Process Management

==========

- A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**.
- Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.
- And even if a user can execute only one program at a time, such as on an **embedded device** that does **not** support **multitasking**, the Operating system may need to support its own internal programmed activities, such as memory management.
- In many respects, all these activities are similar, so we call all of them **processes**.

1 Process Concept:

1.1 The Process :

- 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 generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables).
- A **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.
- The structure of a process in memory

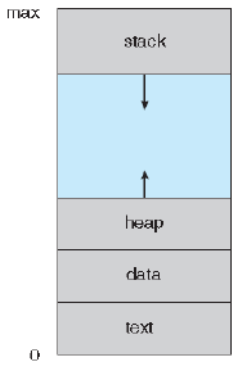


Fig 3.1 process in memory

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**).

In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources.

In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are

1. **Double-clicking** an icon representing the executable file
2. **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 **Example**, 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*.**

1.2 Process State / Process Life Cycle

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

A process may be in one of the following states:

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

- ❖ The states that they represent are found 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 waiting, however. The state diagram corresponding to these states is presented in below figure 2.2

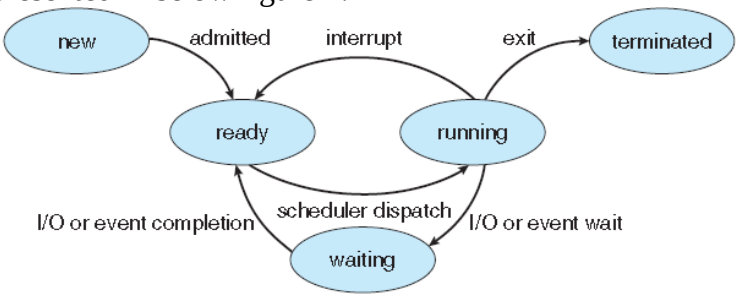


Figure 2.2 Diagram of process state

1.3 Process Control Block

- ✓ 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 2.3. It contains many pieces of information associated with a specific process, including these:

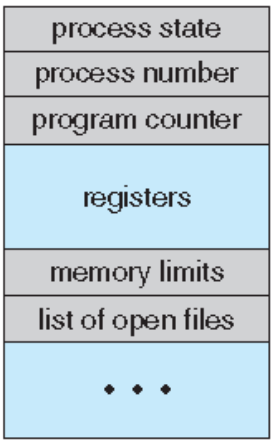


Figure 2.3 PCB

- **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 items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system

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

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

2. Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the 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.

2.1 Scheduling 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. Each PCB includes a pointer field that points to the next PCB in the ready queue.

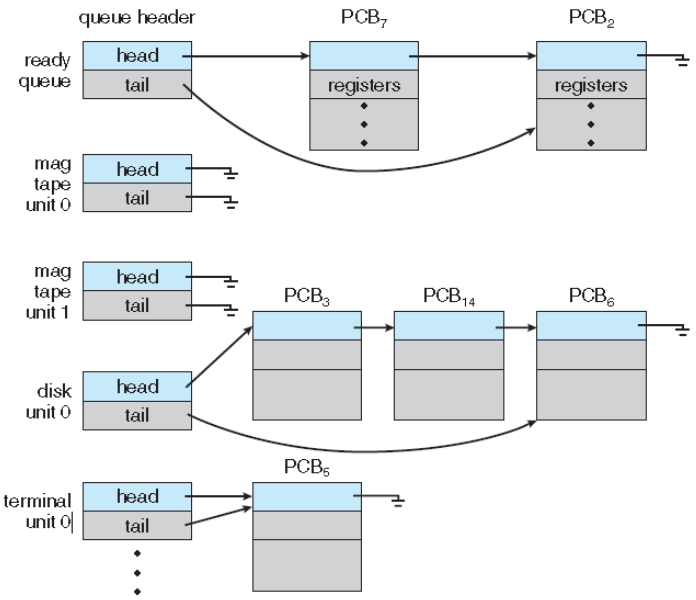


Figure 2.4 The ready queue and various I/O device queues.

- The system also includes other queues.
- 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 of process scheduling is a **queuing diagram**.
- Each **rectangular box** represents a **queue.**
- Two types of queues are present: the **ready queue** and a set of **device queues**.
- The **circles** represent the **resources** that serve the queues, and the arrows indicate the flow of processes in the system.

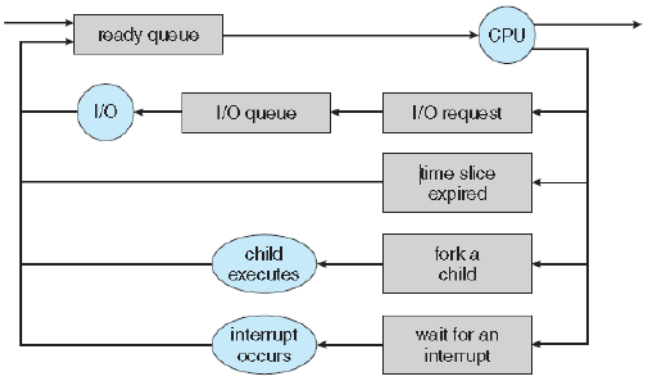


Figure 2.4 Queuing-diagram representation of process scheduling.

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

3.2 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**.

Schedulers are 3 types:

1. **Long-term scheduler**, or **job scheduler**
2. **Short-term scheduler**, or **CPU scheduler**
3. **Medium-term scheduler**

1 Long-term scheduler, or job scheduler:

- Selects processes from this **pool** and loads them into memory for execution.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
- 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.

2. **Short-term scheduler**, or **CPU scheduler:**

- The primary distinction between these two schedulers lies in frequency of execution.
- Selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The short-term scheduler must select a new process for the CPU frequently.
- A process may execute for only a few milliseconds before waiting for an I/O request.
- 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.

3 Medium-term scheduler

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- This **medium-term scheduler** : The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus **reduce the degree of multiprogramming**.

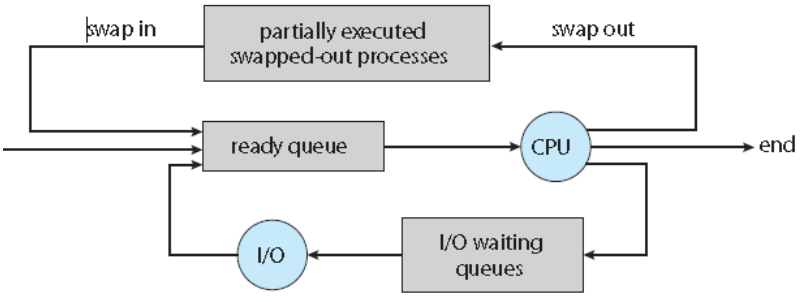


Figure 2.5 Addition of medium-term scheduling to the queueing diagram.

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.

2.3 Context switch:

- **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 running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- The context is represented in the PCB of the process.

- It includes the value of the CPU registers, the process state and memory-management information.

- 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 performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

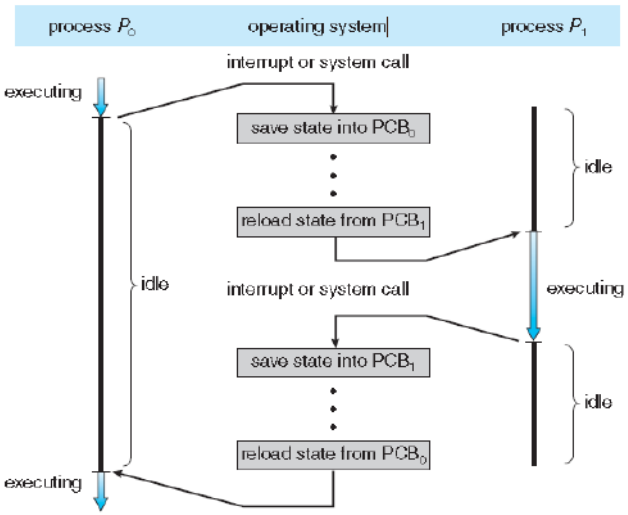


Figure 2.6 Diagram showing CPU switch from process to process.

- 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.

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

- A typical speed is a few milliseconds.
- Context-switch times are highly dependent on hardware support.

3 Operations on Processes

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

3.1 Process Creation

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

- As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process.

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

- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or pid), which is typically an integer number.

- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

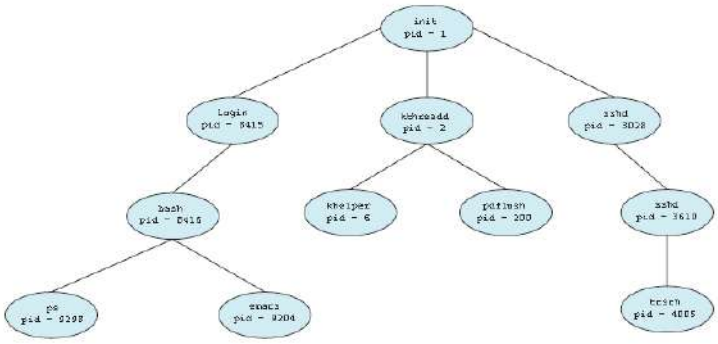


Figure 2.7 A tree of processes on a typical Linux system.

Linux:

- ✓ Figure 3.6 illustrates a typical process tree for the **Linux operating system**, showing the name of each process and its pid. (We use the term ***process*** rather loosely, as Linux prefers the term ***task*** instead.)
- ✓ The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- ✓ Once the system has booted, the **init** process can also create various user processes, such as a web or print server, an ssh server, and the like.

On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.

For example, the command will list complete information for all processes currently active in the system.

ps –el

When a process **creates a new process**, two possibilities for execution exist:

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

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

There are also two address-space possibilities for the new process:

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

Unix :

let's first consider the UNIX operating system. 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 fork(), with one difference:
  - The return code for the **fork()** is **zero** for the new (child) process,
  - Whereas the (nonzero) process identifier of the child is returned to the parent.
  - 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.
  - It can issue a wait() system call to move itself off the ready queue until the termination of the child.

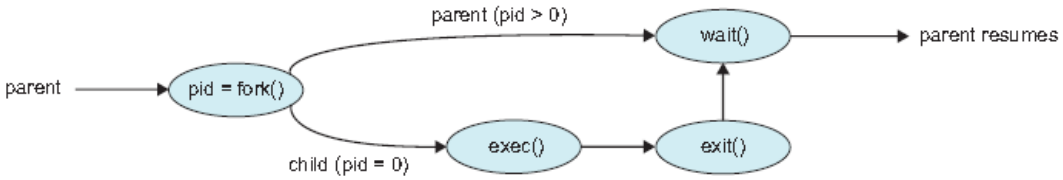


Figure 2.8 Process creation using the fork() system call.

Windows:

- As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the CreateProcess() function.
- The two parameters passed to the CreateProcess() function are instances of the STARTUPINFO and PROCESS INFORMATION structures.
- STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files.
- The PROCESS INFORMATION structure contains a handle and the identifiers to the newly created process and its thread.
- We invoke the ZeroMemory() function to allocate memory for each of these structures before proceeding with CreateProcess().

3.2 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.

- Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, **TerminateProcess()** in Windows).

- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

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

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

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

- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

If a process terminates (either normally or abnormally), then all its children must also be terminated called **cascading termination**.

Linux and Unix:

A process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
/* exit with status 1 */
        exit(1);
```

- A parent process may wait for the termination of a child process by using the wait() system call.

- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child.

- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_ t pid;
int status;
pid = wait(&status);
```

- When a process terminates, its resources are deallocated by the operating system.

- However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

- A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.

4 Interprocess Communication (IPC)

- 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.** (**advantages of interprocess communication**)

1. **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file).

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

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

4. **Convenience**. Even an individual user may work on many tasks at the same time.

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

❖ There are two fundamental models of interprocess communication:
   1. **Shared Memory**
   2 . **Message Passing**.

1. **Shared-Memory Model**: A region of memory that is shared by cooperating processes is established.

Processes can then exchange information by reading and writing data to the shared region.

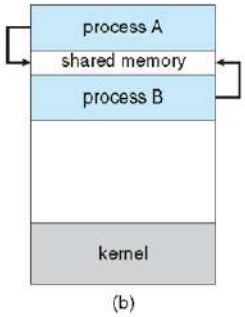


Fig : 2.9 Shared memory.

- ✓ *Interprocess communication* using shared memory requires communicating processes to establish a region of shared memory.
- ✓ Typically, a shared-memory region **resides in the address space of the process** creating the shared-memory segment.
- ✓ Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

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

The producer–consumer problem item next produced;

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

Figure 2.10 The producer process using shared memory.

For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

Two types of buffers can be used.

1 **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.

2 **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.

```
#define BUFFER SIZE 10
typedef struct {
        . . .
}item;

item buffer[BUFFER SIZE];
        int in = 0;
        int out = 0;
item next consumed;
while (true) {
while (in == out)
        ; /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
```

/* consume the item in next consumed */
}

Figure 2.11 The consumer process using shared memory.

The shared buffer is implemented as a circular array with two logical pointers:

- **in** : 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.

Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls

Shared memory suffers from **cache coherency issues**, which arise because shared data migrate among the several caches.

As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

Message-Passing Systems:

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.7

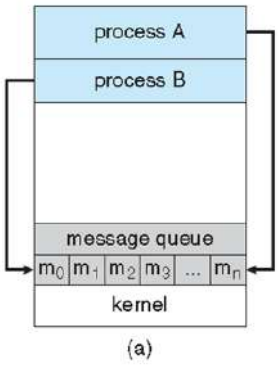


Figure :2.12 Message passing.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations:

- ✓ **send(message)**
- ✓ **receive(message)**

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a **communication link** must exist between them.

This link can be implemented in a variety of ways.

Here are several methods for logically implementing a link and the send()/receive() operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

We look at issues related to each of these features next.

A. Direct communication using Name

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.

In this scheme, the send() and receive() primitives are defined as:

• send(P, message)—Send a message to process P.

• receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process **must name** the other to communicate.

A variant of this scheme employs ***asymmetry*** in addressing.

- Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the **send()** and **receive()** primitives are defined as follows:

• **send(P, message)**—Send a message to process P.

• **receive(id, message)**—Receive a message from any process.

The variable **id** is set to the **name of the process** with which communication has taken place.

- The **disadvantage** in both of these schemes (symmetric and asymmetric) is the **limited modularity** of the resulting process definitions.

Indirect Communication, the messages are sent to and received from **mailboxes,** or **ports.**

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

- Each mailbox has a unique identification.

A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a **shared mailbox**.

The send() and receive() primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.
• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default.

B. Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

• **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.

• **Nonblocking send**. The sending process sends the message and resumes operation.

• **Blocking receive**. The receiver blocks until a message is available.

• **Nonblocking receive**. The receiver retrieves either a valid message or a null.

C. Buffering:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such **queues can be implemented in three ways:**

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it.

- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

❖ The **zero-capacity** case is sometimes referred to as a message system with **no buffering**.

❖ The other cases are referred to as systems with **automatic buffering**.

5 Multithreaded

- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.
- **single-threaded** process and a **multithreaded** process.

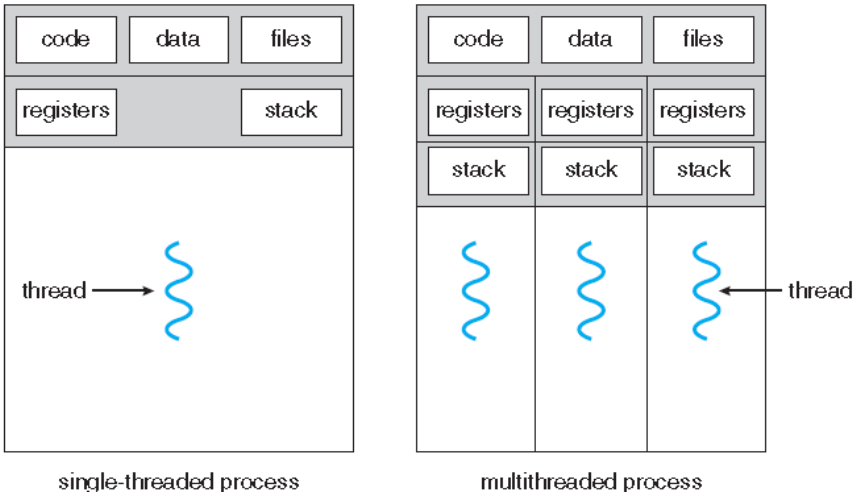


Figure 4.1 Single-threaded and multithreaded processes.

Example:

A web browser might have one thread 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 responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Problem in single thread

If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

Why multithreads

- One solution is to have the server run as a single process that accepts requests.
- When the server receives a request, it creates a separate process to service that request.

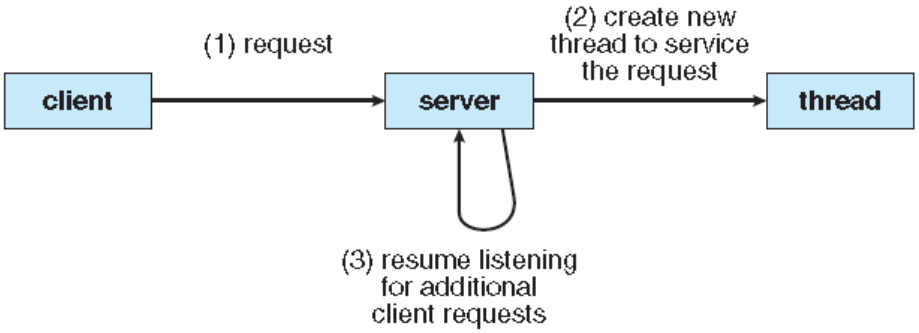


Figure 4.2 Multithreaded server architecture.

.

For example,

Solaris has a set of threads in the kernel specifically for **interrupt handling;**

Linux uses a kernel thread for managing the amount of **free memory** in the system.

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into **four** major categories: (advantages of multithreading)

1. **Responsiveness**.

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

2. **Resource sharing**.

Processes can only share resources through techniques such as **shared memory** and **message passing**. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**.

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

Example :

In Solaris, for example, creating a process is about **thirty times slower than is creating a thread**, and context switching is about **five times slower.**

4. Scalability

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multithreaded programming provides a mechanism for **more efficient use** of these multiple computing cores and **improved concurrency.**

4.3 Multithreading Models

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

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

A relationship must exist between user threads and kernel threads.

Three common ways of establishing such a relationship:

a. many-to-one model,
b. one-to-one model,
c. many-to-many model.

4.3.1 Many-to-One Model:

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.

Example :

Green threads—a thread library available for Solaris

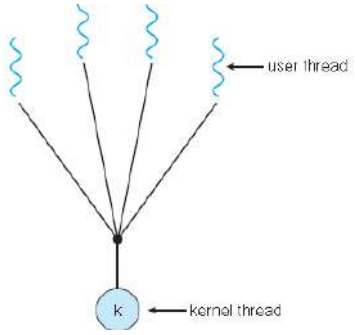


Figure 4.5 Many-to-one model.

4.3.2 One-to-One Model

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

The only **drawback** to this model is that creating a user thread requires creating the corresponding kernel thread.
Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

Example :

Linux, along with the family of Windows operating systems, implement the one-to-one model.

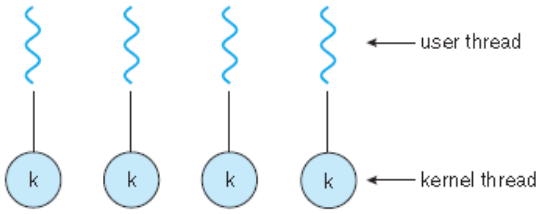


Figure 4.6 One-to-one model.

4.3.3 Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.

The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a **single processor**).

The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

This variation is sometimes referred to as the **two-level model**.

Example :

The Solaris operating system supported the two-level model in versions older than Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.

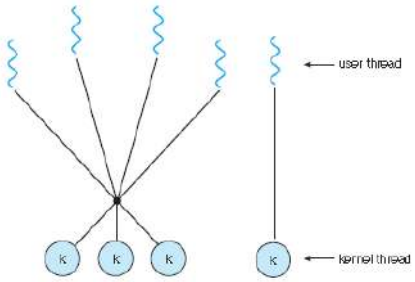


Figure 4.8 Two-level model

Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library.

1. The first approach is to provide a library entirely in user space with no kernel support.

2. The second approach is to implement a kernel-level library supported directly by the operating system.

Three main thread libraries are in use today: **POSIX Pthreads,Windows,** and **Java.**

- **Pthreads**, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.

- The **Windows thread library** is a kernel-level library available on Windows systems.

- The **Java thread API** allows threads to be created and managed directly in Java programs

5. Threading Issues

we discuss some of the issues to consider in designing multithreaded programs.

5.1 The fork() and exec() System Calls

fork():

- The fork() system call is used to create a separate, duplicate process.
- The semantics of the fork() and exec() system calls change in a multithreaded program.

Some UNIX systems have chosen to have two versions of fork(),

1. duplicates all threads
2. duplicates only the thread that invoked the fork() system call.

exec()

1. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

Which of the two versions of fork() to use depends on the application.

2. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.

5.2 Signal Handling

- A **signal** is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either **synchronously or asynchronously**, depending on the source of and the reason for the event being signaled.

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

A signal may be ***handled*** by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.

- ✓ Handling signals in single-threaded programs is straightforward
- ✓ Delivering signals is more complicated in multithreaded programs

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

Example :

The standard UNIX function for delivering a signal is **kill(pid t pid, int signal)**

5.3 Thread Cancellation

Thread cancellation involves terminating a thread before it has completed.

- web page loads using several threads—each image is loaded in a separate thread.
- A thread that is to be canceled is often referred to as the **target thread**.

Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The default cancellation type is deferred cancellation. Here, cancellation occurs only when a thread reaches a **cancellation point.**

5.4 Thread-Local Storage

- Threads belonging to a process share the data of the process.
- Indeed, this data sharing provides one of the benefits of multithreaded programming.

Each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**.)

For example,

- ✓ In a transaction-processing system, we might service each transaction in a separate thread.
- ✓ Furthermore, each transaction might be assigned a unique identifier.

5.5 Scheduler Activations

- A final issue to be considered with multithreaded programs concerns **communication between the kernel** and **the thread library,** which may be required by the many-to-many and two-level models

- Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads.

This data structure—typically known as a **lightweight process**, or **LWP**

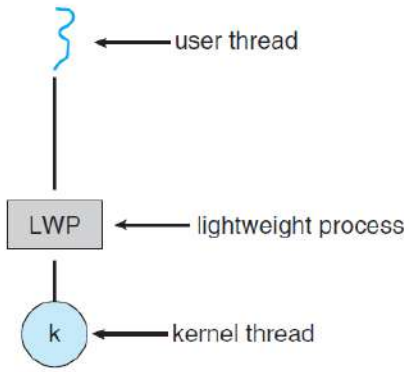


Figure 4.13 Lightweight process (LWP).

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**.

It works as follows:

- ➢ The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor the kernel must inform an application about certain events.

- ➢ This procedure is known as an **upcall**.

- Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor

6. CPU scheduling:

- When a computer is multi-programmed, it frequently has multiple processes or threads competing for the CPU at the same time.
- This situation occurs whenever two or more of them are simultaneously in the ready state.
- If only one CPU is available, a choice has to be made which process to run next.
- The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm.
- CPU scheduling is the basis of multi programmed operating systems.

6.1 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.

The various scheduling algorithms, a **ready queue** can be implemented as a **FIFO queue, a priority queue, a tree, or simply an unordered linked list**.

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

1. Pre-emptive
2. Non-pre-emptive

- ✓ A nonpreemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU.
- ✓ Even if it runs for many hours, **it will not be forcibly suspended**.

Example :

This scheduling method was used by Microsoft Windows 3.x;

- ✓ A preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time.
- ✓ If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is vailable).

Example :

- ✓ Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
- ✓ The Mac OS X operating system for the Macintosh uses preemptive scheduling

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, as the result of an I/O request or an 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, at completion of I/O)

4. When a process terminates

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

6.2 Dispatcher:

- Another component involved in the CPU-scheduling function is the **dispatcher.**
- The dispatcher is the **module** that gives control of the CPU to the process selected by the short-term scheduler.

This function involves the following:

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

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

6.3 Scheduling Criteria

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another.

- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

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.**

- ✓ 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 amount of time that a process spends waiting in the ready queue.
- ✓ *Waiting time* is the **sum** of the periods **spent waiting** in the **ready queue**.

• **Response time.**

- ✓ *response time,* is the time it takes to start responding, not the time it takes to output the response.

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

6.4 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.

This situation arises because different application areas (and different kinds of operating systems) have different goals.

6.4.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first.

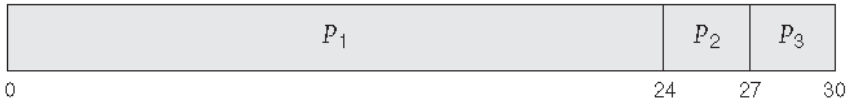
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- The code for FCFS scheduling is simple to write and understand.
- The FCFS scheduling algorithm is nonpreemptive.

- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

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

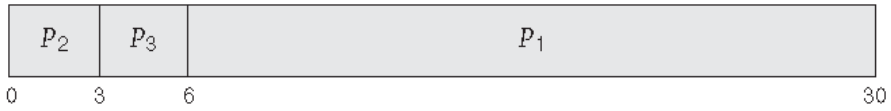
Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order *P*1, *P*2, *P*3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process *P*1, 24 milliseconds for process *P*2, and 27 milliseconds for process *P*3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order *P*2, *P*3, *P*1, however, the results will be as shown in the following Gantt chart:



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

"The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly."

There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

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.

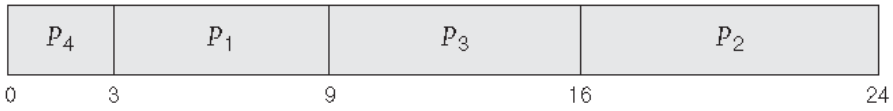
6.4.2 Shortest-Job-First Scheduling

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.

Process	Burst Time
*P*1	6
*P*2	8
*P*3	7
*P*4	3

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



The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

- The real **difficulty** with the **SJF algorithm is knowing the length of the next CPU request**.

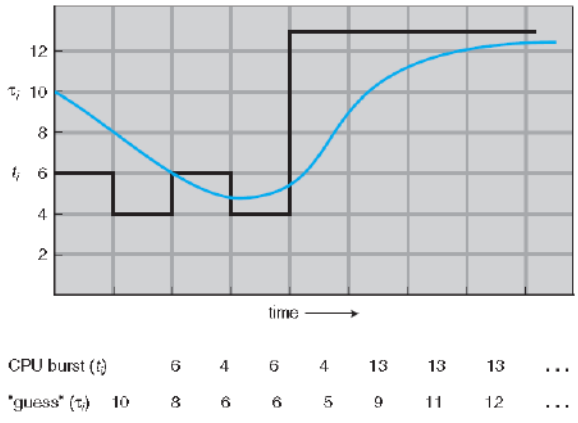
- SJF scheduling is used frequently in long-term scheduling.

One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. **We expect that the next CPU burst will be similar in length to the previous ones.** By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let t_n be the length of the n^{th} CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst.

Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$



The value of contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction.

To understand the behavior of the exponential average, we can expand the formula for $\tau n+1$ by substituting for τn to find.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

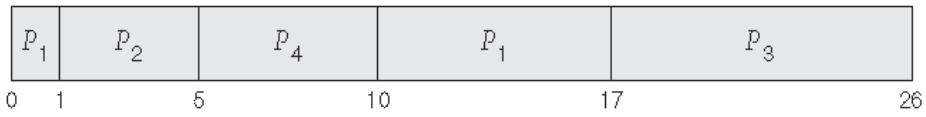
The SJF algorithm can be either **preemptive** or **nonpreemptive**.

- A pre-emptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example,

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	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 *P*1 is started at time 0, since it is the only process in the queue. Process *P*2 arrives at time 1. The remaining time for process *P*1 (7 milliseconds) is larger than the time required by process *P*2 (4 milliseconds), so process ***P*1 is preempted, and process *P*2 is scheduled**. The average **waiting time** for this example is [(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.

6.4.3 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm.

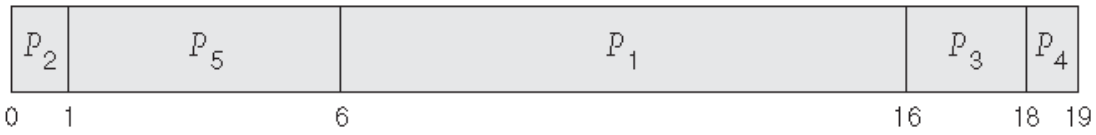
A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

- 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.

Example:

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



- Priority scheduling can be either preemptive or nonpreemptive.
- 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.

A solution to the problem of indefinite blockage of low-priority processes is **aging**.

- Aging involves gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

6.4.4 Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems.
It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

- ✓ A small unit of time, called a **time quantum** or **time slice**, is defined.
  - A time quantum is generally from **10 to 100** milliseconds in length.
  - The ready queue is treated as a **circular queue**.

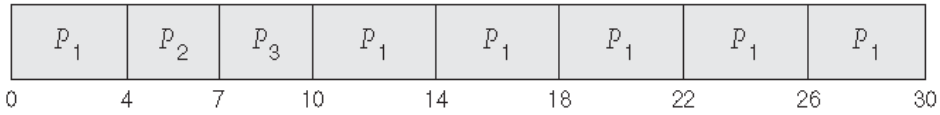
- To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes.

- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

Process	**Burst Time**
P_1	24
P_2	3
P_3	3

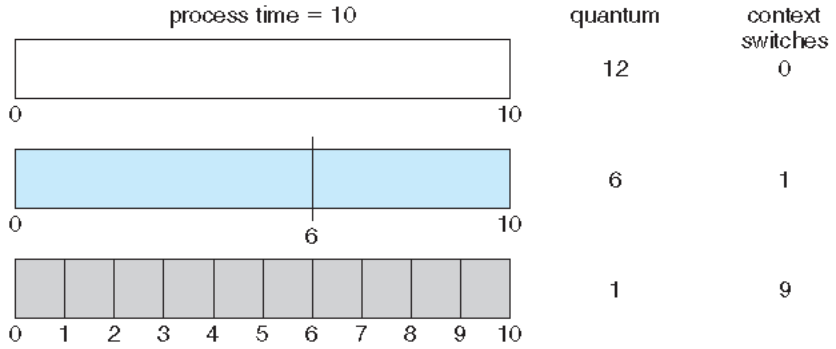
If we use a time quantum of 4 milliseconds,

The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. *P*1 waits for **6** milliseconds (10 - 4), *P*2 waits for **4** milliseconds, and *P*3 waits for **7** milliseconds.

Thus, the **average waiting time is 17/3 = 5.66 milliseconds.**

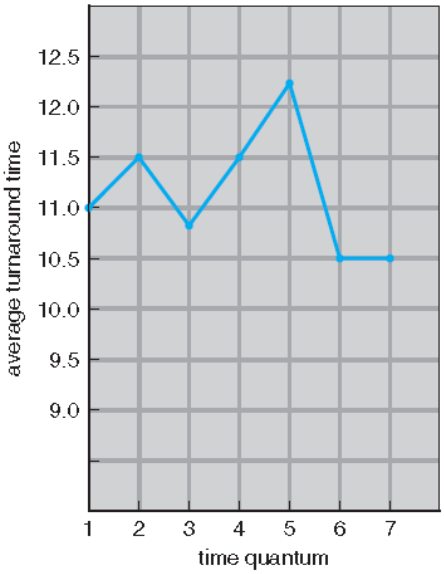


How a smaller time quantum increases context switches.

The performance of the RR algorithm depends heavily on the size of the time quantum.

At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.

In contrast, if the **time quantum is extremely small** (say, 1 millisecond), the RR approach can result in a **large number** of **context switches**.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

How turnaround time varies with the time quantum.

- **Turnaround time also depends on the size of the time quantum.**

Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the **time quantum is too large, RR scheduling degenerates to an FCFS policy.**

- **A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.**

6.4.5 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.

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.
- The foreground queue might be scheduled by an RR algorithm,
- While the background queue is scheduled by an FCFS algorithm.

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

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

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

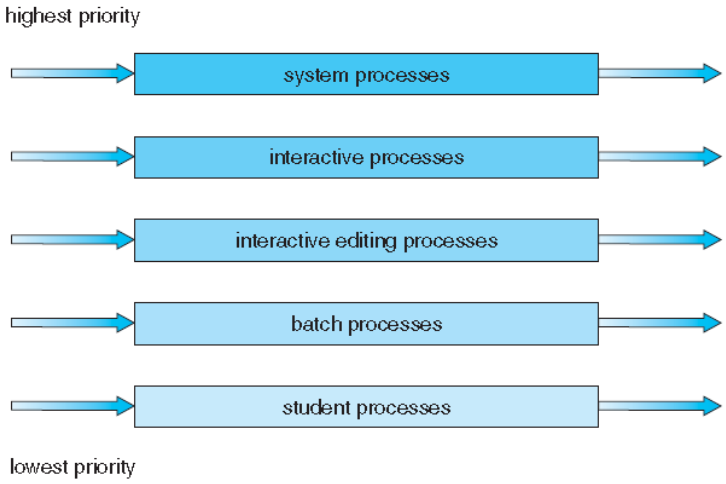


Figure 6.6 Multilevel queue scheduling.

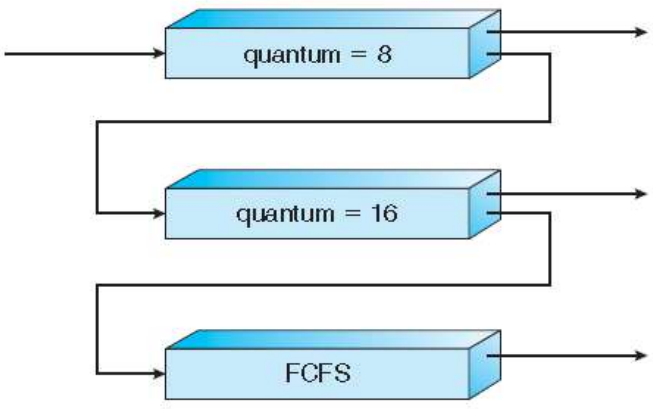
If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

6.4.6 Multilevel Feedback Queue Scheduling

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

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues.

- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.



Multilevel feedback queues.

Explanation :

- 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.

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

• The number of queues

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higher priority queue

• The method used to determine when to demote a process to a lower priority queue

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

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.

It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.