UNIT II

Process Management Process Definition , Process Relationship , Process states , Process State transitions , Process Control Block , Context switching , Threads, Concept of multithreads , Benefits of threads, Types of threads.

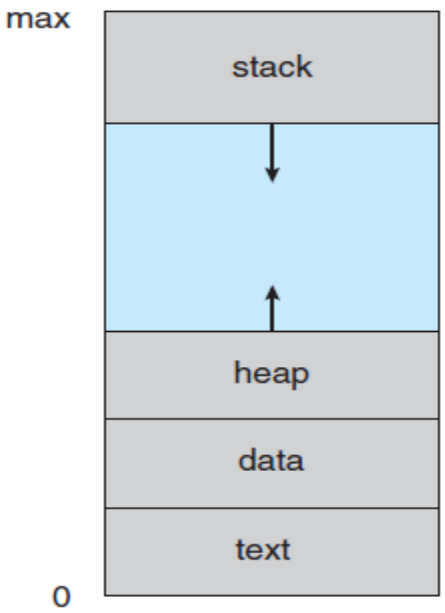
Process Scheduling: Definition, Scheduling objectives, Types of Schedulers, CPU scheduling algorithms, performance evaluation of the scheduling.

Process Management:

- A program does nothing unless their instructions are executed by a CPU.A process is a program in execution. A time shared user program such as a complier is a process. A word processing program being run by an individual user on a pc is a process.
- A system task such as sending output to a printer is also a process. A process needs certain resources including CPU time, memory files & I/O devices to accomplish its task.
- These resources are either given to the process when it is created or allocated to it while it is running.
- The OS is responsible for the following activities of process management.
    - Creating & deleting both user & system processes.
    - Suspending & resuming processes.
    - Providing mechanism for process synchronization.
    - Providing mechanism for process communication.
    - Providing mechanism for deadlock handling.

Process:

- A process or task is an instance of a program in execution.
- The execution of a process must programs in a sequential manner.
- At any time at most one instruction is executed.
- The process includes the current activity as represented by the value of the program counter and the content of the processors registers. Also it includes the process stack which contain temporary data (such as method parameters return address and local variables) & a data section which contain global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.



Process in memory.

Difference between process & program:

- A program by itself is not a process.
- A program in execution is known as a process.
- A **program is a passive entity**, such as the contents of a file stored on disk whereas **process is an active entity** with a program counter specifying the next instruction to execute and a set of associated resources may be shared among several process with some scheduling algorithm being used to determinate when the stop work on one process and service a different one.

Process Relationship:

Process States

As a process executes, it changes state. The state of a process is defined by the correct activity of that process. Each process may be in one of the following states.

- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Terminated:** The process has finished execution.

Process State transitions

Many processes may be in ready and waiting state at the same time. But only one process can be running on any processor at any instant.

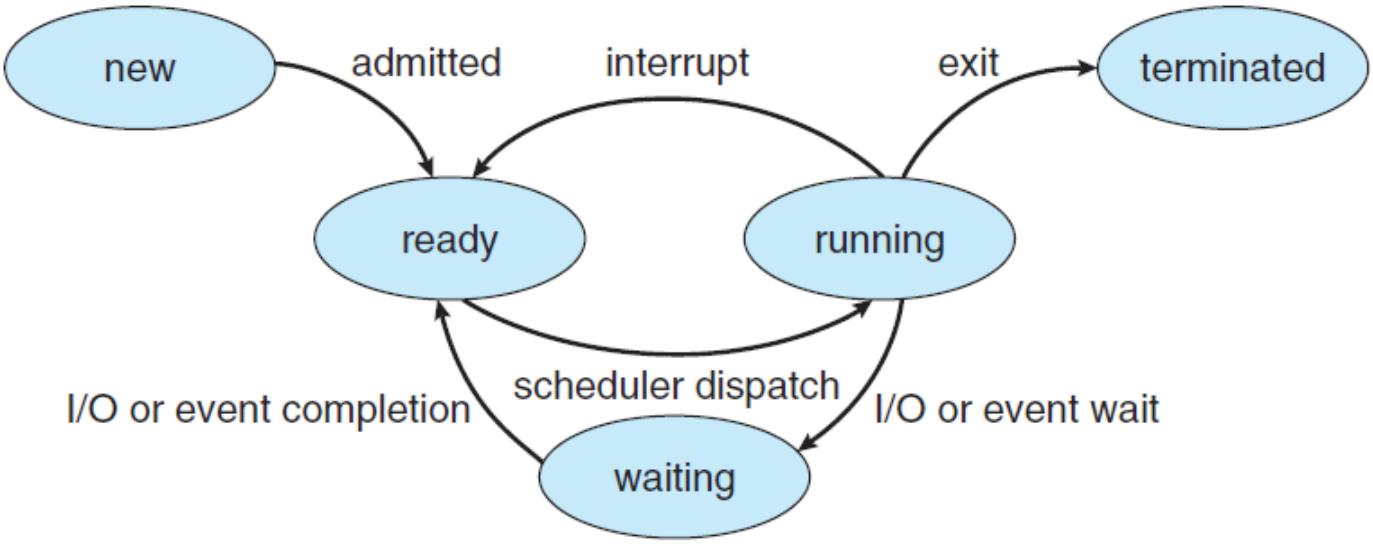
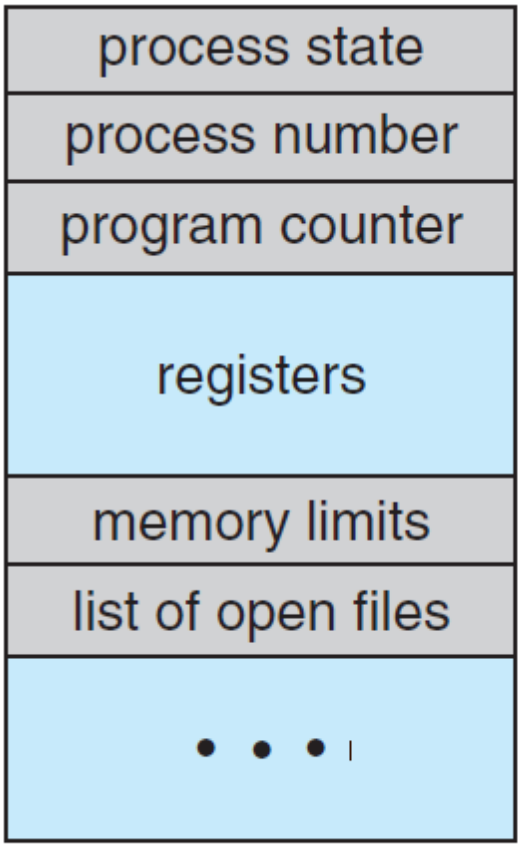


Diagram of process state.

Process Control Block

- Each process is represented in the OS by a process control block. It is also by a process control block.
- It is also known as task control block.
- A process control block contains many pieces of information associated with a specific process.
- It includes the following information's.
  - **Process state:** The state may be new, ready, running, waiting or terminated state.
  - **Program counter:** It indicates the address of the next instruction to be executed for this purpose.
  - **CPU registers:** The registers vary in number & type depending on the computer architecture. It includes accumulators, index registers, stack pointer & general purpose registers, plus any condition- code information must be saved when an interrupt occurs to allow the process to be continued correctly after- ward.
  - **CPU scheduling information:** This information includes process priority pointers to scheduling queues & any other scheduling parameters.
  - **Memory management information:** This information may include such information as the value of the bar & limit registers, the page tables or the segment tables, depending upon the memory system used by the operating system.
  - **Accounting information:** This information includes the amount of CPU and real time used, time limits, account number, job or process numbers and so on.

- **I/O Status Information:** This information includes the list of I/O devices allocated to this process, a list of open files and so on. The PCB simply serves as the repository for any information that may vary from process to process.



Process control block (PCB).

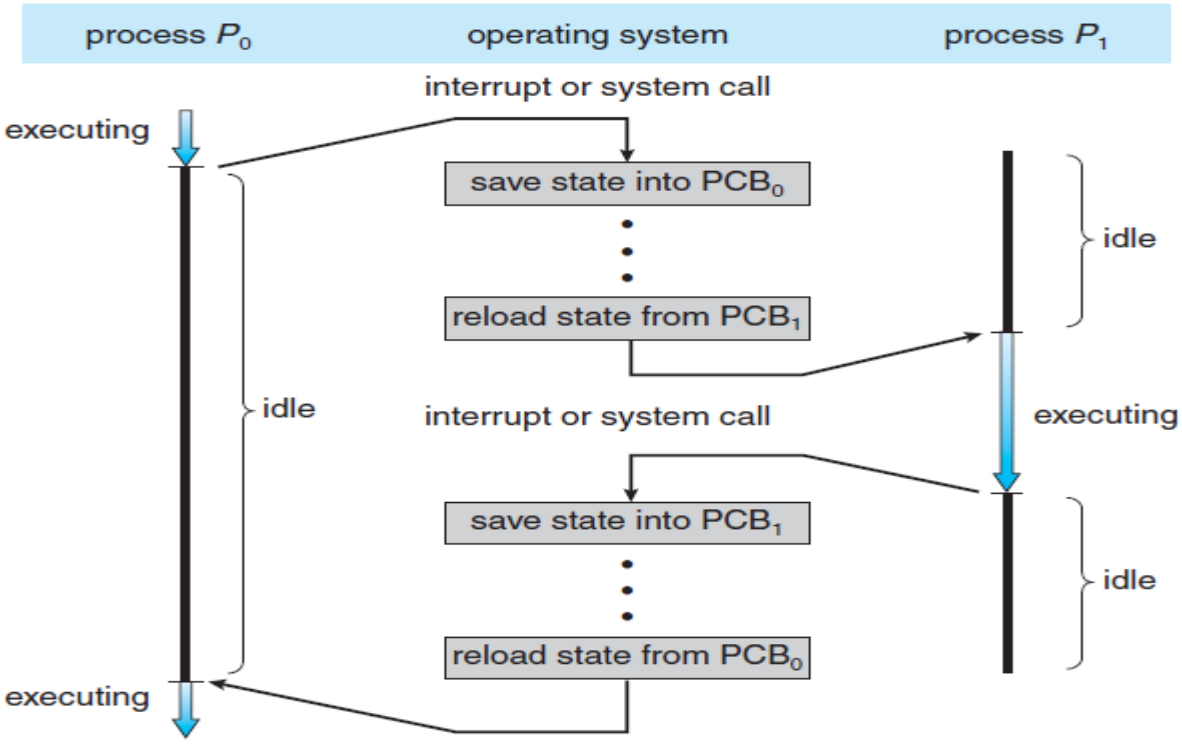


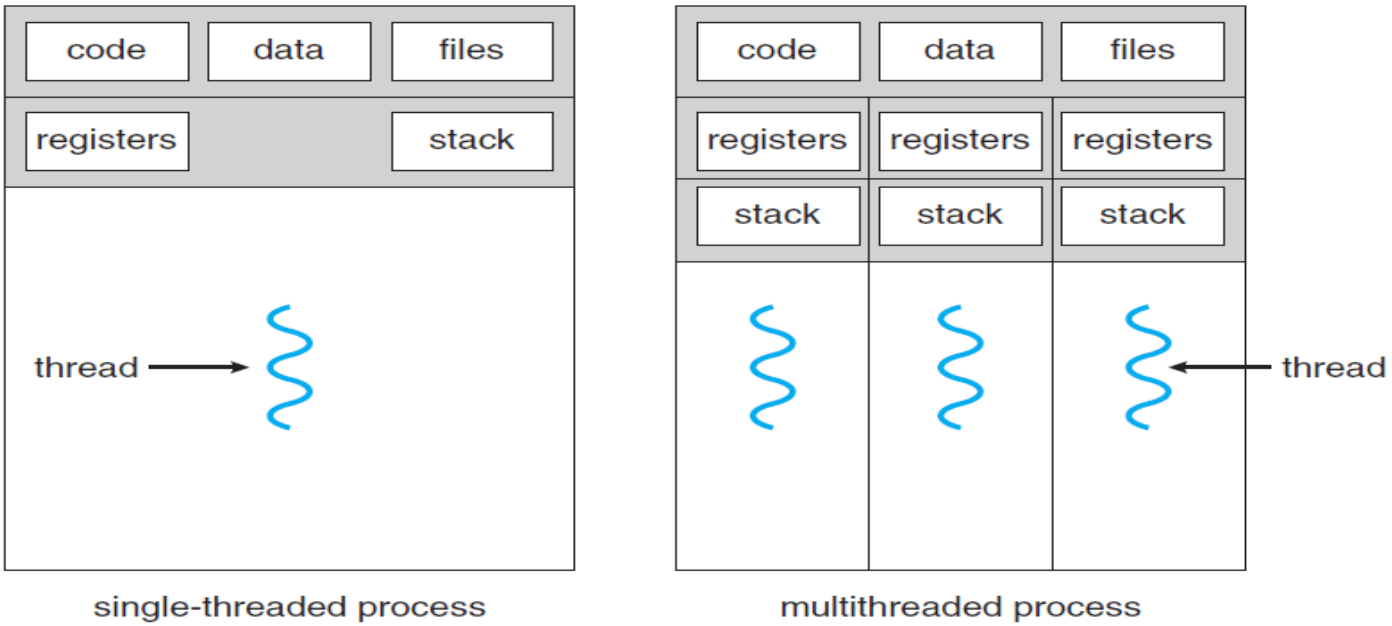
Diagram showing CPU switch from process to process.

Context Switching

1. When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. This is known as context switch.
2. Context-switch time is overhead; the system does no useful work while switching.
3. 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. A typical speed is a few milliseconds.
4. Context switch times are highly dependent on hardware support.

Threads

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- Threads are a popular way to improve application performance through parallelism.
- A thread is sometimes called a light weight process.
- Threads represent a software approach to improving performance of operating system by reducing the over head thread is equivalent to a classical process.
- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control.



Single-threaded and multithreaded processes.

Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Concept of Multithreads

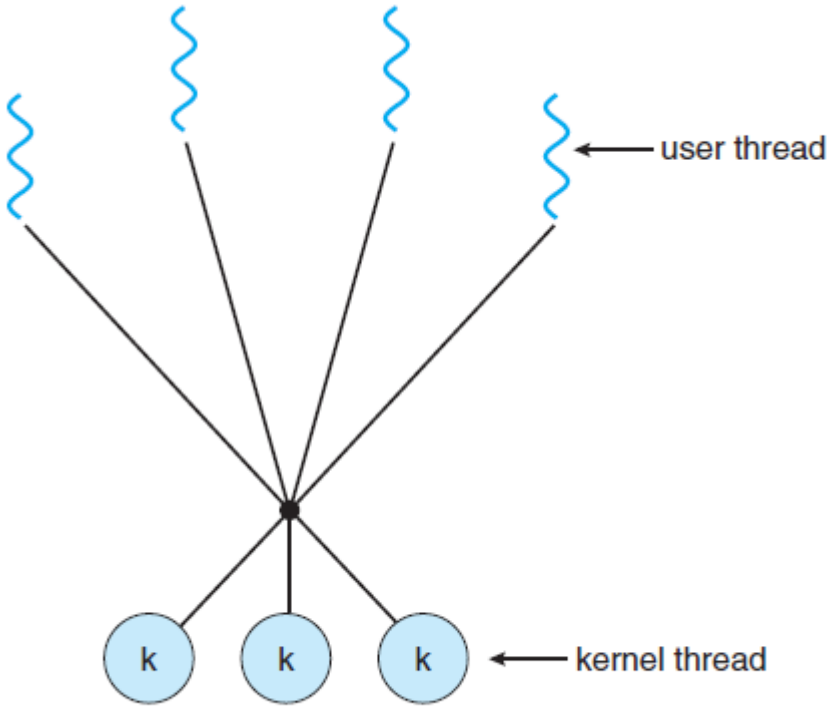
- Some operating systems provide a combined user level thread and Kernel level thread facility.
- Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are **three** types:

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

1. Many to Many Model

- In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers.
- The number of Kernel threads may be specific to either a particular application or a particular machine.



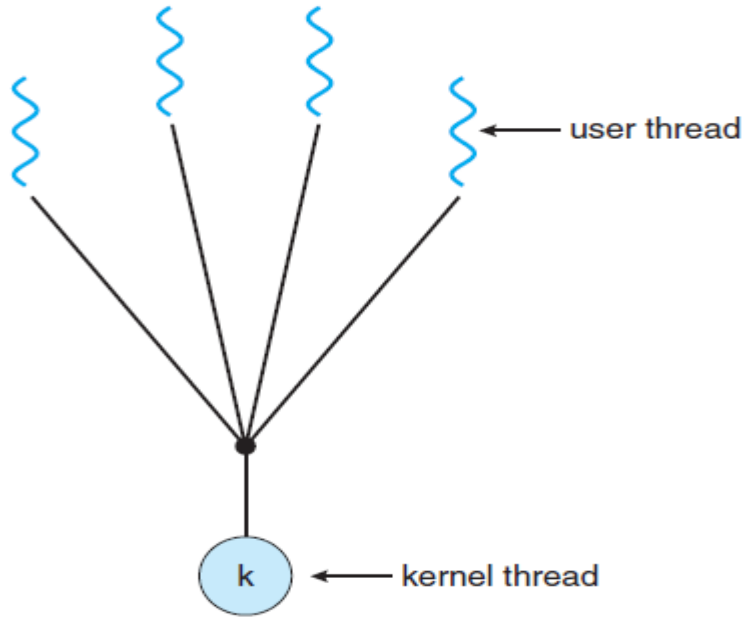
Many-to-many model.

In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.

2. Many to One Model

- Many to one model maps many user level threads to one Kernel level thread.

- Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks.
- Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

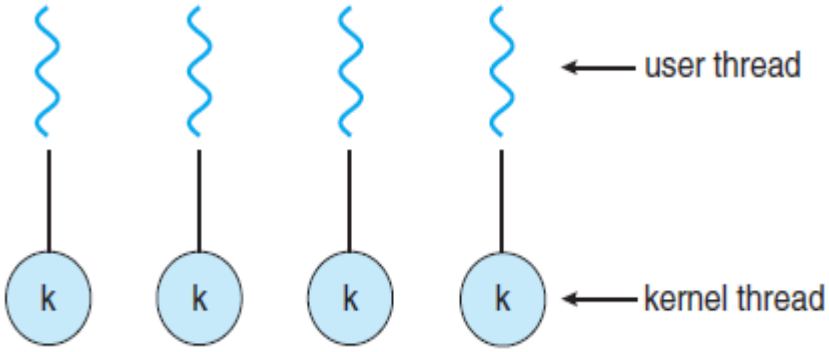


Many-to-one model.

If the user level thread libraries are implemented in the operating system, that system does not support Kernel threads use the many to one relationship modes.

3. One to One Model

- There is one to one relationship of user level thread to the kernel level thread.
- This model provides more concurrency than the many to one model.



One-to-one model.

It allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

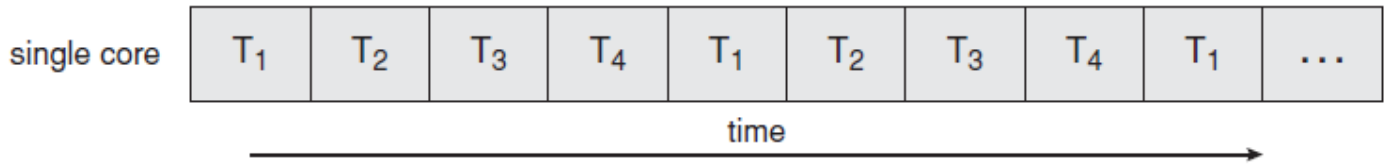
Benefits of Threads

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

1. **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. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.

2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. 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.

3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty time's slower than is creating a thread, and context switching is about five times slower.



Concurrent execution on a single-core system.

4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

Types of Threads

Thread is implemented in two ways:

1. User Level
2. Kernel Level

1. User Level Thread

- In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads.
- The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- The application begins with a single thread and begins running in that thread.
- User level threads are generally fast to create and manage.

Advantage of user level thread over Kernel level thread:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific.
- User level threads are fast to create and manage.

Disadvantages of user level thread:

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

2. Kernel Level Threads

- In Kernel level thread, thread management done by the Kernel.
- There is no thread management code in the application area.
- Kernel threads are supported directly by the operating system.
- Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individual threads within the process.
- Scheduling by the Kernel is done on a thread basis.
- The Kernel performs thread creation, scheduling and management in Kernel space.
- Kernel threads are generally slower to create and manage than the user threads.

Advantages of Kernel level thread:

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

Disadvantages:

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Benefits/Advantages of Thread

- Thread minimizes context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- 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.

Difference between User Level & Kernel Level Thread

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 system support directly to Kernel threads
3	User level thread can run on any operating system	Kernel 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

Difference between Process and Thread

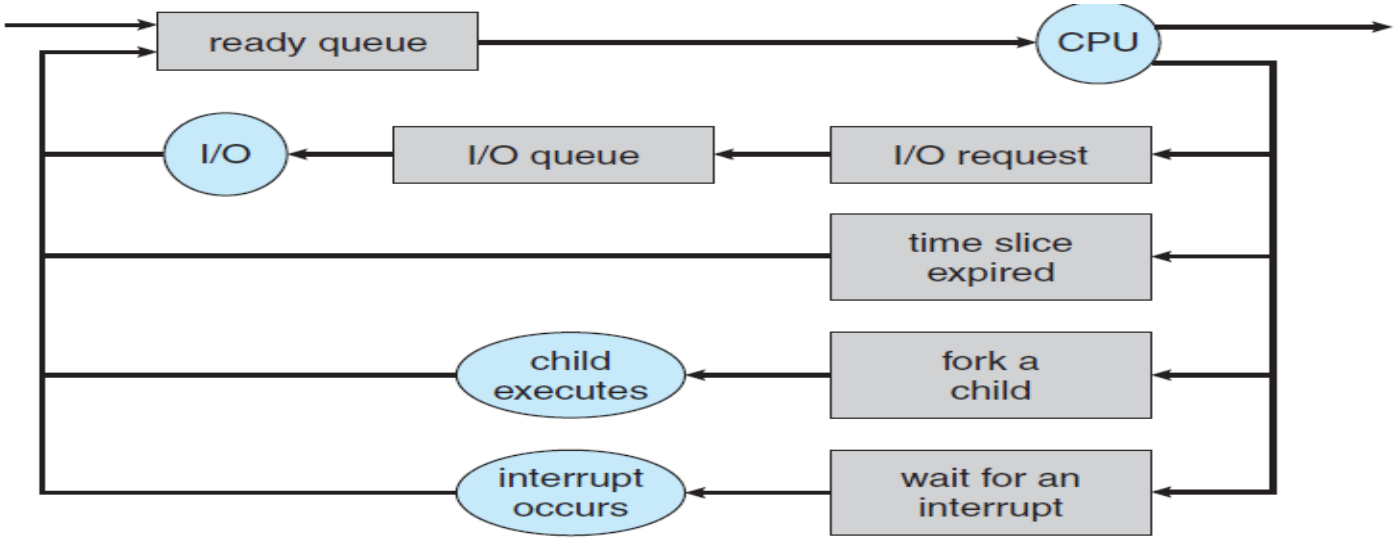
1	Process is called heavy weight process	Thread is called light weight Process
2	Process switching needs interface with operating system	Thread switching does not need to call a operating system and cause an interrupt to the Kernel

3	In multiple process implementation each process executes the same code but has its own memory and file resources	All threads can share same set of open files, 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 process uses fewer resources than multiple redundant process
6	In multiple process each process operates independently of the others	One thread can read, write or even completely wipe out another threads stack

Process Scheduling:

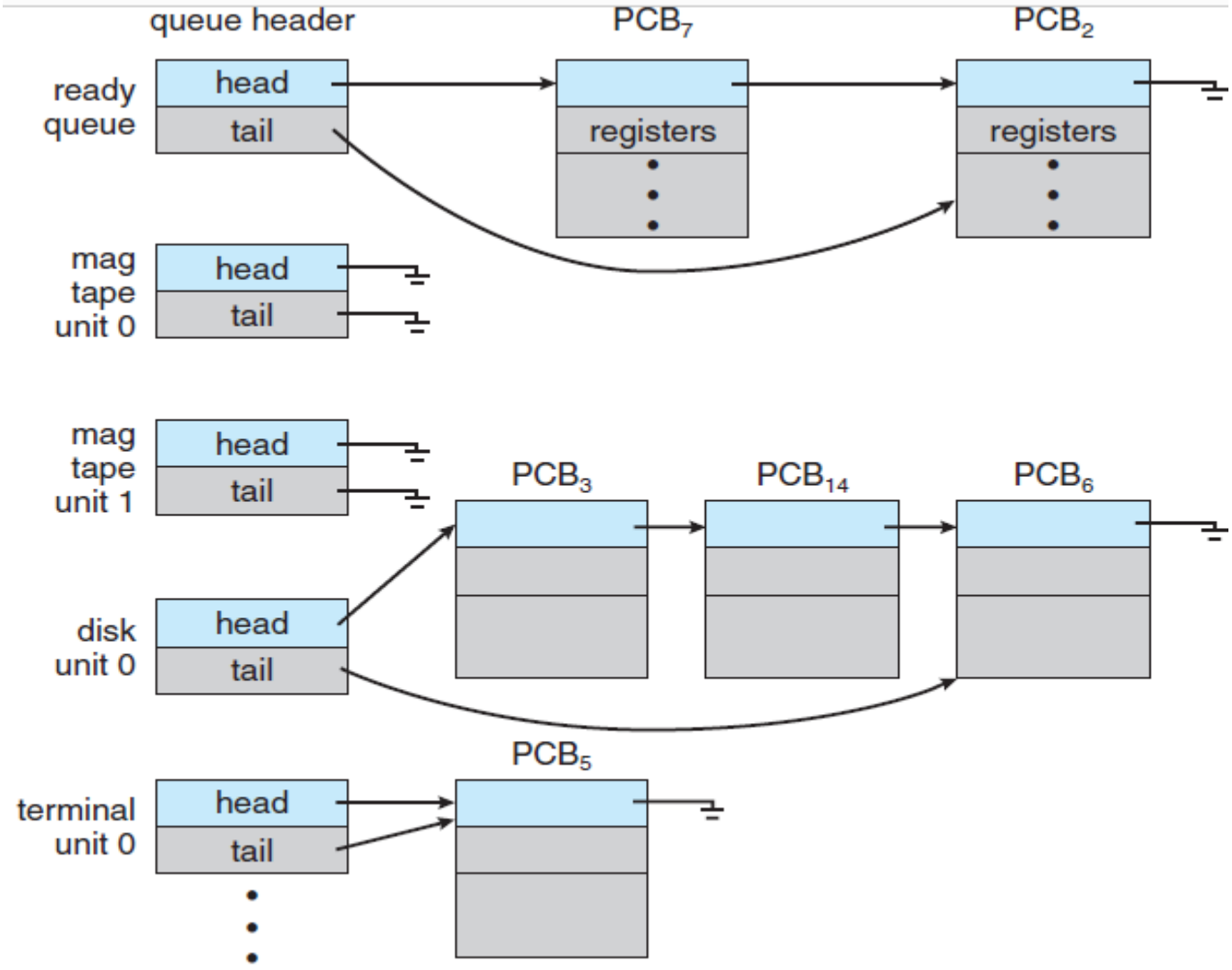
Scheduling is a fundamental function of OS. When a computer is Multiprogrammed, it has multiple processes competing for the CPU at the same time. If only one CPU is available, then a choice has to be made regarding which process to execute next. This decision making process is known as **scheduling** and the part of the OS that makes this choice is called a **scheduler**. The algorithm it uses in making this choice is called **scheduling algorithm**.

Scheduling queues: As processes enter the system, they are put into a job queue. This queue consists of all process in the system. The process that are residing in main memory and are ready & waiting to execute or kept on a list called ready queue.



Queueing-diagram representation of process scheduling.

This queue is generally stored as a linked list. A ready queue header contains pointers to the first & final PCB in the list. The PCB includes a pointer field that points to the next PCB in the ready queue. The lists of processes waiting for a particular I/O device are kept on a list called device queue. Each device has its own device queue. A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution & is given the CPU.



The ready queue and various I/O device queues.

SCHEDULERS:

- A process migrates between the various scheduling queues throughout its life-time purposes. The OS must select for scheduling processes from these queues in some fashion. This selection process is carried out by the appropriate scheduler.
- In a batch system, more processes are submitted and then executed immediately. So these processes are spooled to a mass storage device like disk, where they are kept for later execution.

Types of schedulers:

There are 3 types of schedulers mainly used:

1. Long term scheduler:

- Long term scheduler selects process from the disk & loads them into memory for execution.
- It controls the degree of multi-programming i.e. no. of processes in memory.
- It executes less frequently than other schedulers.
- If the degree of multiprogramming is stable than the average rate of process creation is equal to the average departure rate of processes leaving the system.
- So, the long term scheduler is needed to be invoked only when a process leaves the system.
- Due to longer intervals between executions it can afford to take more time to decide which process should be selected for execution.
- Most processes in the CPU are either I/O bound or CPU bound.
- An I/O bound process (an interactive 'C' program) is one that spends most of its time in I/O operation than it spends in doing I/O operation.
- A CPU bound process is one that spends more of its time in doing computations than I/O operations (complex sorting program).
- It is important that the long term scheduler should select a good mix of I/O bound & CPU bound processes.

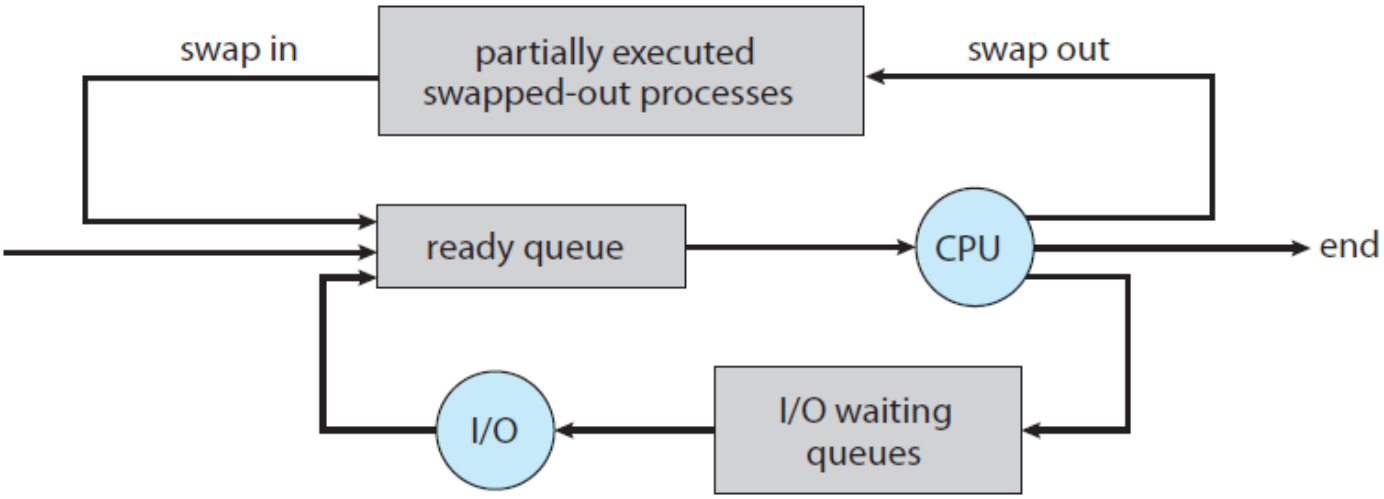
2. Short - term scheduler:

- The short term scheduler selects among the process that are ready to execute & allocates the CPU to one of them.
- The primary distinction between these two schedulers is the frequency of their execution.
- The short-term scheduler must select a new process for the CPU quite frequently.
- It must execute at least one in 100ms. Due to the short duration of time between executions, it must be very fast.

3. Medium - term scheduler:

- Some operating systems introduce an additional intermediate level of scheduling known as medium - term scheduler.
- The main idea behind this scheduler is that sometimes it is advantageous to remove processes from memory & thus reduce the degree of multiprogramming.

- At some later time, the process can be reintroduced into memory & its execution can be continued from where it had left off. This is called as swapping.
- The process is swapped out & swapped in later by medium term scheduler.
- Swapping is necessary to improve the process miss or due to some change in memory requirements, the available memory limit is exceeded which requires some memory to be freed up.



Addition of medium-term scheduling to the queueing diagram.

Scheduling objectives:

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – Number of processes that complete their execution per time unit
3. **Turnaround time** – amount of time to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the ready queue
5. **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

General Goals

Fairness

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement.

Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement

The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time

A scheduler should minimize the response time for interactive user.

Turnaround

A scheduler should minimize the time batch users must wait for an output.

Throughput

A scheduler should maximize the number of jobs processed per unit time.

A little thought will show that some of these goals are contradictory. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

Preemptive Vs Non-preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Non-preemptive Scheduling

A scheduling discipline is non-preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some **characteristics** of non-preemptive scheduling

- In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
- In non-preemptive system, response times are more predictable because incoming high priority jobs cannot displace waiting jobs.

- In non-preemptive scheduling, a scheduler executes jobs in the following two situations.
    - When a process switches from running state to the waiting state.
    - When a process terminates.

Preemptive Scheduling

- A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.
- The strategy of allowing processes that are logically runnable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

CPU scheduling algorithms:

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

Following are some scheduling algorithms we will study.

1. FCFS Scheduling.
2. Round Robin Scheduling.
3. SJF Scheduling.
4. SRT Scheduling.
5. Priority Scheduling.
6. Multilevel Queue Scheduling.
7. Multilevel Feedback Queue Scheduling.

1. First Come, First Served Scheduling (FCFS) Algorithm:

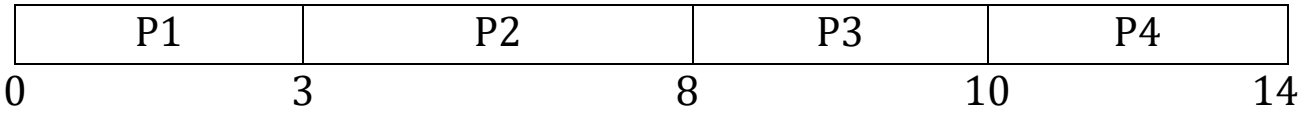
- This is the simplest CPU scheduling algorithm. In this scheme, the process which requests the CPU first, that is allocated to the CPU first.
- The implementation of the FCFS algorithm is easily managed with a FIFO queue.
- When a process enters the ready queue its PCB is linked onto the rear of the queue. The average waiting time under FCFS policy is quiet long.

Consider the following example:

Process	CPU time
P1	3
P2	5
P3	2
P4	4

Using FCFS algorithm find the average waiting time and average turnaround time if the order is P1, P2, P3, P4.

Solution: If the process arrived in the order P1, P2, P3, P4 then according to the FCFS the **Gantt chart** will be:



Waiting time for process P1 = 0, P2 = 3, P3 = 8, P4 = 10
Turnaround time for process P1 = 0 + 3 = 3, P2 = 3 + 5 = 8,
P3 = 8 + 2 = 10, P4 = 10 + 4 =14.
Average waiting time = (0 + 3 + 8 + 10)/4
= 21/4
= 5.25
Average Turnaround time = (3 + 8 + 10 + 14)/4
= 35/4
= 8.75
The FCFS algorithm is non preemptive means once the CPU has been allocated to a process then the process keeps the CPU until the release the CPU either by terminating or requesting I/O.

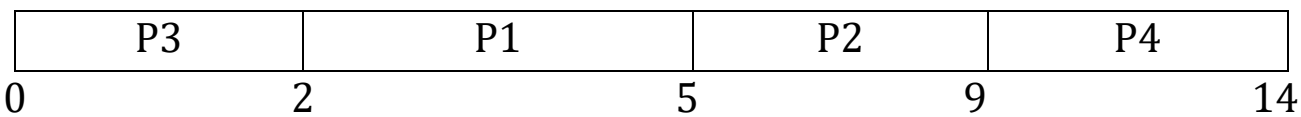
2. Shortest Job First Scheduling (SJF) Algorithm:

- Other name of this algorithm is Shortest-Process-Next (SPN).
- This algorithm associates with each process if the CPU is available.
- This scheduling is also known as shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of the process rather than its total length.

Consider the following example:

Process	CPU time
P1	3
P2	5
P3	2
P4	4

Solution: According to the SJF the Gantt chart will be



Waiting time for process P1 = 0, P2 = 2, P3 = 5, P4 = 9
Turnaround time for process P3 = 0 + 2 = 2, P1 = 2 + 3 = 5,
P4 = 5 + 4 = 9, P2 = 9 + 5 =14.

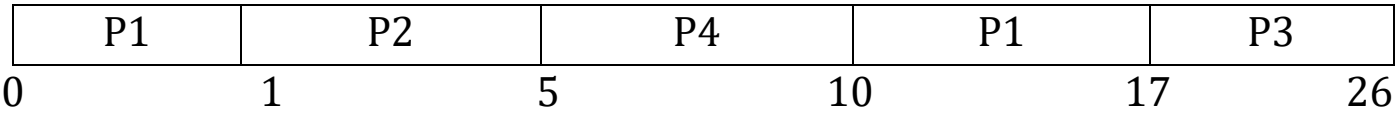
Average waiting time = (0 + 2 + 5 + 9)/4 = 16/4 = 4
Average Turnaround time = (2 + 5 + 9 + 14)/4 = 30/4 = 7.5

The SJF algorithm may be either preemptive or non preemptive algorithm.

The **preemptive SJF** is also known as shortest remaining time first.
Consider the following example.

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

In this case the Gantt chart will be



The Waiting time for process

P1 = 10 - 1 = 9
P2 = 1 – 1 = 0
P3 = 17 – 2 = 15
P4 = 5 – 3 = 2

The Average Waiting time = (9 + 0 + 15 + 2)/4 = 26/4 = 6.5

3. Priority Scheduling Algorithm:

- In this scheduling a priority number (integer) is associated with each process and the CPU is allocated to the process with the highest priority (smallest integer, highest priority).
  - Preemptive
  - Non-preemptive
- Equal priority processes are scheduled in FCFS manner.
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem {Starvation – low priority processes may never execute.
- Solution { Aging – as time progresses increase the priority of the process
- Priority can be defined either internally or externally.

Examples of Internal priorities are

- Time limits.

- Memory requirements.
- File requirements, for example, number of open files.
- CPU Vs I/O requirements.

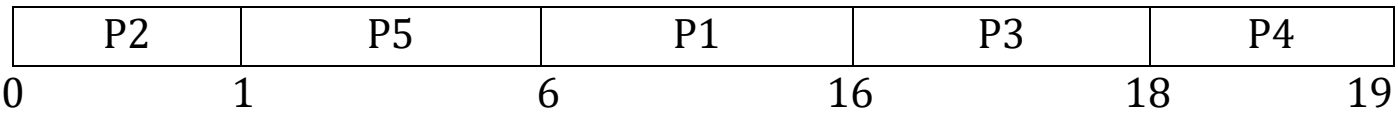
Externally defined priorities are set by criteria that are external to operating system such as

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Consider the following example:

Process	Arrival Time	CPU Time
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

According to the priority scheduling the Gantt chart will be



The Waiting time for process

P1 = 6
P2 = 0
P3 = 16
P4 = 18
P5 = 1

The Average Waiting time = (0 + 1 + 6 + 16 + 18)/5 = 41/5 = 8.2

4. Round Robin Scheduling Algorithm:

- This type of algorithm is designed only for the time sharing system.
- It is similar to FCFS scheduling with preemption condition to switch between processes.
- A small unit of time called quantum time or time slice is used to switch between the processes.
- The average waiting time under the round robin policy is quiet long.

Consider the following example:

Process	CPU time
P1	3
P2	5
P3	2
P4	4

Time Slice = 1 millisecond.

P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P4	P2	P4	P2

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

The Waiting Time for process

P1 = 0 + (4 – 1) + (8 – 5)
= 0 + 3 + 3
= 6

P2 = 1 + (5 – 2) + (9 – 6) + (11 – 10) + (13 – 12)
= 1 + 3 + 3 + 1 + 1
= 9

P3 = 2 + (6 – 3)
= 2 + 3
= 5

P4 = 3 + (7 – 4) + (10 – 8) + (12 – 11)
= 3 + 3 + 2 + 1
= 9

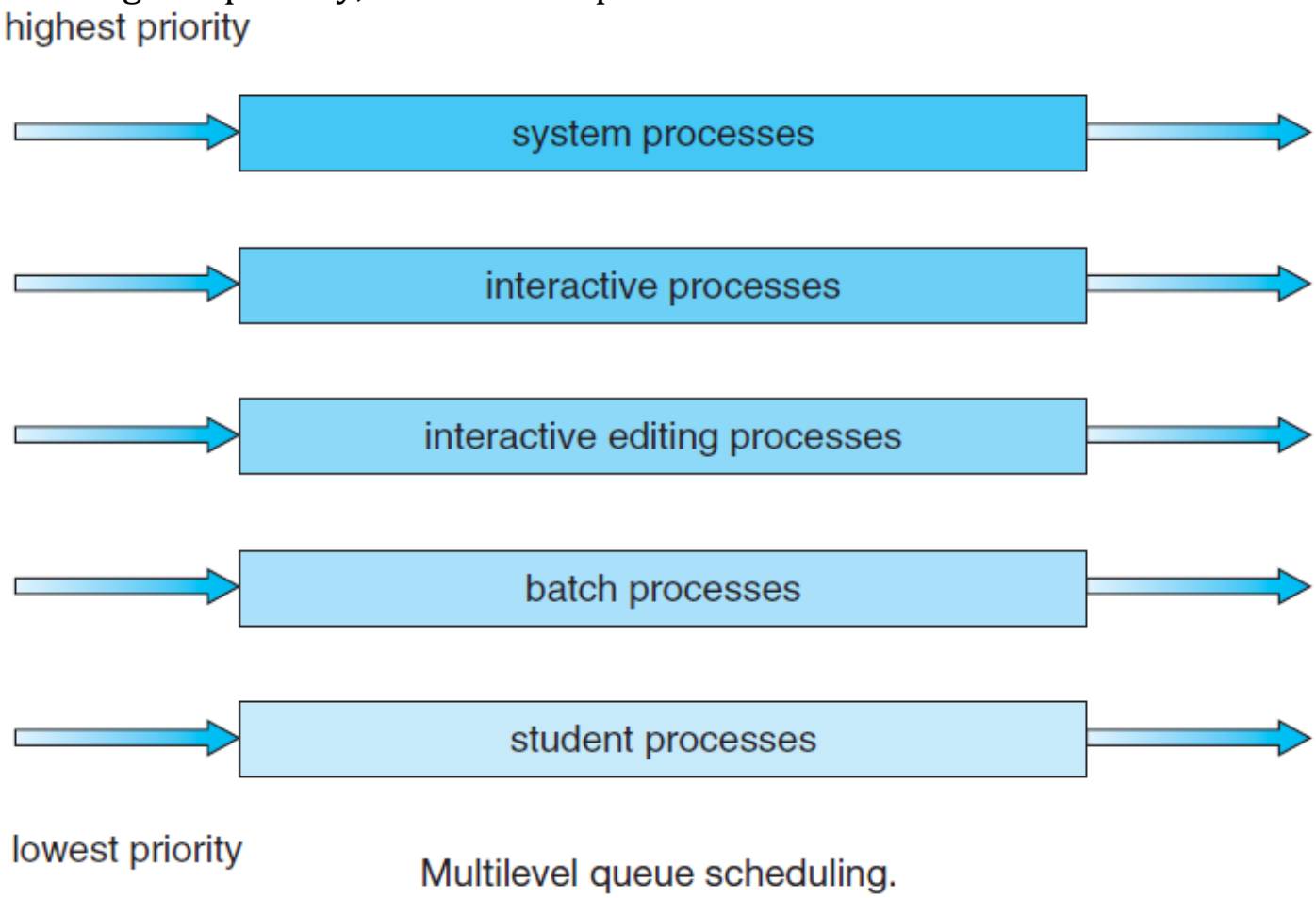
The Average Waiting time = (6 + 9 + 5 + 9)/4
= 7.2

5. Shortest-Remaining-Time (SRT) Scheduling

- The SRT is the preemptive counterpart of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- In SJF scheme, once a job begins executing, it run to completion.
- In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.
- The algorithm SRT has higher overhead than its counterpart SJF.
- The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.
- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

6. Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the ready queue in several separate queues, for instance, in a multilevel queue scheduling processes are permanently assigned to one queues.
- The processes are permanently assigned to one another, based on some property of the process, such as Memory size, Process priority, Process type.
- Algorithm choose the process from the occupied queue that has the highest priority, and run that process either



Multilevel queue scheduling.

7. Multilevel Feedback Queue Scheduling.

- Multilevel feedback queue-scheduling algorithm allows a process to move between queues.
- It uses many ready queues and associates a different priority with each queue.
- The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively or non-preemptively.
- If the process uses too much CPU time it will moved to a lower-priority queue.

- Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest priority queue.

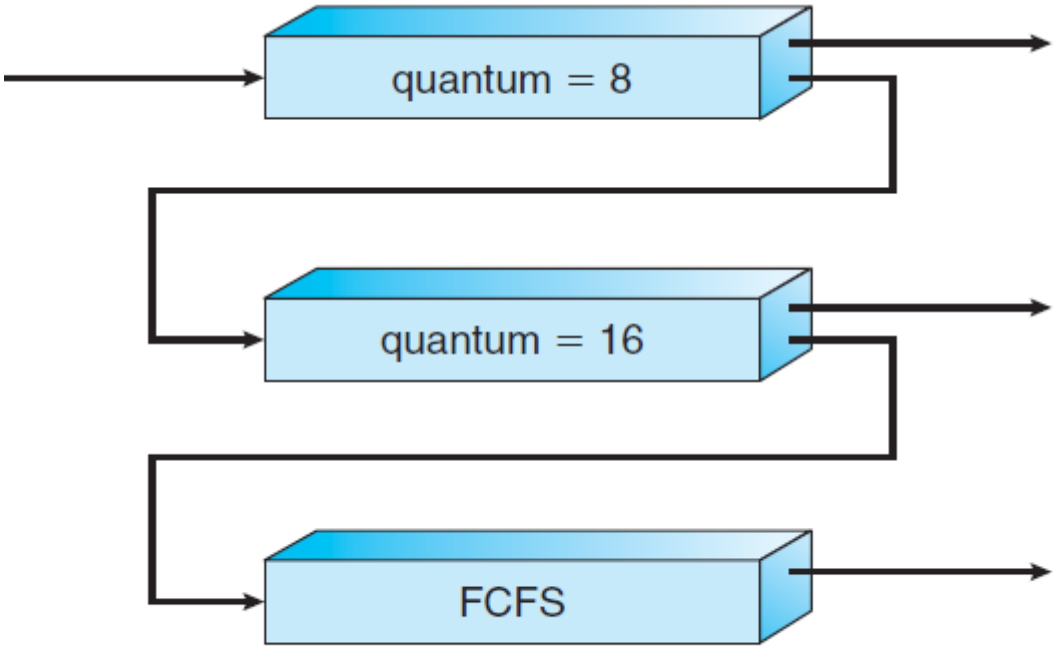
Note that this form of aging prevents starvation.

Example:

- A process entering the ready queue is placed in queue 0.
- If it does not finish within 8 milliseconds time, it is moved to the tail of queue 1.
- If it does not complete, it is preempted and placed into queue 2.
- Processes in queue 2 run on a FCFS basis, only when 2 run on a FCFS basis queue, only when queue 0 and queue 1 are empty.

Three queues:

Q0 – RR with time quantum 8 milliseconds
Q1 – RR time quantum 16 milliseconds
Q2 – FCFS

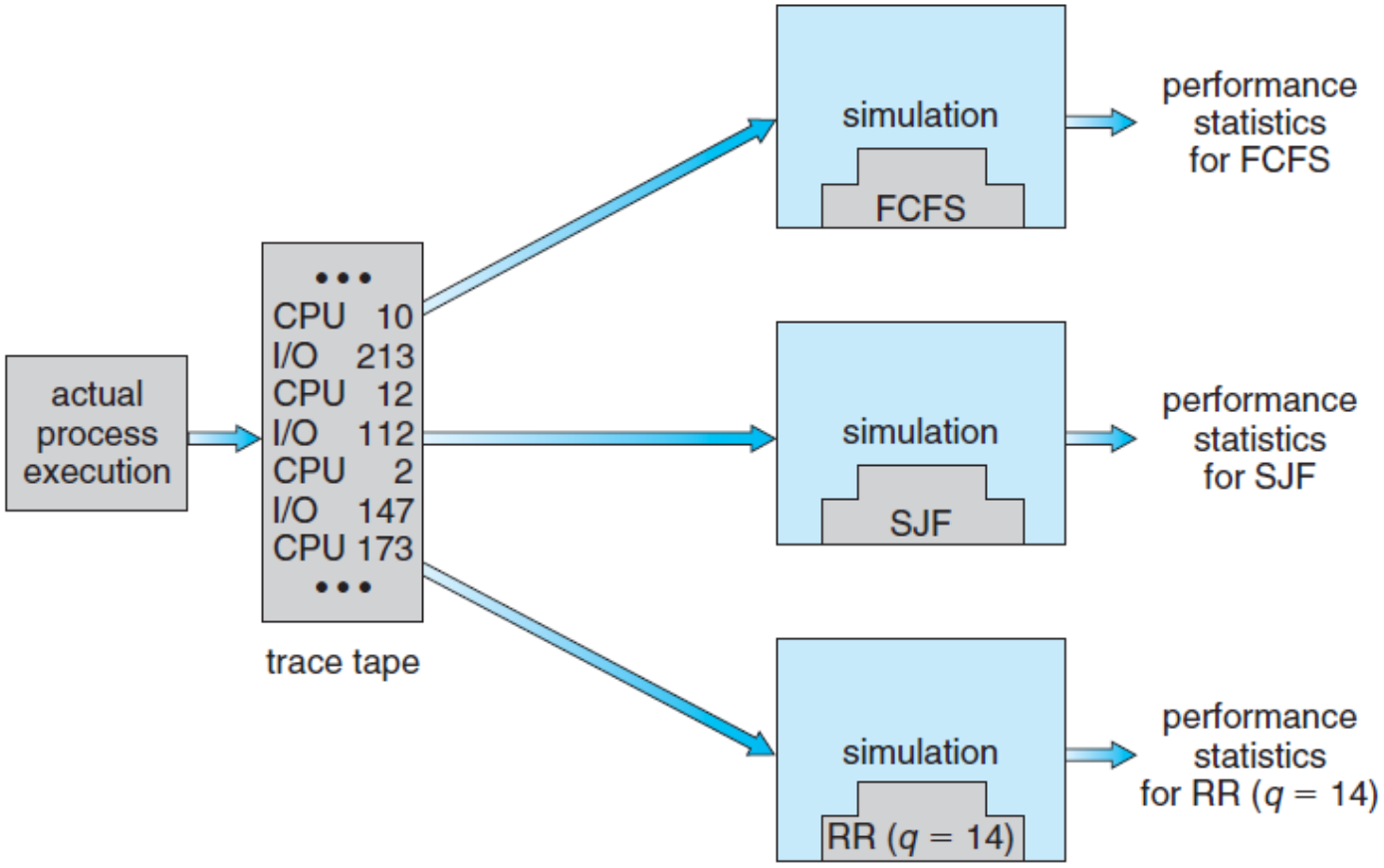


Multilevel feedback queues.

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

Performance evaluation of the scheduling:



Evaluation of CPU schedulers by simulation.