UNIT-2

Process Management

Processes- Process Concepts-Process Schedulling-Operations on Processes-Cooperating Processes-CPU Schedulling- Basic Concepts-Scheduling Criteria-Scheduling Algorithms- Preemptive strategies-Non-preemptive strategies.

Introduction

- A *process* can be thought of as a program in execution, 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.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes:
    - Operating-system processes execute system code, and
    - user processes execute user code.
- All these processes may execute concurrently.
- Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

Processes

- Early computer systems allowed only one program to be executed at a time.
- This program had complete control of the system and had access to all the system's resources.
- In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.
- This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution.
- A process is the unit of work in a modern time-sharing system.
- The more complex the operating system is, the more it is expected to do on behalf of its users.
- Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself.

- A system therefore consists of a collection of processes: operatingsystem processes executing system code and user processes executing user code.
- Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.
- By switching the CPU between processes, the operating system can make the computer more productive.

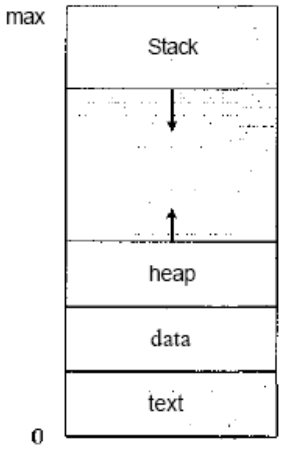
Process Concept

- A question that arises in discussing operating systems involves what to call all the CPU activities.
- A batch system executes *jobs,* whereas a time-shared system has *user programs,* or *tasks.*
- Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package.
- Even if the user can execute only one program at a time, the operating system may need to suppoft its own internal programmed activities, such as memory management.
- In many respects, all these activities are similar, so we call all of them *processes.*
- The terms *job* and *process* are used almost interchangeably used.
- Although we personally prefer the term *process,* much of operating-system theory and terminology was developed during a time when the major activity of operating systems was *job* processing.
- It would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling)* simply because *process* has superseded *job.*

The Process

- An operating system executes a variety of programs:
    - Batch system – jobs
    - Time-shared systems – user programs or tasks
- We use the terms *job* and *process* almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process is a program in execution. A process is more than the program code, which is sometimes known as the text section.

- A process includes:
    - counter
    - program stack
    - data section
        - the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
        - the process **stack** contains temporary data (such as function parameters, return addresses, and local variables),
        - and 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.
- The structure of a process in memory is shown in **Figure(a).**



Process in memory

- We emphasize that a program by itself is not a process; a program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file).**
- Whereas a process is an ***active*** entity, with a program counter pecifying 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.)
- 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.

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.
- Each 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.

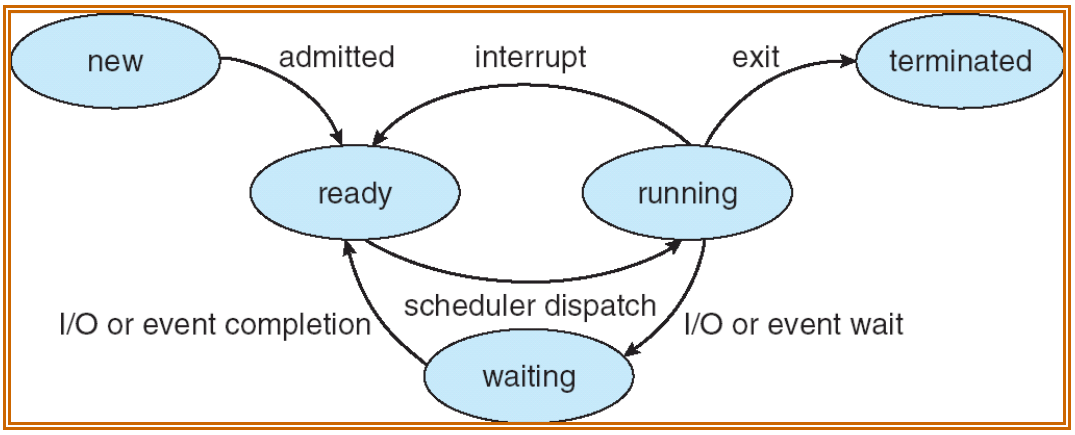
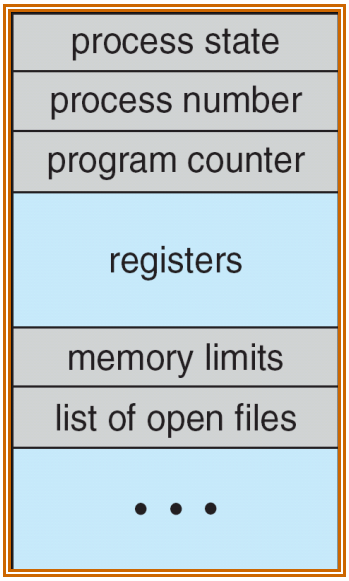


Diagram of Process State

- These names are arbitrary, and they vary across operating systems.
- The states that they represent are fotind on all systems, however.
- Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant.
- Many processes may be *ready* and *limiting,* however. The state diagram corresponding to these states is presented in Figure (b).

Process Control Block (PCB)

- Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block.*
- A PCB is shown in Figure (c).



Process Control Block (PCB)

- It contains many pieces of information associated with a specific process:

  i. Process state

  ii. Program counter

  iii. CPU registers

  iv. CPU scheduling information

  v. Memory-management information

  vi. Accounting information

  vii. I/O status information

  - **Process state:**

- The state may be new, ready, running, waiting, halted, and so on.

- **Program counter:**
  - The counter indicates the address of the next instruction to be executed 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 information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

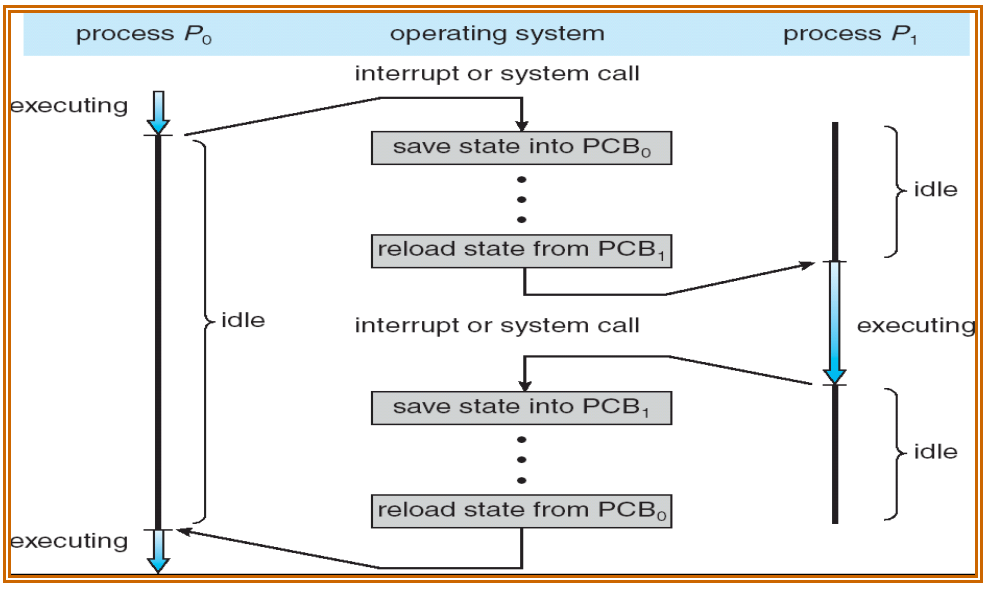


Diagram showing CPU Switch From Process to Process

Accounting information:

- This information includes the amount of CPU and real time used, time limits, account mimbers, 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.

- In brief, the PCB simply serves as the repository for any information that may vary from process to process.

Process Scheduling

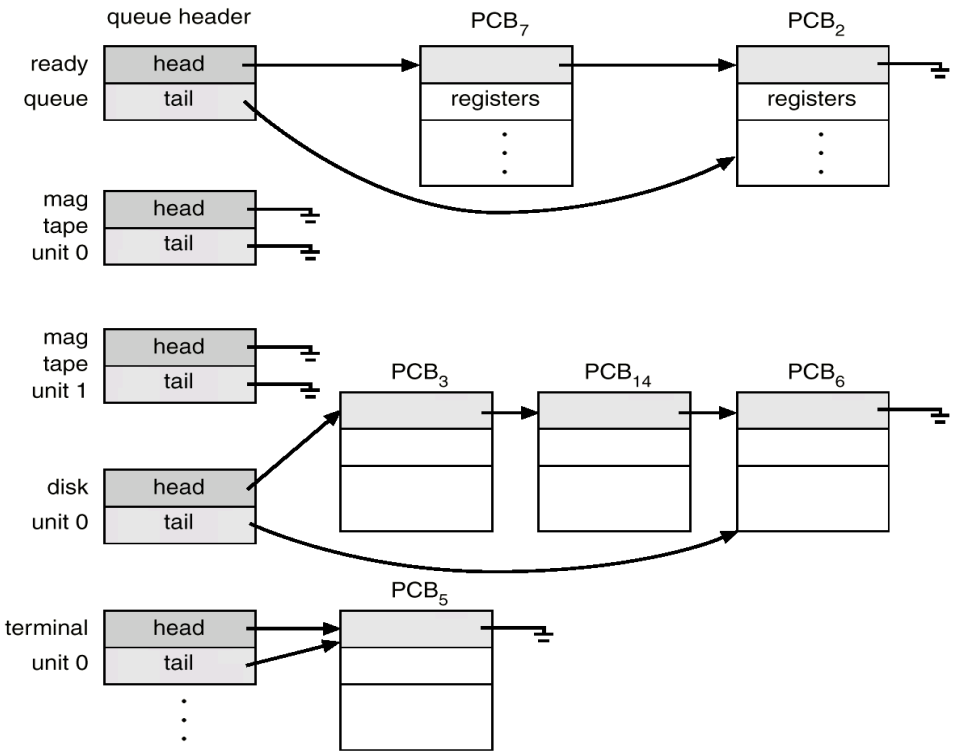
- A uniprocessor system can have only one running process. If more process exist, the rest must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch 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.

Process Scheduling Queues

- Process migration between the various queues:

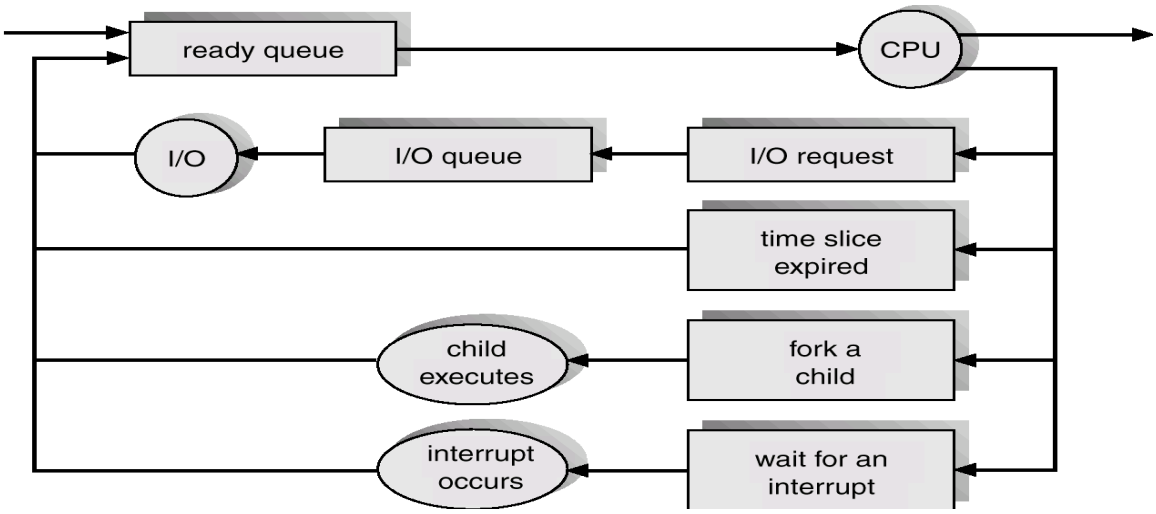
  i. Job queue
  ii. Ready queue
  iii. Device queues

- As processes enter the system, they are put into a **job queue,** which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**
- This queue is generally stored as a linked list.
- A ready-queue header contains pointers to the first and final PCBs in the list.
- We extend, each PCB includes a pointer field that points to the next PCB in the ready queue.



Ready Queue And Various I/O Device Queues

- The Operating system also includes other queues.
- When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
- Suppose the process makes an I/O request to a dedicated tape drive, or to a shared device, such as a disk.
- Since there are many processes in the system, the disk may be busy with the I/O request of some other process.
- The process therefore may have to wait for the disk.
- The list of processes waiting for a particular I/O device is called a **device queue.**
- Each device has its own device queue.
- A common representation for a discussion of process scheduling is a **queuing diagram.**



Queuing-diagram representation of Process Scheduling

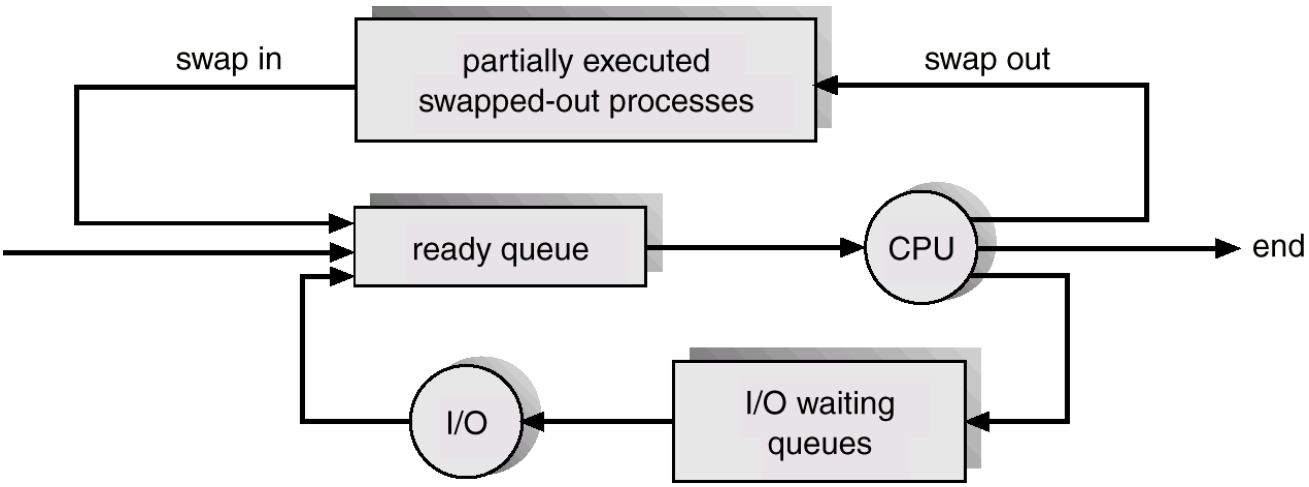
- Each ***rectangular box*** represents a queue.
- Two types of queues are present:
  - the ready queue and
  - a set of device queues.
- The ***circles*** represent the resources that serve the queues, and the ***arrows*** indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or is **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 subprocess and wait for the subprocess'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.

Schedulers

- 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.**
- 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 **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 primary distinction between these two schedulers lies in frequency of execution.
- 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.
- If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.
- 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 make a careful selection.
- In general, most processes can be described as either I/O bound or CPU bound.
    - An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.
    - A **CPU-bound process,** in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.
- On some systems, the long-term scheduler may be absent or minimal.
- For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.
- The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users.
- If the performance declines to unacceptable levels on a multi-user system, some users will simply quit.
- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- **The Medium-term scheduler** is diagrammed in Figure (c).



Addition of Medium Term Scheduling

- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
- At sometime 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.

Context Switch

- As we know interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.
- Such operations happen frequently on general-purpose systems.
- When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information.

- Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.
- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process 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.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its 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).
- Typical speeds are a few milliseconds.
- Context-switch times are highly dependent on hardware support.
- For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers.
- 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 more work must be done during a context switch.

Operations on Processes

- The processes in the system can execute concurrently, and they must be created and deleted dynamically.
- Thus, the operating system must provide a mechanism (or facility) for process creation and termination.
- In this section, we explore the mechanisms involved in creating processes and illustrate process creation on **UNIX** and **Windows** systems.

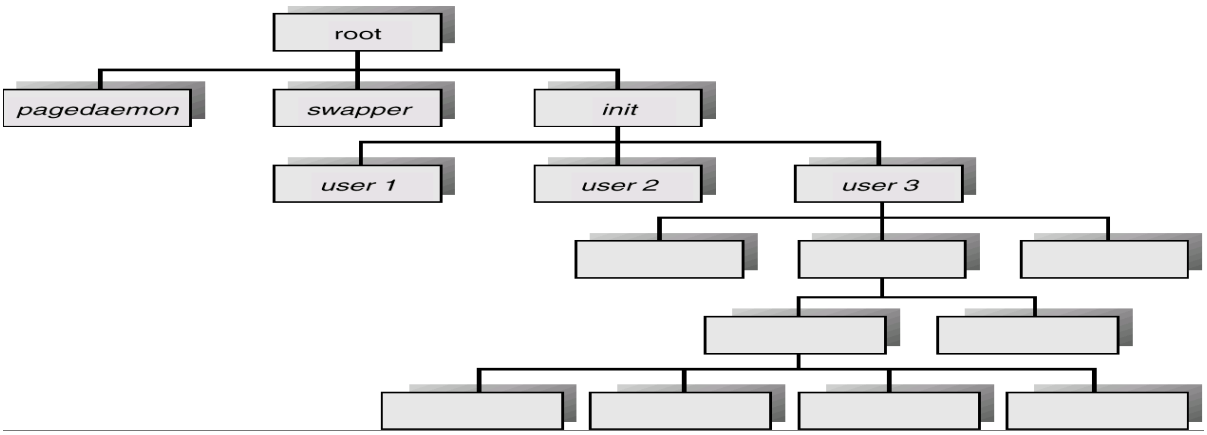
Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing

- Parent and children share all resources.
- Children share subset of parent's resources.
- Parent and child share no resources.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

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



A Tree of Processes On A Typical UNIX System

- Each of these new processes may in turn create other processes, forming a **tree** of processes.
- In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
- In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.
- For example, consider a process whose function is to display the contents of a file—say, *img.jpg*—on the screen of a terminal.
- When it is created, it will get, as an input from its parent process, the name of the file *img.jpg,* and it will use that file name, open the file, and write the contents out.
- It may also get the name of the output device.
- Some operating systems pass resources to child processes.
- On such a system, the new process may get two open files, *img.jpg* and the terminal device, and may simply transfer the datum between the two.
- When a process creates a new process, two possibilities exist in terms of execution:
    - The parent continues to execute concurrently with its children.
    - The parent waits until some or all of its children have terminated.
- There are also two possibilities in terms of the address space of the new process:
    - The child process is a duplicate of the parent process (it has the same
    - program and data as the parent).
    - The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system.

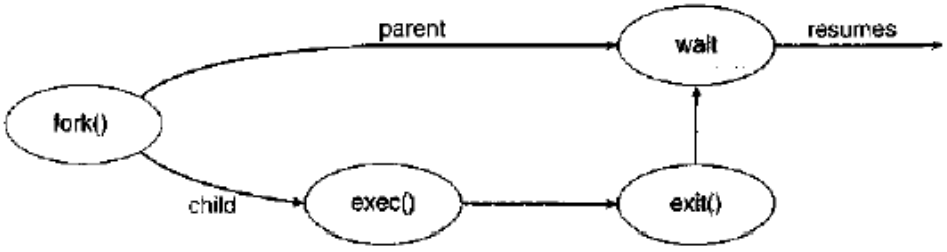
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void main(int argc,char *argv[])
{
int pid;

/* fork a child process */
pid = fork();

if (pid < 0) {/* error occurred */
        fprintf(stderr, "Fork Failed");
        exit (-1) ;
```

```
}

else if (pid == 0} {/* child process */
        execlpf"/bin/Is","Is",NULL);
}

else
{       /* parent process */

        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit (0) ;
}
}
```

C program forking a seperate process

- In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. 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 f ork( ), 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.
- Typically, the exec( ) system call is used after a fork( ) system call by one of the two processes to replace the process's memory space with a new program.
- The exec( ) system call loads a binary file into memory (destroying the memory image of the program containing the exec( ) system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.
- 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.
- The C program illustrates the UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of **pid** for the child process is zero; that for the parent is an integer value greater than zero.
- The child process overlays its address space with the UNIX command / b i n / l s (used to get a directory listing) using the execl( ) system call (execlp( ) is a version of the exec() system call).

- The parent waits for the child process to complete with the wait () system call.
- When the child process completes (by either implicitly or explicitly invoking e x i t ( )) the parent process resumes from the call to wait (), where it completes using the exit( ) system call.



Process creation

Process creation in Windows

- Processes are created in the Win32 API using the CreateProcess( ) function, which is similar to fork( ) in that a parent creates a new child process.
- However, whereas fork( ) has the child process inheriting the address space of its parent, CreateProcess( ) requires loading a specified program into the address space of the child process at process creation.
- Furthermore, whereas fork( ) is passed no parameters, CreateProcess( ) expects no fewer than ten parameters.

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 Win32).

- 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.
- Note that a parent needs to know the identities of its children.
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

    - The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
    - The task assigned to the child is no longer required.
    - The parent is exiting, and the operating system does not allow a child to
    - continue if its parent terminates.

- Some systems, including VMS, do not allow a child to exist if its parent has terminated.
- In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.
- This phenomenon, referred to as cascading termination, is normally initiated by the operating system.
- To illustrate process execution and termination, consider that, in UNIX, we can terminate process by using the exit( ) system call; its parent process may wait for the termination of a child process by using the wait( ) system call.
- The wait( ) system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.
- If the parent terminates, however, all its children have assigned as their new parent the i n i t process. Thus, the children still have a parent to collect their status and execution statistics.

Cooperating Processes

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

- A process is **independent** if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- A process is **cooperating** if **it** can affect or be affected by the other processes
- executing in the system.
- Clearly, any process that shares data with other processes is a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:

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

  Computation speedup
  - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
  - Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

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

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

- Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.
- To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- A **producer** process produces information that is consumed by a **consumer** process.
- For example, a compiler may produce assembly code, which is consumed by an assembler.
- The assembler, in turn, may produce object modules, which are consumed by the loader.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used:
    - Unbounded buffer
    - Bounded buffer

- The **unbounded buffer** places no practical limit on the size of the buffer.
- The consumer may have to wait for new items, but the producer can always produce new items.
- The **bounded buffer** assumes a fixed buffer size.
- In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- Let us illustrate a shared-memory solution to the bounded-buffer problem.
- The producer and consumer processes share the following variables:

```
#define BUFFER_SIZE 10
typedef struct {
.....
}item;

item buffer [BUFFER_SIZE] ;
int in = 0 ;
int out = 0 ;
```

- The shared buffer is implemented as a circular array with two logical pointers: ***in*** and ***out***.
- The variable ***in*** points to the next free position in the buffer; ***out*** points to the first full position in the buffer.
- The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER_SIZE) == out..

- The producer process has a local variable ***nextProduced*** in which the new item to be produced is stored:

```
item nextProduced;
while (true)
{
/* produce an item in nextProduced */
while (((in + 1) % BUFFER-SIZE) = = out)
 /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
`
```

The producer process

- The consumer process has a local variable ***nextConsumed*** in which the item to be consumed is stored:

```
while (true)
{
while (in == out);
 /* do nothing */
nextConsumed  = buffer[out];
out = (out + 1 ) % BUFFER_SIZE;
/*consume the item in nextConsumed */
```

The consumer process

- This scheme allows at most BUFFER_SIZE - l items in the buffer at the same time.

CPU Scheduling

- CPU scheduling is the basis of multiprogrammed operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive.
- However, the terms **process scheduling** and **thread scheduling** are often used interchangeably.

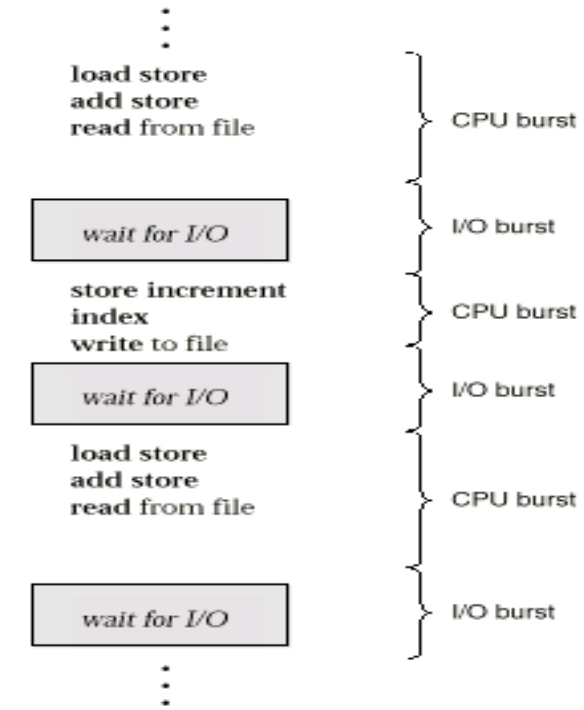
- We use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

Basic concepts

- In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The idea of multiprogramming is relatively simple.
- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle;all this waiting time is wasted;no useful work is accomplished.
- With multiprogramming, we try to use this time productively.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU.
- Scheduling of this kind is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.
- The CPU is, of course, one of the primary computer resources.
- Thus, its scheduling is central to operating-system design.

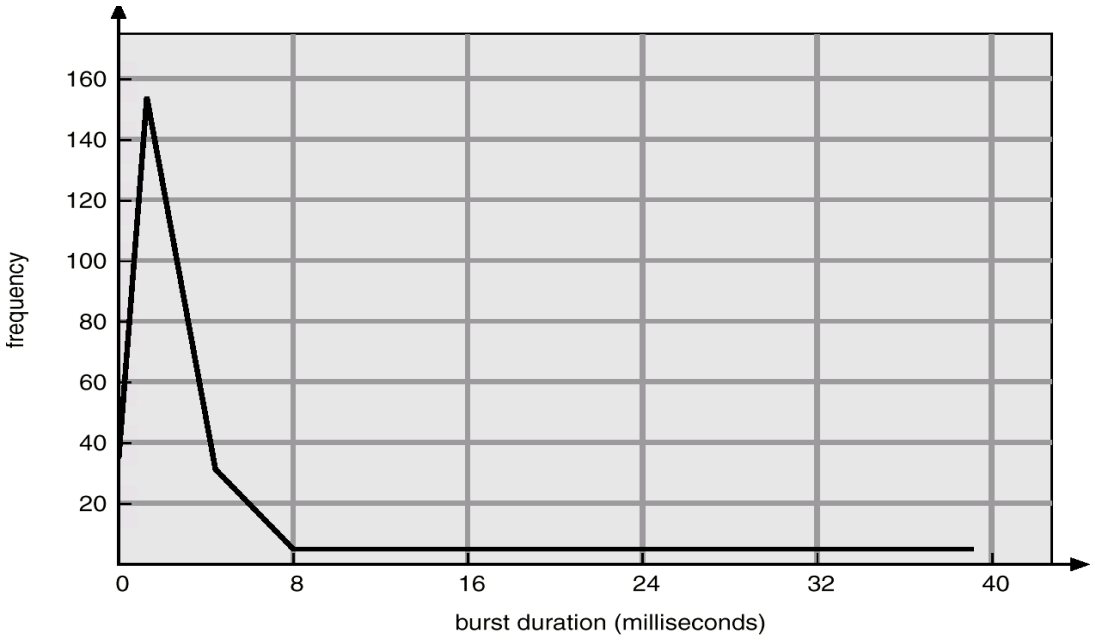
CPU-I/O Burst Cycle

- 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 (**Fig a**).
    - The durations of CPU bursts have been measured extensively.



Alternating sequence of CPU and I/O bursts

- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in **Fig (b).**



Histogram of CPU-burst Times

- The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.
- This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler** (or CPU scheduler).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- It can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally process control blocks (PCBs) of the processes.

Preemptive Scheduling

- CPU-scheduling decisions may take place under the following four circumstances:
    - **i.** When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)**.**
    - **ii.** When a process switches from the running state to the ready state (for example, when an interrupt occurs).
    - **iii.** When a process switches from the waiting state to the ready state (for example, at completion of I/O).
    - **iv.** When a process terminates.

- For situations **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 situations **2** and **3**.
- When scheduling takes place only under circumstances **1** and **4**, we say that the scheduling scheme is **nonpreemptive** or **cooperative;** otherwise, it 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 by switching to the waiting state.
- This scheduling method is used by the Microsoft Windows 3.1 and Apple Macintosh Operating systems.
- Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

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

Dispatcher

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

- The dispatcher should be as fast as possible, since it is invoked during every process switch.

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

Scheduling Criteria

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor 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 used system).

Throughput

- If the CPU is busy executing processes, then work is being done.
- One measure of work is the number of processes that are completed per time unit, called *throughput.*
- For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

Turnaround time

- From the point of view of a particular process, the important criterion is how long it takes to execute that process.
- 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.
- Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
- 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. In most cases, we optimize the average measure.
- However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.
- For example, to guarantee that all users get good service, we may want to minimize the maximum response time.
- Some analysts have suggested that, for interactive systems (such as timesharing systems), it is more important to minimize the *variance* in the response time than to minimize the average response time.
- A system with reasonable and *predictable* response time may be considered more desirable than a system that is faster on the average but is highly variable.
- However, little work has been done on CPU-scheduling algorithms that minimize variance.

Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU scheduling algorithms:

i. First-come, First served scheduling
ii. Shortest-Job-First scheduling
iii. Shortest-remaining-time-first scheduling
iv. Priority scheduling
v. Round-Robin scheduling
vi. Multilevel Queue scheduling
vii. Multilevel Feedback Queue scheduling

First-come, First served scheduling

- By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**
- The **FCFS** scheduling algorithm is nonpreemptive.
- 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.
- The code for FCFS scheduling is simple to write and understand.
- The average waiting time under the FCFS policy, however, is often quite long.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Example 1:

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order : *P1* , *P2* , *P3* *and are served in FCFS order,*
- The **Gantt Chart** for the schedule is:

Waiting time:

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

Turnaround time:

- Turnaround time for P1 = 24
- Turnaround time for P2 = 24 + 3
- Turnaround time for P3 = 24 + 3 + 3

- Average Turnaround time = (24+24+3+24+3+3) / 3
  = 27 milliseconds.

(Or)

= (24*3 + 3*2 + 3*1) / 3
= 27 milliseconds.

- In general we have (n*P1 + (n-1)*P2 + ...) / n
- If we want to minimize this, P1 should be the smallest, followed by P2 and so on.

Example 2:

- If the processes arrive in the order *P2,* P3, P1.

Process	Burst Time
P2	3
P3	3
P1	24

- The Gantt chart for the schedule is:

Waiting time:

- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- **Average waiting time**: (6 + 0 + 3)/3 = 3milliseconds.

Turnaround time:

- Turnaround time for P2 = 3
- Turnaround time for P3 = 3 + 3
- Turnaround time for P1 = 3 + 3+24

- Average Turnaround time = (3*3 + 3*2 + 24*1) / 3
  = 13 milliseconds.

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

Shortest-Job-First scheduling

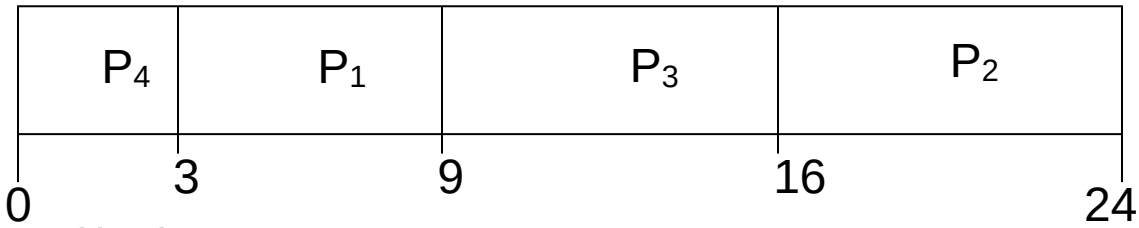
- A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.**
- This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU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

- Note that a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst algorithm,* because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.
- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Example:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

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



Waiting time:

- The waiting time is 3 milliseconds for process *P1,*
- 16 milliseconds for process *P2,*
- 16 milliseconds for process *P3,* and
- 0 milliseconds for process P4.
- Thus, the ***average waiting time*** =(3 + 16 + 9 + 0)/4
  = 7 milliseconds.

Turnaround time:

- Turnaround time for P1 = 9
- Turnaround time for P2 = 24
- Turnaround time for P3 = 16
- Turnaround time for P4 = 3
- Average Turnaround time = (9+24+16+3)/ 4
  = 13 milliseconds.

- 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 decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the *average* waiting time decreases. 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 nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.

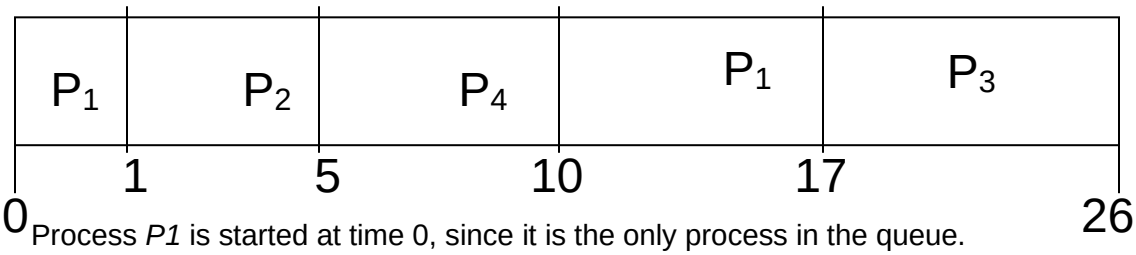
Shortest- Remaining-Time- First

- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling.**
- As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

  Example:

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

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



- Process *P1* is started at time 0, since it is the only process in the queue.

- Process *P2* arrives at time 1.
- Among *P1* and *P2* as P2 is having shortest time, P2 will get executed first.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- After completion of P2, P1 is scheduled.
- And then in the last P4 which is having larger time is sheduled.

Waiting time:

- Waiting time for process p1 = (10 - 1)
- Waiting time for process p2 = (1-1)
- Waiting time for process p3 = (17-2)
- Waiting time for process p4 = (5-3)

- The average waiting time = ((10 - 1) + (1 - 1) + (17-2) + (5-3))/4
  = 26/4
  = 6.5 milliseconds.

- Nonpreemptive **SJF** scheduling would result in an average waiting time of 7.75 milliseconds.

Turnaround time:

- Turnaround time for P1 = (0+8)=8
- Turnaround time for P2 = (7 + 4)
- Turnaround time for P3 = (15+9)
- Turnaround time for P4 = (9 + 5)

- Average Turnaround time = (8+11+24+14) / 4
  = 14.25 milliseconds.

Priority scheduling

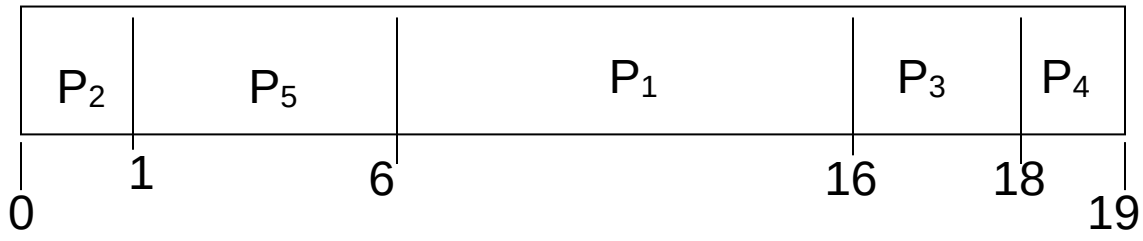
- The SJF algorithm is a special case of the general **priority scheduling algorithm.**
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm 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.

- Note that we discuss scheduling in terms of *high* priority and *low* priority.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- 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. This difference can lead to confusion.
- Here, we assume that low numbers represent high priority.
- As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, ….., *P5,* with the length of the CPU burst given in milliseconds:

Example:

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

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



- Process *P2* is started at time 0, since it is having the highest priority among the processes in the queue.
- After execution of P2 process, process *P5* is started at time 1.
- Then *P1,P3* and *P4* as P2 processes are executed. is having shortest time, P2 will get executed first.

Waiting time:

- Waiting time for process p1 = 6
- Waiting time for process p2 = 0

- Waiting time for process p3 = 16
- Waiting time for process p4 = 18
- Waiting time for process p5 = 1

- The average waiting time = (6+0+16+18+1)/5
  = 41/5
  = 8.2 milliseconds.

Turnaround time:

- Turnaround time for P1 = 6 + 10
- Turnaround time for P2 = 0 + 1
- Turnaround time for P3 = 16 + 2
- Turnaround time for P4 = 18 + 1
- Turnaround time for P5 = 1 + 5

- Average Turnaround time = (16+1+18+19+6) / 5
  = 12 milliseconds

- 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.
- Generally, one of two things will happen.
- Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)
- A solution to the problem of indefinite blockage of low-priority processes is **aging.**
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.
- In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

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 switch between processes.
- A small unit of time, called a **time quantum** or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds.

- *The ready queue is treated as a circular queue*.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.

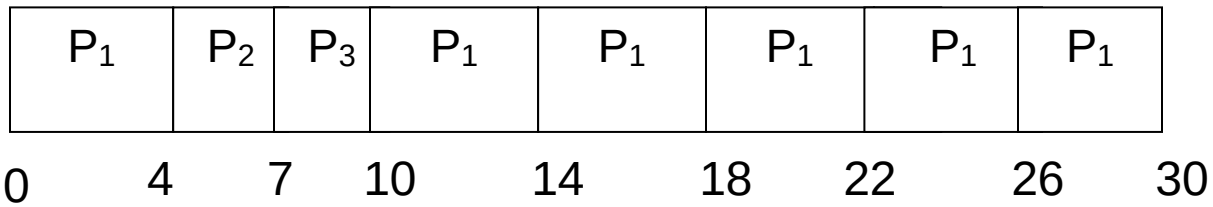
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen:
    - The process may have a CPU burst of less than 1 time quantum.
        - In this case, the process itself will release the CPU voluntarily.
        - The scheduler will then proceed to the next process in the ready queue.
    - Otherwise, 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.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

  Example 1:

Process	Burst Time
P1	24
P2	3
P3	3

  - Time quantum = 4 ms

- The Gnatt chart is:



- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2*
- Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
- The CPU is then given to the next process, process P3.

- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is:

  Waiting time:

  - Waiting time for process p1 = (10 - 4)
  - Waiting time for process p2 = 4
  - Waiting time for process p3 = 7
  - The average waiting time = ((10-4)+4+7) /3

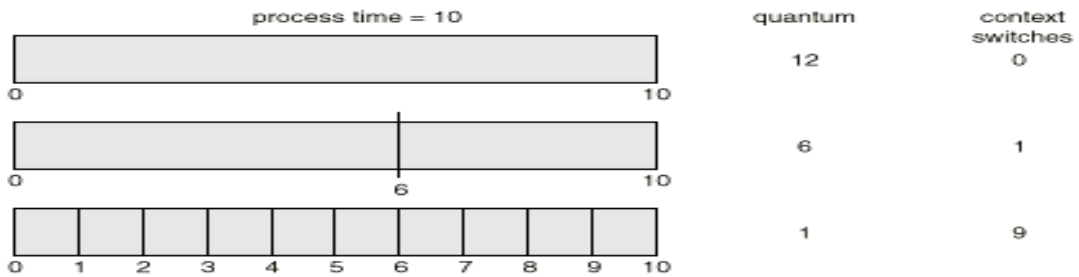
    = 17/3 = 5.66 milliseconds.

  Turnaround time:

  - Turnaround time for P1 = (10-4) + 24
  - Turnaround time for P2 = 4 + 3
  - Turnaround time for P3 = 7 + 3
  - Average Turnaround time = (30+7+10) / 3

    = 15.6 milliseconds

- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thus ***preemptive.***
- If there are *n* processes in the ready queue and the time quantum is *q,* then each process gets *1/n* of the CPU time in chunks of at most *q* time units.
- Each process must wait no longer than *(n* — 1) x *q* time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.
- The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy
- If the time quantum is extremely small (say, 1 millisecond), the RR approach is called **processor sharing**



Showing how a smaller time quantum increases context switches

- In software, we need also to consider the effect of context switching on the performance of RR scheduling.

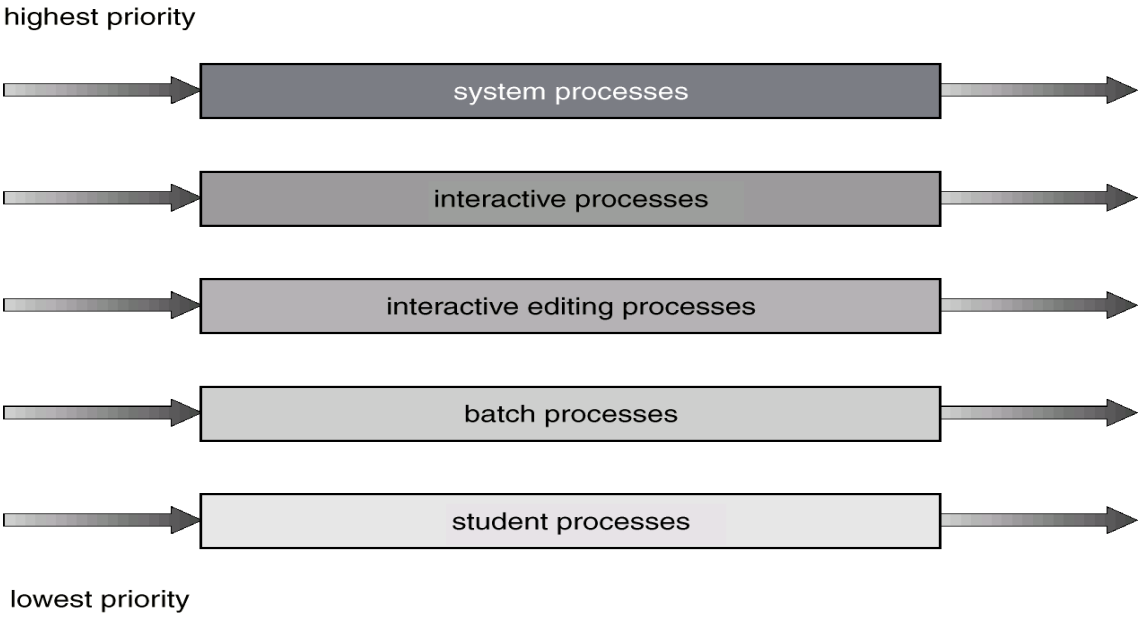
- Let us assume that we have only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.
- Thus, we want the time quantum to be large with respect to the context switch time.
- If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
- In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
- The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Multilevel Queue scheduling

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- 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.



Multilevel queue scheduling

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

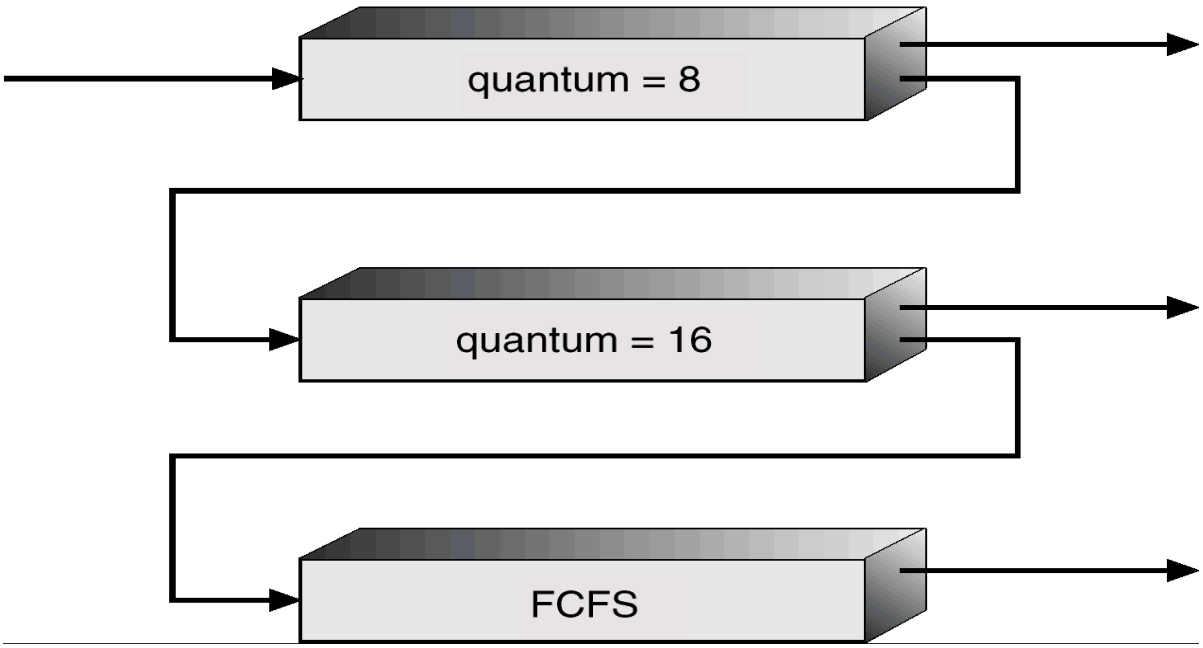
  i. System processes
  ii. Interactive processes
  iii. Interactive editing processes
  iv. Batch processes
  v. Student processes

- 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% of the CPU time for **RR** scheduling among its processes,
    - whereas the background queue receives 20% of the **CPU** to give to its processes on an FCFS basis.

Multilevel Feedback Queue scheduling

- Normally, in the multilevel queue scheduling algorithm, processes are permanently assigned to a queue when they enter to the system.
- Processes do not move between queues.
- For example, if there are separate queues for foreground and background processes, 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.
- Similarly, 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.**



Multilevel feedback queues

- For example, consider a multilevel feedback-queue scheduler with ***three queues***, numbered from 0 to 2 (Figure).
    - The scheduler first executes all processes in queue 0.
    - Only when queue 0 is empty will it execute processes in queue 1.
    - Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.
    - A process that arrives for queue 1 will preempt a process in queue 2.
    - A process that arrives for queue 0 will in turn, preempt a process queue 1.

- 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.

- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.
- Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
- Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

- In general, a multilevel feedback-queue scheduler is defined by the following parameters:
    - The number of queues
    - The scheduling algorithm for each queue
    - The method used to determine when to upgrade a process to a higher priority queue
    - The method used to determine when to demote a process to a lower priority queue
    - The method used to determine which queue a process will enter when that process needs service

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

Algorithm Evaluation

How do we select a CPU scheduling algorithm for a particular system? There are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm. criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:

• Maximizing CPU utilization under the constraint that the maximum

response time is 1 second

• Maximizing throughput such that turnaround time is (on average) linearly
proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

1**.Deterministic Modeling**

One major class of evaluation methods is **analytic evaluation.** Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload. One type of analytic evaluation is **deterministic modeling.** This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

Process	**Burst Time**
P1	10
P2	29
P3	3
P4	7
P5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as

P1	P2	P3	P4	P5	
0	10	39	42	49	61

The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49
milliseconds for process P5. Thus, the average waiting time is (0 + 10 + 39
+ 42 + *49)/5* = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as

P3	P4	P1	P5	P2	
0	3	10	20	32	61

The waiting time is 10 milliseconds for process *P\,* 32 milliseconds for process *P2,* 0 milliseconds for process P3, 3 milliseconds for process P4, and 20 milliseconds for process P5. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as

P1	P2	P3	P4	P5	P2	P5	P2	
0	10	20	23	30	40	50	52	61

The waiting time is 0 milliseconds for process P1, 32 milliseconds for process *P2,* 20 milliseconds for process P3, 23 milliseconds for process P4, and 40 milliseconds for process P5. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We see that, *in this case,* the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value. Deterministic modeling is simple and fast. It gives us exact numbers,allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be

analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

2. **Queueing Models**

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated

or simply estimated. The result is a mathematical formula describing the probability of a particular CPU

burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms. The computer system is described as a network of servers. Each server has a queue of waiting processes. The x W CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis.** As an example, let *n* be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let X be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time *W* that a process waits, X

new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus, this equation n= XxW known as **Little's formula,** is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

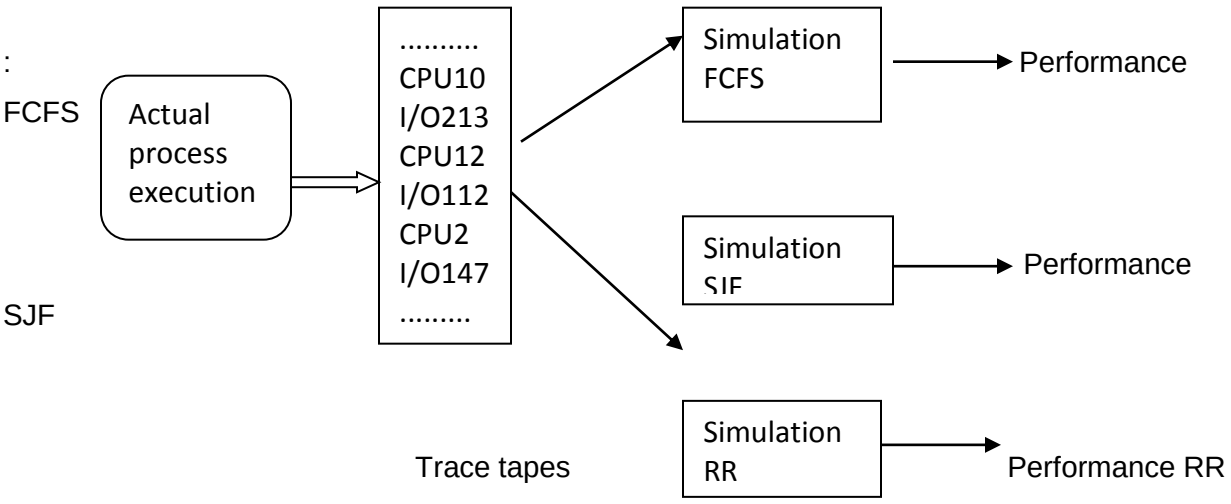
We can use Little's formula to compute one of the three variables, if we know the other two. For example, if we know that 7 processes arrive every second (on average), and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable—but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

3 **Simulations**

To get a more accurate evaluation of scheduling algorithms, we can use **simulations.** Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a

clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation



Algorithm Evaluation

Evaluation of CPU schedulers by simulation. executes, statistics that indicate algorithm performance are gathered and printed. The data to drive the simulation can be generated in several ways. The most

common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation. A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace tapes.** We create a trace tape by monitoring the real system and recording the sequence of actual events (Figure 5.15). We then use this sequence to drive the simulation. Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs. Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

4 **Implementation**

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The major difficulty with this approach is the high cost. The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures) but also in the reaction of the users to a constantly changing operating system. Most users are not interested in building a better operating system; they merely want to get their processes executed and use their results. A constantly changing operating system does not help the users to get their work done. Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use. For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though

the terminal output was completely meaningless. The most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. For instance, a workstation that performs high-end graphical applications may have scheduling needs different from those of a web server or file server. Some operating systems— particularly several versions of UNIX—allow the system manager to fine-tune

the scheduling parameters for a particular system configuration. For example, Solaris provides the dispadmin command to allow the system administrator to modify the parameters of the scheduling classes. Another approach is to use APIs that modify the priority of a process or thread. The Java, /POSIX, and /WinAPI/ provide such functions. The downfall of this approach is that performance tuning a system or application most often does not result in improved performance in more general situations.

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