has finished the task of 2ms already, hence, the waiting time would be

$$11 - 2 \text{ (completed portion)} = 9$$

Waiting time for P2: P2 arrived at the time 2, and given CPU immediately. Then after 2 ms, it has been preempted. Again, CPU was given at the time 5. So, waiting time will be

$$5 - 2 \text{ (arrival time)} - 2 \text{ (completed time)} = 1$$

Waiting time for P3: P3 arrived at the time 4 and was given CPU immediately. It was not preempted during its execution. And hence, the waiting time is 0.

Waiting time for P4: P4 arrived at the time 5 and was given CPU at the time 7. Hence, its waiting time is 2.

The average waiting time = $(9+1+0+2)/4 = 3$ ms.

Turnaround time is calculated as below –

$$\text{Turnaround time for P1} = 16 \text{ (completion time)} - 0 \text{ (arrival time)} = 16$$

$$\text{Turnaround time for P2} = 7 \text{ (completion time)} - 2 \text{ (arrival time)} = 5$$

$$\text{Turnaround time for P3} = 5 \text{ (completion time)} - 4 \text{ (arrival time)} = 1$$

$$\text{Turnaround time for P4} = 11 \text{ (completion time)} - 5 \text{ (arrival time)} = 6$$

$$\text{Average turnaround time} = (16+5+1+6)/4 = 7 \text{ ms}$$

$$\text{Throughput} = 4/16 = 0.25$$

3.11.3 Priority Scheduling

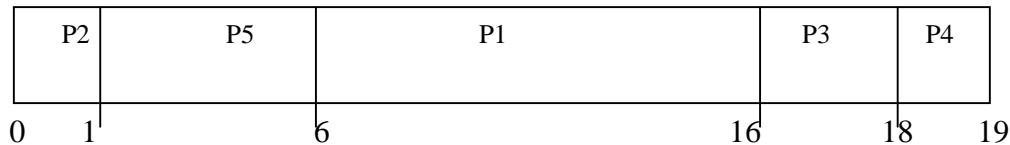
Priority scheduling is a special case of general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. SJF is a priority algorithm where priority is nothing but, its burst time – smaller the burst time, higher the priority. Priority status is normally treated as – lower the number, higher the priority.

The priority scheduling also can be preemptive and non-preemptive. In non-preemptive, normally, the priorities of all the processes are known before. That is, here, we assume that all the processes have arrived at the same time and CPU is assigned based on their priorities. But, sometimes, all the processes may not enter the ready queue at the same time. While one process is running, another process with higher priority than the currently executing process may enter the ready queue. Then currently executing process has to be preempted. So, here, preemption will be based on the arrival of new job with higher priority. This is somewhat similar to preemptive SJF, where new job with lesser burst time will be given a CPU.

Example 1: There are five processes P1, P2, P3, P4 and P5 and have arrived at the time 0 in that order. The priority and burst time are as given below –

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Now, the Gantt chart would be –



The average waiting time = $(6+0+16+18+1)/5 = 8.2$ milliseconds

The average turnaround time = $(16+1+18+19+6)/5 = 12$ ms

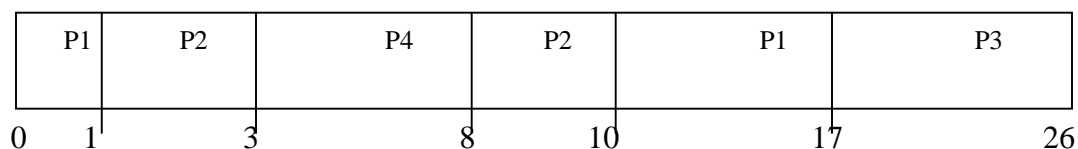
Throughput = $5/19 = 0.2632$

Example 2: Assume there are four processes whose arrival time, burst time and priority have been given as below. Compute average waiting time and turnaround time.

Process	Arrival Time	Burst Time	Priority
P1	0	8	3
P2	1	4	2
P3	2	9	4
P4	3	5	1

Solution:

Note that, the process P1 with priority 3 arrived at the time 0 and no other process is there in a ready queue. Hence, it will be given a CPU. But, just after 1 millisecond, the process P2 with higher priority than P1 arrives and so, P1 has to be preempted and P2 will get CPU. When P2 is under execution, P3 arrives at the time 2. But, its priority is lower than P2. Hence, P2 will continue to execute. Later, P4 arrives at the time 3, which has higher priority than the currently executing P2. So, P2 will be preempted at P4 gets the CPU. Thus, P4 will finish execution before all other processes. Now, the remaining processes will be executed as per their priorities. Thus, the Gantt chart would be –



Waiting time for P1 = $10 - 1$ (completed duration) – 0 (arrival time) = 9

Waiting time for P2 = $8 - 2$ (completed duration) – 1 (arrival time) = 5

Waiting time for P3 = $17 - 2$ (arrival time) = 15

Waiting time for P4 = 0

Average waiting time = $(9+5+15+0)/4 = 7.25$ ms

Turnaround time for P1 = 17 – 0 (arrival time) = 17

Turnaround time for P2 = 10 – 1 (arrival time) = 9

Turnaround time for P3 = 26 – 2 (arrival time) = 24

Turnaround time for P4 = 8 – 3 (arrival time) = 5

Average Turnaround time = (17+9+24+5)/4 = 13.75ms

Throughput = 4/26 = 0.1538

NOTE: The priority scheduling has one drawback: the lower priority processes may never gets executed. This problem is known as **starvation**. As higher priority processes keeps getting added to ready queue, the lower priority processes will find indefinite delay in getting the CPU. In such a situation, either the process will run only when the system becomes free, or eventually, it may crash with all un-finished processes. To solve this problem, there is a remedy called – **aging**. Here, the priority of the process will be increased as the time passes by. Hence, an initially lower priority process will eventually becomes higher priority process and gets the CPU.

3.11.4 Round-Robin Scheduling

This algorithm is designed for time-sharing systems. It is similar to FCFS, but a time quantum is introduced. CPU will be given to the process for one unit of time quantum. After that, the process will be preempted and added back to ready queue. The ready queue will behave as a circular queue. The scheduler will go around the ready queue and allocates CPU for each process for a time interval of one time quantum. The value of time quantum here can be 10ms to 100ms.

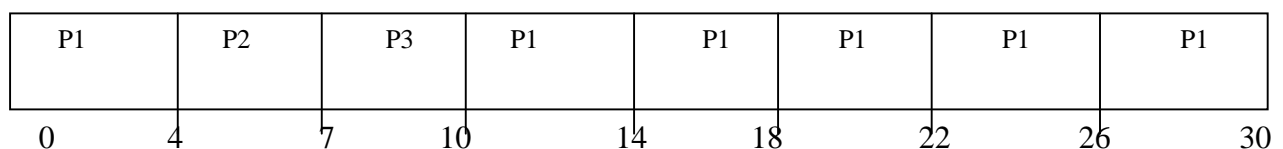
There are two possibilities:

- The process will not be completed in a one time quantum. Then, the context switch will happen and the current process is kept back at the tail of the ready queue. And, the next process in a ready queue is picked and allotted a CPU.
- The process will be completed within the duration of one time quantum. Then the process will give up the CPU voluntarily. Then, next process in a ready queue will be picked for execution.

Example 1: Consider the processes P1, P2 and P3 which are arrived at the time 0. Let the time quantum be 4 milliseconds. The burst time is given as below:

Process	Burst Time
P1	24
P2	3
P3	3

The Gantt chart would be –



Note that, the burst time of processes P2 and P3 are lesser than the actual time quantum and hence, they will give up the CPU immediately after their completion.

Waiting time for P1 = 0 (first pickup) + {10 (next pickup) – 4 (previous CPU release)} = 6

Waiting time for P2 = 4

Waiting time for P3 = 7

Average waiting time = (6+4+7)/3 = 5.67ms

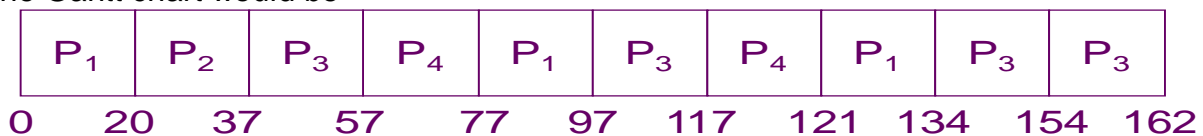
Average turnaround time = (30 +7+10)/3 = 15.67ms

Throughput = 3/30 = 0.1

Example 2: Consider the processes P1, P2, P3 and P4 which are arrived at the time 0. Let the time quantum be 20 milliseconds. The burst time is given as below:

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

The Gantt chart would be –



Waiting time for P1 = 0 (first pickup) + {77 (next pickup) – 20 (previous release)} + {121(next pick up) – 97 (previous release)} = 81

Waiting time for P2 = 20 (first pickup)

Waiting time for P3 = 37 (first pickup) + {97 (next pickup) – 57 (previous release)} + {134 (next pickup) – 117 (previous release)} + {154 (next pickup) - 154 (previous release)} = 94

Waiting time for P4 = 57 (first pickup) + {117 (next pickup) – 77 (previous release)} = 97

Average waiting time = (81+20+94+97)/4 = 73 ms

Average turnaround time = (134+37+162+121)/4 =113.5 ms

Throughput = 4/162 = 0.0247

3.11.5 Multilevel Queue Scheduling

Sometimes, the processes can be easily classified into different groups like –

- Foreground (or interactive) processes
- Background (or batch) processes

These two types have different response – time requirements and different scheduling needs. And, foreground processes may have higher priority over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues as shown in Figure 3.18 . Each queue has its own scheduling algorithm. Normally, foreground queue will have Round Robin algorithm whereas, the background queue has FCFS algorithm.

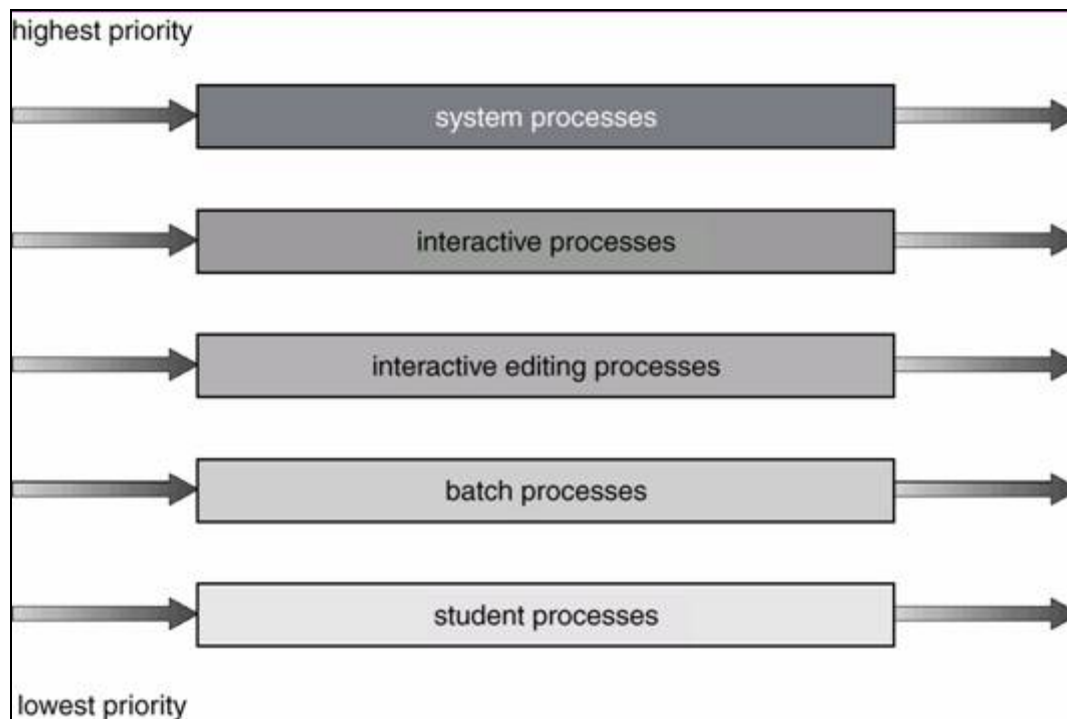


Figure 3.18 Multilevel queue scheduling

3.11.6 Multilevel Feedback Queue Scheduling

In, multilevel queue scheduling algorithm, processes are permanently assigned to a queue as soon as they enter a system. They do not move between the queues. This inflexibility is solved using multilevel feedback queue scheduling algorithm, where each process can move to any other queue. The idea is to separate processes with different CPU – burst characteristics. If a process uses too much CPU time, it will be moved to a lower priority queue. Similarly, if a process waits too long in a lower priority queue, it may be moved to a higher-priority queue.

Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service