UNIT - II
Process and CPU Scheduling

1. The Process

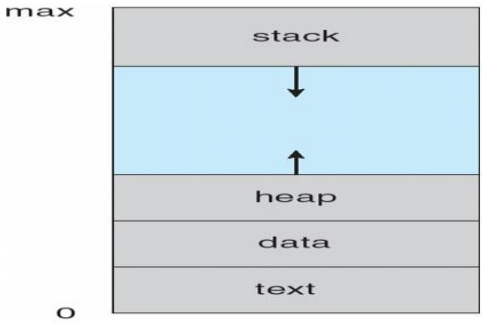
Process Definition:

A process can be thought of as a program in execution. A process is the unit of work in most systems.

A process will need certain resources—such as CPU time, memory, files, and I/O devices to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

Structure of a Process in Memory

- A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables).
- A **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.



When a Program becomes Process?

A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (Often called as **executable file**). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out).

If two processes are associated with the same program, are they same or different? (Or) Explain if you run same program twice, what section would be shared in memory?

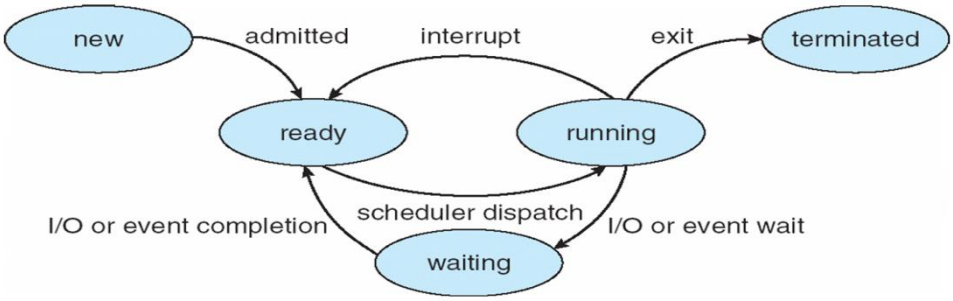
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

2. Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process.

A process may be in one of the following states:

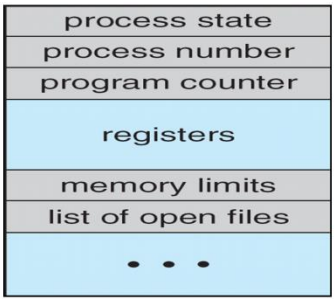
- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.



3. Process Control Block

Each process is represented in the operating system by a **Process Control Block (PCB)** or **Task Control Block.** It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, and waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **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 items as the value of the base and limit registers and 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 is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

1. Scheduling Queues

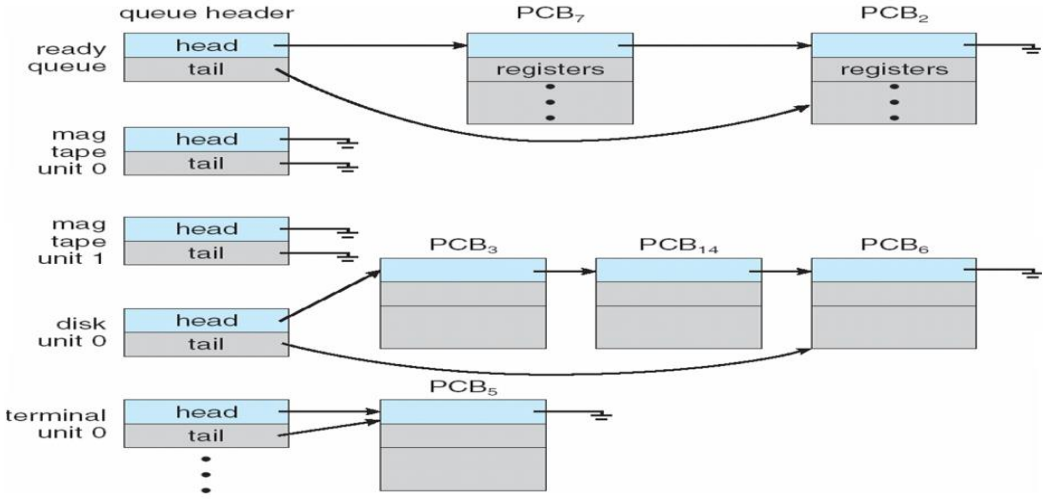
The following are the different queues available,

a. Job Queue

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

b. Ready Queue

- 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.



c. Device Queue

- The list of processes waiting for a particular I/O device is called a **device queue**.
- Each device has its own device queue.

Queuing-diagram representation of process scheduling

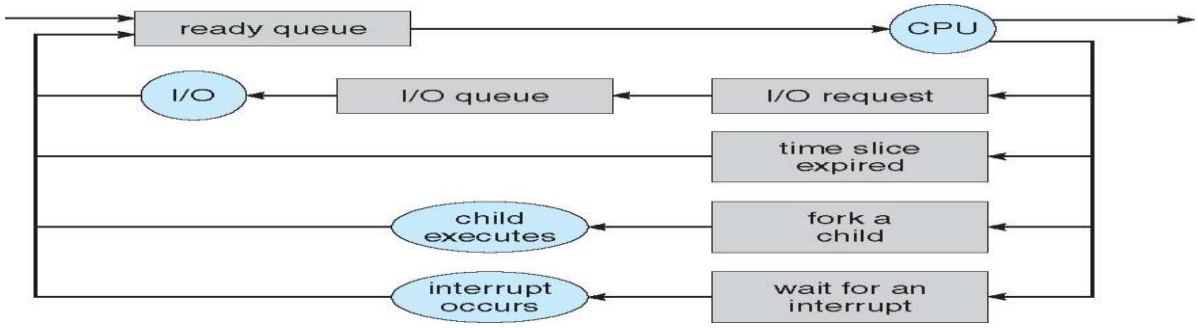
A common representation of process scheduling is a **queuing diagram**. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a

set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.



2. Schedulers

Definition: A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Types of Schedulers

a. Long-Term Scheduler or **Job Scheduler**

- Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.
- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

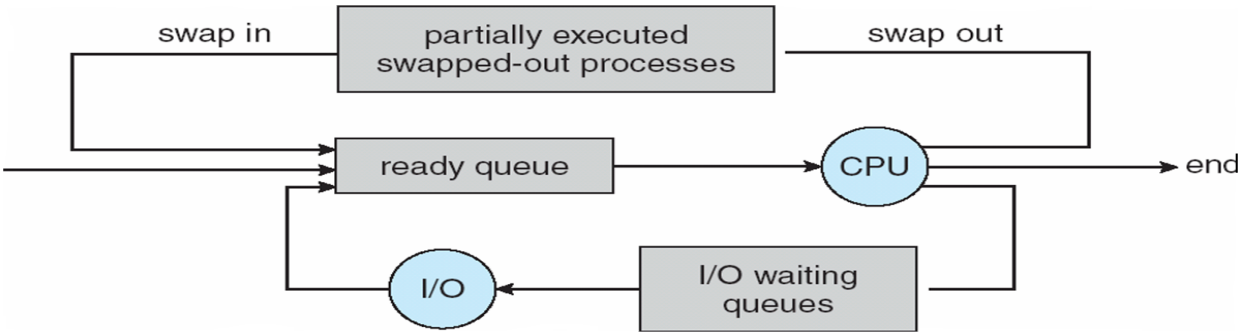
- It is important that the long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes.
- On some systems, the long-term scheduler may be absent or minimal.

b. Short-Term Scheduler, Or **CPU Scheduler**

- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The short-term scheduler must select a new process for the CPU frequently.
- A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds.
- Because of the short time between executions, the short-term scheduler must be fast.

c. Medium-Term Scheduler

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



3. Context Switch

Definition: 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.

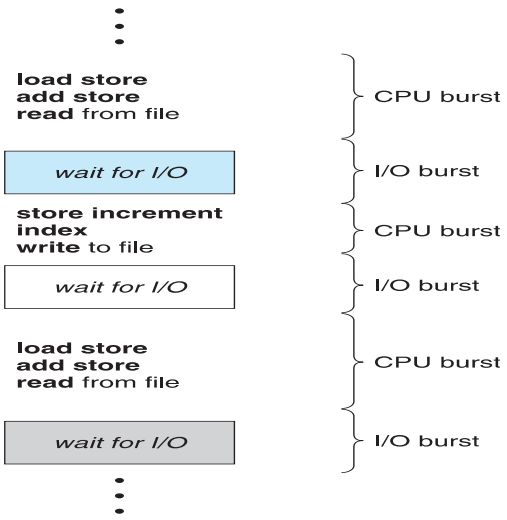
Overhead: Context-switch time is pure overhead, because the system does no useful work while switching.

Switching Speed: Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

Hardware Support: Context-switch times are highly dependent on hardware support. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch

4. 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.



Definition of Non Preemptive Scheduling

Under nonpreemptive 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. This scheduling method was used by Microsoft Windows 3.x.

Definition of Preemptive Scheduling

Under this, a running process may be replaced by higher priority process at any time. Used from Windows 95 to till now. Incurs the cost associated with access to shared data. It also affects the design of OS.

Dispatcher

Another component involved in the CPU-scheduling function is the **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.

Dispatch Latency: The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Operations on processes (OR) System call interface for process management-fork, exit, wait, waitpid, exec

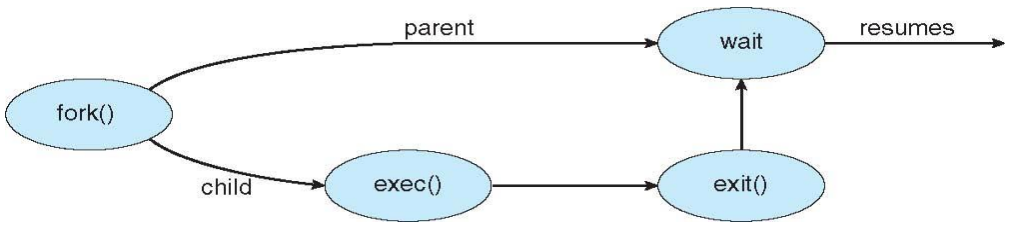
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

a. Process Creation

During the course of execution, a process may create several new processes. 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.

System Calls

- **fork()**
  - Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
  - A new process is created by the fork () system call. The new process consists of a copy of the address space of the original process.
  - This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork (), with one difference: the return code for the fork () is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- **exec()**
  - After a fork () system call, one of the two processes typically uses the exec () system call to replace the process's memory space with a new program.
  - The exec () system call loads a binary file into memory and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.
- **wait()**
  - The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child. Because the call to exec () overlays the process's address space with a new program, the call to exec () does not return control unless an error occurs.



b. 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 (typically an integer) to its parent process (via the wait () system call). All the resources of the process—including physical and virtual memory, open files and I/O buffers—are deallocated by the operating system.

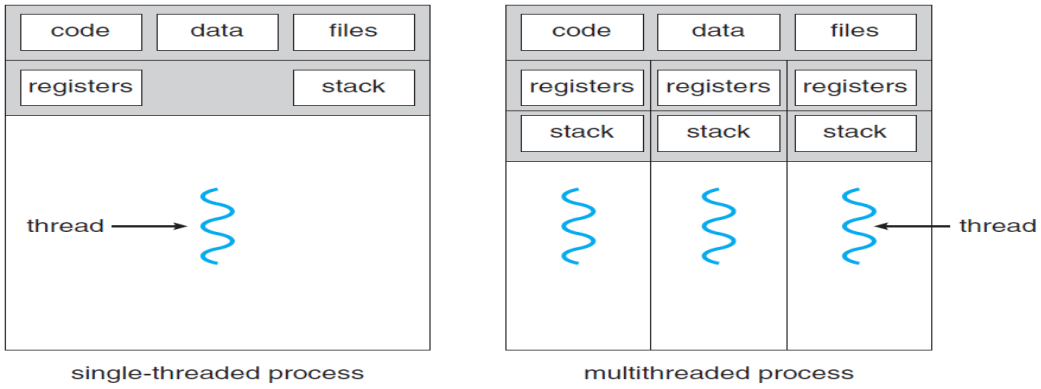
Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

Threads

Defining Thread

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 (or ***heavyweight***) 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.



Single Thread

- A process is a program that performs a single **thread** of execution.
- For example, when a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

Multi Thread

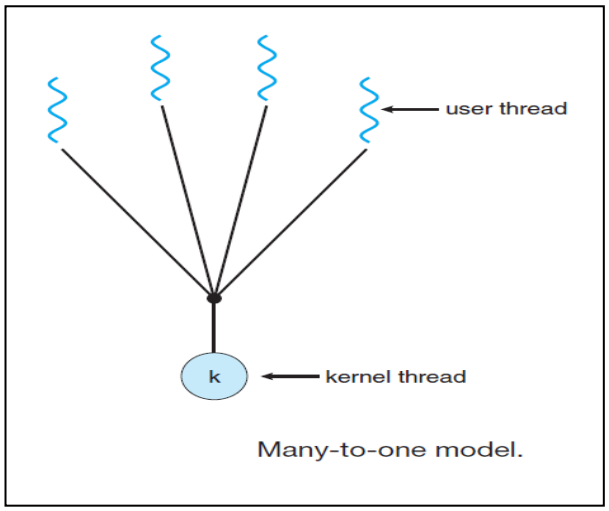
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.
- On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

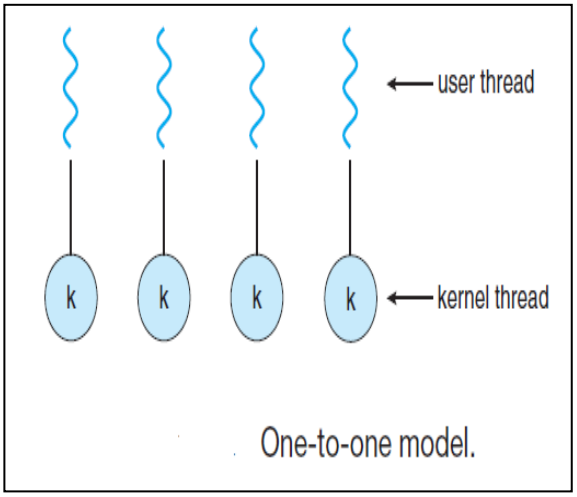
Multithreading Models

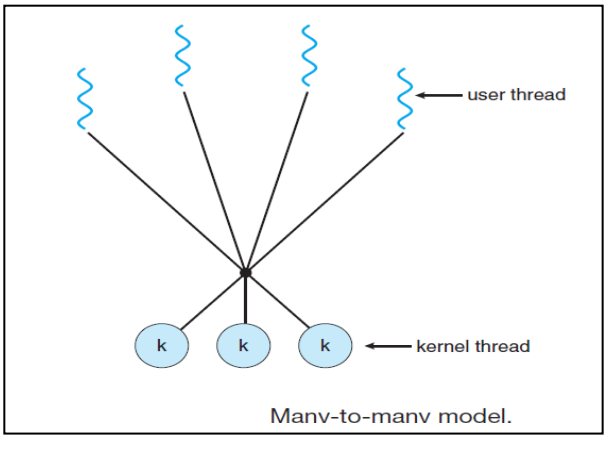
Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. The following are the three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to many models.

1. **Many-to-One Model**
   - The many-to-one model maps many user-level threads to one kernel thread.
2. **One-to-One Model**
   - The one-to-one model maps each user thread to a kernel thread.
3. **Many-to-Many Model**
   - It multiplexes many user-level threads to a smaller or equal number of kernel threads.







Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. 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. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, 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 loaded 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**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **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**. In an interactive system, turnaround time may not be the best criterion. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling algorithms

First-Come, First-Served Scheduling

- **First-Come, First-Served (FCFS)** scheduling algorithm is the simplest CPU-scheduling algorithm.
- With this scheme, 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.
- 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.

- FCFS scheduling algorithm is nonpreemptive. 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, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Example:

Shortest-Job-First Scheduling

- This algorithm associates with each process the length of the process's next CPU burst. 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 the same, FCFS scheduling is used to break the tie.
- 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.
- 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 one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm knows the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. With short-term scheduling, there is no way to know the length of the next CPU burst.
- One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones.
- The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

Example:

Priority Scheduling

- The SJF algorithm is a special case of the 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.

- 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.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority.
- Priorities can be defined either **internally or externally**. **Internally** defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. **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.
- Priority scheduling can be either **preemptive or nonpreemptive**. 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 **nonpreemptive 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 the problem of in definite blockage of low-priority processes is **aging**. Aging involves 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.

Example:

Round-Robin Scheduling

- The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue. To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. 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 then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The

scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer 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.

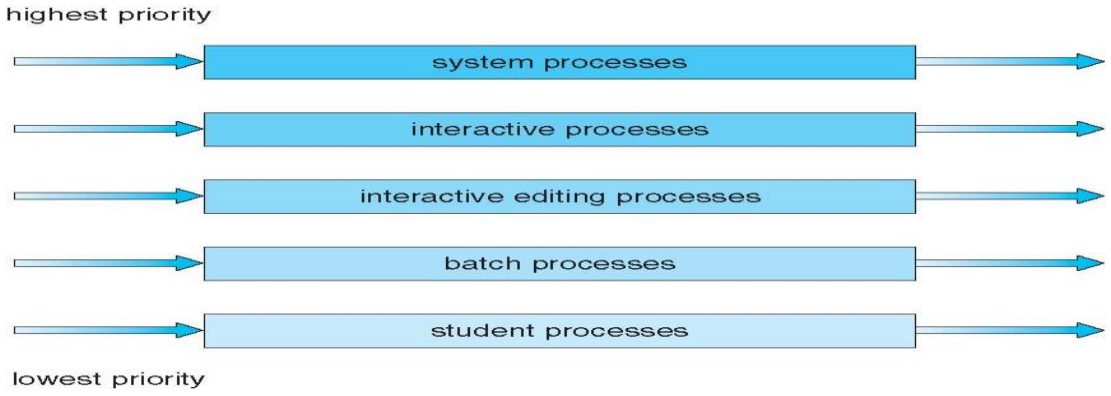
- The average waiting time under the RR policy is often long.
- The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. It creates a processor sharing and creates an appearance that each of n processes has its own processor running at 1/n the speed of the real processor.

Example:

Multilevel Queue Scheduling

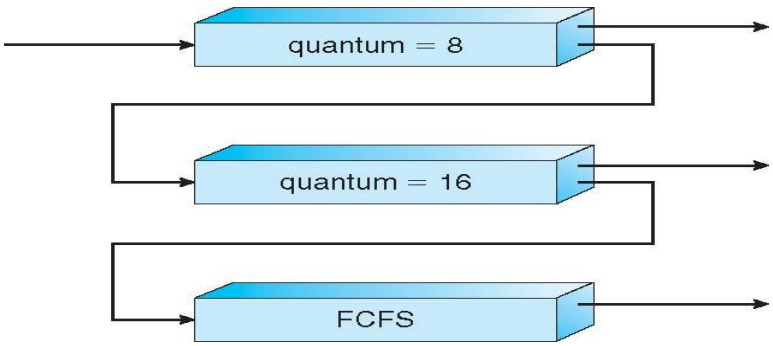
- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally 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.
- Consider the 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. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the

background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.



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. This form of aging prevents starvation.
- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.



- 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.

Multiple- Processor Scheduling

The following are the several concerns in multiprocessor scheduling,

Approaches to Multiple-Processor Scheduling

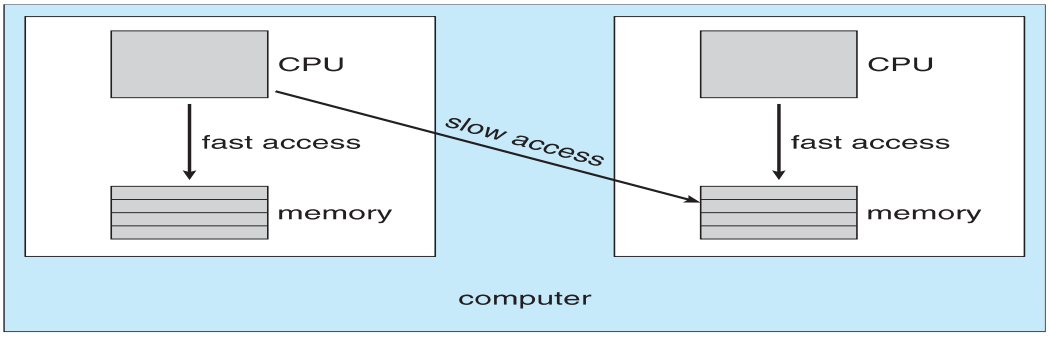
1. **Asymmetric Multiprocessing**
   - All scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code.
   - This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.
2. **Symmetric Multiprocessing (SMP)**,
   - A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
   - Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
   - If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
   - We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.
   - Virtually all modern operating systems support SMP, including Windows, Linux, & Mac OS X.

Processor Affinity

Most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**—that is, a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms,

- **Soft Affinity**: The operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
- **Hard Affinity:** It allows a process to specify a subset of processors on which it may run. The main-memory architecture of a system can affect processor affinity issues. Consider, non-uniform memory access (NUMA). The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system.



Load Balancing

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is necessary only on systems where each processor has its own private queue of eligible processes to execute.

There are two general approaches to load balancing:

1. **Push Migration**
   - With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

2. **Pull Migration**
   - Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.