PROCESSES AND CPU PROCESSING

PROCESS CONCEPT

- A ***process*** is a program in execution. A program by itself is not a process. A program is a *passive entity*, such as the contents of a file stored on disk while a process is an *active entity*.

- A computer system consists of a collection of processes: ***operating-system processes*** execute system code, and ***user processes*** execute user code.

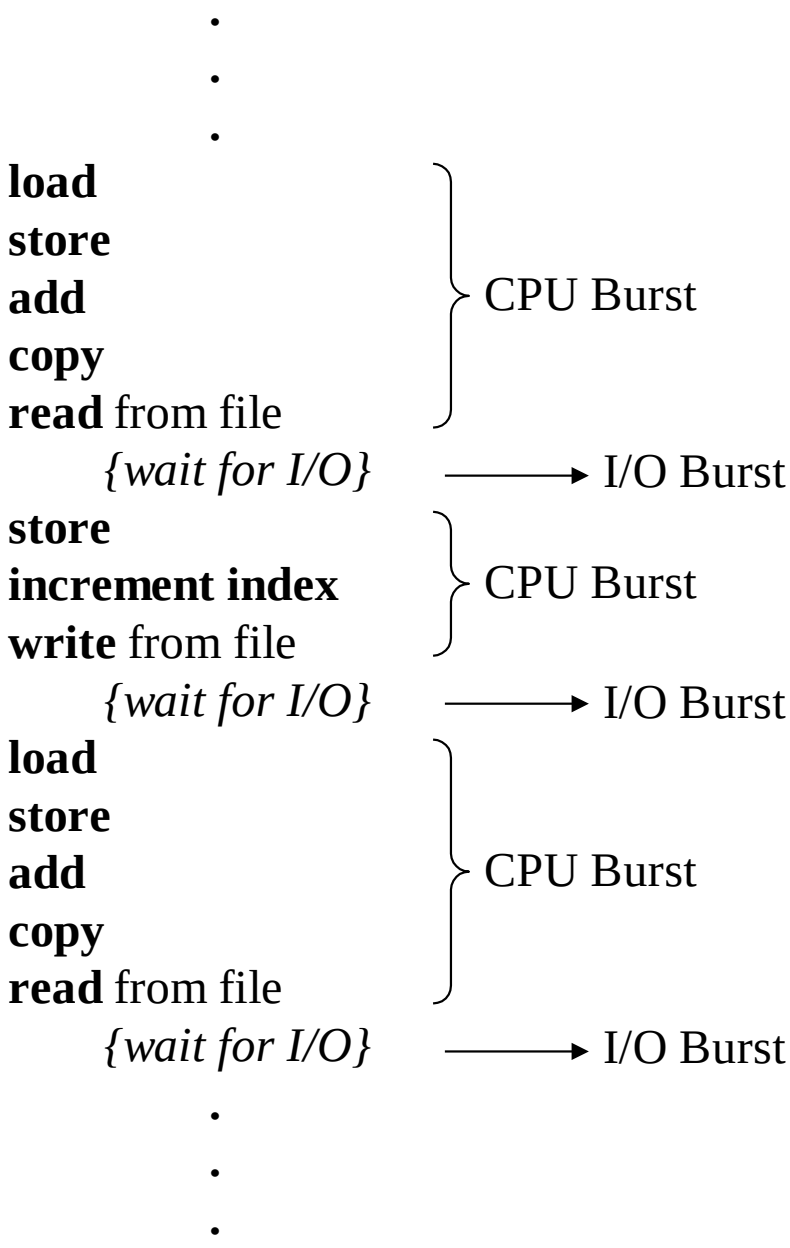
- Although several processes may be associated with the same program, they are nevertheless considered separate execution sequences.

- All processes can potentially execute concurrently with the CPU (or CPUs) multiplexing among them (***time sharing***).

- A process is actually a cycle of CPU execution (***CPU burst***) and I/O wait (***I/O burst***). Processes alternate back and forth 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 last CPU burst will end with a system request to terminate execution.

Example:



☞ Usually, there is a large number of short CPU bursts, or there is a small number of long CPU bursts. An ***I/O-bound program*** would typically have many very short CPU bursts. A ***CPU-bound program*** might have a few very long CPU bursts.

☞ A process is more than the program code, which is sometimes known as the ***text section***. It also includes:

1. the ***current activity***, as represented by the contents the program counter and the contents the CPU's registers.

2. the ***process stack***, which contains temporary data (such as subroutine parameters, return addresses, and local variables).

3. a ***data section***, which contains global variables.

☞ As a process executes, it changes ***state***. The current activity of a process partly defines its state. Each sequential process may be in one of following states:

1. ***New***. The process is being created.

2. ***Running***. The CPU is executing its instructions.

3. ***Waiting***. The process is waiting for some event to occur (such as an I/O completion).

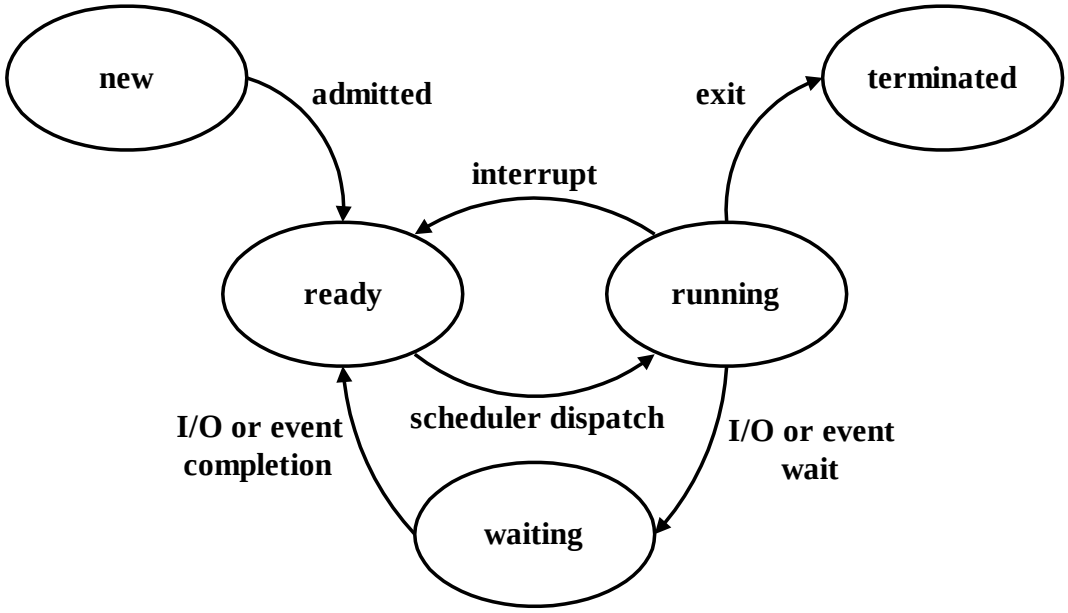
4. ***Ready***. The process is waiting for the OS to assign a processor to it.

5. ***Terminated***. The process has finished execution.

These names are arbitrary, and vary between operating systems. The states that they represent are found on all systems, however. Certain operating systems also distinguish among more finely delineating process states.

It is important to realize that only one process can be *running* at any instant. Many processes may be *ready* and *waiting*, however.

☞ Process state diagram:



☞ Each process is represented in the operating system by a ***process control block*** (PCB) – also called a ***task control block***. A PCB is a data block or record containing many pieces of the information associated with a specific process including:

1. ***Process state***. The state may be new, ready, running, waiting, or halted.

2. ***Program Counter***. The program counter indicates the address of the next instruction to be executed for this process.

3. ***CPU Registers***. These include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

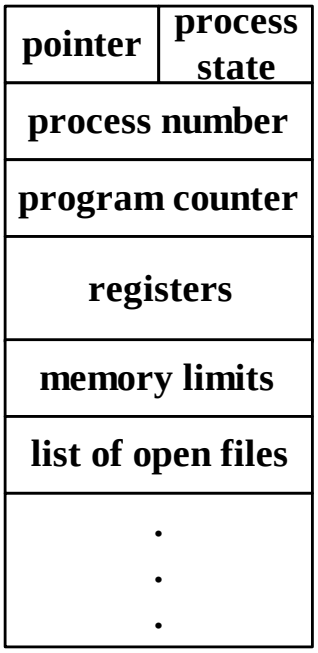
4. ***CPU Scheduling Information***. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

5. ***Memory Management Information***. This information includes limit registers or page tables.

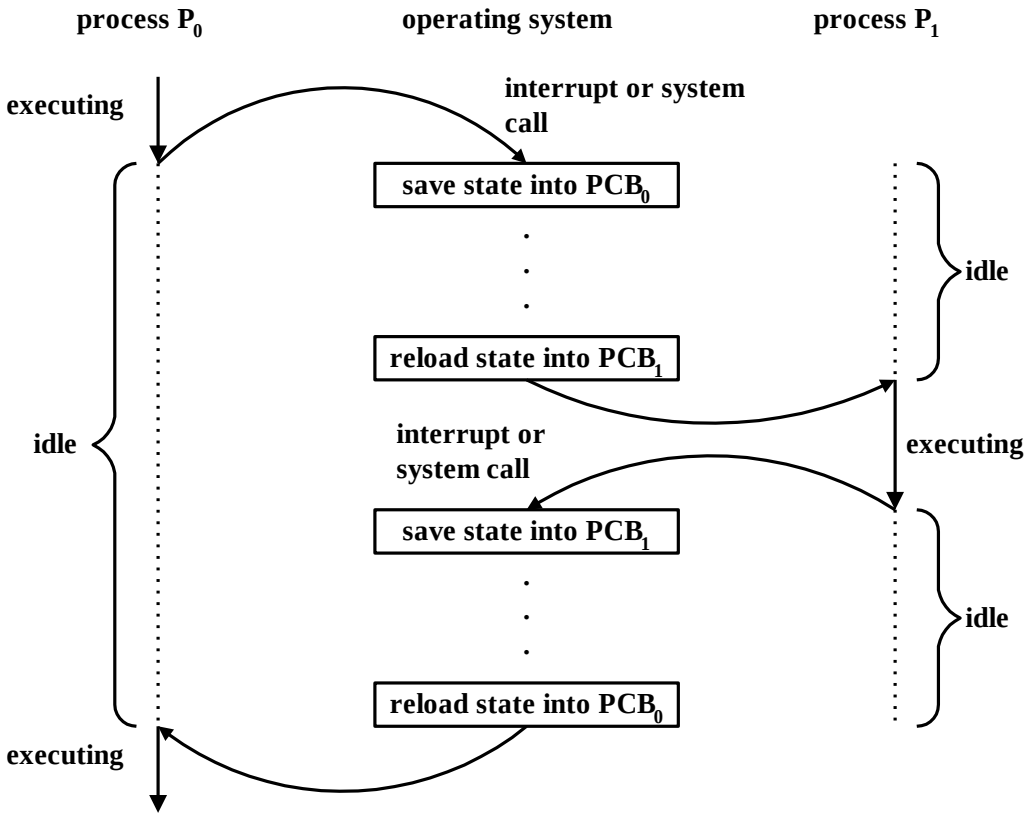
6. ***Accounting Information***. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

7. ***I/O Status Information***. This information includes outstanding I/O requests, I/O devices (such as disks) 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.



☞ Example of the CPU being switched from one process to another.



CONCURRENT PROCESSES

- ☞ The processes in the system can execute concurrently; that is, many processes may be multitasked on a CPU.

- ☞ A process may create several new processes, via a ***create-process*** system call, during the course of execution.

- ☞ The creating process is the ***parent*** process whereas the new processes are the ***children*** of that process. Each of these new processes may in turn create other processes, forming a *tree of processes*.

- ☞ When a process creates a subprocess, the subprocess may be able to obtain its resources directly from the operating system or it may use a subset of the resources of the parent process. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many processes.

- ☞ When a process creates a new process, two common implementations exist in terms of execution:

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

    2. The parent waits until all its children have terminated.

- ☞ A process terminates when it finishes its last statement and asks the operating system to delete it using the ***exit*** system call.

- ☞ A parent may terminate the execution of one of its children for a variety of reason, such as

    1. The child has exceeded its usage of some of the resources it has been allocated.

    2. The task assigned to the child is no longer required.

    3. The parent is exiting, and the OS does not allow a child to continue if its parent terminates. In such systems, if a process terminates, then all its children must also be terminated by the operating system. This phenomenon is referred to as ***cascading termination***.

- ☞ The concurrent processes executing in the operating system may either be ***independent processes*** or ***cooperating processes***.

- A process is independent if it cannot affect or be affected by the other processes. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent.

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

- Concurrent execution of cooperating process requires mechanisms that allow processes to communicate with one another and to synchronize their actions.

- Take note that a process is a program that performs a *single thread of execution*. For example, if 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 one time.

  In order to accomplish several tasks, a process has to create other processes. Creating and maintaining processes incurs overhead particularly if resources have to be allocated to the newly-created processes.

THREADS

- A ***thread*** is a separate part of a process. A process can consist of several threads, each of which execute separately. For example, one thread could handle screen refresh and drawing, another thread printing, another thread the mouse and keyboard.

- In other words, a thread is a portion of a program that can run independently of and concurrently with other portions of the program.

- Many software packages that run on modern desktop PCs are ***multithreaded***. An application typically is implemented as a separate process with several threads of control.

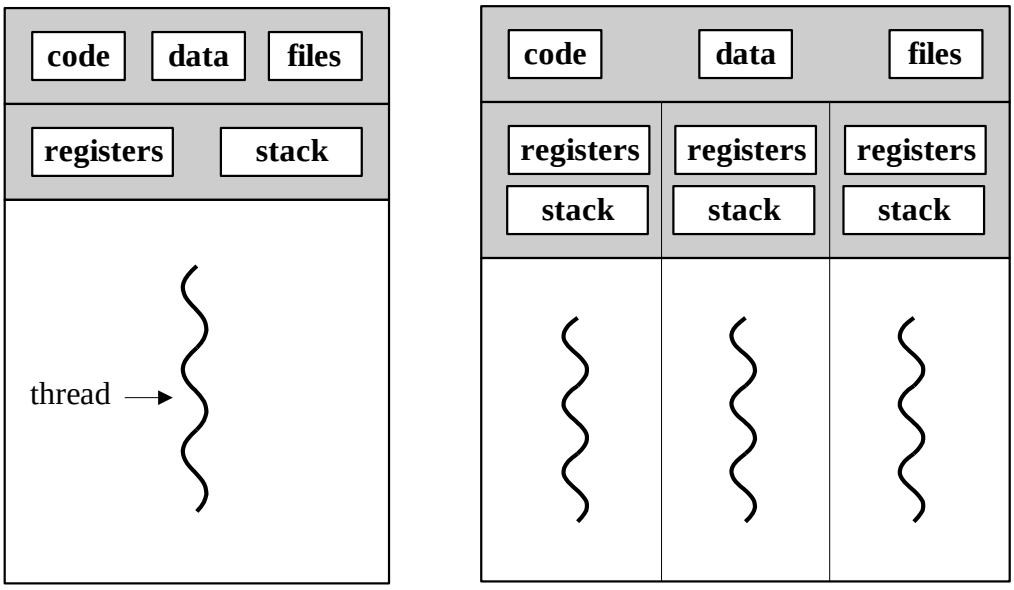
  Examples:

  A web browser might have one thread to display images or text while another thread retrieves data from the network.

  A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

☞ A traditional process has a single thread of control. A traditional process is therefore called a ***heavyweight process***.

☞ Single- and multithreaded processes



Single-threaded Process **Multithreaded Process**

A thread shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files. Each thread has its own register set and stack.

A thread is sometimes called a ***lightweight process***.

☞ Benefits of Multithreaded Programming

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, hereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

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

3. ***Economy***. Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.

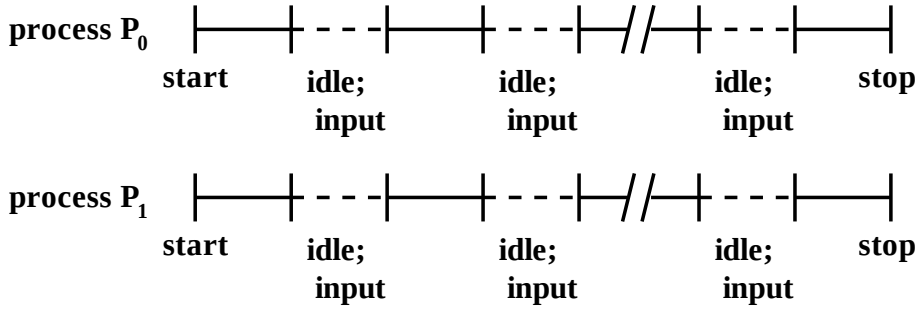
4. ***Utilization of Multiprocessor Architectures***. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor.

SCHEDULING CONCEPTS

- ☞ The objective of ***multiprogramming*** is to have some process running at all times, to maximize CPU utilization. Multiprogramming also increases throughput, which is the amount of work the system accomplishes in a given time interval (for example, 17 processes per minute).

  Example:

  Given two processes, P_0 and P_1.

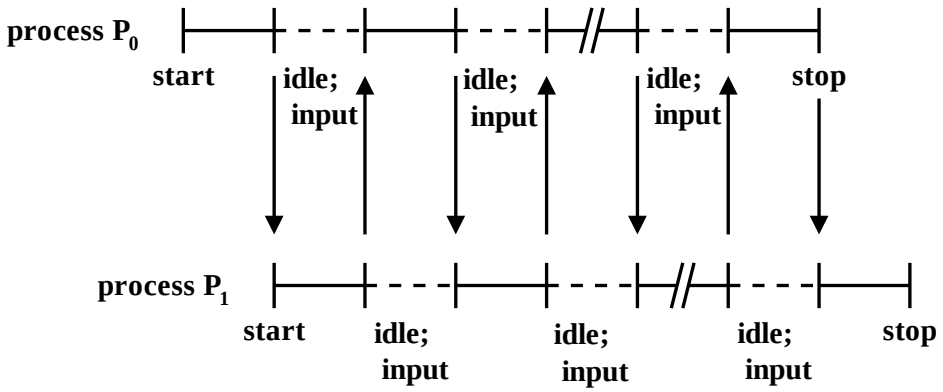


  If the system runs the two processes sequentially, then CPU utilization is only 50%.

- ☞ The idea of multiprogramming is if one process is in the *waiting* state, then another process which is in the *ready* state goes to the *running* state.

Example:

Applying multiprogramming to the two processes, P_0 and P_1,



then CPU utilization increases to 100%.

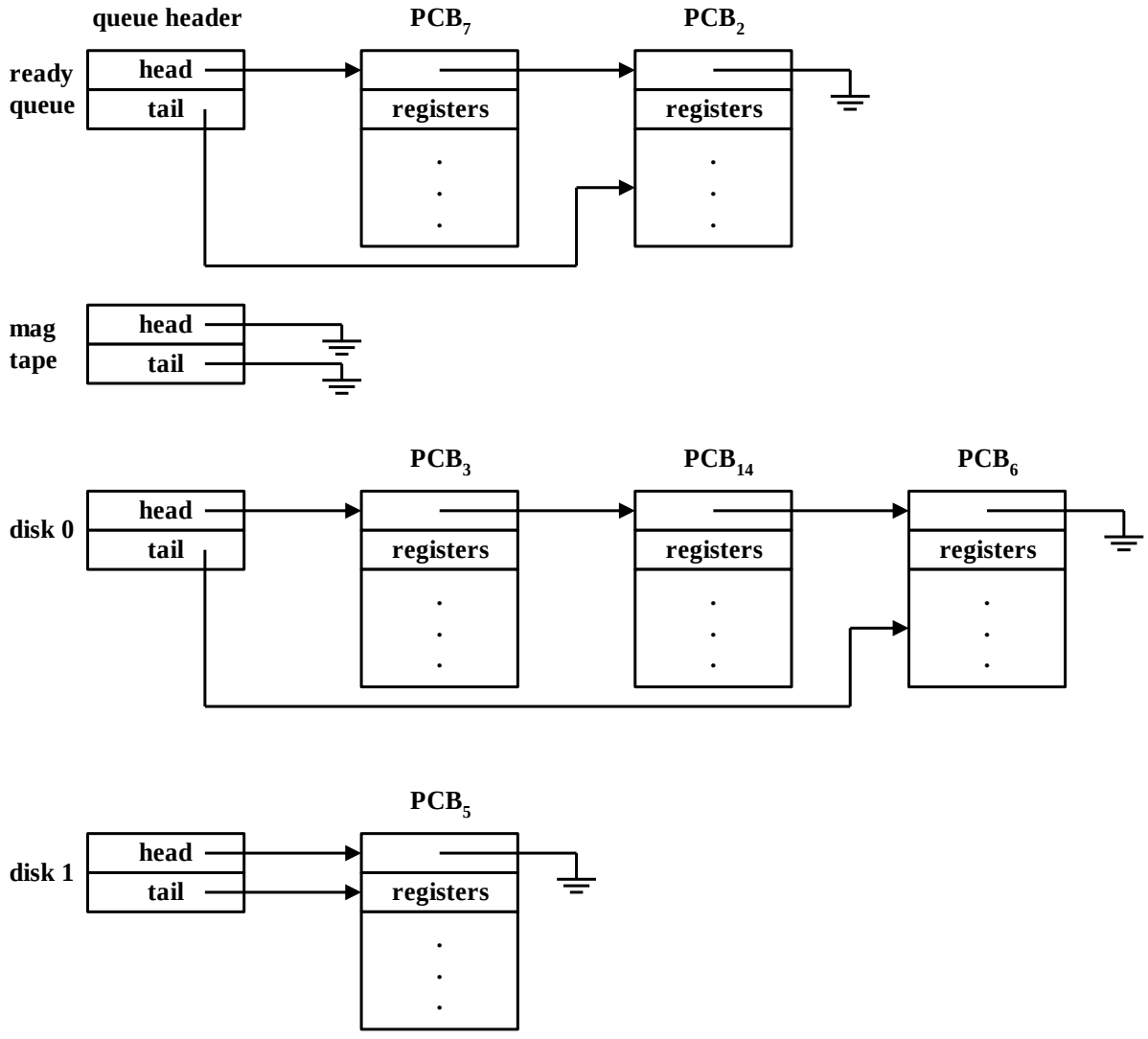
- As processes enter the system, they are put into a ***job queue***. This queue consists of all processes on the disk that are waiting to be brought into memory for execution.

- The processes that are residing in main memory and are ready and waiting to execute are kept on another queue which is the ***ready queue***.

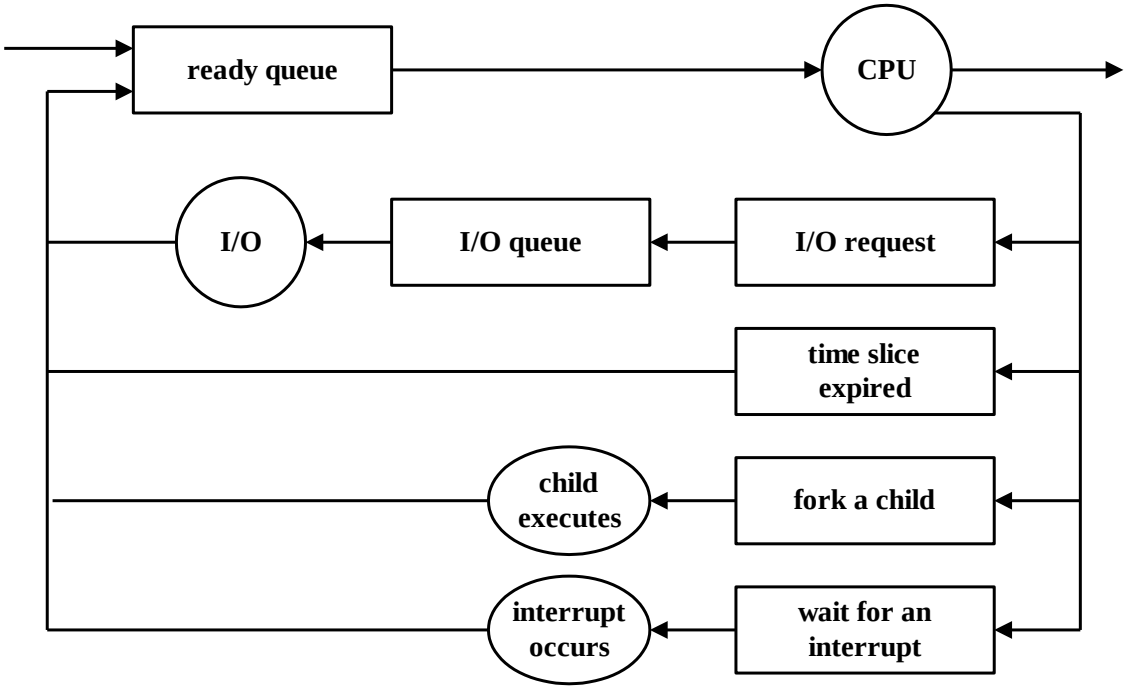
- The ready queue is generally stored as a linked list. Each node in this linked list is a PCB. Therefore, each PCB has a pointer field that points to the next process in the ready queue. A ready-queue header will contain pointers to the first and last PCB's in the list.

☞ The are also other queues in the system. When a process needs an I/O device which is currently in use by another process, then the former has to wait in the ***device queue***. Each device in the system has its own device queue.

Example:



☞ Queuing-diagram representation of process scheduling



A new process initially goes in the ready queue. It waits in this queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of several events could occur:

1. The process could issue an I/O request, and then be placed in an I/O queue.
2. The process could create a new subprocess and wait for its termination.
3. The process could be forcibly removed from the CPU, as a result of an interrupt, and put back in the ready queue.

- A process migrates between the various scheduling queues throughout its lifetime. The operating system must select processes from these queues in some fashion. The selection process is the responsibility of the appropriate ***scheduler***.

- The ***long-term scheduler*** (or ***job scheduler***) selects processes from the secondary storage and loads them into memory for execution. The ***short-term scheduler*** (or ***CPU scheduler***) selects a process 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. Because of the brief time between executions, the short-term scheduler must be very fast.

- The long-term scheduler executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the ***degree of multiprogramming*** – the number of processes in memory. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

  The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes.

☞ Some operating systems may have a ***medium-term scheduler***. This removes (swaps out) certain processes from memory to lessen the degree of multiprogramming (particularly when thrashing occurs). At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called ***swapping***.

☞ Switching the CPU to another process requires some time to save the state of the old process and loading the saved state for the new process. This task is known as ***context switch***.

Context-switch time is pure overhead, because the system does no useful work while switching and should therefore be minimized.

Context-switch time varies from machine to machine, depending on the memory speed, the number of registers to be copied, and the existence of special instructions (such as a single instruction to load or store all registers).

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch simply includes changing the pointer to the current register set.

CPU SCHEDULER

- Whenever the CPU becomes idle, the operating system (particularly the CPU scheduler) must select one of the processes in the ready queue for execution.

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

    1. When a process switches from the running state to the waiting state (for example, I/O request, invocation of wait for the termination of one of the child processes)

    2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).

    3. When a process switches from the waiting state to the ready state (for example, completion of I/O).

    4. When a process terminates.

- For circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for circumstances 2 and 3.

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is ***nonpreemptive***; otherwise, the scheduling scheme is ***preemptive***.

  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 switching states.

  Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted, and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms thus are needed to coordinate access to shared data.

SCHEDULING ALGORITHMS

- Different CPU-scheduling algorithms have different properties and may favour one class of processes over another.

- Many criteria have been suggested for comparing CPU-scheduling algorithms. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm. The criteria should include the following:

1. ***CPU Utilization***. This measures how busy is the CPU. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40% (for a lightly loaded system) to 90% (for a heavily loaded system).

2. ***Throughput***. This is a measure of work (number of processes completed per time unit). For long processes, this rate may be one process per hour; for short transactions, throughput might be 10 processes per second.

3. ***Turnaround Time***. This measures how long it takes to execute a process. Turnaround time is the interval from the time of submission to the time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing in the CPU, and doing I/O.

4. ***Waiting Time***. 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 a process spends waiting in the ready queue. Waiting time is the total amount of time a process spends waiting in the ready queue.

5. ***Response Time***. The time from the submission of a request until the system makes the first response. It is the amount of time it takes to start responding but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

A good CPU scheduling algorithm maximizes CPU utilization and throughput and minimizes turnaround time, waiting time and response time.

In most cases, the average measure is optimized. However, in some cases, it is desired to optimize the minimum or maximum values, rather than the average. For example, to guarantee that all users get good service, it may be better to minimize the maximum response time.

For interactive systems (time-sharing systems), some analysts suggests that minimizing the variance in the response time is more important than averaging response time. A system with a reasonable and predictable response may be considered more desirable than a system that is faster on the average, but is highly variable.

☞ ***First-Come First-Served (FCFS) Scheduling Algorithm***

This is the simplest CPU-scheduling algorithm. The process that requests the CPU first gets the CPU first.

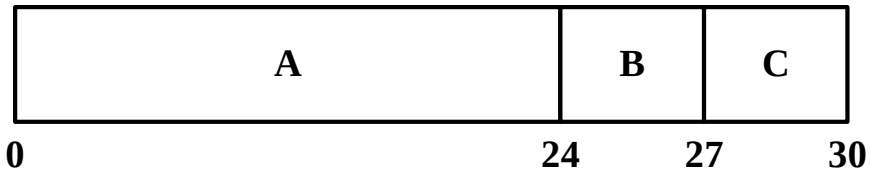
The average waiting time under the FCFS policy is often quite long.

Example 1:

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

Process Name	**Execution Time**
A	24
B	3
C	3

If the processes arrive in the order *A*, *B*, *C*, and are served in FCFS order, the system gets the result shown in the following ***Gantt chart***:

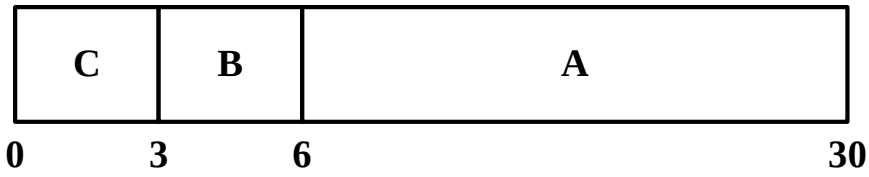


Therefore, the waiting time for each process is:

WT for A	=	0 - 0	=	0
WT for B	=	24 - 0	=	24
WT for C	=	27 - 0	=	27

Average waiting time = (0 + 24 + 27) / 3
= 17 ms

If the processes arrive in the order *C*, *B*, *A*, however, the results will be:



Therefore, the waiting time for each process is:

WT for A	=	6 - 0	=	6
WT for B	=	3 - 0	=	3
WT for C	=	0 - 0	=	0

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

The average waiting time under a FCFS policy is generally not minimal, and may vary substantially if the process CPU-burst times vary greatly.

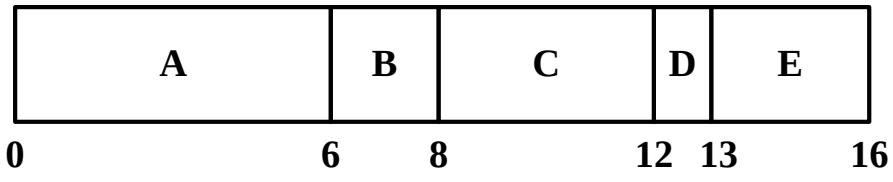
Example 2:

Consider the following set of processes with their respective arrival times at the ready queue:

Process Name	Arrival Time	Execution Time
A	0	6
B	1	2
C	2	4
D	3	1
E	4	3

Assume that the execution or CPU burst time is in milliseconds.

If the processes arrive in the given order and are served in FCFS order, the system gets the result shown in the following ***Gantt chart***:



Therefore, the waiting time for each process is:

WT for A	=	0 - 0	=	0
WT for B	=	6 - 1	=	5
WT for C	=	8 - 2	=	6
WT for D	=	12 - 3	=	9
WT for E	=	13 - 4	=	9

Average waiting time = (0 + 5 + 6 + 9 + 9) / 5
= 5.8 ms

The FCFS algorithm is ***nonpreemptive***. Once the CPU has been allocated to a process, the process keeps the CPU until it wants to release the CPU, either by terminating or be requesting I/O. The FCFS algorithm is particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

☞ ***Shortest-Job-First (SJF) Scheduling Algorithm***

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

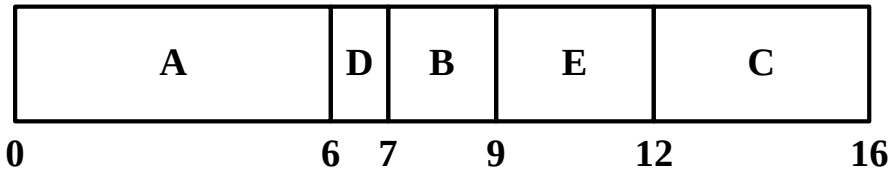
Example:

Consider the following set of processes with their respective arrival times at the ready queue:

Process Name	Arrival Time	Execution Time
A	0	6
B	1	2
C	2	4
D	3	1
E	4	3

Assume that the execution or CPU burst time is in milliseconds.

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



Therefore, the waiting time for each process is:

WT for A	=	0 - 0	=	0
WT for B	=	7 - 1	=	6
WT for C	=	12 - 2	=	10
WT for D	=	6 - 3	=	3
WT for E	=	9 - 4	=	5

Average waiting time = (0 + 6 + 10 + 3 + 5) / 5
= 4.8 ms

If the system were using the FCFS scheduling, then the average waiting time would be 5.8 ms.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term scheduling. There is no way to know the length of the next CPU burst. The only alternative is to predict the value of the next CPU burst.

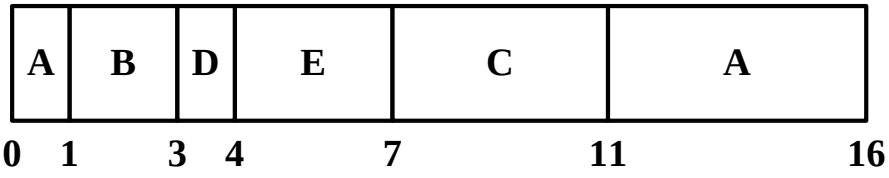
The SJF algorithm may be either preemptive or nonpreemptive. A new process arriving may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process. Preemptive SJF scheduling is sometimes called ***shortest-remaining-time-first*** scheduling.

Example:

Consider the following set of processes with their respective arrival times at the ready queue:

Process Name	Arrival Time	Execution Time
A	0	6
B	1	2
C	2	4
D	3	1
E	4	3

Using SRTF, the system would schedule these processes according to the following Gantt chart:



Therefore, the waiting time for each process is:

WT for A = (0 - 0) + (11 - 1) = 10
WT for B = 1 - 1 = 0
WT for C = 7 - 2 = 5
WT for D = 3 - 3 = 0
WT for E = 4 - 4 = 0

Average waiting time = (10 + 0 + 5 + 0 + 0) / 5
= 3.0 ms

☞ ***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 next CPU burst (τ).

$$p = 1 / \tau$$

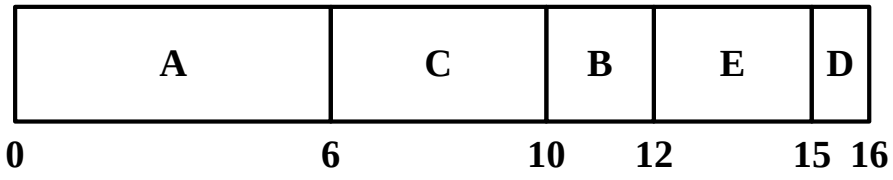
The larger the CPU burst, the lower the priority, and vice versa.

Example:

Consider the following set of processes with their respective arrival times at the ready queue:

Process Name	Arrival Time	Priority	Execution Time
A	0	3	6
B	1	2	2
C	2	1	4
D	3	5	1
E	4	4	3

Using the priority algorithm, the schedule will follow the Gantt chart below:



Therefore, the waiting time for each process is:

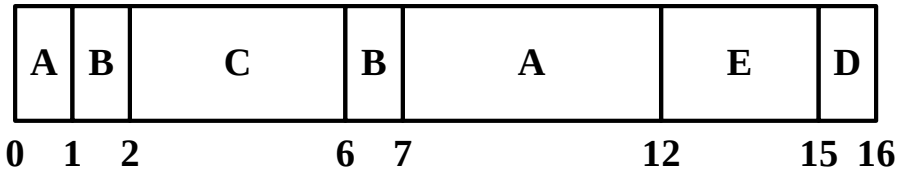
WT for A	=	0 - 0	=	0
WT for B	=	10 - 1	=	9
WT for C	=	6 - 2	=	4
WT for D	=	15 - 3	=	12
WT for E	=	12 - 4	=	8

Average waiting time = (0 + 9 + 4 + 12 + 8) / 5
= 6.6 ms

Priority scheduling can either be preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority at 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 currently running process.

Example:

Using the pre-emptive priority algorithm, the schedule will follow the Gantt chart below:



Therefore, the waiting time for each process is:

WT for A	=	(0 - 0) + (7 - 1)	=	6
WT for B	=	(1 - 1) + (6 - 2)	=	4
WT for C	=	2 - 2	=	0
WT for D	=	15 - 3	=	12
WT for E	=	12 - 4	=	8

Average waiting time = (6 + 4 + 0 + 12 + 8) / 5
= 6.0 ms

A major problem with the priority scheduling algorithm is ***indefinite blocking*** or ***starvation***. 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 indefinite blockage is ***aging***. Aging is the technique of gradually increasing the priority of processes that wait in the system for a long time.

☞ ***Round-Robin (RR) Scheduling Algorithm***

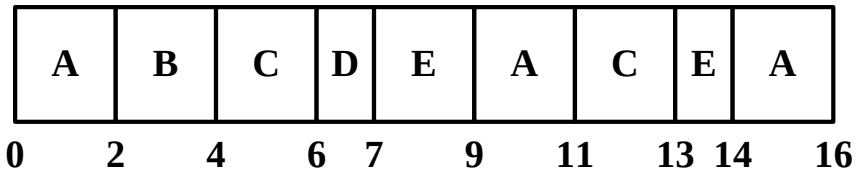
This algorithm is specifically for time-sharing systems. A small unit of time, called a ***time quantum*** or ***time slice***, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. The RR algorithm is therefore preemptive.

Example:

Consider the following set of processes with their respective arrival times at the ready queue:

Process Name	Arrival Time	Execution Time
A	0	6
B	1	2
C	2	4
D	3	1
E	4	3

If the system uses a time quantum of 2 ms, then the resulting RR schedule is:



Therefore, the waiting time for each process is:

WT for A	=	(0 - 0) +		
		(9 - 2) +		
		(14 - 11)	=	10
WT for B	=	2 - 1	=	1
WT for C	=	(4 - 2) +		
		(11 - 6)	=	7
WT for D	=	6 - 3	=	3
WT for E	=	(7 - 4) +		
		(13 - 9)	=	7

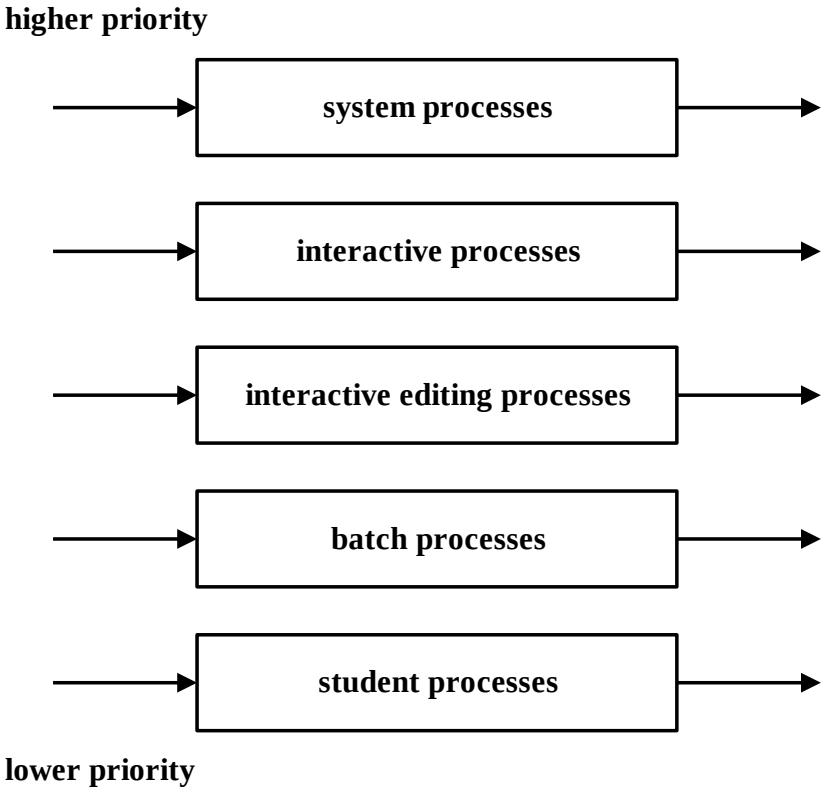
Average waiting time = (10 + 1 + 7 + 3 + 7) / 5
= 5.6 ms

The performance of the RR algorithm depends heavily on the size of the time quantum. If the time quantum is too large (infinite), the RR policy degenerates into the FCFS policy. If the time quantum is too small, then the effect of the context-switch time becomes a significant overhead. As a general rule, 80 percent of the CPU burst should be shorter than the time quantum.

☞ *Multilevel Queue Scheduling Algorithm*

This algorithm partitions the ready queue into separate queues. Processes are permanently assigned to one queue, generally based on some property of the process, such as memory size or process type. Each queue has its own scheduling algorithm. And there must be scheduling between queues.

Example:



In this example, no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.

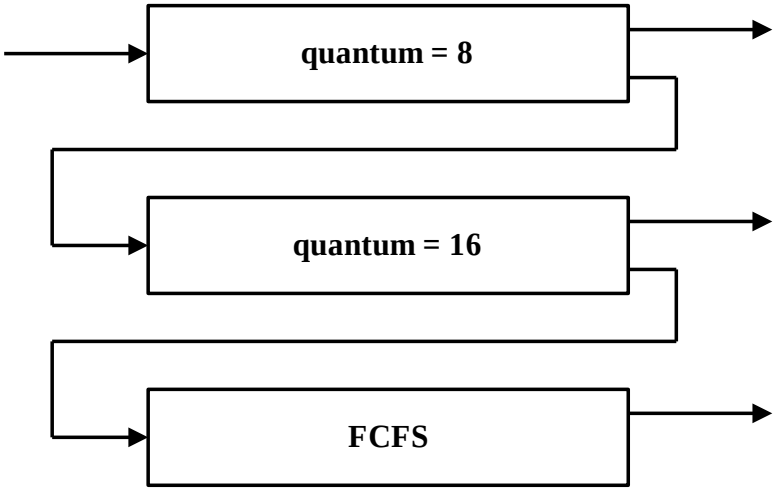
☞ *Multilevel Feedback Queue Scheduling Algorithm*

This algorithm is similar to the multilevel queue scheduling algorithm except that it allows processes to move between queues.

The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

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

Example:



In this example, the scheduler will first execute all processes in the first queue. Only when this queue is empty will the CPU execute processes in the second queue.

If a process in the first queue does not finish in 8 ms, it is moved to the tail end of the second queue. If a process in the second queue did not finish in 16 ms, it is preempted also and is put into the third queue. Processes in the third queue are run on an FCFS basis, only when the first and second queues are empty.

In general, the following parameters define a multilevel queue schedule:

1. The number of queues.

2. The scheduling algorithm for each queue.

3. The method used to determine when to upgrade a process to a higher-priority queue.

4. The method used to determine when to demote a process to a lower-priority queue.

5. The method used to determine which queue a process will enter when that process needs service

Although the multilevel feedback queue is the most general scheme, it is also the most complex.