UNIT-2

(A30516) OPERATING SYSTEMS

UNIT-I
Operating System Introduction, Structures - Simple Batch, Multi-programmed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls.

UNIT –II
Process and CPU Scheduling - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads, and Interposes Communication, Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling.
System call interface for process management-fork, exit, wait, waitpid, exec

PROCESS CONCEPTS

I)THE PROCESS: A Process is program in execution.

For example: A user may run program, word processor, a web browser and email.

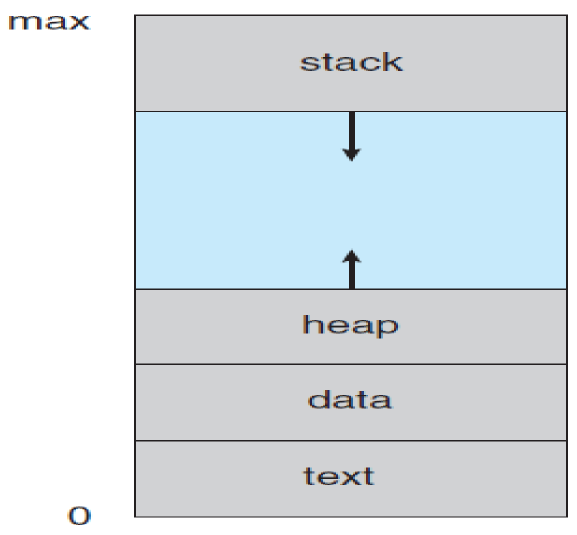


Figure 3.1 Process in memory.

A process generally also includes the process stack, which contains temporary data(such as function parameters, return address and local variables) and data section which contains global variables.

A process may also include a Heap, which is memory that is dynamically allocated during process runtime.

A program is passive entity such as a file containing a list of instructions stored on disk(executables files).

Whereas a process is an active entity, with a program counter and a set of associated resources.

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

There are two common techniques for loading executable files are double clicking icon representing the executable file and entering the name of the executable file on command line.

II)PROCESS STATE DIAGRAM:

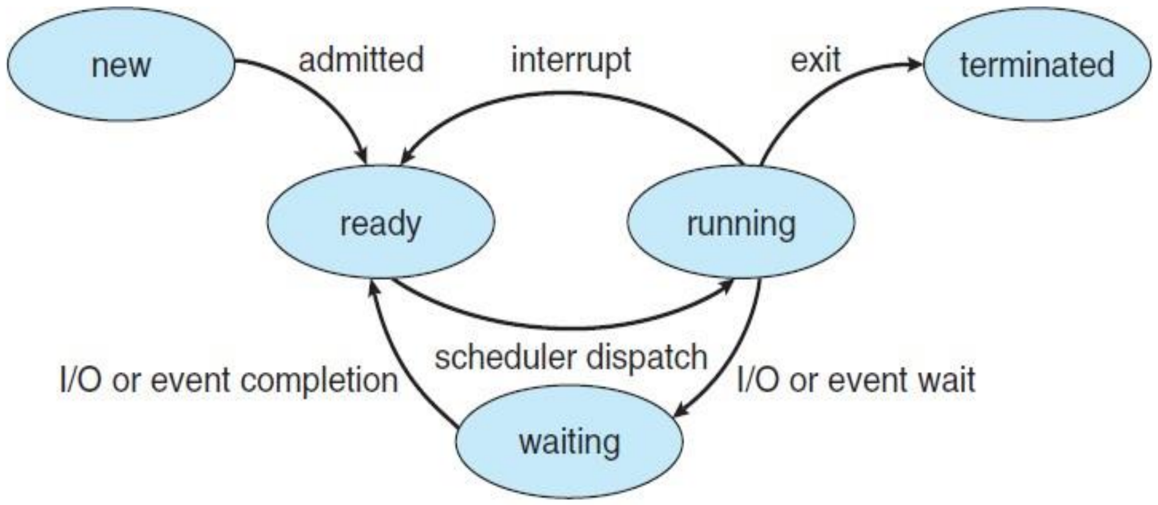


Figure 3.2 Diagram of process state.

- **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.

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

It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting state.

PROCESS CONTROL BLOCK.

Each process is represented in the OS by a Process Control Block(PCB), also called as "Task Control Block".

It contains many pieces of information associated with specific process.

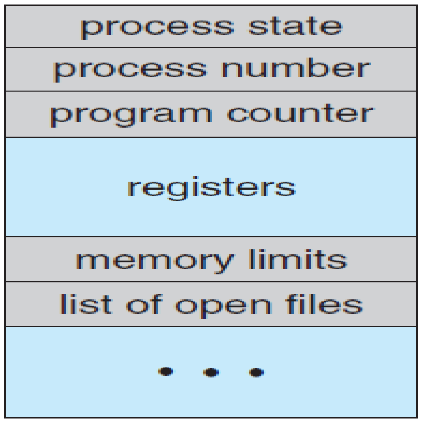


Figure 3.3 Process control block (PCB).

1. Process state
2. Program Counter
3. CPU Registers
4. CPU Scheduling information
5. Memory management information
6. Accounting information
7. I/O Status information

Process state. The state may be new, ready, running, waiting, halted, and so on.
Program counter. The counter indicates the address of the next instruction to be executed for this process.
CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

THREADS

A process is program that performs a single thread of execution.

A thread is a basic unit of CPU utilization; it comprises a thread id, a program counter, a register set and a stack.

It shares with other threads belonging to the same process its code section, data section and other OS resources.

For ex 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.

Many modern OS's have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. On a system that supports threads, the pcb is expanded to include information for each thread.

PROCESS SCHEDULING

SCHEDULING QUEUES

Job Queue: As processes enter the system they are put into the job queue. This consists of all processes in the system.

Ready Queue: The processes that are residing in the main memory and they are Ready and waiting to execute are kept on a list called the Ready Queue.

Device Queue: The list of processes waiting for a particular I/O device is called a Device Queue.

Each Device has its own Device Queue.

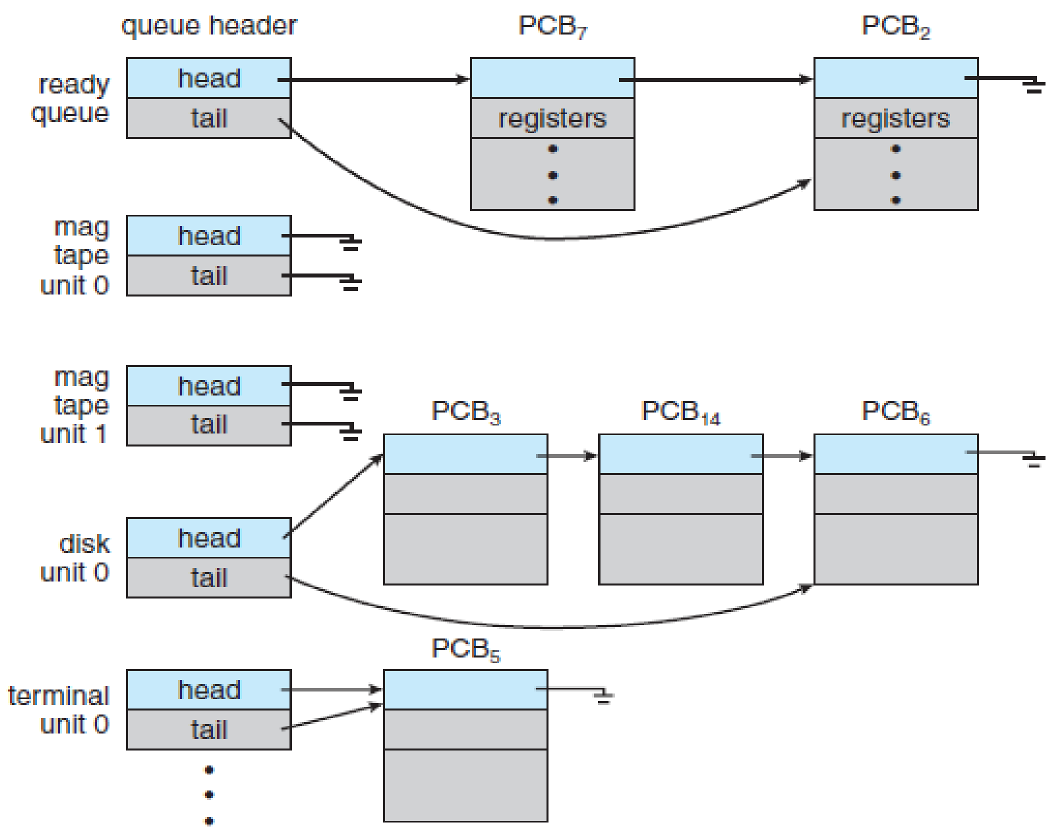


Figure 3.5 The ready queue and various I/O device queues.

The queue is generally stored as a linked list.

A ready queue header contains pointers to the first and final PCB's in the list.

Each PCB includes a pointer field that points to the next PCB in the Ready Queue.

Queuing Diagram.

A new process is initially put in the Ready Queue. And it waits until it is selected for its execution.

Once the process is allocated the CPU and is executing, one of the several events could occur.

A common representation of process scheduling is queuing diagram.

In this each rectangular box represents a queue(Ready and I/O queue).

The circle represents the resources that serve the queues and arrows indicate the flow of process in the system.

1. The process could issue I/O request and then be placed in I/O queue.
2. The process could create a new sub process and wait for the sub process termination.
3. The process could be removed forcefully from the CPU as a result of an interrupt and then be put back in the Ready Queue.

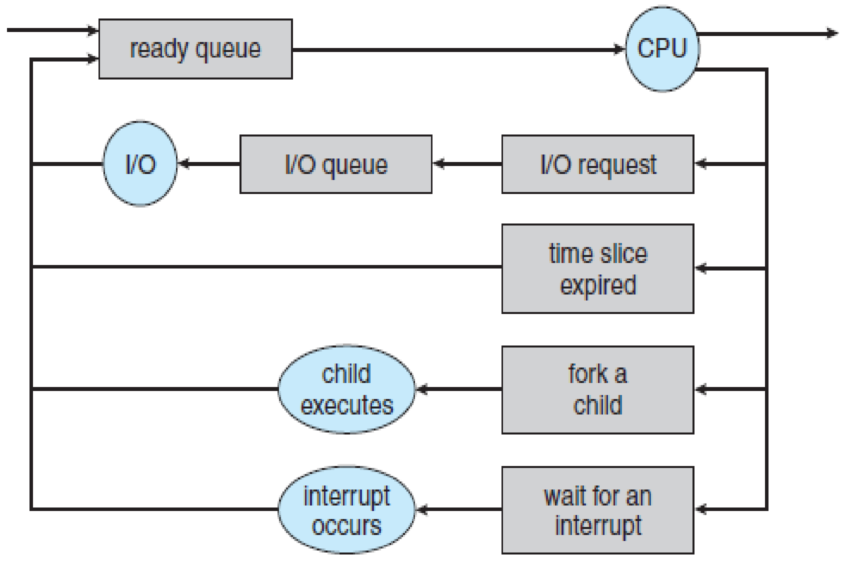
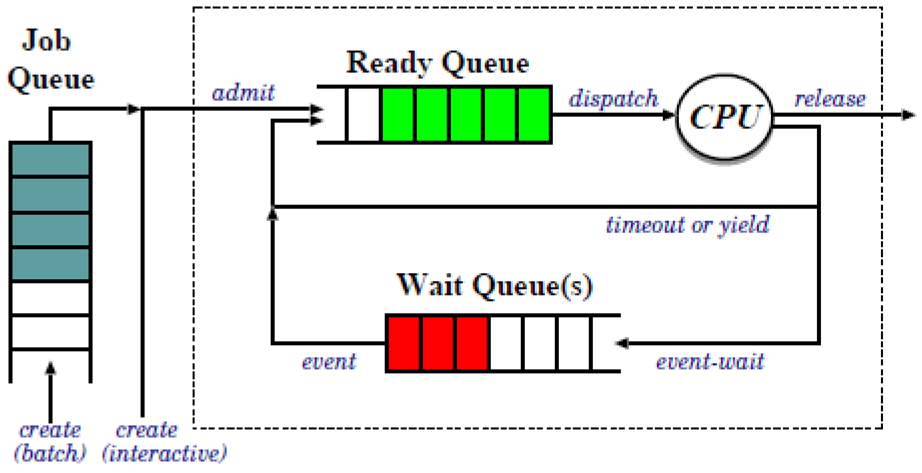


Figure 3.6 Queueing-diagram representation of process scheduling.



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

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**.

1. **LONG TERM SCHEDULERS**
2. **SHORT TERM SCHEDULERS**
3. **MEDIUM TERM SCHEDULERS**

1. Long Term Schedulers Or Job Scheduler
   Which selects processes form this pool and loads them into memory for execution?
2. Short term scheduler or CPU Scheduler:
   Which selects from among the process that is ready to execute and allocate the CPU to one of them?
   The short term scheduler must select a new process for the CPU frequently.
   Difference between long and short term scheduler:
   The long term scheduler will execute much less frequently.
   It is important long term scheduler makes a careful selection.
   In general most processes can be either I/O Bound process or CPU Bound process. I/O Bound process is one that spends more of its time doing I/O. CPU Bound process 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 CPU bound the I/O waiting queue always be empty.
If all processes are I/O bound the ready queue, the ready queue will be empty.

3.Medium Term Scheduler

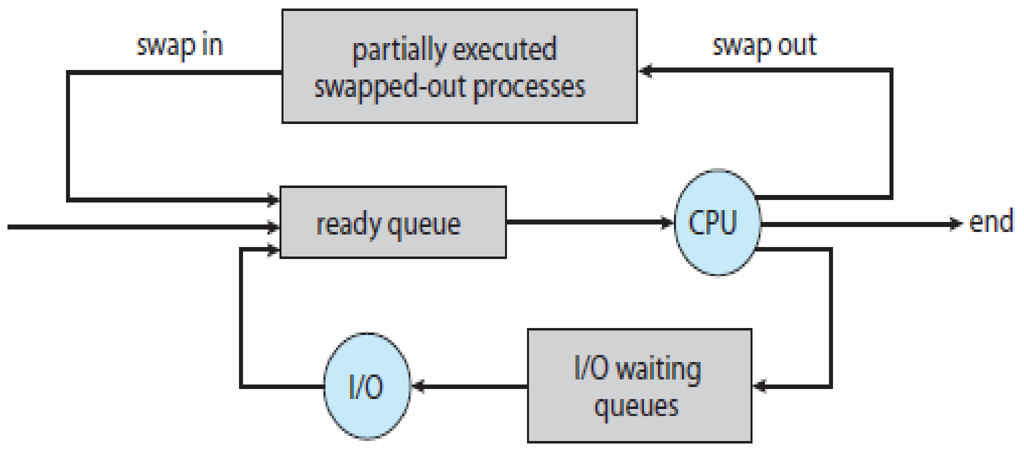


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Some operating systems such as time sharing systems may introduce an additional scheduler.
The medium term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
The process is swapped out and is later swapped in by medium term scheduler.

Later the process can be reintroduced into memory and its execution can be continued where it left. This scheme is called **swapping**.

CONTEXT SWITCH:

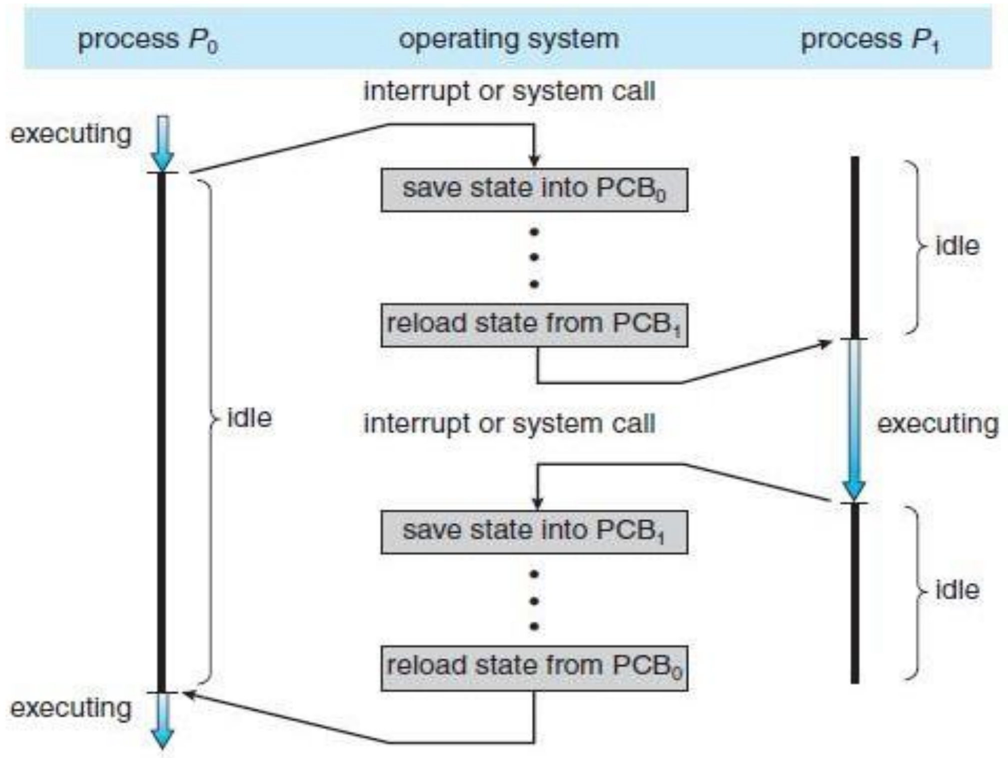


Figure 3.4 Diagram showing CPU switch from process to process.

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.

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.

Switching the CPU to another process requires performing a state save the current process and a state restore of a different process. This task is known as **context switch.**

Context switch times are highly dependent on hardware support.

Process scheduling

Basic Concepts:

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU–I/O Burst Cycle

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.

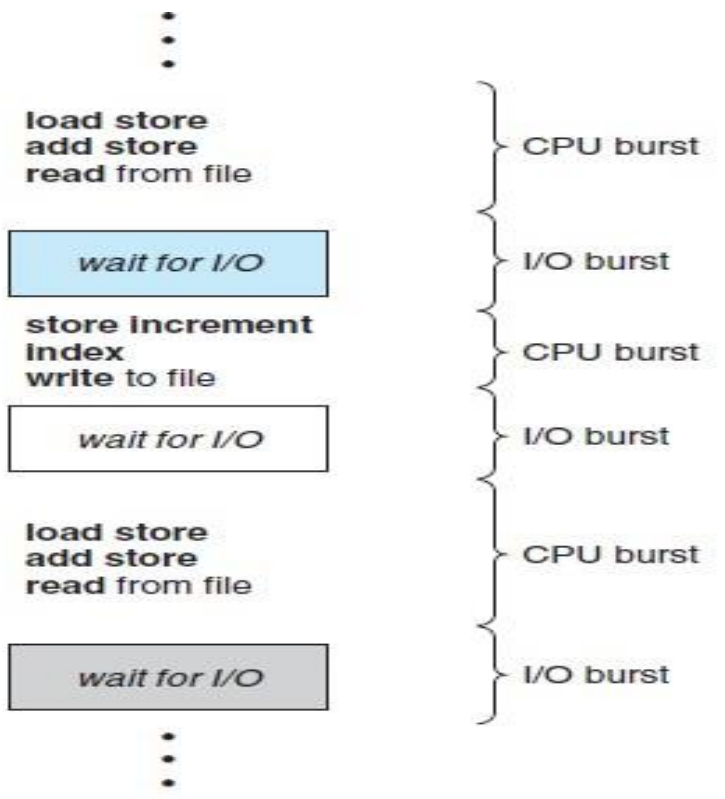


Figure 6.1 Alternating sequence of CPU and I/O bursts.

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.

Preemptive Scheduling

CPU scheduling decisions may take place under the following four circumstances.

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

When scheduling takes place only under circumstances 1 and 4 , we say that the scheduling scheme is **Non-preemptive or cooperative.**

Under **Non-preemptive** 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.

Preemptive

This scheduling method used by Windows 95 and all subsequent versions of the Microsoft Windows OS's and Max OS X OS for the Macintosh OS.

In this scheduling a running process may be replaced by higher priority or the CPU burst time of current executing process is greater than of a new process at any time.

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

a) Switching context

b) Switching to user mode

c) 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.

Dispatch latency: The time it take for the dispatcher to stop one process and start another running is known as the Dispatch Latency.

Operation on Processes

1) **Process Creation**
2) **Process Termination**

i)Process Creation

- 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.
- During the course of execution, a process may create several new processes via a create _process system call.
- 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.
- Figure 3.8 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.

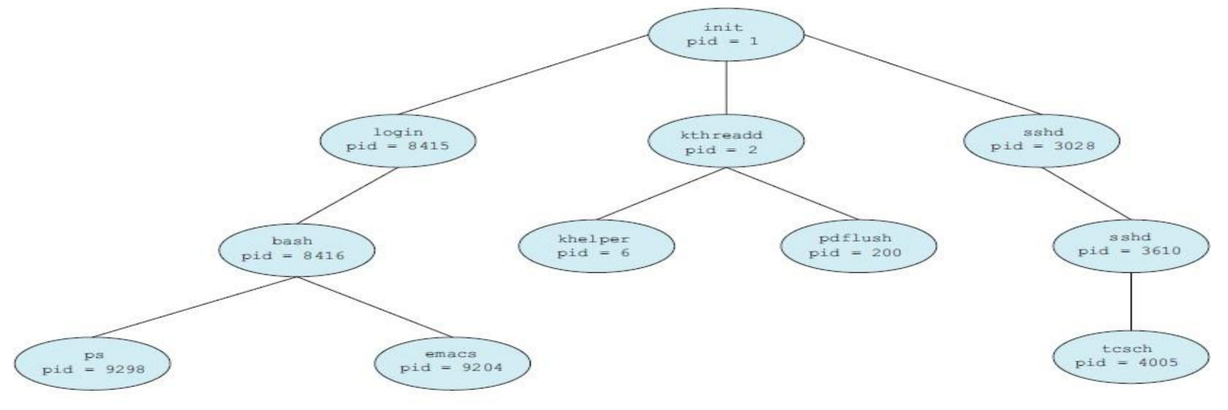


Figure 3.8 A tree of processes on a typical Linux system.

-

- The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- The init process can also create various user processes.
- We see two children of init—kthreadd and sshd.
- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.
- 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.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- When a process creates a new process, two possibilities for execution exist:

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

  2. The parent waits until some or all of its children have terminated.
- There are also two address-space possibilities for the new process:

  1. The child process is a duplicate of the parent process (it has the

same program and data as the parent).

2. The child process has a new program loaded into it.

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

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX fork() system call.

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.
- The exec() system call loads a binary file into memory and starts

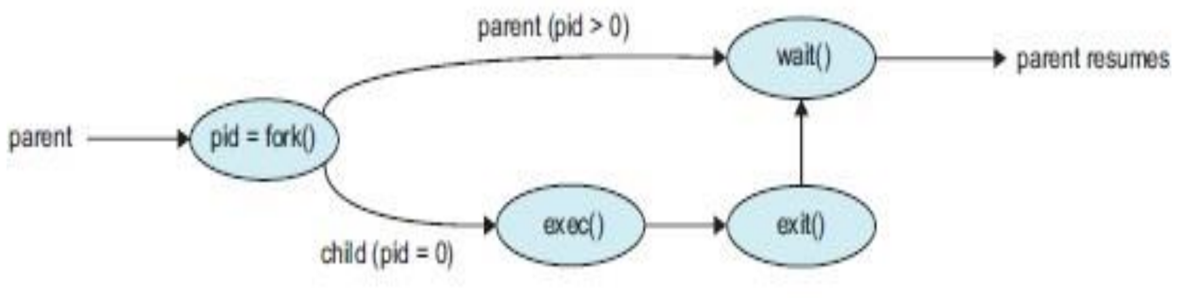


Figure 3.10 Process creation using the fork() system call.

its execution.

ii) 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.
- Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children if it is to terminate them.
- 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:

a) The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have

a mechanism to inspect the state of its children.)

b) The task assigned to the child is no longer required.

c) 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);

Cooperating Processes

- Processes.
    1) Independent Processes
    2) Cooperating Processes
- Inter process Communication
    1) Shared Memory Model
    2) Message Passing Model
- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system.
- Clearly, any process that shares data with other processes is a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:
- **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **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 **Inter process Communication (IPC) mechanism** that will allow them to exchange data and information. There are two fundamental models of Inter process 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.
- Message passing is useful for exchanging smaller amounts of data.
- Message passing is also easier to implement in a distributed system than shared memory.

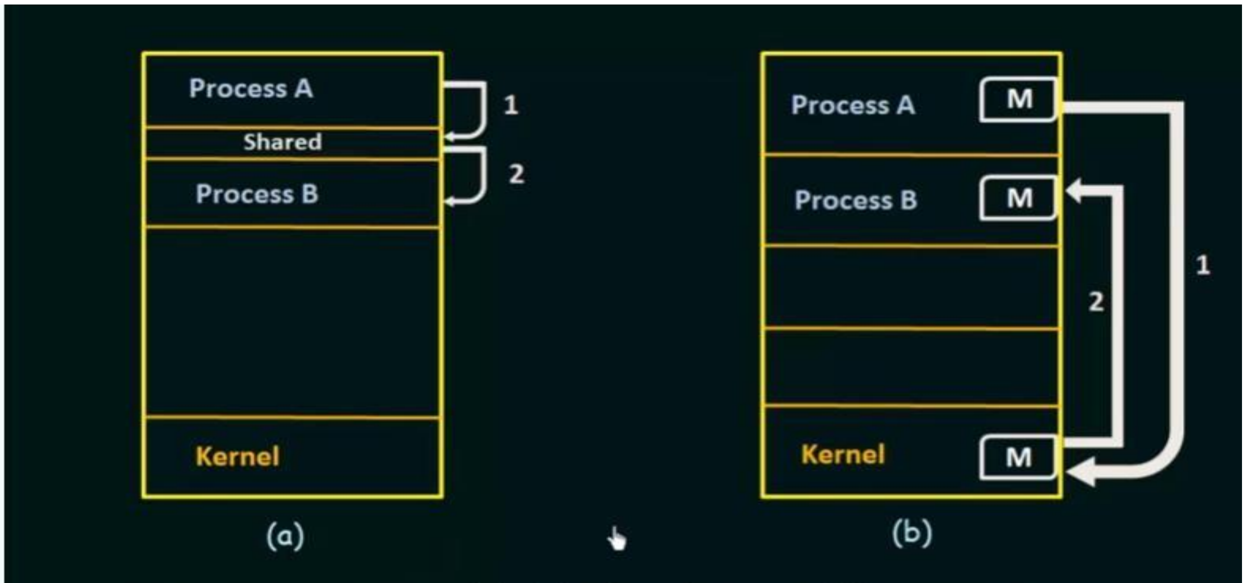
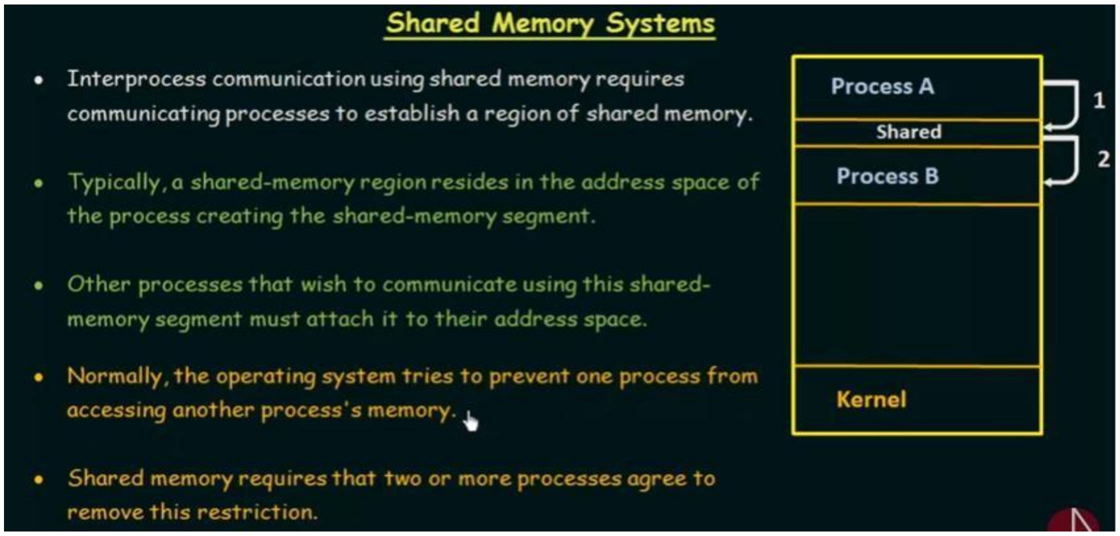


Fig: Communications models, (a) Shared memory, (b) Message Passing.

Shared-Memory Systems:

- Inter process 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
- They can then exchange information by reading and writing data in the shared areas.
- 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.

- The form of the data and the location are determined by these processes and are not under the operating system's control.



Shared Memory Systems

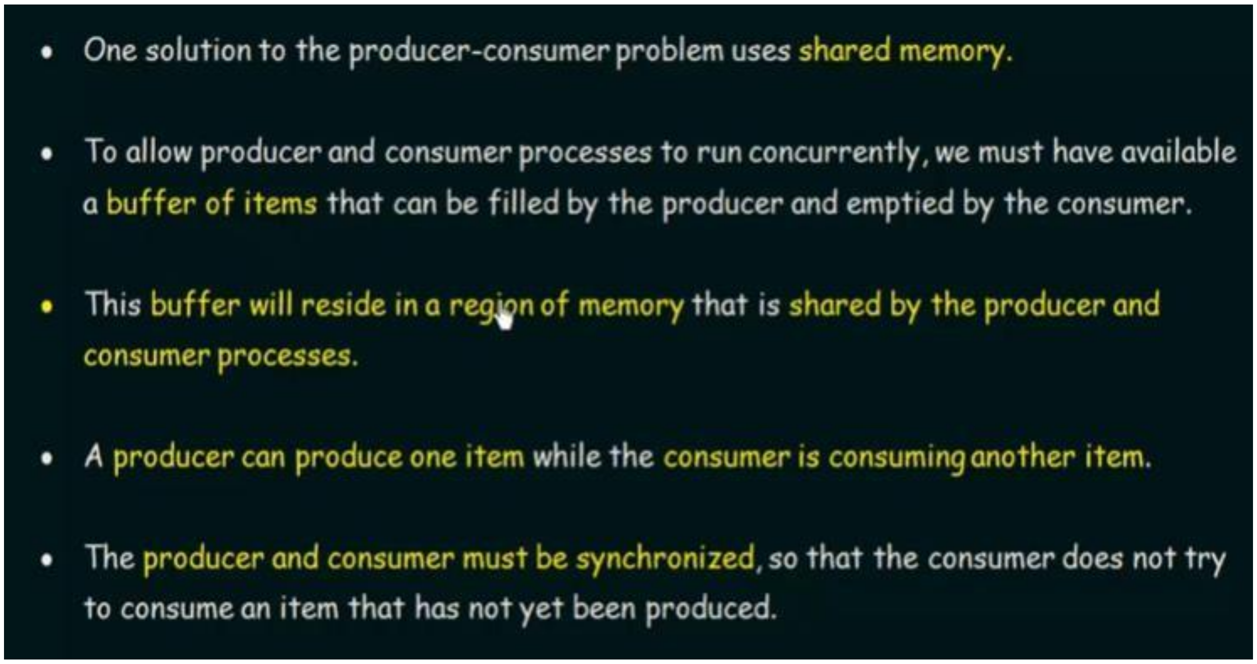
- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory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

- To illustrate the concept of cooperating processes, let's consider the producer–consumer problem
- 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.

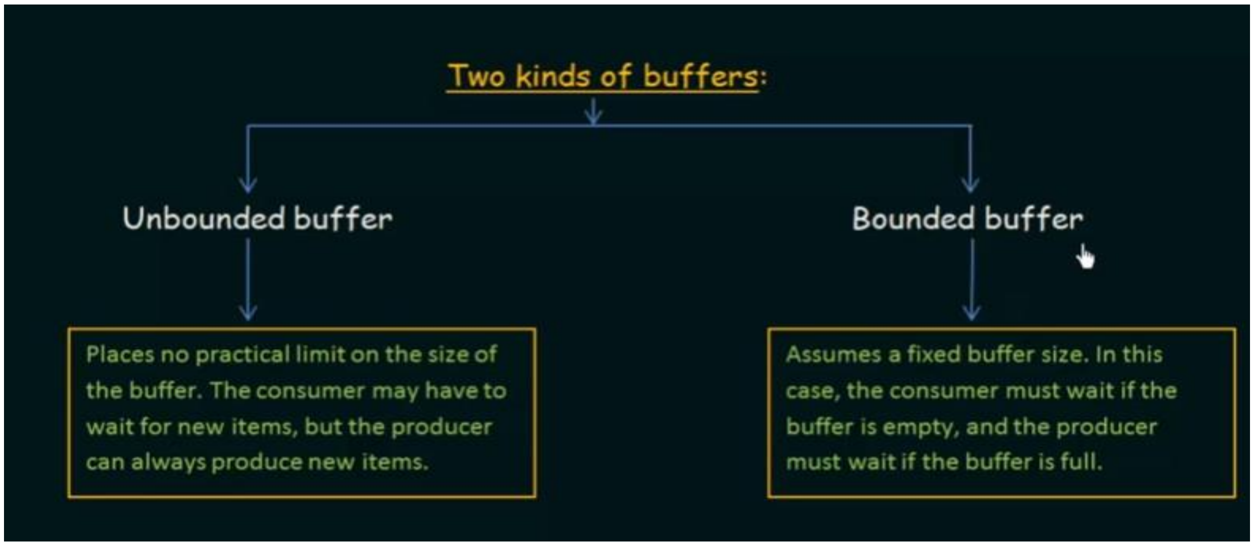
- So that the consumer does not try to consume an item that has not yet been produced.



- 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.

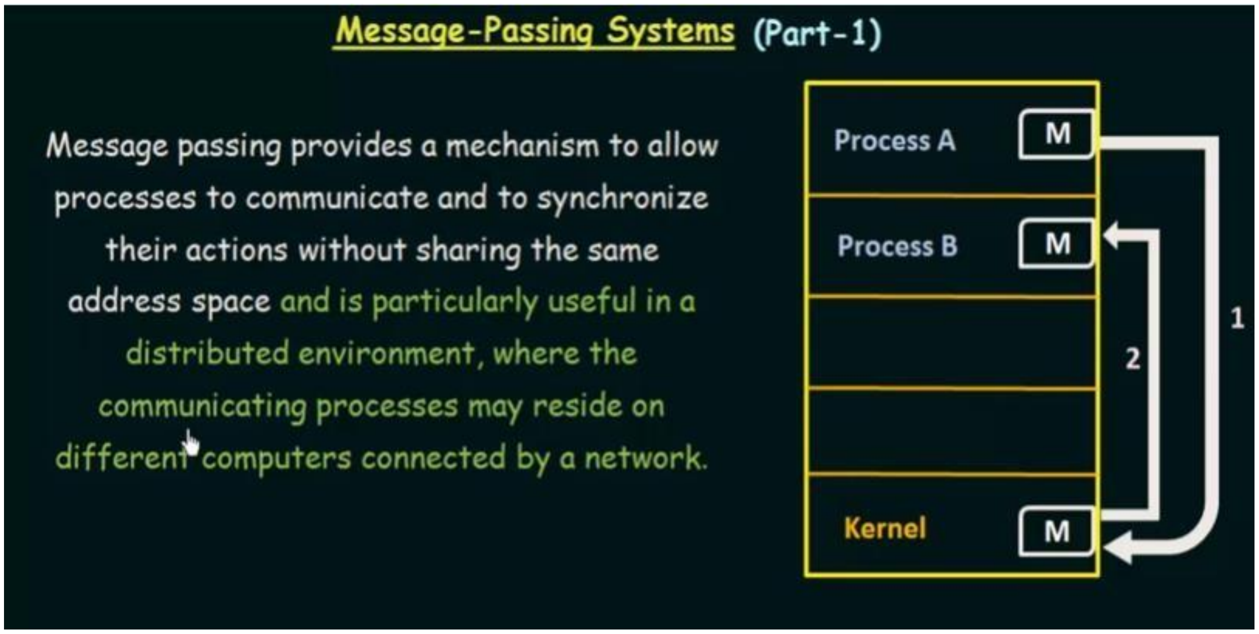
- Two types of buffers can be used. The **unbounded buffer places no practical** limit on the size of the buffer.
- The consumer may have to wait for new items, but the producer can always produce new items.

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.


Two kinds of buffers:
Unbounded buffer
Bounded 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.
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.

Message-Passing Systems

- 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.



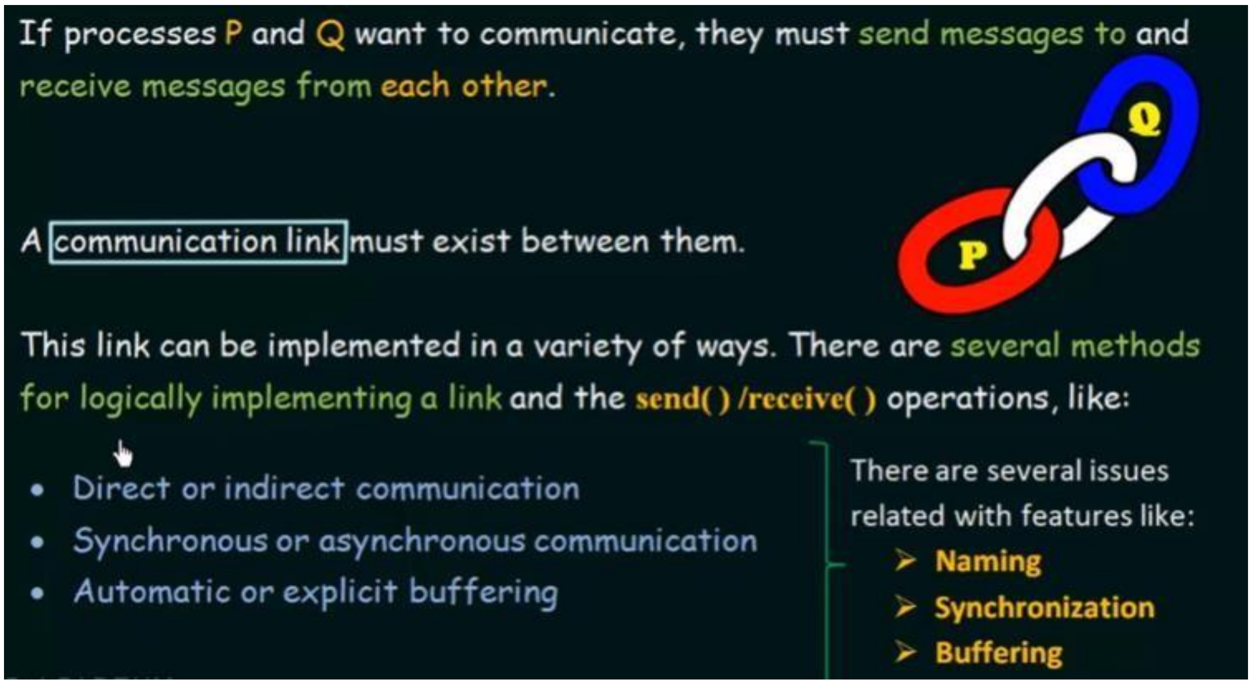
Message-Passing Systems (Part-1)

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

- A message-passing facility provides at least two operations.
- send(message)
- receive(message)
- 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 straightforward.
- This restriction, however, makes the task of programming more difficult.
- 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.
    - Direct or Indirect communication
    - Synchronous or Asynchronous communication

Automatic or Explicit buffering



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.

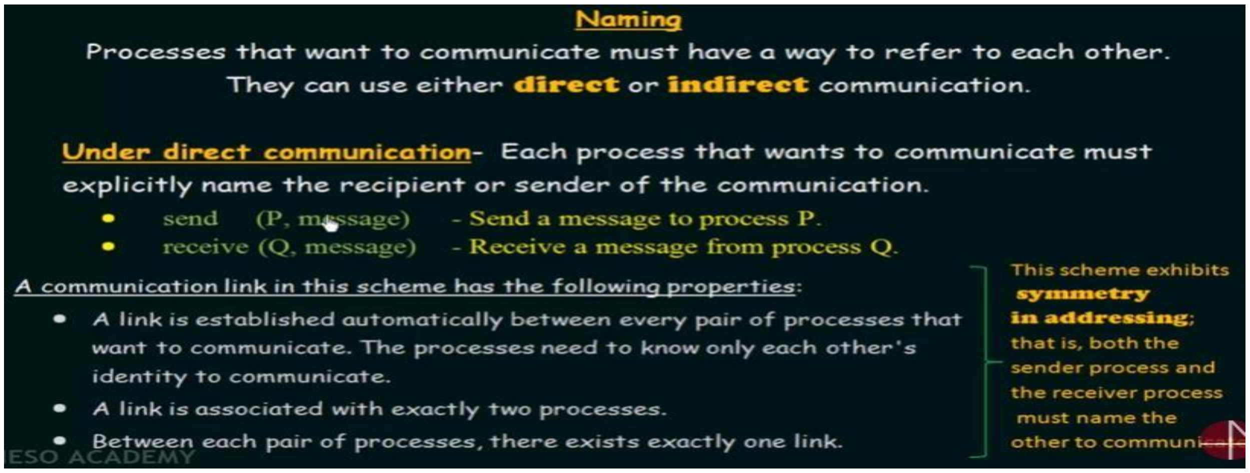
This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send( ) /receive( ) operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:

- Naming
- Synchronization
- 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.



Naming

Processes that want to communicate must have a way to refer to each other.
They can use either **direct** or **indirect** communication.

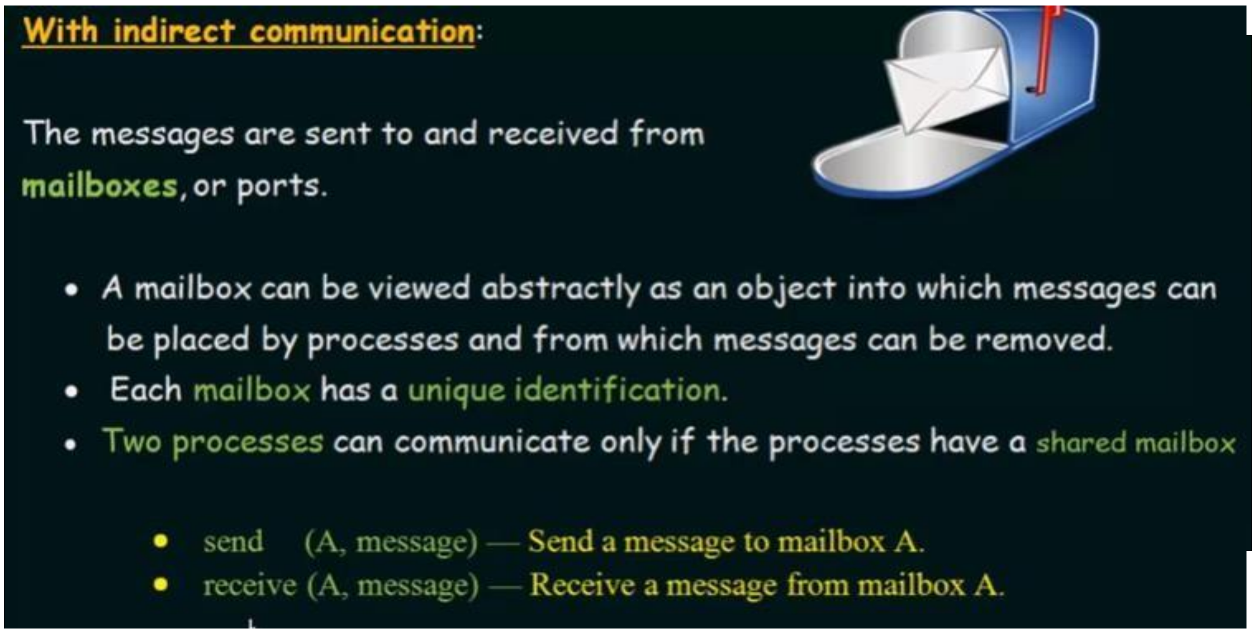
Under direct communication- Each process that wants to communicate must explicitly name the recipient or sender of the communication.

- 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 communi


With indirect communication:

The messages are sent to and received from mailboxes, or ports.

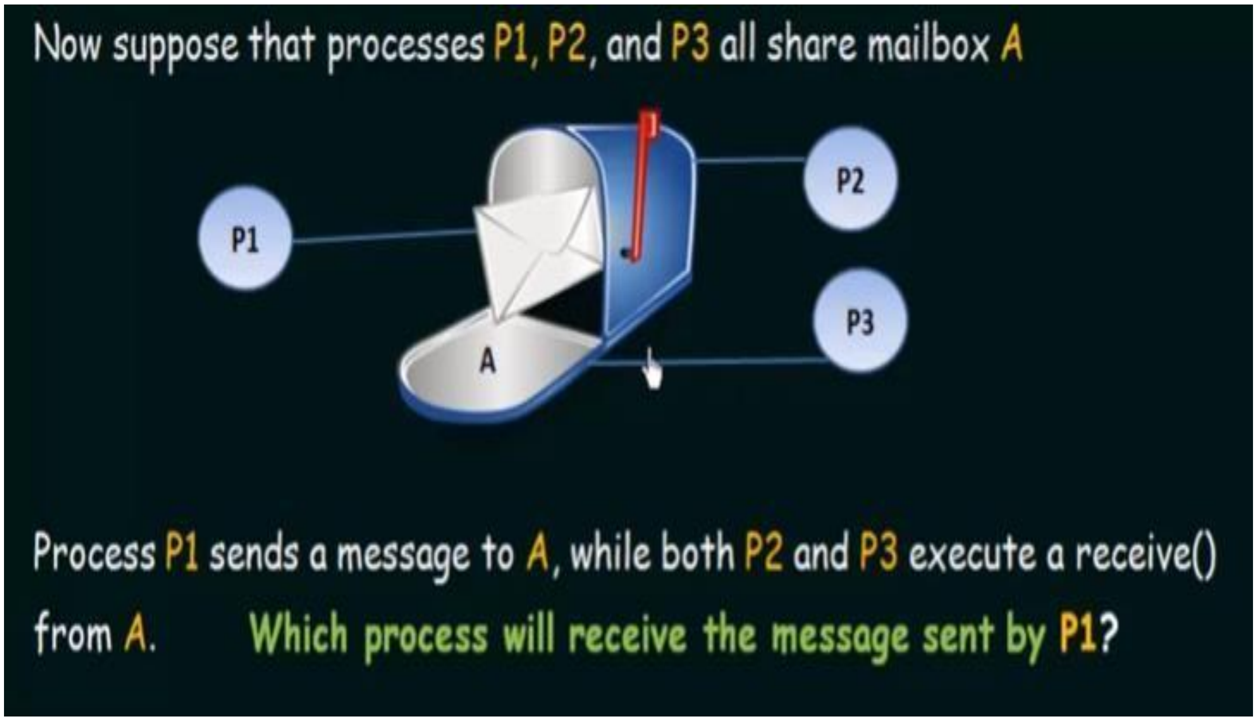
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- Two processes can communicate only if the processes have a shared mailbox

  - send (A, message) — Send a message to mailbox A.
  - receive (A, message) — Receive a message from mailbox A.

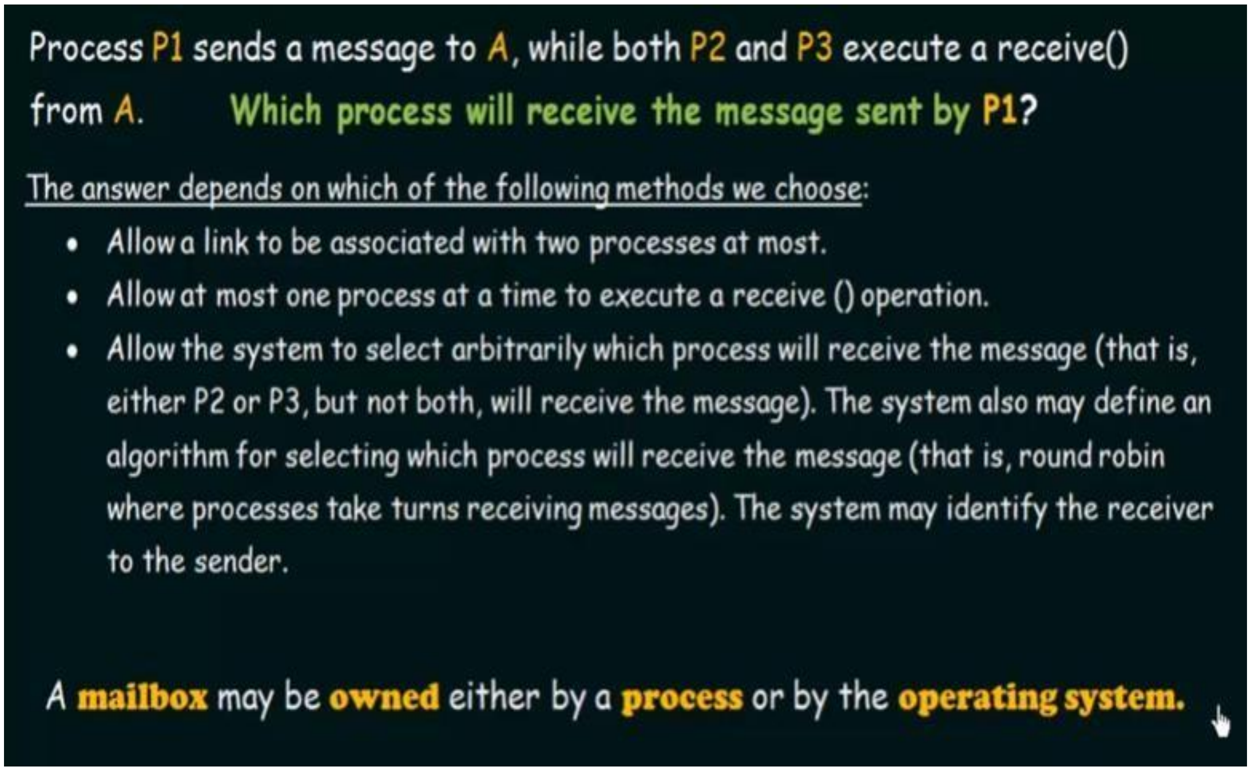
- With Indirect Communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- The send() and receive() 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 P1, P2, and P3 all share mailbox A

Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?


Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?

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 P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, 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.**

Synchronization

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

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

  ii)Nonblocking send. The sending process sends the message and resumes operation.

  iii)Blocking receive. The receiver blocks until a message is available.

  iv)Nonblocking receive. The receiver retrieves either a valid message or a null.

Buffering

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

  i)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.

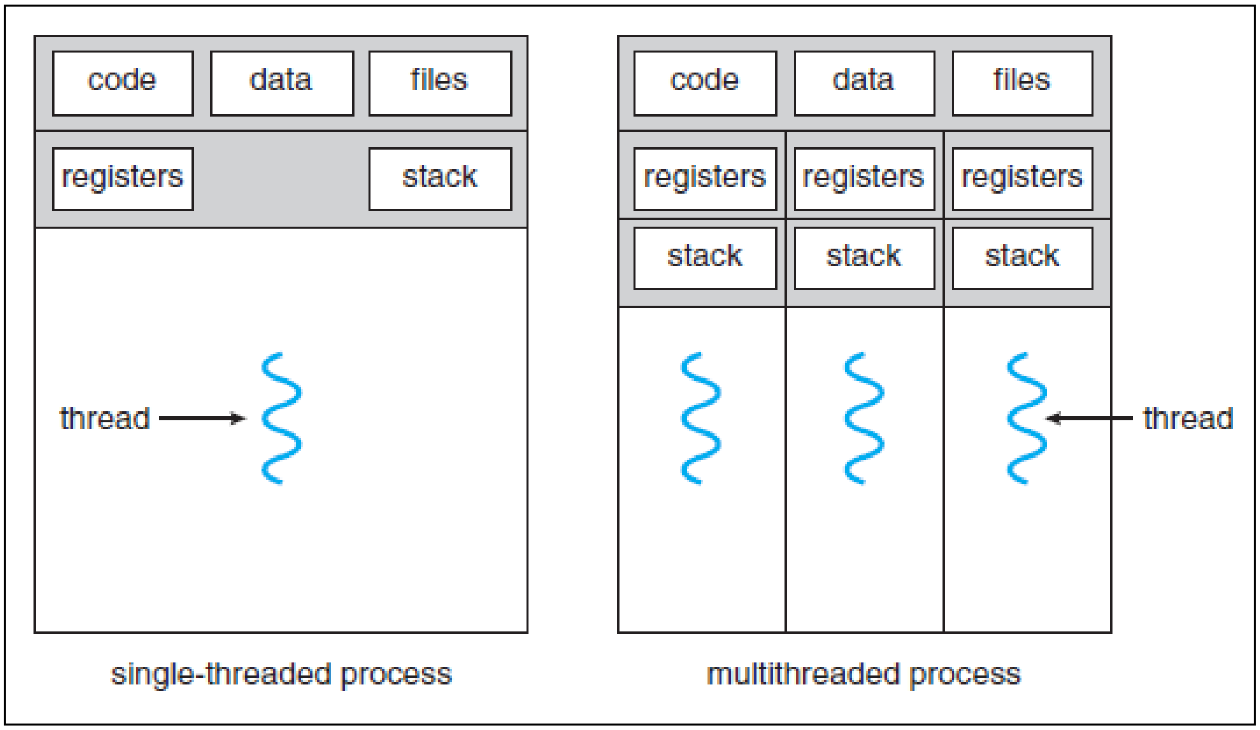
  ii) Bounded capacity. The queue has finite length *n; thus, at most n messages* can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting.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

iii)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

Threads

- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.



- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separate process with several threads of control.
- For Example, A web browser might have one thread display images or text while another thread retrieves data from the network.
- Another Example, A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
- When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.
- This is illustrated in Figure 4.2.

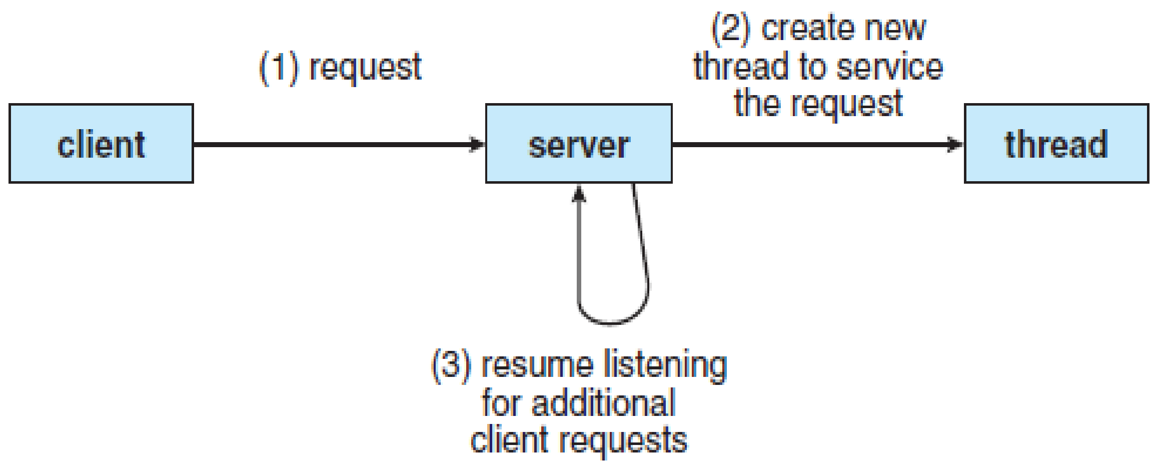

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Figure 4.2 Multithreaded server architecture.

Benefits

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

2. Resource sharing. Processes can only share resources through techniques such as shared memory and message passing.

- Threads share the memory and the resources of the process to which they belong by default.
- The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

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

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

Multithreading Models

- Our discussion so far has treated threads in a generic sense.
- However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads.**
- User threads are supported above the kernel and are managed without kernel support.
- Whereas kernel threads are supported and managed directly by the operating system.
- All operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.
- Ultimately, a relationship must exist between user threads and kernel threads.
- In this section, we look at three common ways of establishing such a relationship: the Many-to-One model, the One-to-One model, and the Many-to Many model.
- The Many-to-One model (Figure 4.5) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.
- One thread can access the kernel at a time,multiple threads are unable to run in parallel on multicore systems.

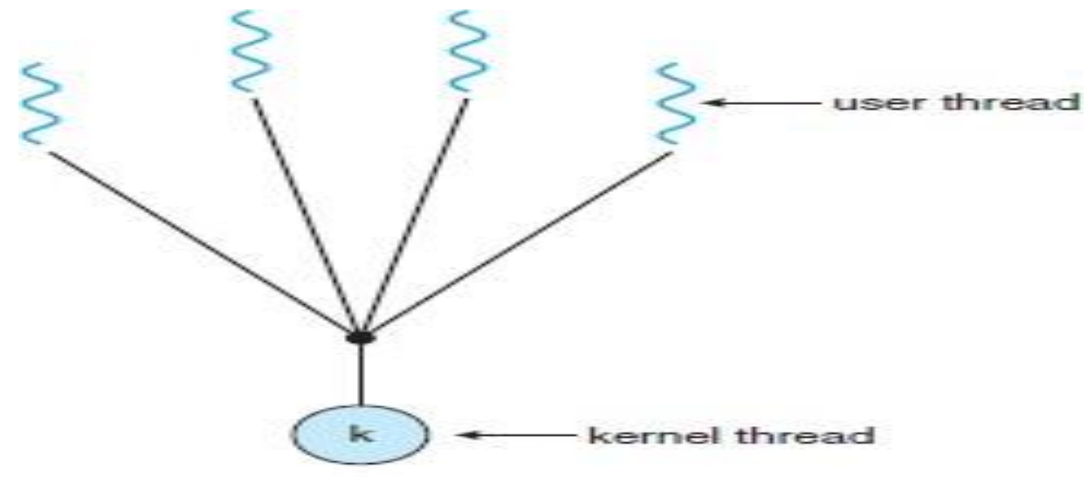


Figure 4.5 Many-to-one model.

One-to-One Model

- The one-to-one model (Figure 4.6) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

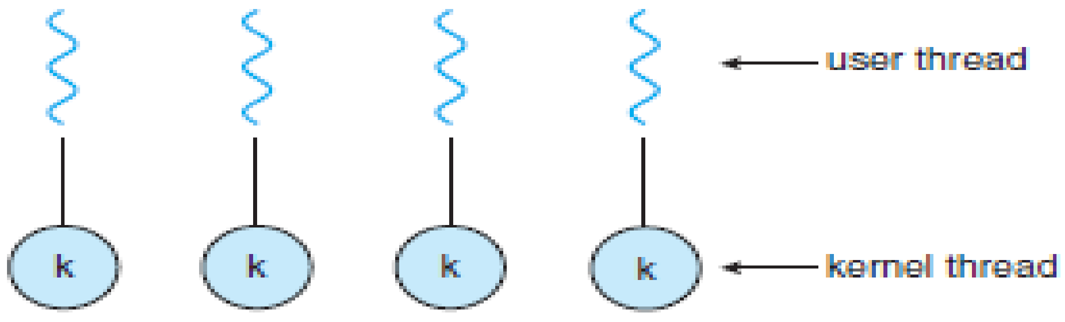


Figure 4.6 One-to-one model.

Many-to-Many Model

- The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.
- The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- 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).

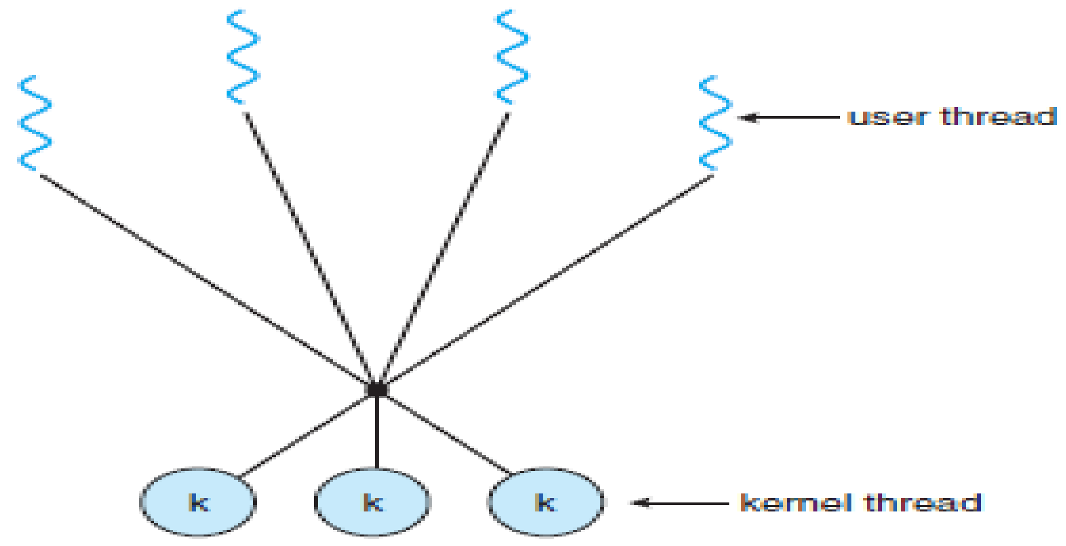


Figure 4.7 Many-to-many model.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a lightweight process, or LWP—is shown in Figure 4.13.

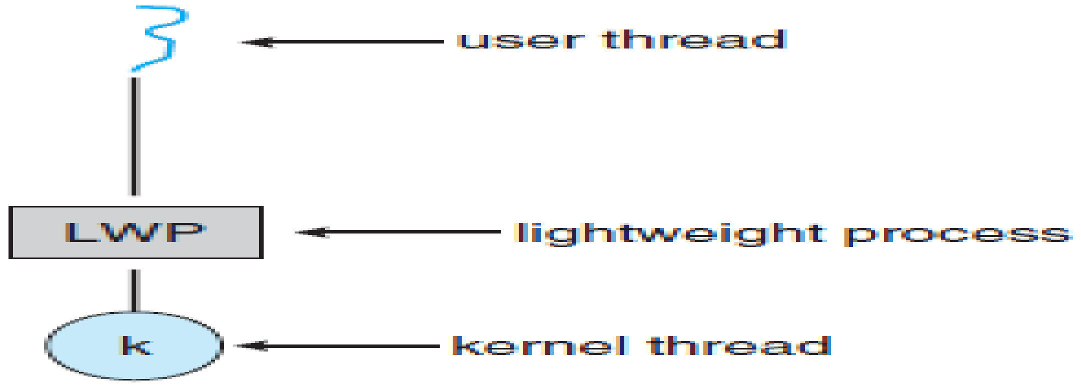


Figure 4.13 Lightweight process (LWP).

Scheduling Criteria:

Criteria can be used for comparing CPU Scheduling algorithms. The criteria includes

a) **CPU Utilization**: We want to keep the CPU as busy as possible , the cpu utilization can range from 0- 100%

b) **Throughput**: The number of processes that are completed per unit time. For Long process this rate may be 1 process/hr, for short process this rate may be 10 processes/sec

iii) **Turnaround Time**: It is the sum of the periods spent on waiting to get into the memory and executing on the cpu.

iv) **Waiting time**: It is the time , the sum of the periods spent waiting in ready queue.

v) **Response time**: It is the sum of a request until the first response is produced.

CPU Schedudling Algorithms

1) FCFS(FIRST COME FIRST SERVE)

2) SJF(Shortest Job First)

3) Priority

4) Round Robin

1)FCFS: (First Come First Serve)

In this scheme the process that requests the cpu first is allocated first. The implementation of this policy is by FIFO Queue. When process enters into ready queue, its PCB is linked on to the tail of queue.

When cpu is free it is allocated to the process at the head of the queue, then the running process is removed from the queue.

The average waiting time under this policy is quite long and may vary substantially, if the process CPU burst time vary greatly.

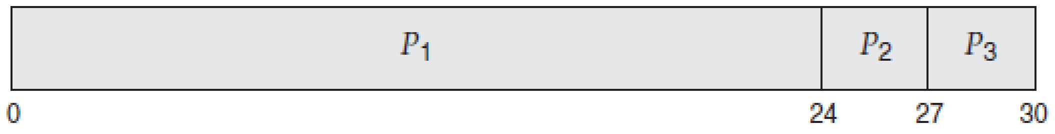
FCFS is for time sharing systems where each user gets a share of CPU at regular intervals.

FCFS scheduling algorithm is Nonpreemptive.

Example:

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.

2) SJF(Shortest Job First):

This algorithm associates with each process the length of the processes 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 burst of processes is the same, then FCFS scheduling is used.

SJF gives the minimum average waiting time for a given set of processes.

Moving to a short process before a long one decreases the waiting time, but it increases the waiting time of the long process.

This scheduling is used frequently in long term scheduling.

The real difficulty of this algorithm is knowing the length of the next CPU burst.

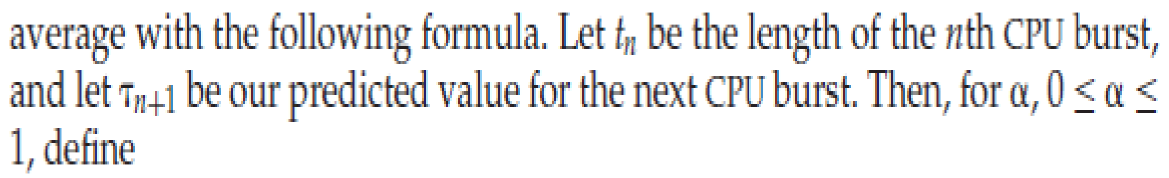
SJF scheduling algorithm may be either Non preemptive or Preemptive.

When a new process arrives at the ready queue while previous process is still executing, the CPU burst time of the new arriving process may be shorter than what is left of the currently executing process.

A preemptive SJF algorithm will preempt the currently executing process where as a non preemptive SJF algorithm will allow the currently running process to finish its CPU burst time.

Preemptive SJF scheduling is sometimes called "**Shortest Remaining Time First**" Scheduling.

The next CPU burst is predicted as an exponential average of the measured lengths of the previous CPU bursts.



average with the following formula. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

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

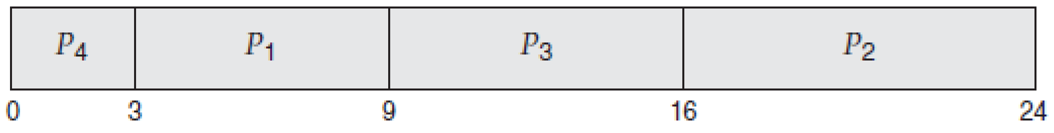
The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted.

Example for NON- Preemptive SJF CPU Scheduling:

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

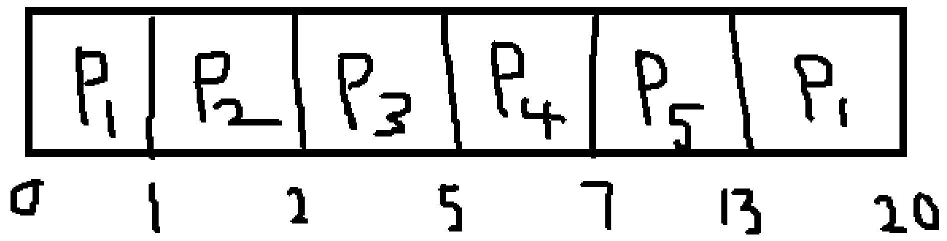
Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



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.

Example for Preemptive SJF CPU Scheduling:

Process	**Arrival Time**	**Burst Time(Mille Seconds)**
P1	0	8
P2	1	1
P3	2	3
P4	3	2
P5	4	6



PNO	Arrival time(AT)	Burst Time(BT)	Remaining Time(RT)	Completion time(CT)	Turn Around Time(CT-AT)	Waiting Time(TAT-BT)
P1	0	8	0	20	20	12
P2	1	1	0	2	1	0
P3	2	3	0	5	3	0
P4	3	2	0	7	4	2
P5	4	6	0	13	9	3

Average Waiting Time=17/5=3.4 ms

Average Turn Around Time =37/5=7.4 ms

3) <u>PRIORITY CPU SCHEDULING ALGORITHM:</u>

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.

Priorities are generally indicated by some fixed range of numbers such as 0 to 7. However there is no general agreement on whether '0' is the highest or lower priority. Some systems use low numbers to represent low priority others use low numbers for high priority.

Priority can be either preemptive or non preemptive. When a new 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 non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue .

The main problem with the priority scheduling is indefinite blocking or starvation. A process that is ready to run but waiting for the CPU can be considered blocked.

A solution to the problem of indefinite blockage of low priority process is **Aging.**

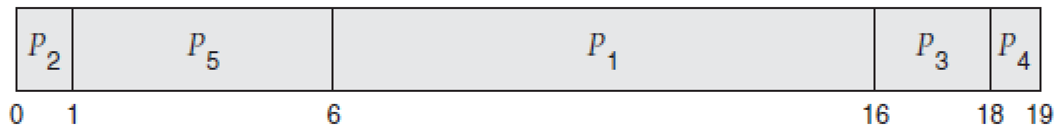
Aging is the technique of gradually increasing the priority of processes that wait in the system for a long time.

Example for Non-Preemptive Priority CPU Scheduling:

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \cdots, P_5$, 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

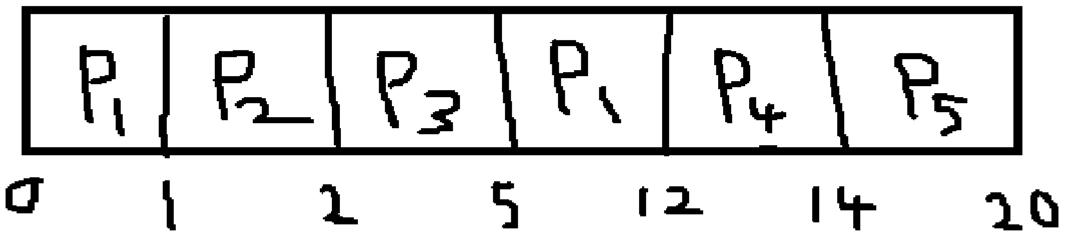
Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Example for Preemptive Priority CPU Scheduling:

Process	Arrival Time	Burst Time(Mille Seconds)	Priority
P1	0	8	3
P2	1	1	1
P3	2	3	2
P4	3	2	3
P5	4	6	4



PNO	Arrival time(AT)	Burst Time(BT) Mille Seconds	Priority	Remaining Time(RT)	Completion time(CT)	Turn Around Time(CT-AT)	Waiting Time(TAT-BT)
P1	0	8	3	0	12	12	4
P2	1	1	1	0	2	1	0
P3	2	3	2	0	5	3	0
P4	3	2	3	0	14	11	9
P5	4	6	4	0	20	16	10

Average Waiting Time=23/5=4.6 ms

Average Turn Around Time =43/5=8.6 ms

4) ROUND ROBIN CPU SCHEDULING ALGORITHM.

This algorithm is designed for time sharing systems. It is similar to FCFS scheduling but pre-emption is added to switch between processes.

A small unit of time called time quantum or time slice. A time quantum is generally from 10 to 100 ms.

Here the ready queue is treated as circular queue. The CPU scheduler goes around the ready queue allocating the CPU up to one time quantum.

The CPU scheduler picks the first process from the ready queue sets a timer to interrupt after one time quantum and dispatches the process.

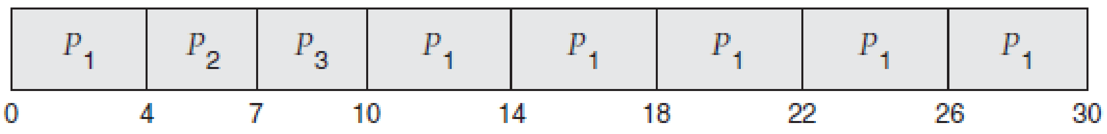
The process may have a CPU burst of less than 1 time quantum, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to OS. And 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.

Example for Preemptive Priority CPU Scheduling:

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.

Q) Consider the following set of processes, with the length of the cpu burst given in milliseconds.

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

The processes are assumed to have arrived in the order P1,P2,P3,P4,P5 all at time 0.

a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms. FCFS, SJF, NON-Preemptive Priority(smaller number implies higher priority) and Round Robin(time quantum=1).
b) What is the turnaround time and waiting of each process for each of these scheduling algorithms?
c) Which of the algorithms results in the minimum average waiting time?

5) Multilevel Queue Scheduling

Another class of scheduling algorithm has been created in which processes are easily classified into different groups.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority or process type.

Each queue has its own scheduling algorithm.

For Example foreground queue might be scheduled by a RR algorithm

Background queue is scheduled by an FCFS algorithm.

Look at an example of a multilevel queue scheduling algorithm with five queues.

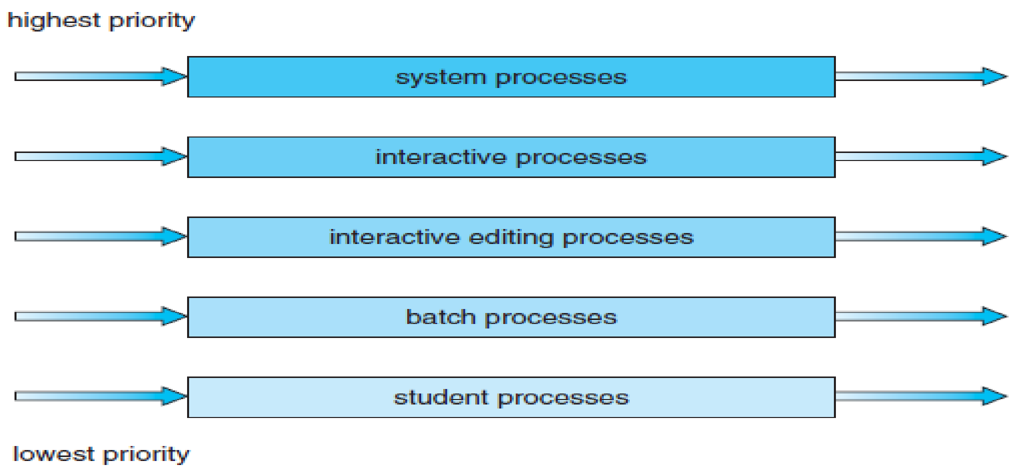


Figure 6.6 Multilevel queue scheduling.

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

6) Multilevel Feedback queue scheduling

The multilevel feedback queue scheduling algorithm 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 schemes leaves I/O bound and interactive processes in the higher priority queues.

A process that waits too long in a lower priority queue may be moved to a high priority queue.

This form of aging prevents starvation.

For example consider a multilevel feedback queue scheduler with three queues, numbered 0 to 2.

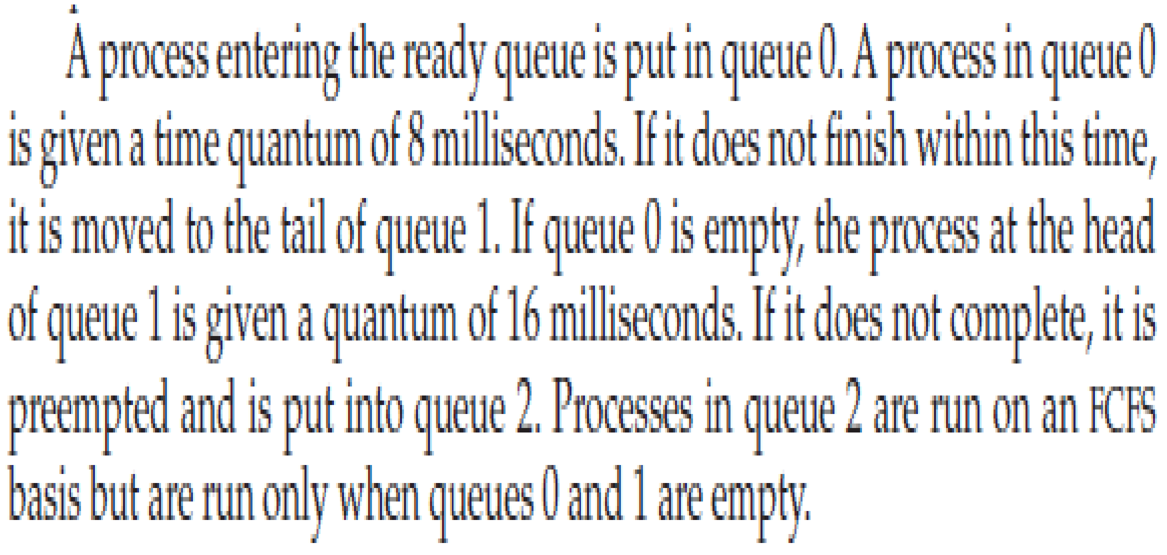
The scheduler first executes all processes in queue 0.

Only when queue0 is empty will it executes processes in queue 1.

Similarly, processes in queue 2 will only be executed if queue 0 and queue 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.

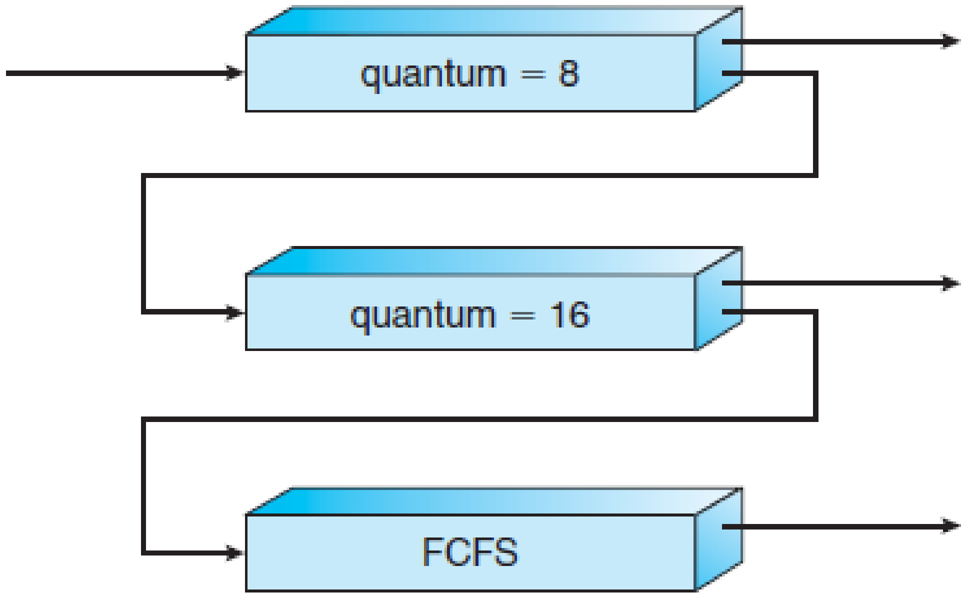


Figure 6.7 Multilevel feedback queues.

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

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

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.

Here, we discuss several concerns in multiprocessor scheduling. We concentrate on systems in which the processors are identical homogeneous in terms of their functionality.

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.

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.

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.

There are two types of processor affinity:

1. **Soft Affinity –** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity –** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

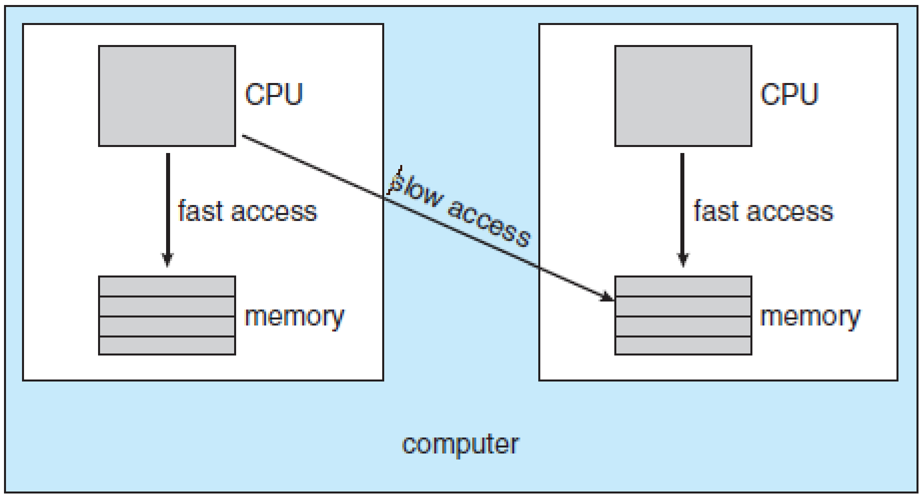


Figure 6.9 NUMA and CPU scheduling.

Load Balancing

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute.
Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue.

On SMP (symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.
There are two general approaches to load balancing :

1. **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors

In multicore processors **multiple processor** cores are places on the same physical chip.

Each core has a register set to maintain its architectural state and thus appears to the operating system as 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.
However multicore processors may **complicate** the scheduling problems.

When processor accesses memory then it spends a significant amount of time waiting for the data to become available.

This situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory.

In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory.

To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core.

Therefore if one thread stalls while waiting for the memory, core can switch to another thread.
There are two ways to multithread a processor :

1. **Coarse-Grained Multithreading** – In 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.
   The cost of switching between threads is high as the instruction pipeline must be terminated 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.
2. **Fine-Grained Multithreading** – This multi-threading switches between threads at a much finer level mainly at the boundary of an instruction cycle.
   The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.