MARRI LAXMAN REDDY
Institute of Technology and Management
(UGC – AUTONOMOUS)

LECTURE NOTES ON

OPERATING SYSTEMS

Course Objectives:

1. To understand the OS role in the overall computer system
2. To study the operations performed by OS as a resource manager
3. To understand the scheduling policies of OS

Learning Outcomes:

The student will be able to

1. Learn about minimization of turnaround time, waiting time and response time and also maximization of throughput by keeping CPU as busy as possible.

2. Understand process concepts and process states.

3. Implement operations on processes

4. Understand system call interface for process management.

Program Outcomes

The Program Outcomes (POs) of the department are defined in a way that the Graduate Attributes are included, which can be seen in the Program Outcomes (POs) defined. The Program Outcomes (POs) of the department are as stated below:

a) An ability to apply knowledge of Science, Mathematics, Engineering & Computing fundamentals for the solutions of Complex Engineering problems.
b) An ability to identify, formulates, research literature and analyze complex engineering problems using first principles of mathematics and engineering sciences.
c) An ability to design solutions to complex process or program to meet desired needs.
d) Ability to use research-based knowledge and research methods including design of experiments to provide valid conclusions.

e) An ability to use appropriate techniques, skills and tools necessary for computing practice.
f) Ability to apply reasoning informed by the contextual knowledge to assess social issues, consequences & responsibilities relevant to the professional engineering practice.
g) Ability to understand the impact of engineering solutions in a global, economic, environmental, and societal context with sustainability.
h) An understanding of professional, ethical, Social issues and responsibilities.
i) An ability to function as an individual, and as a member or leader in diverse teams and in multidisciplinary settings.
j) An ability to communicate effectively on complex engineering activities within the engineering community.
k) Ability to demonstrate and understanding of the engineering and management principles as a member and leader in a team.
l) Ability to engage in independent and lifelong learning in the context of technological change.

Program Educational Objectivies

PEO1: Establish a successful professional career in industry, government or academia.

PEO2: Gain multidisciplinary knowledge providing a sustainable competitive edge in higher studies or Research.

PEO3: Promote design, analyze, and exhibit of products, through strong communication, leadership and ethical skills, to succeed an entrepreneurial.

Scope:

Process management is an integral part of any modern day operating system. The operating system must allocate resources to processes, enable processes to share and exchange information, protect the resources of each process from other processes and enable synchronization among processes. To meet these requirements, the operating system must maintain a data structure for each process, which describes the state and resource ownership of that process and which enables the operating system to exert control over each process.

UNIT-II

Process Management

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.

The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Early computers 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, contemporary computer systems allow multiple pro-grams to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various pro-grams; 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: operating-system 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. In this chapter, you will read about what processes are and how they work.

OBJECTIVES

1. To introduce the notion of a process, forms the basis of all computation.
2. To describe the various features of processes, including scheduling, creation, and termination.
3. To explore interprocess communication using shared memory and message passing.
4. To describe communication in client – server systems.
5. To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
6. To present both software and hardware solutions of the critical-section problem.
7. To examine several classical process-synchronization problems.
8. To explore several tools that are used to solve process synchronization problems.
9. To introduce CPU scheduling, the basis for multiprogrammed operating systems.
10. To describe various CPU-scheduling algorithms.
11. To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
12. To examine the scheduling algorithms of several operating systems.

Process Concept

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**.

The Process

Informally, a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), 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 2.1.

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. 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).

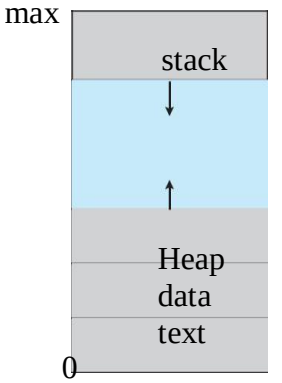


Figure 2.1 Process in memory.

Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

New. The process is being created.

Running. Instructions are being executed.

Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Ready. The process is waiting to be assigned to a processor.

Terminated. The process has finished execution.

The state diagram corresponding to these states is presented in Figure2.2.

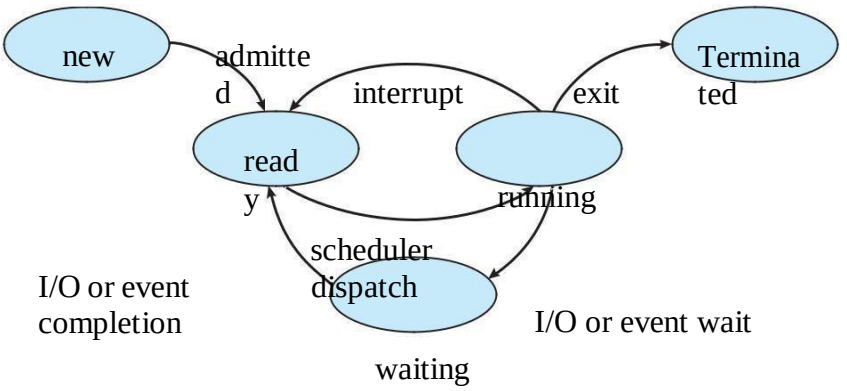


Figure 2.2 Diagram of process state

3.1.3 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:

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 (Figure 3.4).

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.

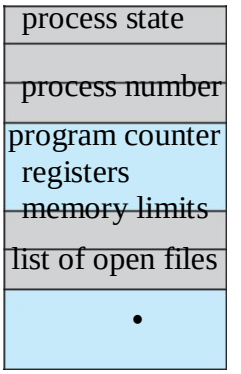


Figure 2.3 Process control block (PCB).

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.

2.1.4 Threads

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

Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

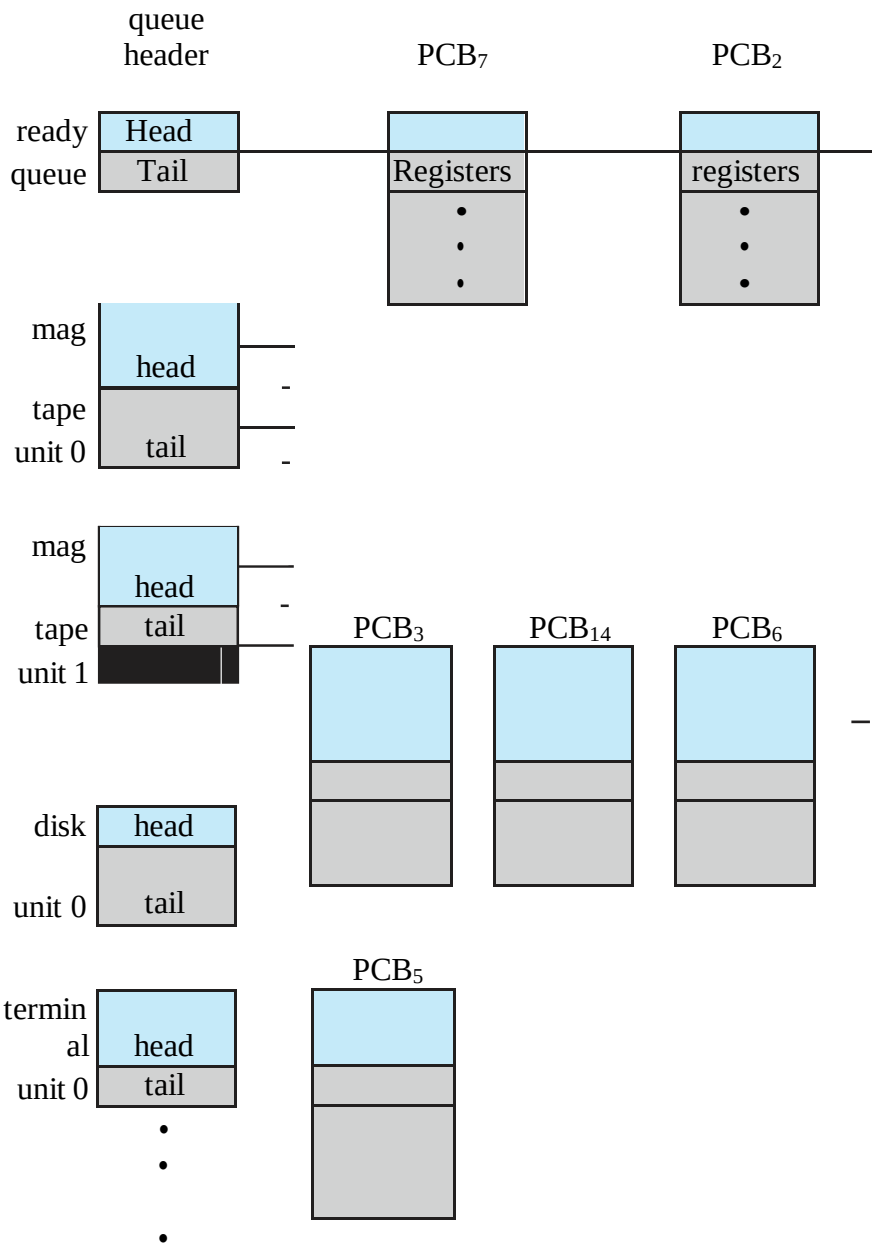


Figure 2.4 The ready queue and various I/O device queues.

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.

The 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 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 (Figure 2.4).

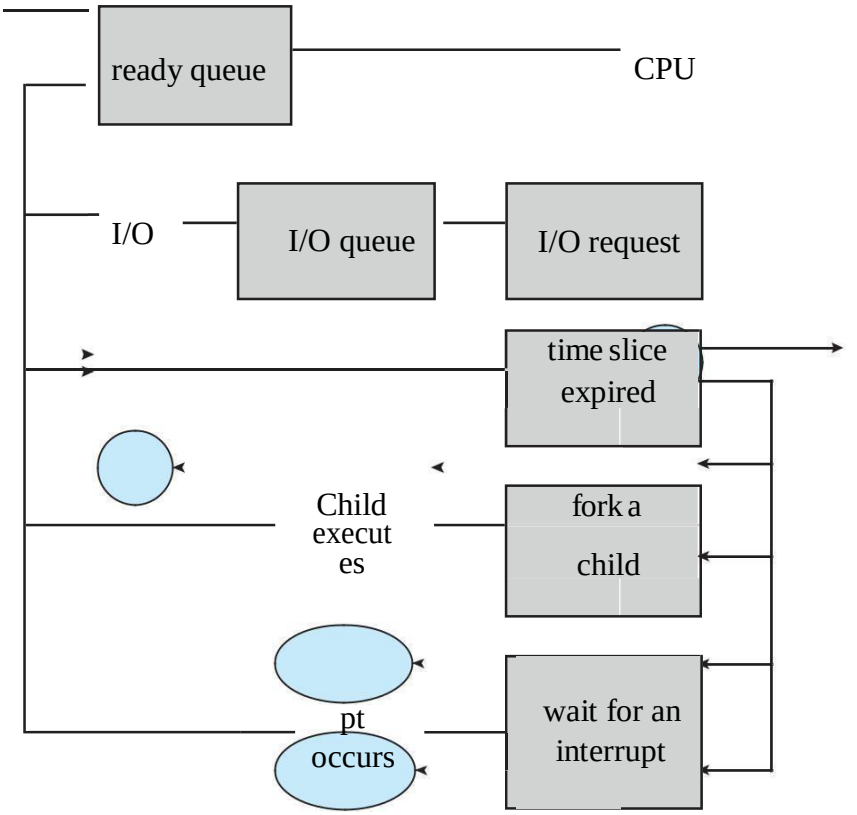


Figure 2.5 Queueing-diagram representation of process scheduling.

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 2.5. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

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

The process could issue an I/O request and then be placed in an I/O queue.

The process could create a new child process and wait for the child's termination.

The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

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

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 long-term scheduler executes much less frequently. 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.

This **medium-term scheduler** is diagrammed in Figure 2.6. 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.

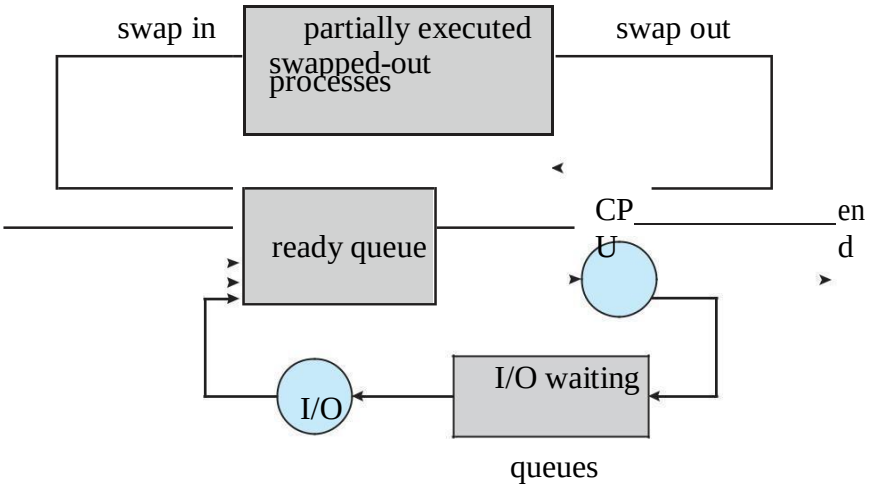


Figure 2.6 Addition of medium-term scheduling to the queueing diagram.

Context Switch

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 (see Figure 2.2), 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 performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. 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.

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.

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.

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. In Figure 2.7, we see two children of init— kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush). The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell***). The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ps as well as the emacs editor.

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

```
ps -el
```

will list complete information for all processes currently active in the system. It is easy to construct a process tree similar to the one shown in Figure 3.8 by recursively tracing parent processes all the way to the init process.

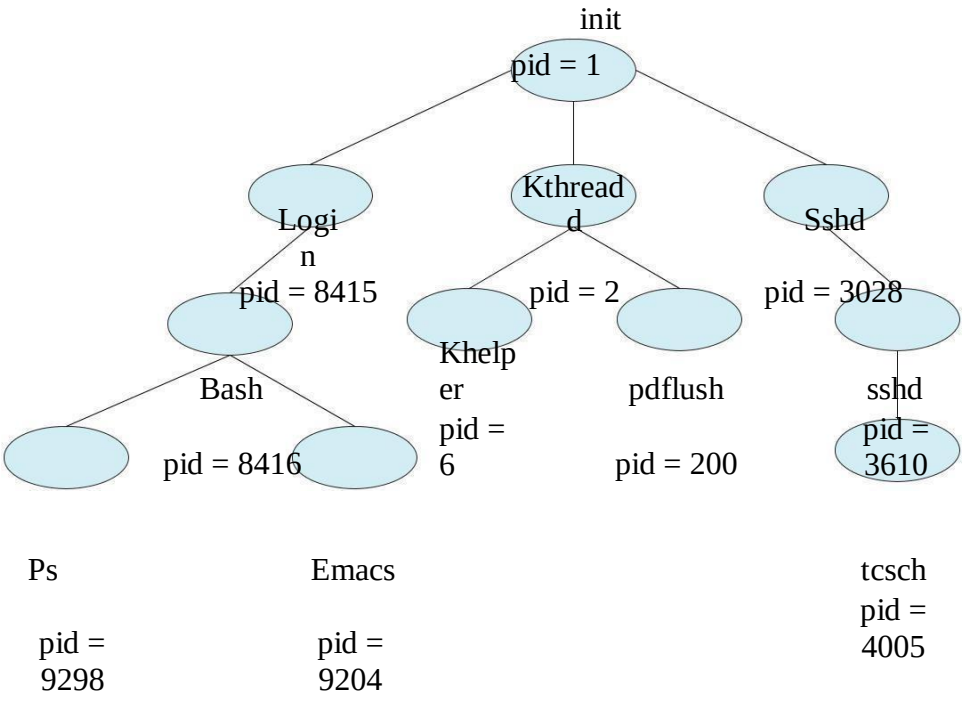


Figure 2.7 A tree of processes on a typical Linux system.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process 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.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file — say, image.jpg— on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file ***image.jpg***. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, image.jpg and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:
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 address-space possibilities for 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. 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.

After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (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. Because the call to exec() overlays the process's address space with a new program, the call to exec() does not return control unless an error occurs.

We now have two different processes running copies of the same program. The only difference is that the value of pid (the process identifier) for the child process is zero, while that for the parent is an integer value greater than

zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() 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 exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in Figure 2.8.

Of course, there is nothing to prevent the child from ***not*** invoking exec() and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

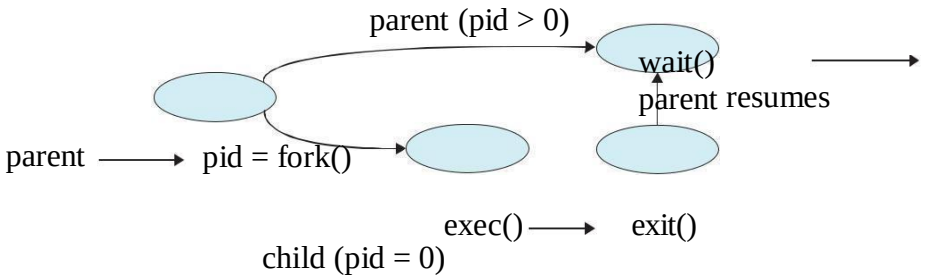


Figure 2.8 Process creation using the fork() system call.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows 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.

A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be

terminated 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.

Some systems 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 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);
```

In fact, under normal termination, exit() may be called either directly (as shown above) or indirectly (by a return statement in main()).

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.

Interprocess Communication

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:

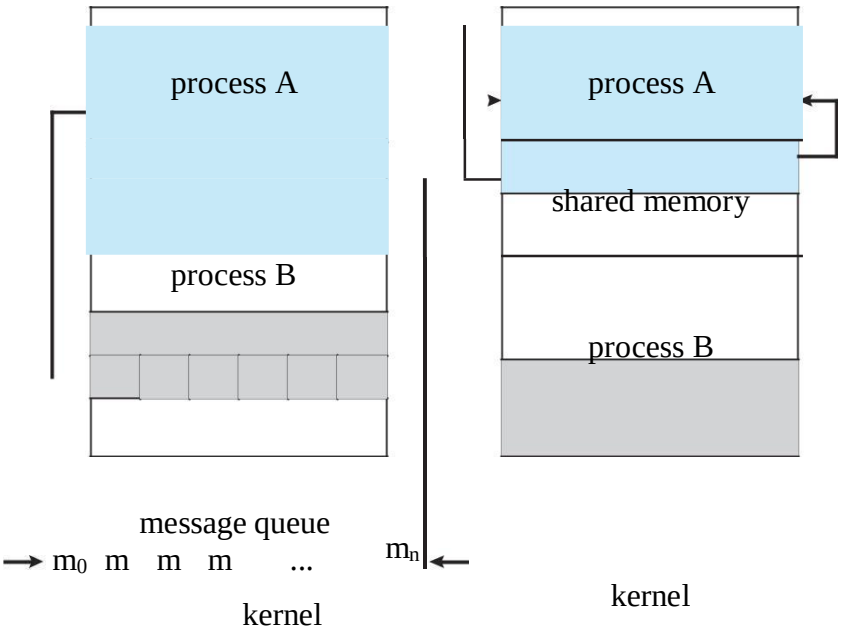
Information sharing. Several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

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.

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, listening to music, and compiling in parallel.

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: **shared memory** and **message passing**. In the 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. 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 2.9.



(a) Message passing (b) (b) Shared memory.

Figure 2.9 Communications models.

Shared-Memory Systems

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. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

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 that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. 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.

also provides a useful metaphor for the client – server paradigm. We generally think of a server as a producer and a client as a consumer. 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.

One solution to the producer – consumer problem uses shared memory. 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. 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's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

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

item        buffer[BUFFER
SIZE]; int in = 0;
int out = 0;
```

Figure: 2.11The producer process using shared memory

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 code for the producer process is shown in Figure 2.11, and the code for the consumer process is shown in Figure 2.12. The producer process has a

```
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 2.12 The consumer process using shared memory.

local variable next produced in which the new item to be produced is stored. The consumer process has a local variable next consumed in which the item to be consumed is stored.

This scheme allows at most BUFFER SIZE − 1 items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which BUFFER SIZE items can be in the buffer at the same time.

Message-Passing Systems

Shared-memory environment scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

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. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

send(message) receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straight-forward. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

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

Naming

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. Changing the identifier of a process may necessitate examining all other process definitions.

With ***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. For example, POSIX message queues use an integer value to identify a mailbox. 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.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a receive() from A. Which process will receive the message sent by P_1? The answer depends on which of the following methods we choose:

Allow a link to be associated with two processes at most.

Allow at most one process at a time to execute a receive() operation.

Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, ***round robin,*** where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send

messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

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.

Buffering

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

```
message next produced;
while (true) {
        /* produce an item in next produced */
        send(next produced);
}
```

Figure 2.13 The producer process using message passing.

Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

Bounded capacity. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

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.

Pipes

A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations

Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer – consumer fashion: the producer writes to one end of the pipe (the **write-end**) and the consumer reads from the other end (the **read-end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. In both program examples, one process writes the message Greetings to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

pipe(int fd[])

This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end.

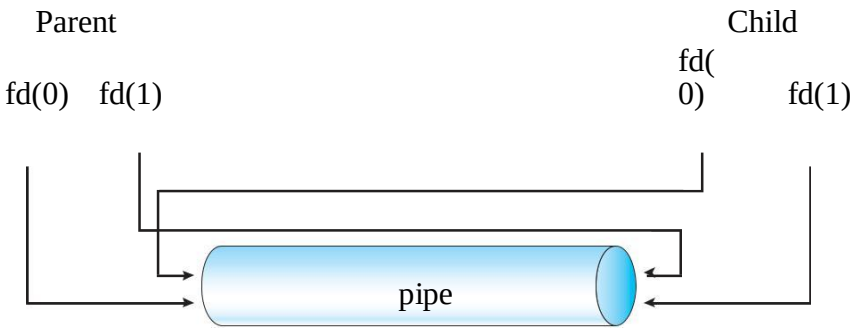


Figure 2.14 File descriptors for an ordinary pipe.

UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via fork().

Named Pipes

Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent – child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(), and close() system calls. It will continue to exist until it is explicitly deleted from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the CreateNamedPipe() function, and a client can connect to a named pipe using ConnectNamedPipe(). Communication over the named pipe can be accomplished using the ReadFile() and WriteFile() functions.

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

In a single-processor system, only one process can run at a time. 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 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 (Figure 2.34).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer. 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.

•

•

•

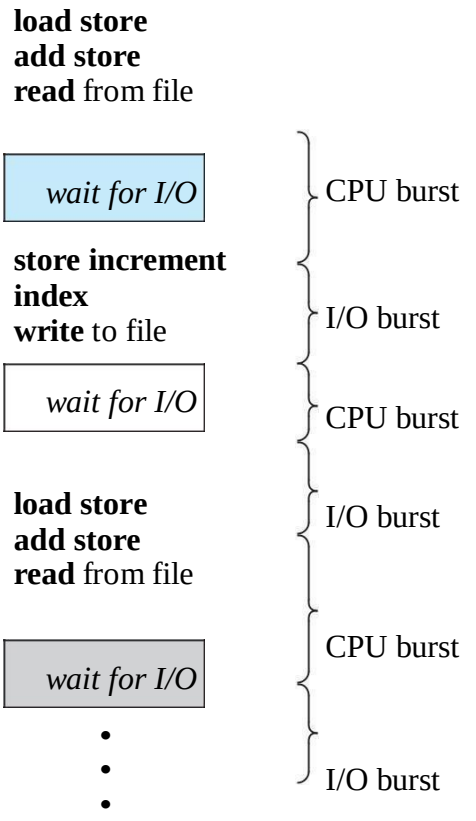


Figure 2.34 Alternating sequence of CPU and I/O bursts.

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. As we shall see when we consider 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. 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:

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 a child process)

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

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

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 was used by Microsoft Windows 3.x. 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 also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. 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 can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process 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.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting

real-time computing where tasks must complete execution within a given time frame.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

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**.

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. Concep-tually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

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

Turnaround time. 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, we prefer 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.

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. In this section, we describe several of them.

First-Come, First-Served Scheduling

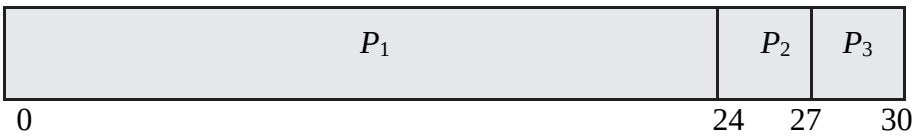
By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready 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.

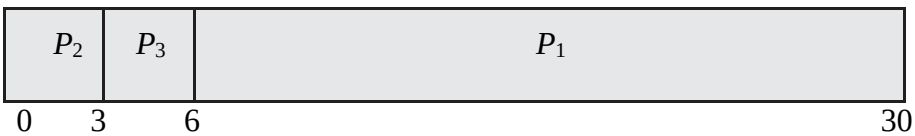
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

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

Note also that 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 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.

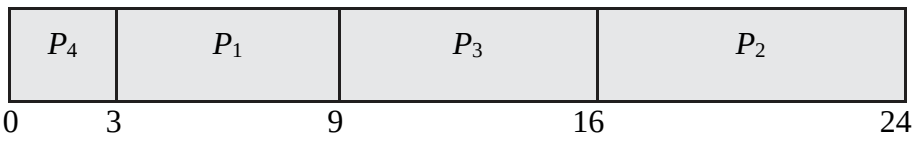
Shortest-Job-First Scheduling

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.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

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. 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 real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. In this situation, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response but too low a value will cause time-limit-exceeded error and require resubmission. SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next

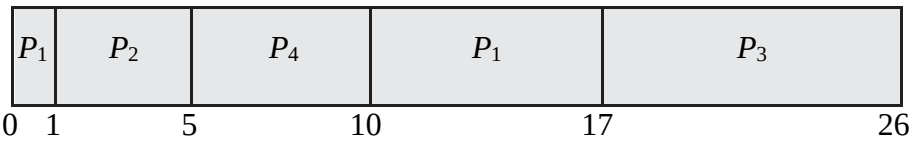
CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

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.

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

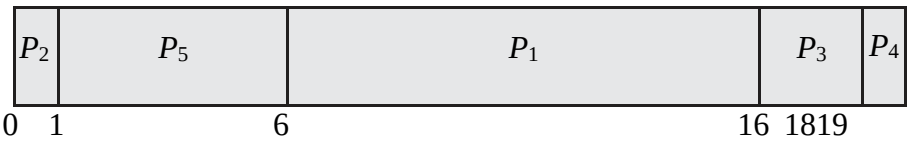
FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

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. In this text, 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 P_1, P_2, · · · , P_5, with the length of the CPU burst given in milliseconds:

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

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



The average waiting time is 8.2 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 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. 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 time-sharing 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.

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

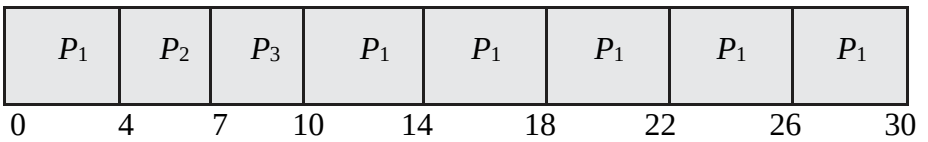
One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider 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 we use a time quantum of 4 milliseconds, then process P_1 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 P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. 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.

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.

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.

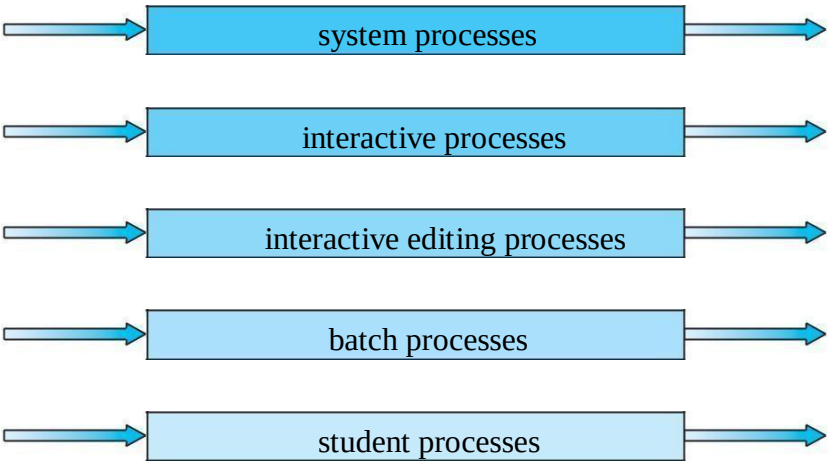
A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues (Figure 2.35). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling 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.

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

System processes
Interactive processes
Interactive editing processes
Batch processes
Student processes

highest priority



lowest priority

Figure 2.35 Multilevel queue scheduling.

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground – background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Multilevel Feedback Queue Scheduling

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

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

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 2.36). 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 be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

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 is also the most complex algorithm, since

defining the best scheduler requires some means by which to select values for all the parameters.

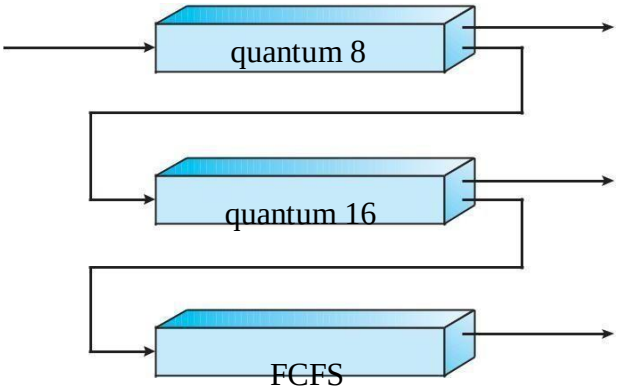


Figure 2.36 Multilevel feedback queues.

Thread Scheduling

On operating systems that support them, it is kernel-level threads — not processes — that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled.The thread library schedules user-level threads to run on an available LWP. This scheme is known as **process-contention scope (PCS)**, since competition for the CPU takes place among threads belonging to the same process. (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**. Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.

Typically, PCS is done according to priority — the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is

important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 6.3.4) among threads of equal priority.

Pthread Scheduling

We highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD SCOPE PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.6.5). The PTHREAD SCOPE SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC provides two functions for getting — and setting — the contention scope policy:

```
pthread attr setscope(pthread attr t *attr, int scope)
pthread attr getscope(pthread attr t *attr, int *scope)
```

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread attr setscope() function is passed either the PTHREAD SCOPE SYSTEM or the PTHREAD SCOPE PROCESS value, indicating how the contention scope is to be set. In the case of pthread attr getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, **load sharing** becomes possible — but scheduling problems become correspondingly more complex. Many possibilities have been tried; and as we saw with single-processor CPU scheduling, there is no one best solution.

Here, we discuss several concerns in multiprocessor scheduling. We concentrate on systems in which the processors are identical — homogeneous — in terms of their functionality. We can then use any available processor to run any process in the queue. Note, however, that even with homogeneous multiprocessors, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.

Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread t tid[NUM THREADS];
   pthread attr t attr;

   /* get the default attributes */
   pthread attr init(&attr);

   /* first inquire on the current scope */
   if (pthread attr getscope(&attr, &scope) != 0) fprintf(stderr,
      "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD SCOPE PROCESS)
        printf("PTHREAD SCOPE PROCESS");
      else if (scope == PTHREAD SCOPE SYSTEM)
        printf("PTHREAD SCOPE SYSTEM");
      else
        fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */ pthread attr
   setscope(&attr, PTHREAD SCOPE SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
        pthread create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}

/* Each thread will begin control in this function */ void
*runner(void *param)
{
   /* do some work ... */
```

```
    pthread_exit(0);
}
```

Figure 2.36 Pthread scheduling API.

A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. As we saw in Chapter 5, if we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully. We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue. Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.

Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity** — that is, a process has an affinity for the processor on which it is currently running.

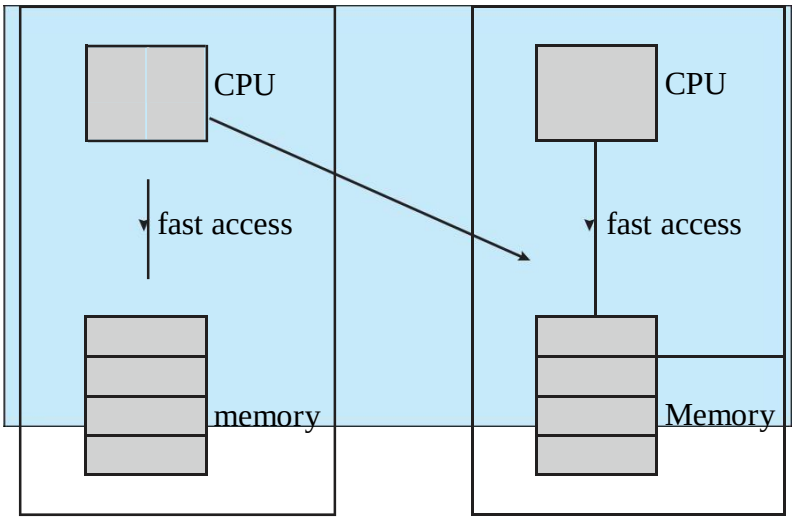
Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor — but not guaranteeing that it will do so — we have a situation known as **soft affinity**. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors. In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it may run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity.

The main-memory architecture of a system can affect processor affinity issues. Figure 2.37 illustrates an architecture featuring non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory than to other parts. Typically, this occurs in systems containing combined CPU and memory boards. The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system. If the operating system's CPU scheduler and memory-placement algorithms work together, then a

process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides. This example also shows that operating systems are frequently not as cleanly defined and implemented as described in operating-system textbooks. Rather, the ―solid lines‖ between sections of an operating system are frequently only ―dotted lines,‖ with algorithms creating connections in ways aimed at optimizing performance and reliability.

Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.



computer

Figure 2.37 NUMA and CPU scheduling.

Otherwise, one or more processors may sit idle while other processors have high workloads, along with lists of processes awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue. It is also important to note, however, that in most contemporary operating systems supporting SMP, each processor does have a private queue of eligible processes.

There are two general approaches to load balancing: **push migration** and **pull migration**. With push migration, a specific task periodically checks the load on each processor and — if it finds an imbalance — evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems. For example, the Linux schedulercand the ULE scheduler available for FreeBSD systems implement both techniques.

The benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory. Either pulling or pushing a process from one processor to another removes this benefit. As is often the case in systems engineering, there is no absolute rule concerning what policy is best. Thus, in some systems, an idle processor always pulls a process from a non-idle processor. In other systems, processes are moved only if the imbalance exceeds a certain threshold.

Multicore Processors

Traditionally, SMP systems have allowed several threads to run concurrently by providing multiple physical processors. However, a recent practice in computer

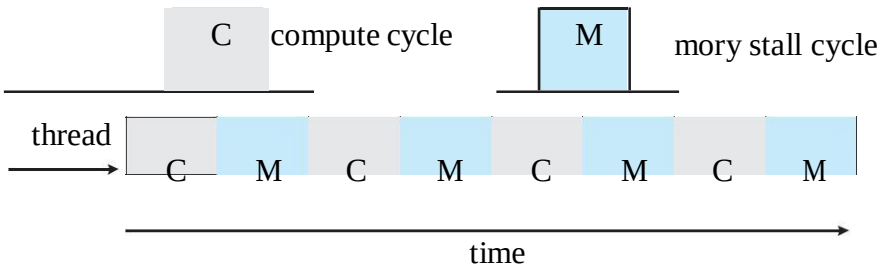


Figure 2.38 Memory stall.

hardware has been to place multiple processor cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor. SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, may occur for various reasons, such as a cache miss (accessing data that are not in cache memory). Figure 2.38 illustrates a memory stall. In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory. To remedy this situation, many recent hardware designs have implemented multithreaded processor cores in which two (or more) hardware threads are assigned to each core. That way, if one thread stalls while waiting for memory, the core can switch to another thread. Figure 2.39 illustrates a dual-threaded processor core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating-system perspective, each hardware thread

appears as a logical processor that is available to run a software thread. Thus, on a dual-threaded, dual-core system, four logical processors are presented to the operating system. The UltraSPARC T3 CPU has sixteen cores per chip and eight hardware threads per core. From the perspective of the operating system, there appear to be 128 logical processors.

In general, there are two ways to multithread a processing core: **coarse-grained** and **fine-grained** multithreading. With coarse-grained multithreading, a thread executes on a processor until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the processor must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity — typically at the boundary of an instruction cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

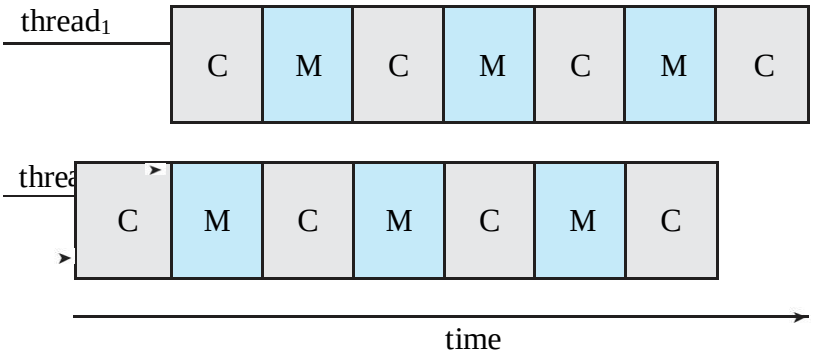


Figure 2.39 Multithreaded multicore system.

Notice that a multithreaded multicore processor actually requires two different levels of scheduling. On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical processor). For this level of scheduling, the operating system may choose any scheduling algorithm. A second level of scheduling specifies how each core decides which hardware thread to run. There are several strategies to adopt in this situation. The UltraSPARC T3, mentioned earlier, uses a simple round-robin algorithm to schedule the eight hardware threads to each core. Another example, the Intel Itanium, is a dual-core processor with two hardware-managed threads per core. Assigned to each hardware thread is a dynamic *urgency* value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest. The Itanium identifies five different events that may trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the

processor core.

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process.

  Cooperating process can affect or be affected by the execution of another process.

  Advantages of process cooperation
- Information sharing
- Computation speed-up
- Modularity
- Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- *unbounded-buffer* places no practical limit on
- the size of the buffer *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
        while (true) {
  /* Produce an item */
     while (((in = (in + 1) % BUFFER SIZE count) == out)
            ;   /* do nothing
           -- no free buffers
           */ buffer[in] =
           item;
           in = (in + 1) % BUFFER SIZE;
   }
```

Bounded Buffer – Consumer

```
while (true) {
        while (in == out)
```

```
        ; // do nothing --
        nothing to
        consume //
        remove an item
        from the buffer
        item =
        buffer[out];
        out = (out + 1) % BUFFER SIZE;
    return item;
}
```

SYSTEM CALL INTERFACE FOR PROCESS MANAGEMENT

Process management uses certain system calls. They are explained below.

1. To create a new process – fork () is used.
2. To run a new program = exec () is used.
3. To make the process to wait = wait () is used.
4. To terminate the process – exit () is used..

fork

A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

Fork

The fork system call is the primary (and historically, only) method of process creation in Unix-like operating systems.

```
#include <unistd.h>

pid_t fork(void);
```

Wait

The wait system call blocks the caller until one of its child process terminates. If the caller doesn't have any child processes,returns immediately without blocking the caller. Using the parent can obtain the exit status of the terminated child.

```
#include
<sys/types.h>
#include
<sys/wait.h> pid_t
wait(int status);
```

Exit

Processes on a system terminate by executing the exit system call. The syntax for the call is
exit(status);

Where the value of status is returned to the parent process for its examination The exit() function terminate the calling function.

Waitpid: The waitpid function doesn't wait for the child that terminates first,it has a number of options that control which process it waits for.

The waitpid function provides three features that are not provided by the wait function.

1) The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.

2) The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status , but we don't want to block.

3) The waitpid function provides support for job control with the WUNtrACED and WCONTINUED options.

EXEC: The exec system call causes a calling process to change its content and execute a different program.

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.

University Questions

Part- A

1. What is preemptive scheduling? [JNTUH Nov/Dec-2018]
2. In general one process is not allowed to access the memory of another process, then how shared memory is working? [JNTUH MAY-2018]
3. Can a thread ever be preempted by a clock interrupt? If so, what circumstances? If not, why not? [JNTUH MAY-2018]
4. What do you mean turn around time? [JNTUH Nov/Dec-2017]
5. Define process control block. [JNTUH Nov/Dec-2017]
6. What is context switching? [JNTUH Feb/Mar -2016]
7. Define multithreading. [JNTUH Feb/Mar -2016]
8. Explain System call interface for process management. [JNTUH 19,20]

Part - B

1. Explain FCFS, RR, SJF scheduling algorithm with illustratins. [JNTUH Nov/Dec-2018]
2. Explain about multiple- processor scheduling and real- time scheduling. [JNTUH Nov/Dec-2018]
3. How parent and child relationship is created between processes? Explain how parent and child behave on its termination? [JNTUH MAY-2018]
4. Consider a system where counting semaphore initialized to +17, on this semaphore variable the various operations like 23P, 18V, 16P, and 1P are performed. Then what is the final value of semaphore? [JNTUH MAY-2018]
5. Describe the actions taken by a thread library to context switch between user level threads. [JNTUH MAY-2018]
6. Differentiate between Long term, Short term, Medium term Scheduler. [JNTUH Nov/Dec-2017]
7. By illustrating the structure of process P1, explain the Petersons solution to critical section problem. [JNTUH Nov/Dec-2017]
8. Discuss in detail about the Dining – Philosophers solution using monitors. [JNTUH Feb/Mar -2016]
9. Illustrate the semaphore functions with examples. [JNTUH Feb/Mar -2016]
10. Explain the following: (a) Multilevel Q Scheduling
    (b) Multilevel Feedback Scheduling (c) Real time Scheduling. [JNTUH Feb/Mar -2016]
11. Explain in detail about the semaphores and monitors. [JNTUH Feb/Mar -2016]

12. What is scheduler? State and explain the various states of a process with a neat diagram. [JNTUH NOV/DEC-2015]

13. What is monitor? How are monitors used in solving Dining Philosophers problem? Explain. [JNTUH NOV/DEC-2015]

14. What is starvation? Illustrate with an example [JNTUH NOV/DEC-2015]

15. Consider the following four processes, with the length of the CPU burst given in milliseconds.

Process	AT	BT
P1	0	5
P2	1	6
P3	2	2
P4	3	8

Calculate the average waiting time and average turnaround time for non preemptive SJF scheduling.

16. State and explain critical section problem. [JNTUH DEC-2014]

17. What is monitor? How are monitors used in solving the dining philosophers problem?

Tutorial Questions

1. For the following example. Calculate average turn around time and average waiting time for the FCFS algorithm.

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

2. Draw the gnatt chart for the above data.
3. Draw the gnatt chart using priority scheduling for preemptive. A larger number has the highest priority

Process	Arrival Time	Burst Time	Priority
P1	0.0	6	4
P2	3.0	5	2
P3	3.0	3	6
P4	5.0	5	3

4. Calculate the turnaround time for each scheduling algorithm.
5. What is the waiting time of each process for each one of the scheduling algorithm using FCFS and SJF

Process	WT
P1	0
P2	10
P3	11
P4	13
P5	14

6. Consider the following process with the CPU burst time given in milliseconds

Process	BT	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

   Calculate turn around time for each scheduling algorithm.
7. Calculate the waiting time for the above processes using FCFS, SJF.
8. Calculate the average waiting time and average turnaround time for the above the processes using FCFS, SJF.

9. Schedule the following process in SRTF. Find average waiting time.

Process	AT	BT
P1	0	20
P2	15	25

P3	30	10
P4	45	15

10. Schedule the above process in SRTF. Find average turnaround time.
11. Draw the gnatt chart to the above processes using SJF.
12. Draw the gnatt chart to the above processes using FCFS.
13. Draw the gnatt chart to the above processes using Priority.
14. Draw the gnatt chart to the above processes using round robin with ts=2 sec.
15. Draw the gnatt chart to the above processes using SRTF.