UNIT 2

PROCESS MANAGEMENT

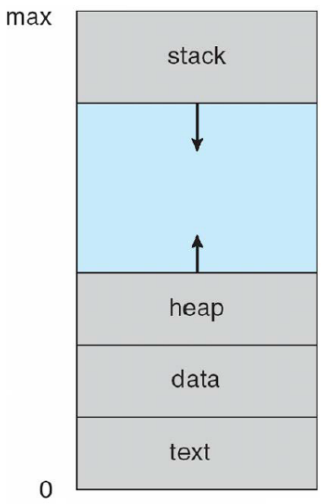
Process concept, The process, Process State Diagram, Process control block, Process Scheduling-Scheduling Queues, Schedulers, Operations on Processes, Inter process Communication, Threading Issues, Scheduling-Basic Concepts, Scheduling Criteria, Scheduling Algorithms.

2.1 Process concept

A process is basically a program in execution. The execution of a process must progress in a sequential fashion. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

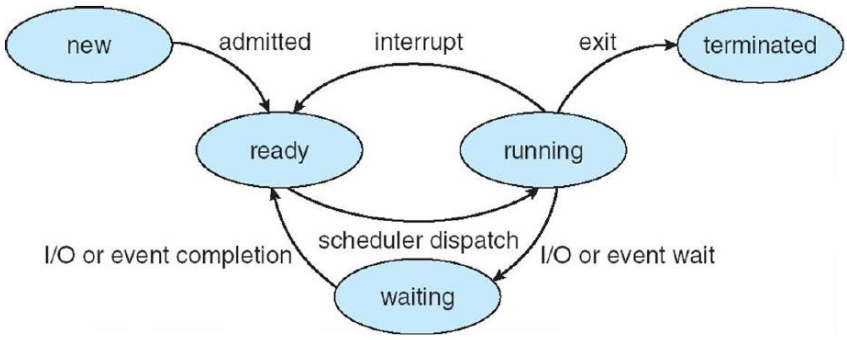
Process memory is divided into four sections for efficient working:

- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.



Process States:

- A process changes states while executing.
- The current activity of a process is called as Process state.
- Process state contains five states and each process can be in any one of the states.
- The various states are New, Ready, Running, Waiting and Terminated.



1. **New:** a process that's has just been created.
2. **Ready:** Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
3. **Running:** These are the processes that are currently being executed. A running process possesses all the resources needed for its execution, including the processor.
4. **Waiting:** a process that cannot execute until some event occurs such as the completion of an I/O operation. The running process may become suspended by invoking an I/O routine.
5. **Terminated:** A process is terminated when it is released from the pool of executable processes by the operating system.

Only one process can be running on any processor at any instant and many processes may be ready and waiting state. Whenever the process changes state, the operating system changes its PCB with respect to the new state.

Suspended processes: the various features of a suspended process are

- They are not readily available for execution.
- The process may or may not be waiting on any event.
- For preventing the execution, process is suspended by OS, parent process, process itself.

The various reasons for suspension are

Swapping: OS needs to release required main memory to bring in a process that is ready to execute.

Timing: Process may be suspended while waiting for the next time interval.

Interactive user request: Process may be suspended for debugging purpose by user.

Parent process request: To modify the suspended process or to co-ordinate the activity of various descendants.

Process Control Block:

Each process is represented in the operating system by a process control block (PCB) enclosing all the information about the process. It is a data structure, which contains the following:

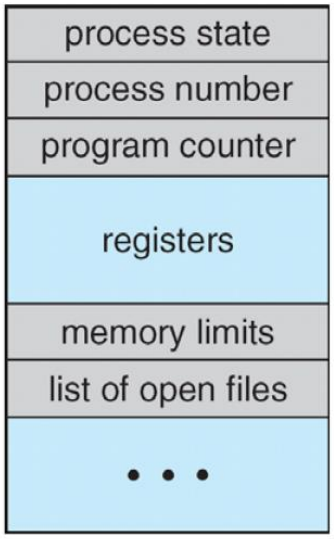
- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

• **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

• **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

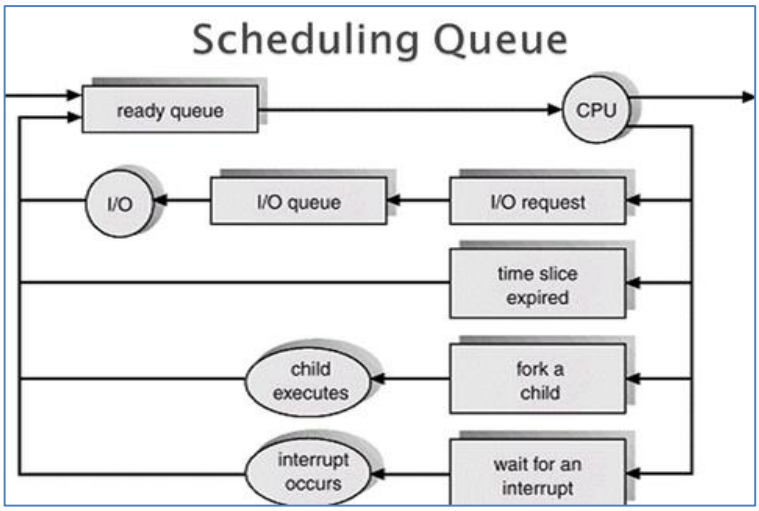


Process Scheduling

The objective of multiprogramming allows some process running at all times, to maximize CPU utilization which allows to switch the CPU among processes so frequently that users can interact with each program while it is running.

For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues:



Queuing-diagram representation of process scheduling.

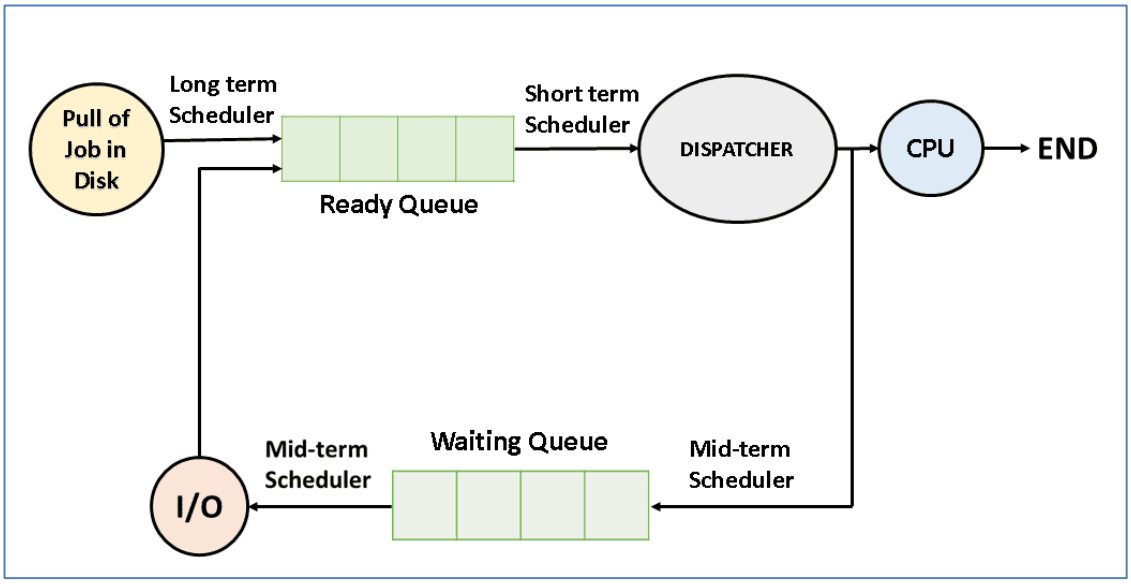
As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler



Long Term Scheduler

- Sometimes the numbers of processes submitted to the system are more than it can be executed immediately. Then in such cases, the processes are spooled on the mass storage, where they reside to get executed later.
- The **Long-Term Scheduler** then selects the process from this spool which is also called as **Job Pool** and loads them in the **Ready Queue** for their further execution.
- It is also called as the **Job Scheduler**.
- The **frequency** of Long-Term Scheduler to pick up the processes from Job pool is **less** as compared to the Short-Term Scheduler.
- Long-Term Scheduler controls the **Degree of Multiprogramming**, which is stable if the rate of creation of the new processes is equal to the average rate of departure of the processes leaving the system.
- The Long-Term Scheduler executes when a process leaves the system.

Short Term Scheduler

- Short-Term Scheduler is also called a **CPU Scheduler**.
- The purpose of Short-Term Scheduler is to select the process from the **Ready Queue** that is ready for the execution and allocate **CPU** to it for its execution.
- The execution of Short-Term Scheduler is very **frequent** as compared to Long-Term Scheduler.
- The Short-Term Scheduler has **less control** over the **Degree of Multiprogramming**.

Medium Term Scheduler

- Medium-term scheduling is a part of **swapping**.
- It removes the processes from the memory.
- It reduces the degree of multiprogramming.
- The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request.
- A suspended process cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out.
- Swapping may be necessary to improve the process mix.

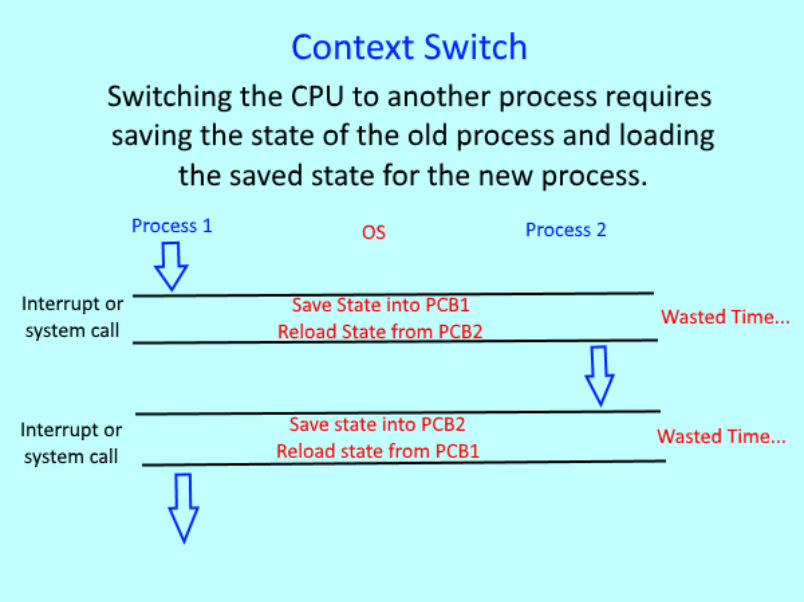
Key Differences between Long-Term Scheduler and Short-Term Scheduler

1. The Long -Term Scheduler select the processes from the Job pool. On the other hand, the Short-Term Scheduler selects the processes from the Ready queue.
2. The Short Term Scheduler executes more frequently as compared to the Long-Term Scheduler.
3. Long-Term scheduler controls the degree of multiprogramming whereas, the Short-Term Scheduling has less control over the degree of Multiprogramming.
4. Long-Term Scheduling is also called Job Scheduler. On the other hand, the Short-Term Scheduling is also called CPU Scheduler.

Context Switch

When an interrupt occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory-management information.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.



Preemptive and Non-Preemptive Scheduling in OS

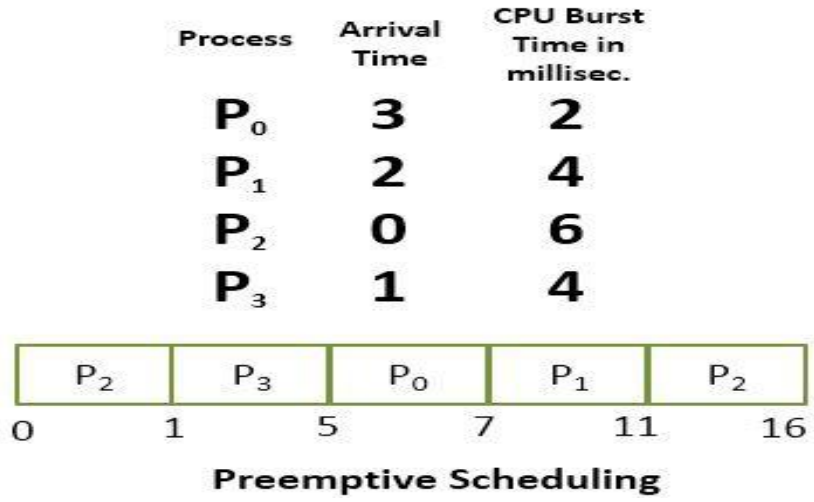
Preemptive Scheduling

Preemptive scheduling is one which can be done in the circumstances when a process switches from running state to ready state or from waiting state to ready state. Here, the resources (CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is placed back in the ready queue again if it still has CPU burst time remaining. The process stays in ready queue till it gets next chance to execute.

If a process with high priority arrives in the ready queue, it does not have to wait for the current process to complete its burst time. Instead, the current process is interrupted in the middle of execution and is placed in the ready queue till the process with high priority is utilizing the CPU cycles. In this way, each process in the ready queue gets some time to run CPU. It makes the preemptive scheduling flexible but, increases the overhead of switching the process from running state to ready state and vise-verse.

Algorithms that work on preemptive scheduling are Round Robin. Shortest Job First (SJF) and Priority scheduling may or may not come under preemptive scheduling.

Let us take an example of Preemptive Scheduling, look in the picture below. We have four processes P0, P1, P2, P3. Out of which, P2 arrives at time 0. So the CPU is allocated to the process P2 as there is no other process in the queue. Meanwhile, P2 was executing, P3 arrives at time 1, now the remaining time for process P2 (5 milliseconds) which is larger than the time required by P3 (4 milli-sec). So CPU is allocated to processor P3.



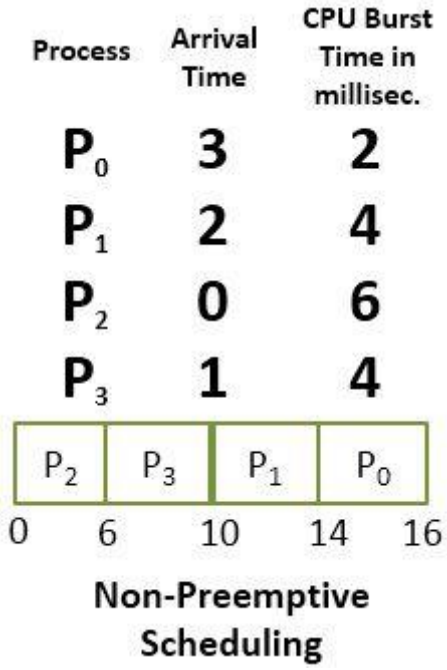
Meanwhile, P3 was executing, process P1 arrives at time 2. Now the remaining time for P3 (3 milliseconds) is less than the time required by processes P1 (4 milliseconds) and P2 (5 milliseconds). So P3 is allowed to continue. While P3 is continuing process P0 arrives at time 3, now the remaining time for P3 (2 milliseconds) is equal to the time required by P0 (2 milliseconds). So P3 continues and after P3 terminates the CPU is allocated to P0 as it has less burst time than other. After P0 terminates, the CPU is allocated to P1 and then to P2.

Non-Preemptive Scheduling

Non-preemptive Scheduling is one which can be applied in the circumstances when a process terminates, or a process switches from running to waiting state. In Non-Preemptive Scheduling, once the resources (CPU) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state.

Unlike preemptive scheduling, non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits for the process to complete its CPU burst time and then it can allocate the CPU to another process.

In Non-preemptive scheduling, if a process with long CPU burst time is executing then the other process will have to wait for a long time which increases the average waiting time of the processes in the ready queue. However, the non-preemptive scheduling does not have any overhead of switching the processes from ready queue to CPU but it makes the scheduling rigid as the process in execution is not even preempted for a process with higher priority.



Non-Preemptive Scheduling

Let us solve the above scheduling example in non-preemptive fashion. As initially the process P2 arrives at time 0, so CPU is allocated to the process P2 it takes 6 milliseconds to execute. In between all the processes i.e. P0, P1, P3 arrives into ready queue. But all waits till process P2 completes its CPU burst time. Then process that arrives after P2 i.e. P3 is then allocated the CPU till it finishes it's burst time. Similarly, then P1 executes, and CPU is later given to process P0.

Differences Between Preemptive and Non-Preemptive Scheduling

1. The basic difference between preemptive and non-preemptive scheduling is that in preemptive scheduling the CPU is allocated to the processes for the limited time. While in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution.
3. Preemptive Scheduling has the overhead of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. On the other hands, non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a process with high priority frequently arrives in the ready queue then the process with low priority have to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process with larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling is quite flexible because the critical processes are allowed to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. The Preemptive Scheduling is cost associative as it has to maintain the integrity of shared data which is not the case with Non-preemptive Scheduling.

Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process.

Each of these new processes may in turn create other processes, forming a tree of processes.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a sub-process, that sub-process may be able to obtain its resources directly from the operating system.

The parent may have to partition its resources among its children, or it may be able to share some resources among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many sub-processes.

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.

When a process creates a new process, two possibilities exist in terms of execution:

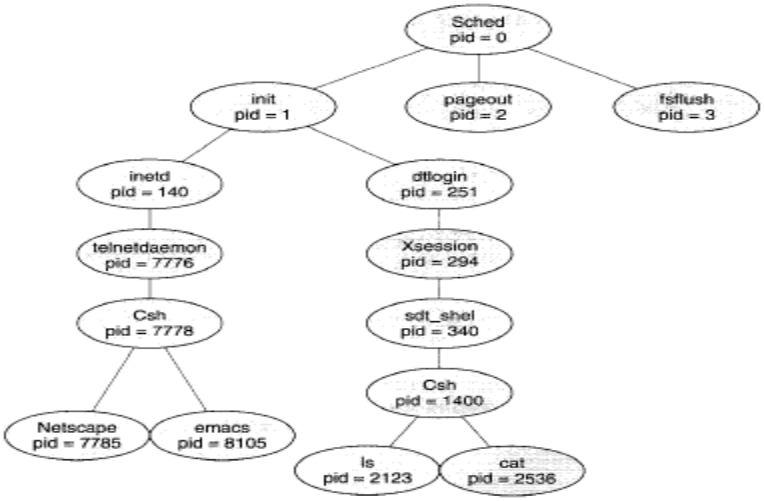
1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it



A tree of processes on a typical Solaris system

Fig 2.4 illustrates a typical process tree for the Solaris operating system, showing the name of each process and it's pid. In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes—including pageout and fsflush. These processes are responsible for managing memory and file systems. The sched process also creates the init process, which serves as the root parent process for all user processes. In Fig 4, we see two children of init — inetd and dtlogin. inetd is responsible for networking services such as telnet and ftp; dtlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process. Below sdt_shel, a user's command-line shell—the C-shell or csh—is created. It is this command line interface where the user then invokes various child processes, such as the Is and cat commands. We also see a csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value to its parent process.

All the resources of the process including physical and virtual memory, open files, and I/O buffers are de-allocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. A parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- **The child has exceeded its usage of some of the resources that it has been allocated.**
- **The task assigned to the child is no longer required.**
- **The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.**

Interprocess Communication:

Processes executing concurrently in the operating system may be either independent or cooperating processes.

A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows **process cooperation**:

1. **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

2. **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

3. **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

4. **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

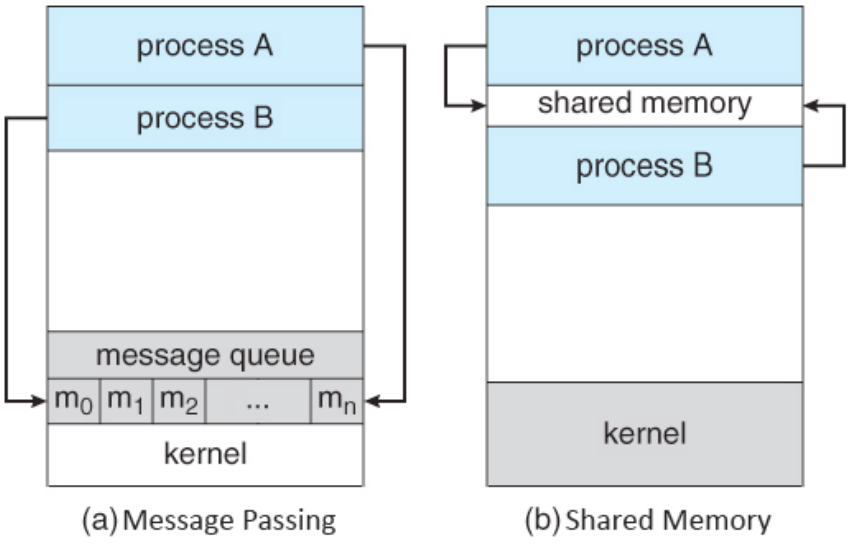
Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication:
(1) Shared memory and
(2) Message passing.

In the **shared-memory** model, a region of memory that is shared by cooperating processes is established. **Processes can then exchange information by reading and writing data to the shared region. S**hared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.

In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

The two communications models are contrasted in Fig below.



Communications models. (a) Message passing. (b) Shared memory.

THREADS:

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional process has a single thread of control.

If a process has multiple threads of control, it can perform more than one task at a time.

Fig illustrates the difference between a traditional single-threaded process and a multithreaded process.

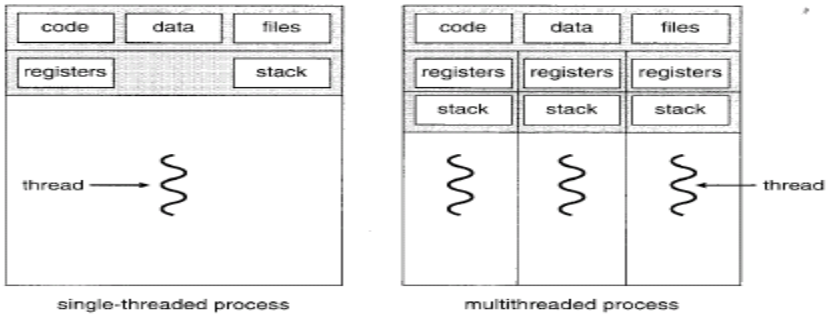


Fig : Single-threaded and Multithreaded processes

Benefits

The **benefits of multithreaded programming** can be broken down into four major categories:

Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource sharing: By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.

Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

Multithreading Models

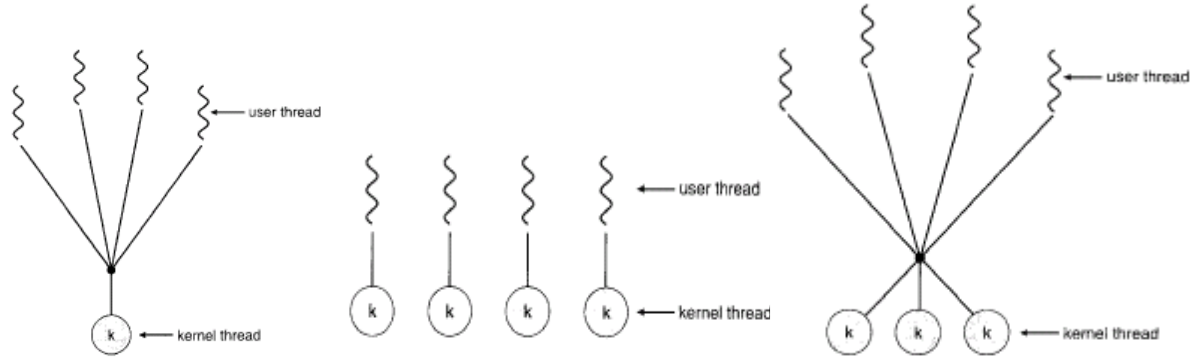
Threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.

User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

Ultimately, there must exists a relationship between user threads and kernel threads.

Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.



(a) (b) (c)

Fig: (a) Many-to-one model. (b) One-to-one model. (c) Many-to-many model.

One-to-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.

Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

PROCESS SCHEDULING CRITERIA AND ALGORITHMS:

Basic Concepts

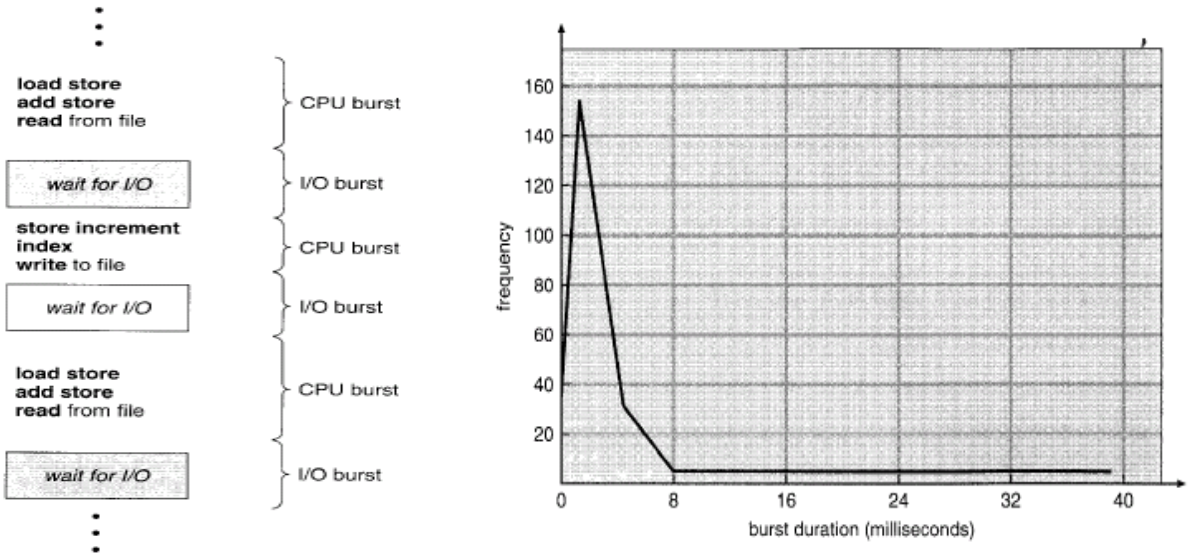
In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to maximize **CPU utilization.** A process is executed until it wait, typically for the completion of some

I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.

Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process and this pattern continues. Scheduling of this kind is a fundamental operating-system function.

CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed 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, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution shown in fig 2.8.



(a) (b)

Fig : (a) Alternating sequence of CPU and I/O burst (b) Histogram of CPU – burst

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Fig. The curve is generally given as with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

Scheduling Methods:

Scheduling methods may be different criteria for selecting process from the ready list.

There are two ways to define algorithms preemptive and non- preemptive. 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

For situations 1 and 4, there is no choice in terms of scheduling. A new process must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive.

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. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware needed for preemptive scheduling.

Unfortunately, preemptive scheduling incurs a cost associated with access to shared data. Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data.

Dispatcher:

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

- 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, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria

Different CPU scheduling algorithms have different properties. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms.

The criteria include the following:

• **CPU utilization:** It is the average function of time during which the CPU is busy. The load on the system affects the level of utilization that can be achieved. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

• **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called 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:** Response time is the time from the submission of a request until the first response is produced.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.

SCHEDULING ALGORITHMS

The problem of deciding which of the processes in the ready queue is to be allocated the CPU is called CPU scheduling. There are many different CPU scheduling algorithms.

First-Come, First-Served Scheduling:

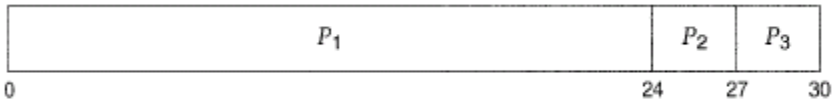
FCFS is the simplest CPU-scheduling algorithm. In this, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

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. The average waiting time under the FCFS policy, however, is often quite long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

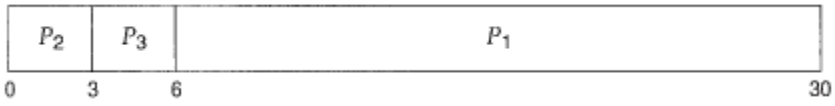
Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6+0+3+)/3 = 3 milliseconds. This reduction is substantial.

Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle.

Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move, back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Shortest-Job-First Scheduling

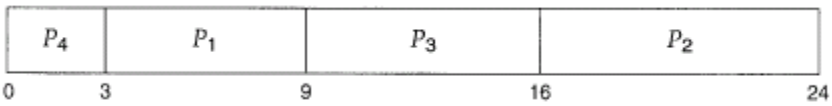
A different approach to CPU scheduling is the shortest-job-first (SIF) scheduling algorithm. When the CPU is available, it is assigned to the process that has the **smallest next CPU burst**. **If the next CPU bursts of two processes are same, FCFS scheduling is used to break the tie.**

Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

SJF algorithm is used frequently in long term scheduling. SJF algorithm may be either preemptive or non-preemptive. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The turnaround time is the sum of burst time plus waiting time of each process. The turnaround time of P1 is (3+6) = 9, P2 is (16+8)= 24, P3 is (9+7)= 16 and P4 is (0+3)= 03. The average turnaround time is (9+24+16+3)/4=13.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long process decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

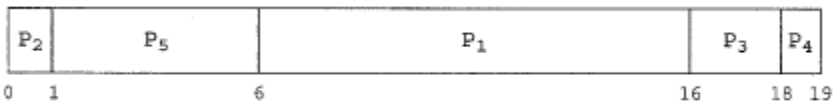
Priority Scheduling

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. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

We can say scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. Some systems use low numbers to represent low priority; others use low numbers for high priority. Let us assume that low numbers represent high priority.
Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds:

Using priority scheduling, we would schedule these processes according to the following Gantt chart:
The average waiting time is 8.2 milliseconds

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- Priority scheduling is preemptive or nonpreemptive.
- Priorities can be defined either internally or externally.
- Internally defined priorities use some quantities to compute the priority of a process like time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used.

- External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation.

- A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- A solution to this kind of problem is aging.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

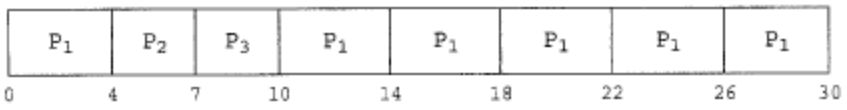
Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is used for time sharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will happen in RR. If the process may have a CPU burst of less than 1 time quantum, the process itself will release the CPU voluntarily and the scheduler will then proceed to the next process in the ready queue. If the process may have a CPU burst of more than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3



If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Since process P2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The average waiting time is 17/3 = 5.66 milliseconds.

The RR scheduling algorithm is thus preemptive. If there are n processes in the ready queue and the time quantum is q, then each process gets 1/w of the CPU time in chunks of at most q time units. Each process must wait no longer than (n — 1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds. The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1/ni the speed of the real processor.

Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes 2. Interactive processes 3. Interactive editing processes 4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

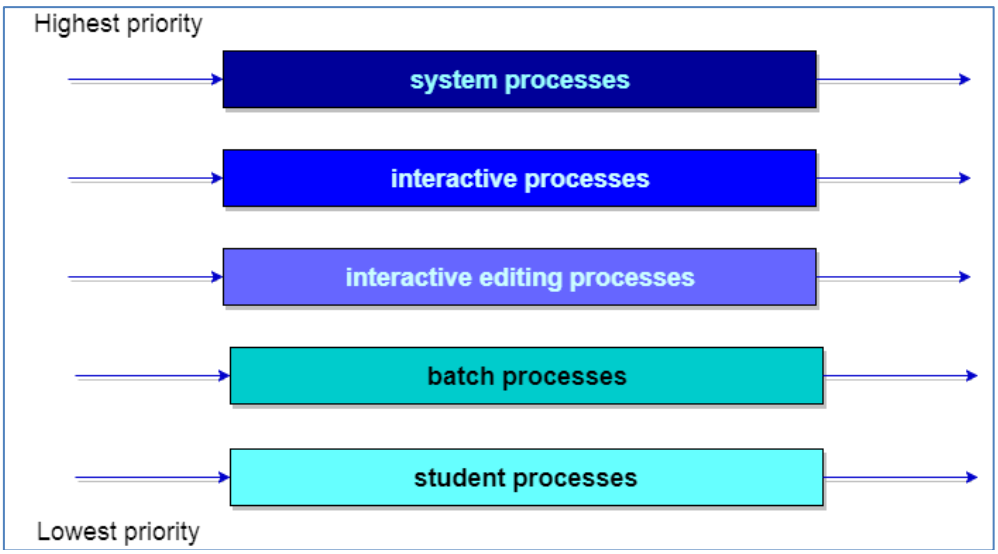


Fig: Multilevel queue scheduling.

Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

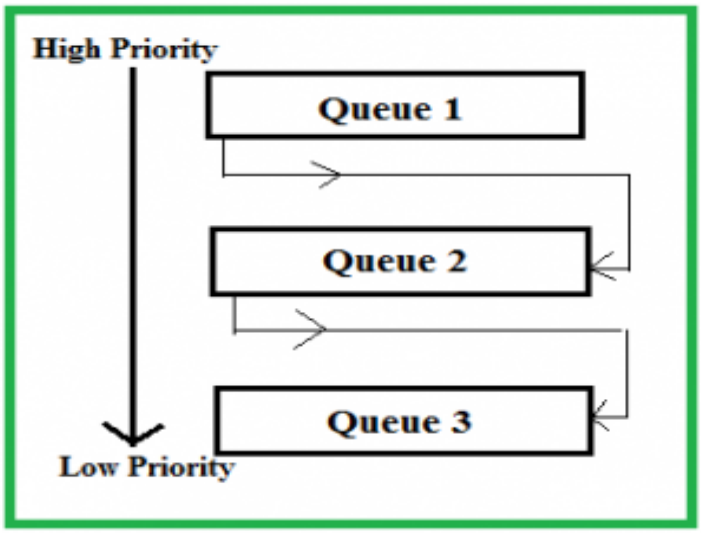
The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue.

This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. It prevents starvation.

In general, a multilevel feedback-queue scheduler is defined by the following parameters:

• The number of queues
• The scheduling algorithm for each queue
• The method used to determine when to upgrade a process to a higher-priority queue
• The method used to determine when to demote a process to a lower-priority queue
• The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback-queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.



Comparison between process and program

S No	Process	Program
1.	Process is active entity	Program is passive entity
2.	Process is a sequence of instruction executions	Program contains the instructions
3.	Process exists in a limited span of time	A program exists at single place and continues to exist
4.	Process is a dynamic entity	Program is a static entity

Comparison between Process and thread

S No	Process	Thread
1.	Process is called heavy weight process	Thread is called light weight process
2.	Process switching needs interface with operating system	Thread switching does not need to call a operating system and cause an interrupt to the kernel
3.	In multiple program implementation	All threads can share same set of open files,

	each process executes the same code but has its own memory and file resources	child processes
4.	If one server process is blocked no other server process can execute until the first process unblocked	While one server thread is blocked and waiting, second thread in the same task could run
5.	Multiple redundant process uses more resources than multiple threaded.	Multiple threaded processes uses fewer resources than multiple redundant process.
6.	In multiple process each process operates independently of the others	One thread cannot read, write or even completely wipe out another threads stack

Comparison between User and Kernel level threads

S No	User level threads	Kernel level threads
1.	User level threads are faster to create and manage	Kernel level threads are slower to create and manage
2.	Implemented by a thread library at the user level	Operating systems support directly to kernel threads
3.	User level threads can run on any operating system	User level threads are specific to the operating system
4.	Support provided at the user level called user level thread	Support may be provided by kernel is called kernel level threads
5.	Multithread application cannot take advantage of multiprocessing	Kernel routines themselves can be multithreaded.

Comparison between Preemptive and Non preemptive scheduling

S No	Preemptive	Non preemptive
1.	Preemptive scheduling allows a process to be interrupted in the midst of its execution , taking the CPU away and allocating it to another process	Non preemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.
2.	Preemptive scheduling incurs a cost associated with access shared data.	Non preemptive scheduling does not increase the cost.
3.	It also affects the design of the operating system	It does not affects the design of OS kernel
4.	Preemptive scheduling is more complex	Simple, but very different
5.	Round robin method is preemptive	FCFS is non preemptive

Comparison between schedulers

S No	Long term	Short term	Medium term
1.	It is a job scheduler	It is a CPU scheduler	It is swapping
2.	Speed is less than short term	Speed is very fast	Speed is in between both
3.	It controls the degree of multiprogramming	Less control over degree of multiprogramming	Reduce the degree of multiprogramming
4.	Absent or minimal in time sharing system	Minimal in time sharing system	Time sharing system use medium term scheduler

Comparison of FCFS and Round robin

S No	FCFS	Round robin
1.	FCFS decision made is non preemptive	RR decision made is preemptive
2.	It has minimum overhead	It has low overhead
3.	Response time may be high	Provides good response time for short process
4.	It is troublesome for time sharing systems	It is mainly designed for time sharing systems
5.	The workload is simply processes in the order of arrival	It is similar like FCFS but uses time quantum

Comparison of scheduling algorithms

Scheduler	Type	Used in	Advantages	Disadvantages
FCFS	Non preemptive	Batch	1. Easy to implement 2. Minimum overhead	1. Unpredictable turnaround time 2. Average waiting is more
RR	Preemptive	Interactive	1. Provides fair CPU allocation 2. Provides reasonable response times to interactive users	1. Requires selection of good time slice
Priority	Non preemptive	Batch	1. Ensures fast completion of important jobs	1. Indefinite postponement of jobs 2. Faces starvation problem
SJF	Non preemptive	Batch	1. Minimizes average waiting time 2. SJF algorithm is optimal	1. Indefinite postponement of some jobs 2. Cannot be implemented at the level of short term scheduling 3. Difficulty in knowing the length of the next CPU request.
Multilevel queues	Preemptive or Non preemptive	Batch/ Interactive	1. Flexible 2. Gives fair treatment to CPU bound jobs	1. Overhead is incurred by monitoring of queues