UNIT- II PROCESS MANAGEMENT

Processes - Process Concept, Process Scheduling, Operations on Processes, Inter-process Communication; CPU Scheduling - Scheduling criteria, Scheduling algorithms, Multiple-processor scheduling, Real time scheduling; Threads- Overview, Multithreading models, Threading issues; Process Synchronization - The critical-section problem, Synchronization hardware, Mutex locks, Semaphores, Classic problems of synchronization, Critical regions, Monitors; Deadlock - System model, Deadlock characterization, Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

PROCESS:

A Process is defined as a program in execution.

A program is a passive entity, such as a file containing a list of instructions stored on disk

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.

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)

It contains a data section, which contains global variables.

A Process may also include a heap which is a memory that is dynamically allocated during process run time.

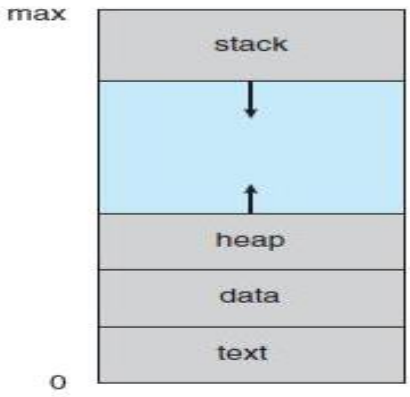


Figure 3.1 Process in memory.

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
Ready. The process is waiting to be assigned to a processor.
Terminated. The process has finished execution.

Downloaded from: annauniversityedu.blogspot.com

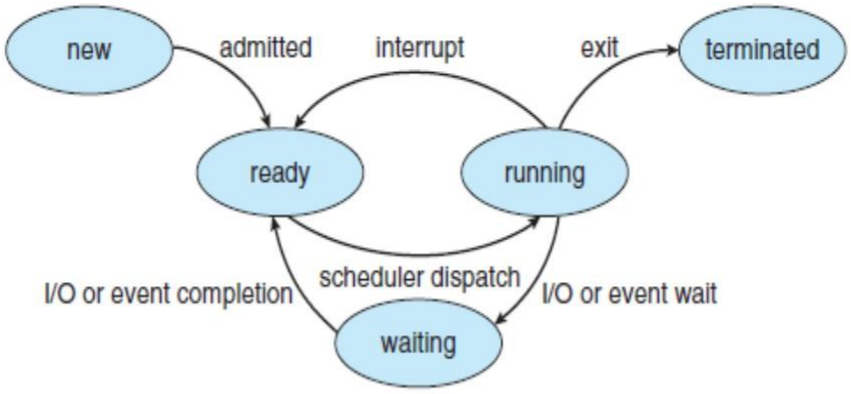


Figure 3.2 Diagram of process state.

Process Control Block:

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
It contains many pieces of information associated with a specific process
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
CPU-scheduling information. This information includes a process priority, pointers to scheduling queues
Memory-management information. This information may include such items as the value of the base and limit registers and the page tables
Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers,
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.

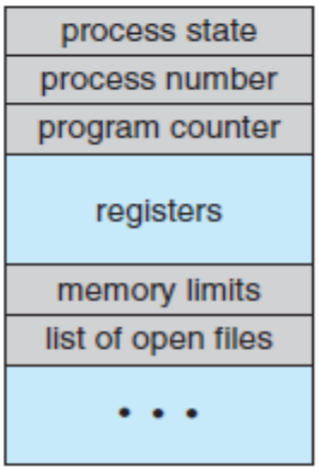


Figure 3.3 Process control block (PCB).

Downloaded from: annauniversityedu.blogspot.com

PROCESS SCHEDULING:

The **process scheduler** selects an available process 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.

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.

Scheduling Queues:

The Scheduling Queues are of three types

Job Queue

Ready Queue

Device Queue

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**

A ready-queue header contains pointers to the first and final PCBs in the list.

Each process that requires I/O Operation may have to wait for the device. The list of processes waiting for a particular I/O device is called a **device queue**.

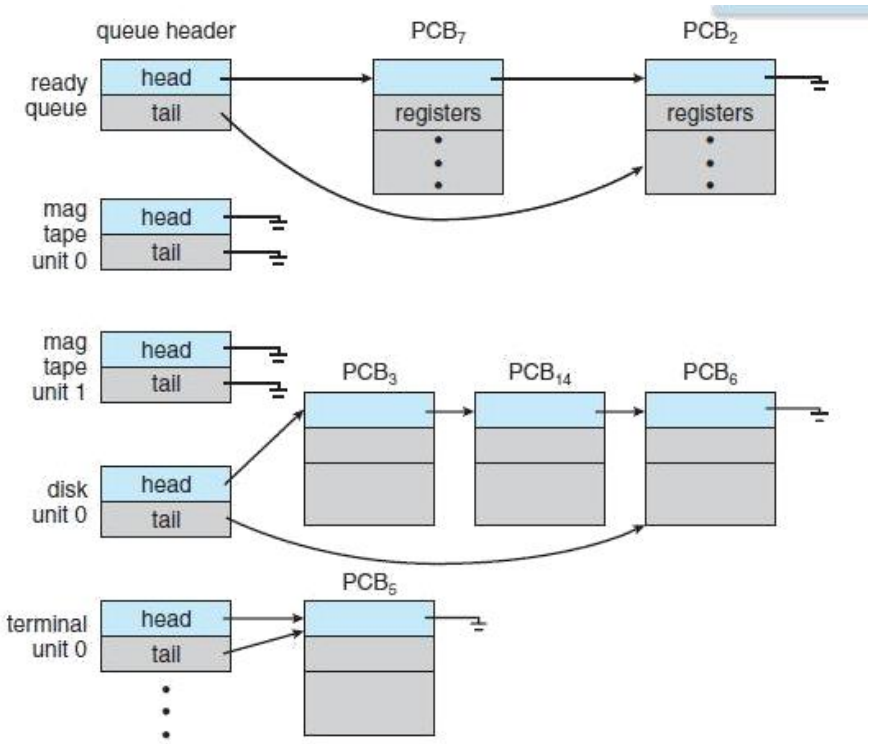


Figure 3.5 The ready queue and various I/O device queues.

A common representation of process scheduling is a **queuing diagram**

Downloaded from: annauniversityedu.blogspot.com

Two types of queues are present: **Ready queue and a set of device queues.** The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

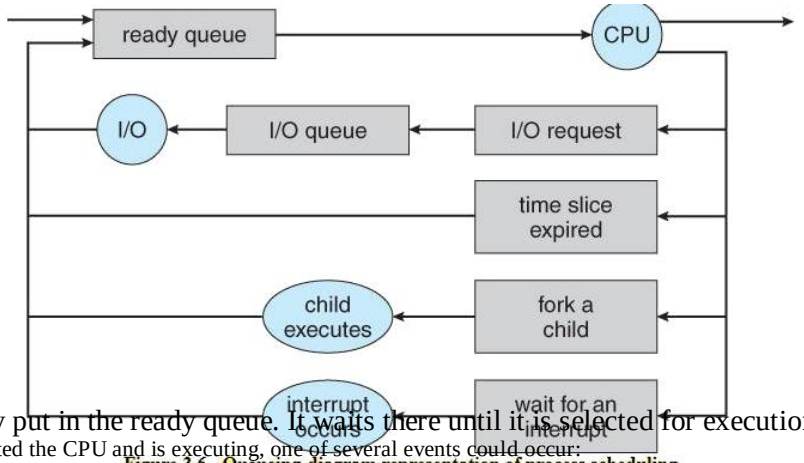


Figure 3.6 - Queueing-diagram representation of process scheduling

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched.**

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

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.

Schedulers:

The operating system must select, for scheduling purposes, processes from the queues in some approach. The selection process is carried out by the appropriate **scheduler**.

It makes use of two types of schedulers

Long term scheduler or job scheduler
Short term scheduler or CPU scheduler.
Medium term scheduler

The **long-term scheduler**, or **job scheduler**, selects processes from the job queue 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 must have a careful selection of both I/O Bound and CPU Bound process.

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.

If all processes are I/O bound, the ready queue will almost always be empty, If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused.

The medium-term scheduler is used to remove a process from memory to reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

The process is swapped out, and is later swapped in, by the medium-term scheduler.

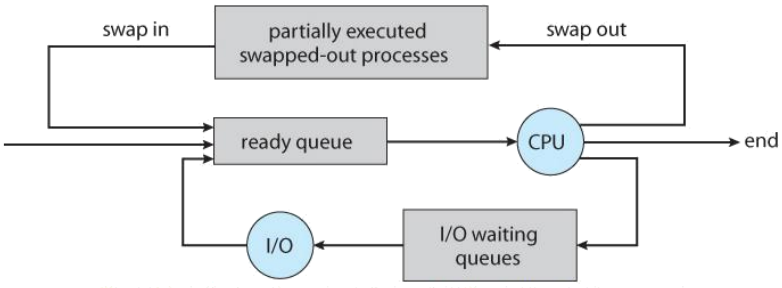


Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

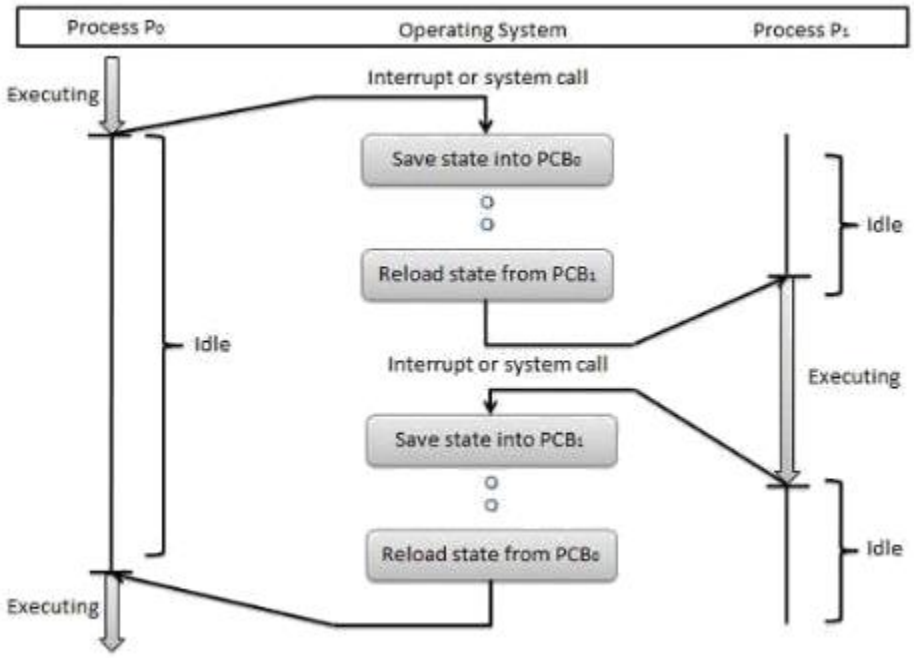
Downloaded from: annauniversityedu.blogspot.com

Context Switch:

The process of switching the CPU from one process 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 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.

The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory management information.



OPERATIONS ON PROCESSES:

The operating system must provide a mechanism for process creation and termination. The process can be created and deleted dynamically by the operating system.

The Operations on the process includes

Process creation

Process Termination

Process Creation:

During Execution a process may create several new processes.

The creating process is called as the **parent process** and the newly created process is called as the **child process**.

The operating systems identify the processes according to their unique process identifier.

Downloaded from: annauniversityedu.blogspot.com

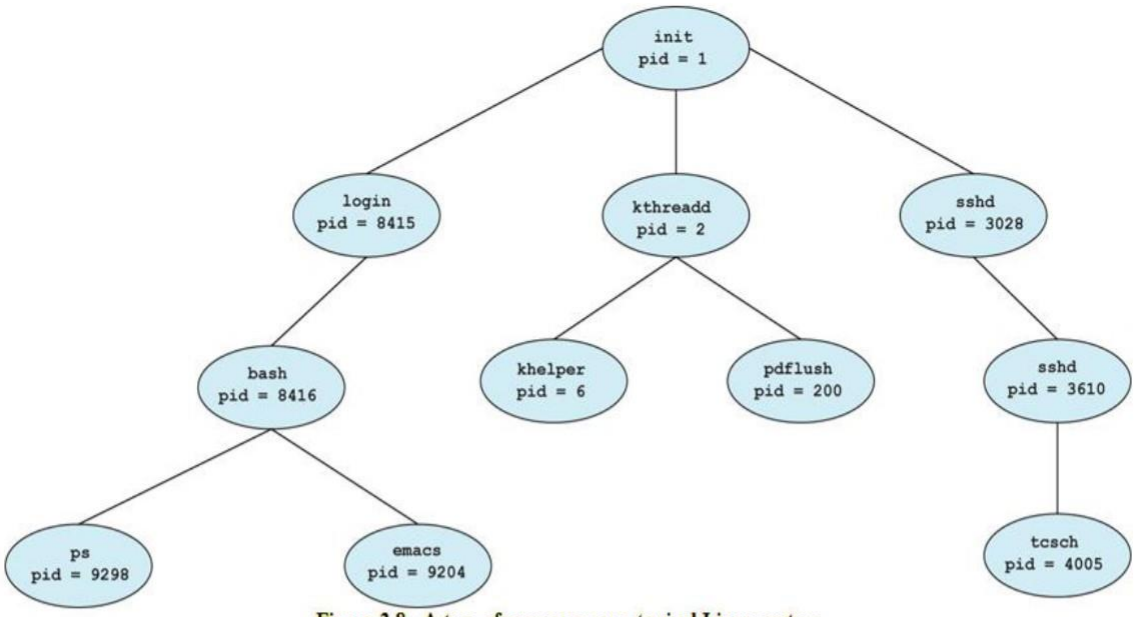


Figure 3.8 - A tree of processes on a typical Linux system

Example: Process tree for Linux operating system.

The init process serves as the root parent process for all the user process.
Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server.
The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel
The sshd process is responsible for managing clients that connect to the system by using ssh(Secure shell)
The login process is responsible for managing clients that directly log onto the system
The command ps –el will list complete information for all processes currently active in the system.

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 as the parent).
- The child process has a new program loaded into it.

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.
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.

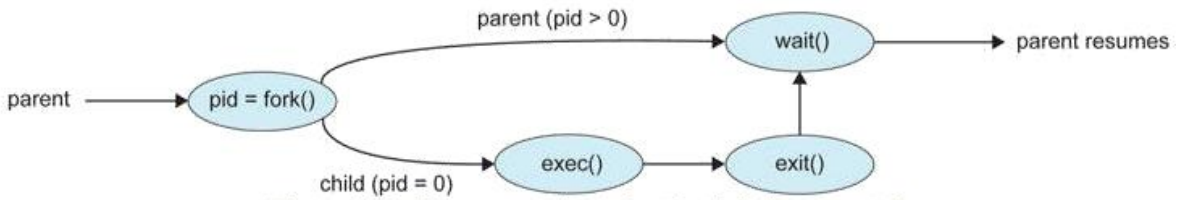


Figure 3.10 - Process creation using the fork( ) system call

Creating a separate process using the UNIX fork() system call:

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.

Downloaded from: annauniversityedu.blogspot.com

```
#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;
}
```

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.

All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system

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

- 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 is referred to as **cascading termination.**

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

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);
```

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

INTERPROCESS COMMUNICATION:

A process can be either an independent process or a cooperating process.

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.

Advantages of cooperating process i) Information sharing ii) Computation speedup iii) Modularity iv) Convenience**.**

Downloaded from: annauniversityedu.blogspot.com

DEFINITION:

An Interprocess communication is a mechanism that allows the cooperating process to exchange data and communication among each other.

There are two fundamental models of Interprocess communication

Shared Memory model

Message passing model

In shared memory model a region of memory is shared by the cooperating process. 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 also easier to implement in a distributed system than shared memory.

The shared memory is faster than that of message passing.

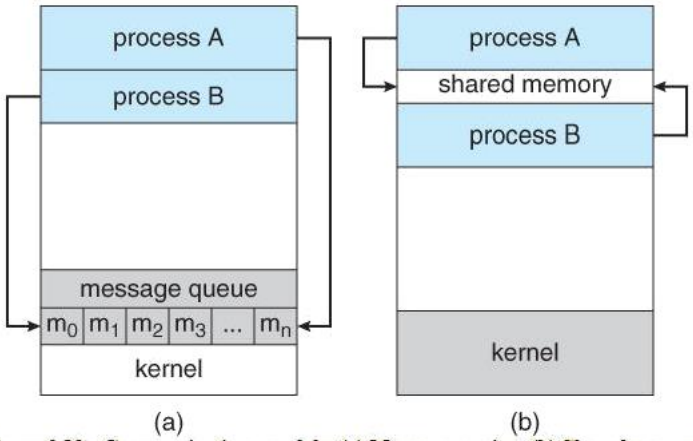


Figure 3.12 - Communications models: (a) Message passing. (b) Shared memory.

Shared-Memory Systems:

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

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.

EXAMPLE: PRODUCER – CONSUMER PROCESS:

A **producer** process produces information that is consumed by a **consumer** process.

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.

Two types of buffers can be used.

Bounded Buffer.

Unbounded Buffer.

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

The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The following variables reside in a region of memory shared by the producer and consumer processes:

Downloaded from: annauniversityedu.blogspot.com

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

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.

CODE FOR PRODUCER PROCESS:

```
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;
}
```

The producer process has 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.

CODE FOR CONSUMER PROCESS

```
item next consumed;
while (true) {
while (in == out)
; /* do nothing */ 10
```

Direct or indirect communication

Direct Communication:

Each process that wants to communicate must explicitly name the recipient or sender of the communication.
Direct communication can be done in two ways symmetric addressing and asymmetric addressing.

In Symmetric addressing both the sender process and the receiver process must name the other to communicate.

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

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

In Asymmetric addressing only the sender names the recipient; the recipient is not required to name the sender.

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

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

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.

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.

Downloaded from: annauniversityedu.blogspot.com

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.
A mailbox may be owned either by a process or by the operating system.

Synchronous or asynchronous communication

Communication between processes takes place through calls to send() and receive() primitives.
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.
When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver.
Automatic or explicit buffering:
Messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- o Zero capacity.
- o Bounded capacity
- o unbounded capacity

Zero capacity. The queue has a maximum length of zero 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.

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

```
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

Message-Passing Systems:
Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space
A message-passing facility provides at least two operations:

Receive(message)

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

THREADS-OVERVIEW:

A thread is a basic unit of CPU utilization. It is a smallest unit of execution of a program that determines the flow of control of execution.
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.
Within a process, there may be one or more threads, each with the following:
A thread execution state (Running, Ready, etc.).

Downloaded from: annauniversityedu.blogspot.com

A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.

An execution stack

Access to the memory and resources of its process

A process with a single flow of control is called as heavy weighted process.

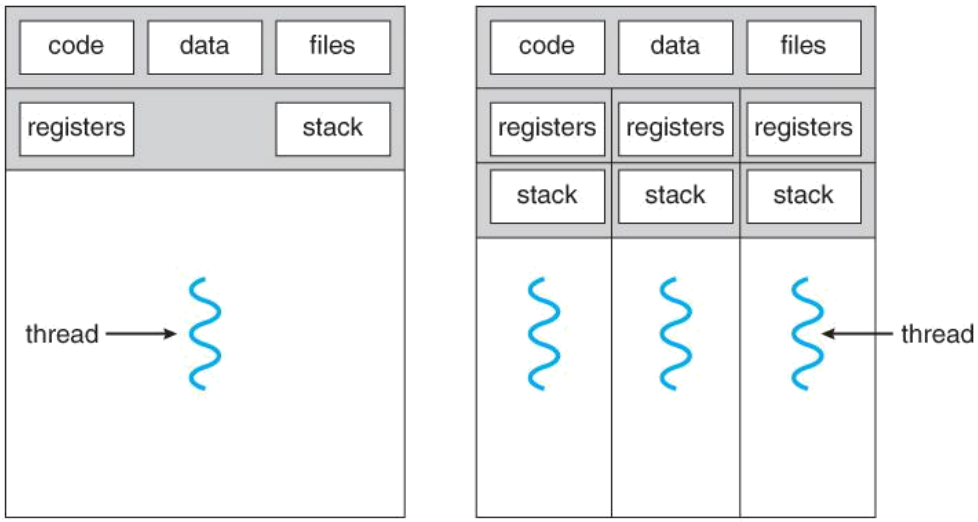


Figure 4.1 - Single-threaded and multithreaded processes

MULTITHREADING: Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

Example: Multithreaded server architecture:

A web server accepts client requests for web pages, images, sound, and so forth.

A Busy web server may have several clients concurrently accessing it.

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

One solution is to have the server run as a single process that accepts requests.

When the server receives a request, it creates a separate process to service that request.Here Process creation is time consuming and resource intensive

Rather, 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.

Threads also play a vital role in remote procedure call (RPC) systems.

ADVANTAGES OF MULTITHREADING:

Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked.

Resource sharing: threads share the memory and the resources of the process to which they belong by default.

Economy of scale: Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

Effective multiprocessor utilization: The benefits of multithreading can be even greater in a multiprocessor architecture

The process of placing multiple computing cores on a single chip is called as multicore programming.

Each core appears as a separate processor to the operating system, whether the cores appear across CPU chips or within CPU chips.

Downloaded from: annauniversityedu.blogspot.com

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

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core

A system is parallel if it can perform more than one task simultaneously.

A concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism.

There are two types of parallelism

- o **Data Parallelism**
- o **Task parallelism.**

Data parallelism :

This focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

This focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

➢

Task parallelism:

This involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data**.**

Thus data parallelism involves the distribution of data across multiple cores and task parallelism on the distribution of tasks across multiple cores.

This involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

Thus data parallelism involves the distribution of data across multiple cores and task parallelism on the distribution of tasks across multiple cores

The challenges in the design of programming for multicore systems includes

Identifying tasks. This involves examining applications to find areas that can be divided into separate, concurrent tasks.

Balance. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.

Data splitting. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency

Testing and debugging. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult .

User level threads(ULTs)

Kernel-level threads (KLTs).

Downloaded from: annauniversityedu.blogspot.com

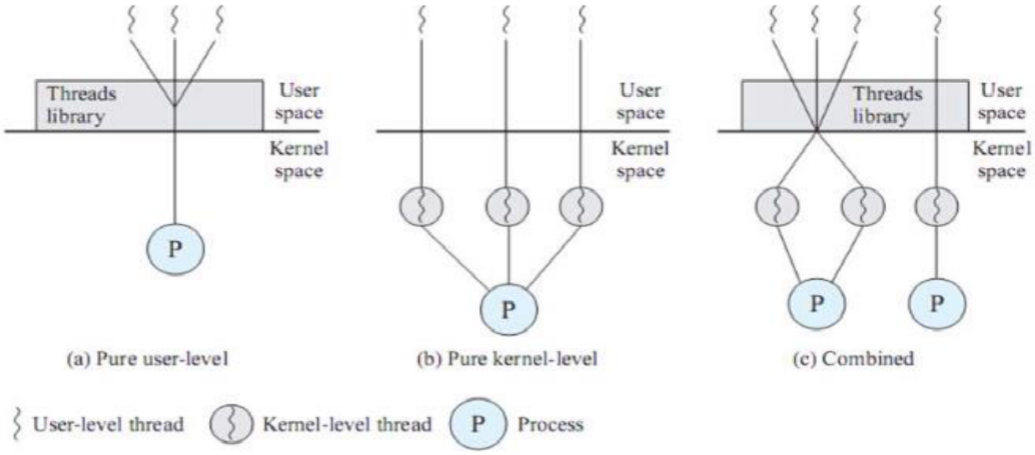


Figure 4.6 User-Level and Kernel-Level Threads

User level thread:

All of the thread management is done by the application and the kernel is not aware of the existence of threads.

The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

At any time that the application is running the application may create a new thread to run within the same process.

Advantages of user level threads:

There are a number of advantages to the use of ULTs instead of KLTs

Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS.

Disadvantages of user level threads:

When a ULT executes a blocking system call, not only is that thread blocked, but also all of the threads within the process are blocked.

A Multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time.

Kernel level threads:

All of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility.

The kernel maintains context information for the process as a whole and for individual threads within the process.

Advantages of Kernel level threads:

The kernel can simultaneously schedule multiple threads from the same process on multiple processors.

If one thread in a process is blocked, the kernel can schedule another thread of the same process.

Disadvantages of kernel level threads:

The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combined approach:

Some operating systems provide a combined ULT/KLT facility

In a combined system, thread creation is done completely in user space

The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs.

In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process.

Relationship between user threads and kernel threads:

Many-to-one model,

One-to-one model,

Many-to many model.

Downloaded from: annauniversityedu.blogspot.com

MANY TO ONE MODEL:

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

The entire process will block if a thread makes a blocking system call.

Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. Example: Green threads in Solaris Operating system

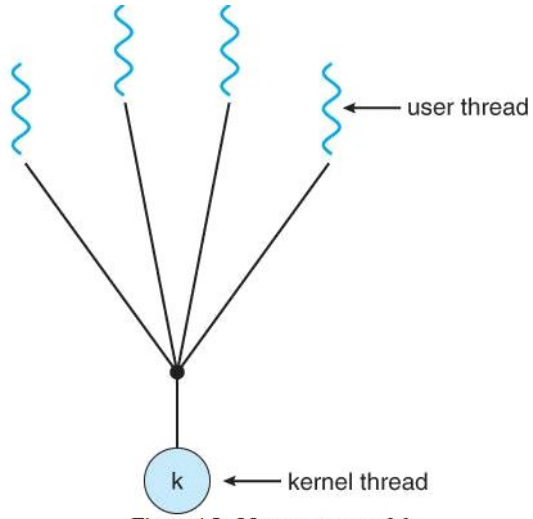


Figure 4.5 - Many-to-one model

ONE TO ONE MODEL:

The one-to-one model maps each user thread to a kernel thread.

It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

It also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

Example: Linux and Windows operating system

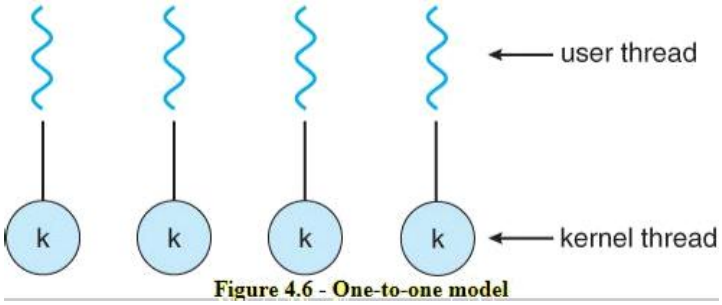


Figure 4.6 - One-to-one model

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 developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

Downloaded from: annauniversityedu.blogspot.com

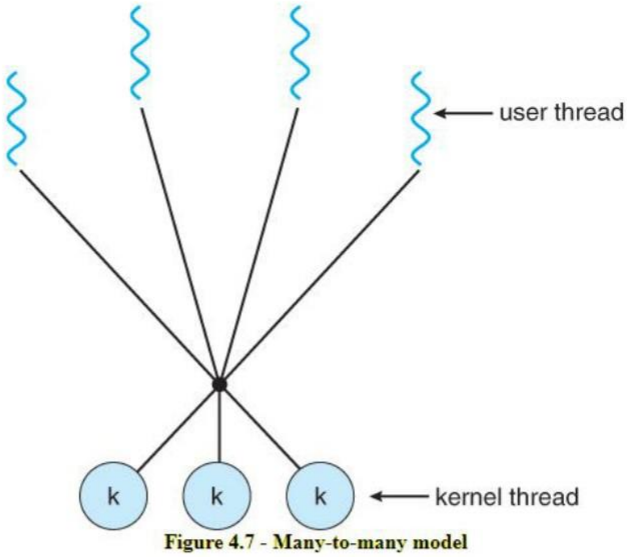


Figure 4.7 - Many-to-many model

TWO LEVEL MODELS:

Many-to-many model multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.

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

WINDOWS 7 AND SMP MANAGEMENT:

The important characteristics of Windows processes are,

i) Windows processes are implemented as objects

ii) An executable process may contain one or more threads.

iii) Both process and thread objects have built-in synchronization capabilities

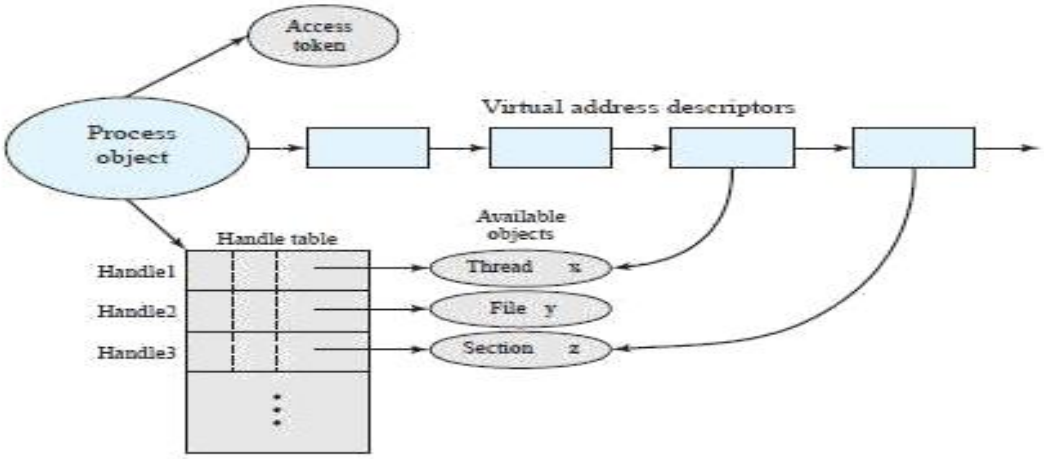


Figure 4.12 A Windows Process and Its Resources

Each process is assigned a security access token, called the primary token of the process When a user first logs on,

Windows creates an access token that includes the security ID for the user.

Every process that is created by or runs on behalf of this user has a copy of this access token.

Windows uses the token to validate the user's ability to access secured objects or to perform restricted functions on the system

Process contains a series of blocks that define the virtual address space currently assigned to this process.

The process includes an object table, with handles to other objects such as threads, files and data known to this process.

Downloaded from: annauniversityedu.blogspot.com

Process and Thread Objects:

Windows makes use of two types of process-related objects:

processes

Threads

A process is an entity corresponding to a user job or application that owns resources, such as memory, and opens files.

A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Each process is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform.

Process ID	A unique value that identifies the process to the operating system.
Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Each thread is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform.

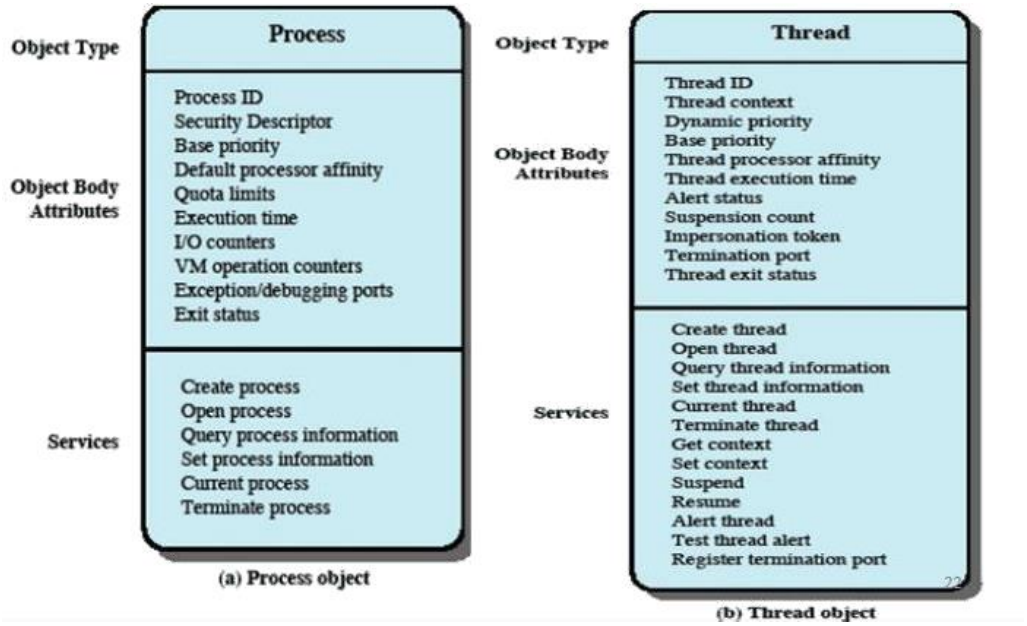
Process ID	A unique value that identifies the process to the operating system.
Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Downloaded from: annauniversityedu.blogspot.com

A Windows process must contain at least one thread to execute. That thread may then create other threads.

In a multiprocessor system, multiple threads from the same process may execute in parallel

An attribute called thread processor affinity is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the process processor affinity.



Multithreading:

Windows supports concurrency among processes because threads in different processes may execute concurrently.

Multiple threads within the same process may be allocated to separate processors and execute simultaneously.

Threads within the same process can exchange information through their common address space and have access to the shared resources of the process.

Threads in different processes can exchange information through shared memory that has been set up between the two processes.

Thread States:

An existing Windows thread is in one of six states

Ready: May be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.

Standby: A standby thread has been selected to run next on a particular Processor. Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher priority thread

Waiting: A thread enters the Waiting state when (1) it is blocked on an event such as I/O operation or it voluntarily waits for synchronization purposes

Transition: A thread enters this state after waiting if it is ready to run but the resources are not available.

Terminated: A thread can be terminated by itself, by another thread, or when its parent process terminates.

Downloaded from: annauniversityedu.blogspot.com

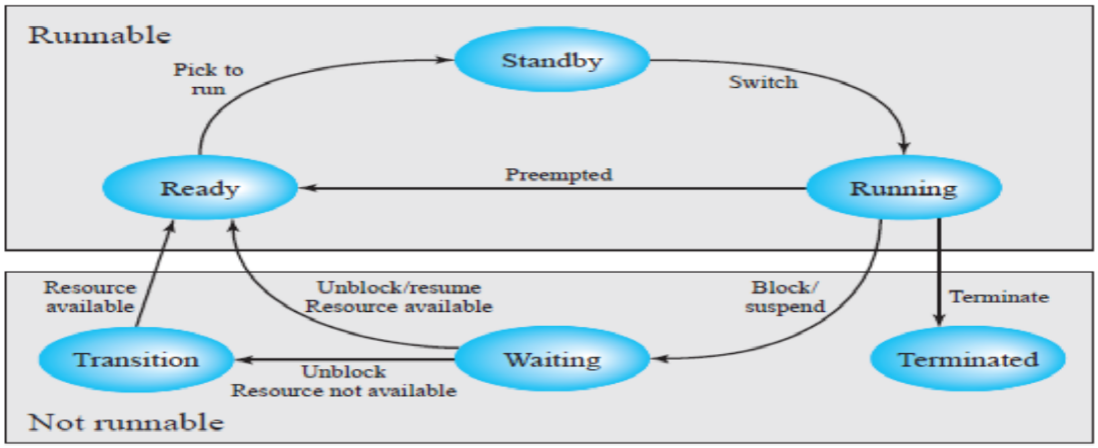


Figure 4.14 Windows Thread States

SYMMETRIC MULTIPROCESSOR SUPPORT:

The threads of any process can run on any processor.
In the absence of processor affinity the microkernel assigns a ready thread to the next available processor.
This assures that no processor is idle or is executing a lower-priority thread when a higher-priority thread is ready.
As a default, the microkernel uses the policy of soft affinity in assigning threads to processors.
The dispatcher tries to assign a ready thread to the same processor it last ran on. This helps reuse data still in that processor's memory caches from the previous execution of the thread. This is called as soft affinity.
It is possible for an application to restrict its thread execution to certain processors (hard affinity).

PROCESS SYNCHRONIZATION:

Process Synchronization is defined as the process of sharing system resources by cooperating processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.

EXAMPLE:

Consider the producer consumer process that contains a variable called counter.
Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
The code for the producer process is

```
while (true) {
/* produce an item in next produced */
while (counter == BUFFER SIZE)
  /* do nothing */ 20 buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
counter++;
}
```

The code for the consumer process is **while**

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

Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements —counter++‖ and —counter--‖.
Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

Downloaded from: annauniversityedu.blogspot.com

When several processes access and manipulate the same data concurrently the outcome of the execution depends **on the particular order in which the access takes place, is called a race condition.**
To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter.

CRITICAL SECTION PROBLEM:

The **critical-section problem** is to design a protocol that the processes can use to cooperate.
Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
When one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The structure of critical section problem is

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.
The critical section may be followed by an **exit section**.
The remaining code is the **remainder section**.

A solution to the critical-section problem must satisfy the following three requirements:

1**. Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next.

Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

SOLUTIONS TO CRITICAL SECTION PROBLEM:

PETERSON'S SOLUTION:

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process.
Peterson's solution requires the two processes to share two data items:

int turn;
boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

Downloaded from: annauniversityedu.blogspot.com

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby checking that if the other process wishes to enter the critical section, it can do so.

Similarly to enter the critical section, process Pj first sets flag[j] to be true and then sets turn to the value i, thereby checking that if the other process wishes to enter the critical section.

The solution is correct and thus provides the following.

Mutual exclusion is preserved.

The progress requirement is satisfied.

The bounded-waiting requirement is met.

MUTEX LOCKS:

Mutex locks are used to protect critical regions and thus prevent race conditions

A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

The acquire() function acquires the lock, and the release() function releases the lock.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not.

If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.

A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of acquire() is as follows:

```
acquire()
{
while (!available)
  /* busy wait */
available = false;;
}
```


Downloaded from: annauniversityedu.blogspot.com

The definition of release() is as follows:

```
release()
{
available = true;
}
```

The main disadvantage of the implementation given here is that it requires **busy waiting**.
While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
This type of mutex lock is also called a **spinlock** because the process ―spins‖ while waiting for the lock to become available.
Busy waiting wastes CPU cycles that some other process might be able to use productively.
Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time

SEMAPHORES:

A **semaphore** S is an integer variable that, is accessed only through two standard atomic operations: wait() and signal().
The wait() operation was originally termed P and the meaning is to test, the signal() was originally called V and the meaning is to increment.

The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
  // busy
wait S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
S++;
}
```

Operating systems often distinguish between counting and binary semaphores.
The value of a **counting semaphore** can range over an unrestricted domain.
The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks.
Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
In this case the semaphore is initialized to the number of resources available.
Each process that wishes to use a resource performs a wait() operation on the
semaphore When a process releases a resource, it performs a signal() operation.
When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

SEMAPHORE IMPLEMENTATION:

The main disadvantage of semaphore is that it requires busy waiting. When one process is in its critical section any other process that tries to enter the critical section must loop continuously in the entry code.
The mutual exclusion implementation with semaphores is given
by **do { wait(mutex);**
//critical section signal(mutex);
//remainder section }while(TRUE);

Downloaded from: annauniversityedu.blogspot.com

To overcome the need for busy waiting, we can modify the wait() and signal() operations.
When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.
The block operation places a process into a waiting queue associated with the semaphore.
Then control is transferred to the CPU scheduler, which selects another process to execute.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
The process is restarted by a wakeup() operation, which changes the process fromthe waiting state to the ready state.
To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
int value;
struct process *list;
} semaphore
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes.
A signal() operation removes one process from the list of waiting processes and awakens that process.

The wait() semaphore operation can be defined as

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

The signal() semaphore operation can be defined as

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

The Block() and wakeup() operations are provided by the operating system as system calls.

DEADLOCKS AND STARVATION:

The implementation of a semaphore with waiting queue may result in a situation where two or more process are waiting indefinitely for an event that can be caused only by one of the waiting process**. This situation is called as deadlock**.

Example: Consider a system consisting of two process p0 and p1, each accessing two semaphores that is set to value 1.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Downloaded from: annauniversityedu.blogspot.com

P0 executes wait(S) and p1 executes Wait(Q).
When P0 executes wait(Q), it must wait until P1 executes signal(Q).
Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).
Here the signal() operations cannot be executed, P0 and P1 are deadlocked.

PRIORITY INVERSION:

Assume we have three processes—L, M, and H—whose priorities follow the order L < M < H.
The process H requires resource R, which is currently being accessed by process L.
Process H would wait for L to finish using resource R. Suppose that process M becomes runnable, thereby preempting process L.
Indirectly, a process with a lower priority—process M—has affected process H that is waiting for L to release resource R.**This problem is known as priority inversion**
Priority-inheritance can solve the problem of priority inversion.
According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources that are requested.
When they are finished, their priorities revert to their original values.

CLASSIC PROBLEMS OF SYNCHRONIZATION:

1. Bounded Buffer problem:

In this problem, the producer process produces the data and the consumer processes consumes the data. Both of the process share the following data structures:

```
int n;
Semaphore mutex = 1;
Semaphore empty = n;
Semaphore full = 0
```

Assume that the pool consists of n buffers, each capable of holding one item.
The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers.
The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

Code for producer process:

```
do{
/* produce an item in next_produced */
……
wait(empty);
wait(mutex);
…..
/ * add next produced to the buffer
*/ signal(mutex);
signal(full); }
while(true);
```

Code for Consumer process:

```
do{
wait(full);
wait(mutex);
……
/* remove an item from buffer to next_consumed */
signal(mutex);
signal(empty);
……..
/* consume the item in next_consumed */
} while(true);
```


Downloaded from: annauniversityedu.blogspot.com

THE DINING-PHILOSOPHERS PROBLEM:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
When a philosopher gets hungry she tries to pick up the two chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks.
One simple solution is to represent each chopstick with a semaphore.
A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal () operation on the appropriate semaphores.
The shared data is semaphore chopstick [5]; where all the elements of the chopsticks are initialized to

```
1. do {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        ……
        /* eat for awhile */
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        …..
        /* think for awhile */
 } while (true);
```

Several possible remedies to the deadlock problem are replaced by:

Allow at most four philosophers to be sitting simultaneously at the table.
Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick

THE READERS WRITERS PROBLEM:

A database is to be shared among several concurrent processes.
Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**.
If two readers access the shared data simultaneously, no effects will result.

Downloaded from: annauniversityedu.blogspot.com

However, if a writer and some other process (either a reader or a writer) access the database simultaneously, problems may occur.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

Semaphore rwmutex = 1;
Semaphore mutex = 1;
int read count = 0;

The semaphores mutex and rwmutex are initialized to 1; read count is initialized to 0. The semaphore rwmutex is common to both reader and writer

Code for writer process:

```
do {
wait(rw mutex);
. . .
/* writing is performed */
. . .
signal(rw mutex);
} while (true);
```

The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
The read count variable keeps track of how many processes are currently reading the object.
The semaphore rwmutex functions as a mutual exclusion semaphore for the writers.

Code for readers process:

```
do {
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex);
signal(mutex);
. . .
/* reading is performed */
. . .
wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```

MONITORS:

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect.

EXAMPLE: Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution: signal(mutex);

```
...
critical section
...
wait(mutex);
```


Downloaded from: annauniversityedu.blogspot.com

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
...
critical section
...
wait(mutex);
```

In this case, a deadlock will occur. To deal with such errors one fundamental high-level synchronization constructs called the **monitor** type is used.

A **monitor type** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.

The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

```
monitor monitor name
{/* shared variable declarations */
        function P1(. .....){
        ... .
        }
        function P2(. .....){
        ....
        }
                .
                .
        function Pn(. .....){
        ....
        }
        initialization_code( ..... ) {
        ....
        }
}
```

Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor.

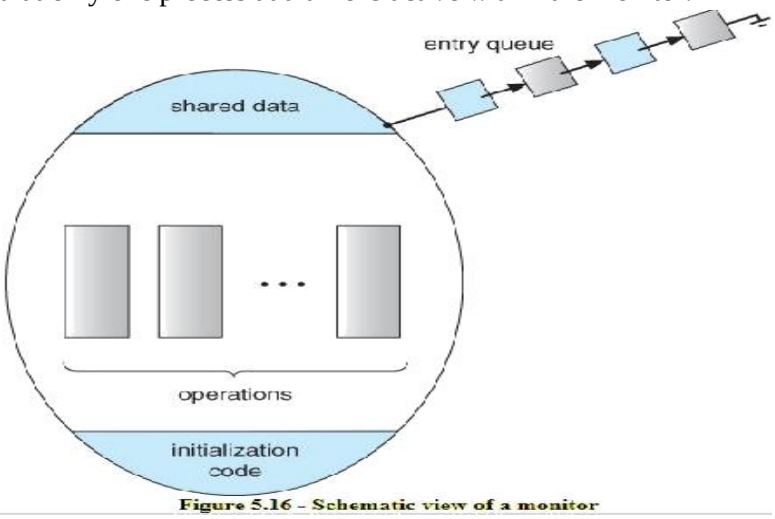


Figure 5.16 – Schematic view of a monitor

Downloaded from: annauniversityedu.blogspot.com

The monitors also provide mechanisms of synchronization by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition: Condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal().
The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal();
The x.signal() operation resumes exactly one suspended process
Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process associated with condition x.
Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor.
Note, however, that conceptually both processes can continuewith their execution. Two possibilities exist:

Signal and wait. P either waits until Q leaves the monitor or waits for another condition.
Signal and continue. Q either waits until P leaves the monitor or waits for another condition.

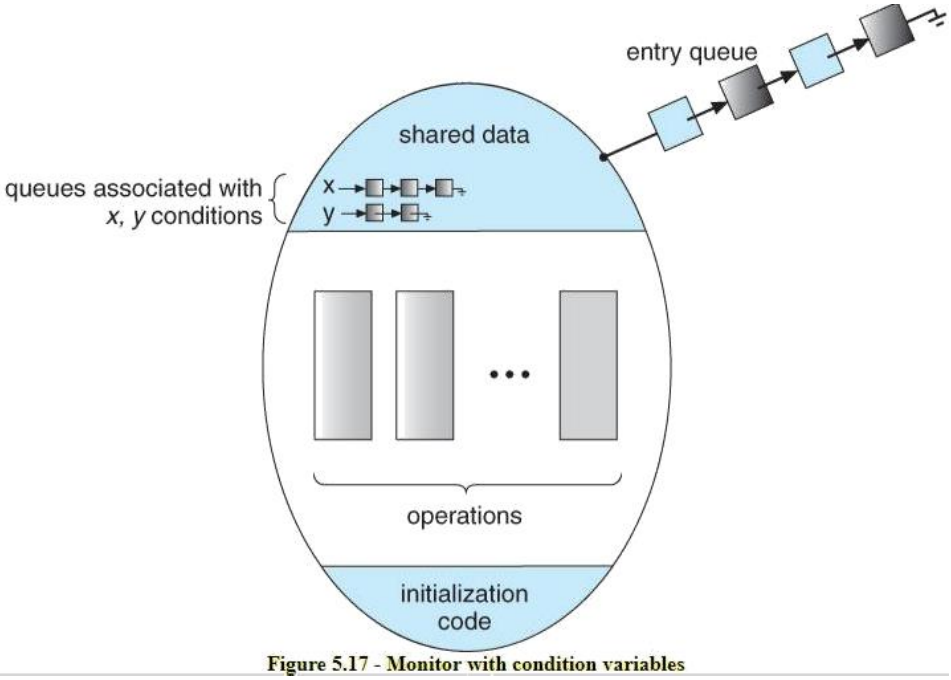


Figure 5.17 - Monitor with condition variables

Dining-Philosophers Solution Using Monitors:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
When a philosopher gets hungry she tries to pick up the two chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks.
The solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Downloaded from: annauniversityedu.blogspot.com

```
monitor DiningPhilsophers
{
        enum {THINKING, HUNGRY, EATING} state[5];
        condition self[5];

        void pickup(int i) {
                state[i] = HUNGRY;
                test(i);
                if(state[i]!= EATING)
                        self[i].wait();
        }

        void putdown(int i){
                state[i] = THINKING;
                test((i+4)%5);
                test((i+1)%5);
        }

        void test(int i) {
                if((state[(i+4)%5] != EATING) &&
                  (state[i] == HUNGRY) &&
                  (state[(i+1)%5] != EATING)){
                        state[i] = EATING;
                        self[i].signal();
                  }
        }

        initialization_code() {
                for(int i=0; i < 5; i++)
                        state[i] = THINKING;
        }
}
```

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating:

(state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

Also declare condition self[5]; that allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Each philosopher, before starting to eat, must invoke the operation pickup().

This may result in the suspension of the philosopher process.

After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation.

Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

DiningPhilosophers.pickup(i);

eat

DiningPhilosophers.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Downloaded from: annauniversityedu.blogspot.com

CPU SCHEDULING ALGORITHM:

CPU–I/O Burst Cycle:

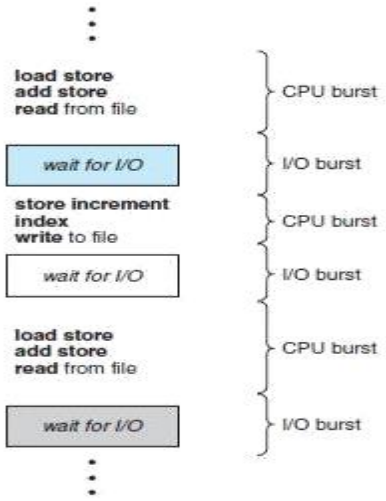
The process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states.

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.

CPU–I/O Burst Cycle:

The process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states.



Preemptive Scheduling and Non Preemptive scheduling:

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.

Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

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

When a process switches from the running state to the waiting state
When a process switches from the running state to the ready state
When a process switches from the waiting state to the ready state
When a process terminates

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

Scheduling Criteria:

The criteria include the following:

CPU utilization. The CPU Should be kept as busy as possible for effective CPU utilization.

Throughput: The total number of process completed per unit time is called as throughput.

Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time = Waiting time + Burst time

Waiting time: Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: The time from the submission of a request until the first response is produced.

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.

First-Come, First-Served Scheduling

Downloaded from: annauniversityedu.blogspot.com

Shortest-Job-First Scheduling
Priority Scheduling
Round-Robin Scheduling

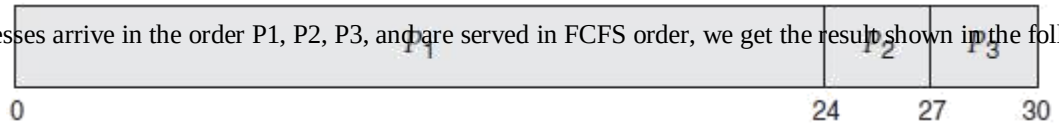
FIRST COME FIRST SERVE SCHEDULING:

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

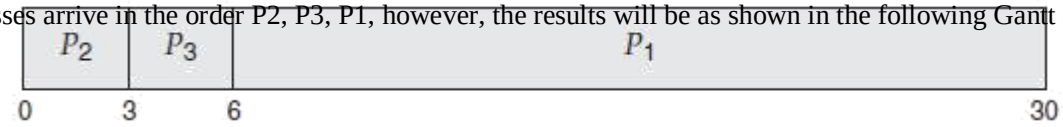
EXAMPLE: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**.



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3.
Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. Thus, the average waiting time under an FCFS policy is generally not minimal.
Assume that we have one CPU-bound process and many I/O-bound processes. The CPU-bound process will get and hold the CPU.
During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
While the processes wait in the ready queue, the I/O devices are idle.
Now the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. Now the CPU sits idle. This is called as CONVOY EFFECT which results in lower CPU and device utilization.
The FCFS scheduling algorithm is non preemptive. 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.

ADVANTAGES:

Better for long processes
Simple method (i.e., minimum overhead on processor)
No starvation

Downloaded from: annauniversityedu.blogspot.com

DISADVANTAGES:

Waiting time can be large if short requests wait behind the long ones.

It is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval.

A proper mix of jobs is needed to achieve good results from FCFS scheduling.

SHORTEST-JOB-FIRST SCHEDULING:

With this algorithm the process that comes with the shortest job will be allocated the CPU first.

This algorithm includes 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.

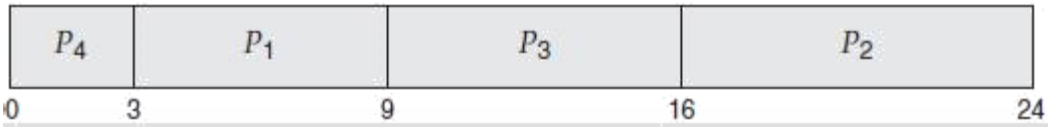
If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

This is also called as shortest next CPU burst algorithm.

EXAMPLE: 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 P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.

Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

The SJF algorithm can be either preemptive or nonpreemptive.

When a new process arrives at the ready queue while a previous process is still executing and the next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, then a preemptive or non preemptive approach can be chosen.

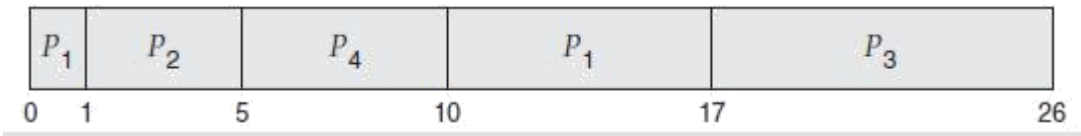
A preemptive SJF algorithm will preempt the currently executing process.

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.

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

The preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in queue. Process P2 arrives at time 1.

The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

Downloaded from: annauniversityedu.blogspot.com

The average waiting time 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

ADVNATAGES:

The SJF scheduling algorithm has minimum average waiting time for a given set of processes.
Gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
Throughput is high.

DISADVANTAGES:

The difficulty with the SJF algorithm is knowing the length of the next CPU request
Starvation may be possible for the longer processes.

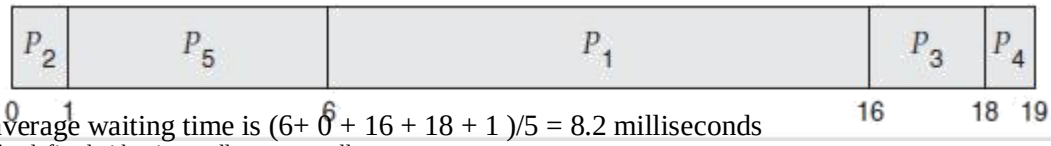
PRIORITY SCHEDULING:

Apriority 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.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

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:.



Thus, the average waiting time is (6+ 0 + 16 + 18 + 1)/5 = 8.2 milliseconds
Priorities can be defined either internally or externally
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 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. Hence the higher-priority processes can prevent a low-priority process from ever getting the CPU. This problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
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.

ADVANTAGES:

1.Simplicity.
Reasonable support for priority.
Suitable for applications with varying time and resource requirements.

Downloaded from: annauniversityedu.blogspot.com

DISADVANTAGES

Indefinite blocking or starvation.

A priority scheduling can leave some low priority waiting processes indefinitely for CPU.

If the system eventually crashes then all unfinished low priority processes gets lost.

ROUND ROBIN SCHEDULING:

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems.

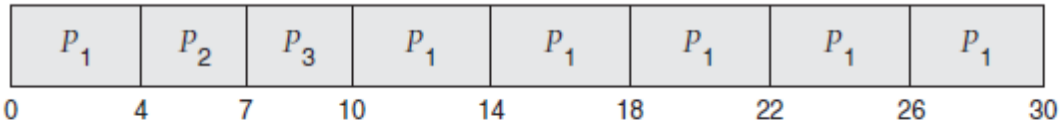
A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length.

The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

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.

EXAMPLE: 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.

*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.

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

If the time quantum is extremely large, the RR policy is the same as the FCFS policy.

If the time quantum is extremely small the RR approach can result in a large number of context switches.

ADVANTAGES:

Does not suffer by starvation.

There is Low throughput.

There are Context Switches.

MULTI LEVEL QUEUE SCHEDULING:

The processes are divided into foreground process and background process.

These two types of processes have different response-time requirements and have different scheduling needs.

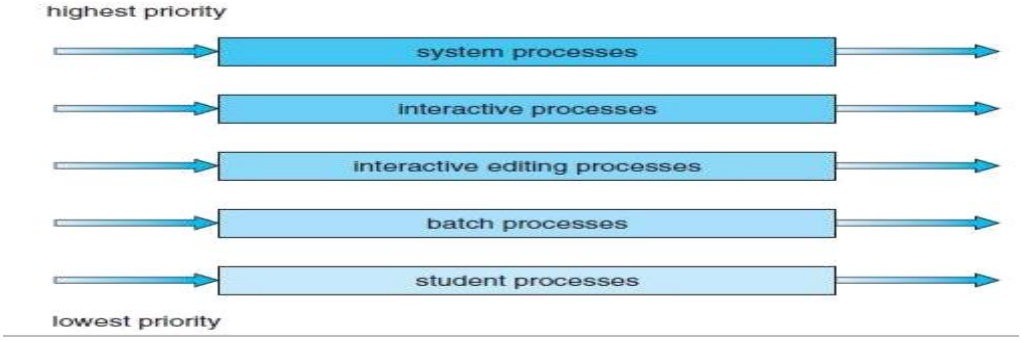
The foreground processes may have priority over background processes.

A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues.

The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.

Downloaded from: annauniversityedu.blogspot.com

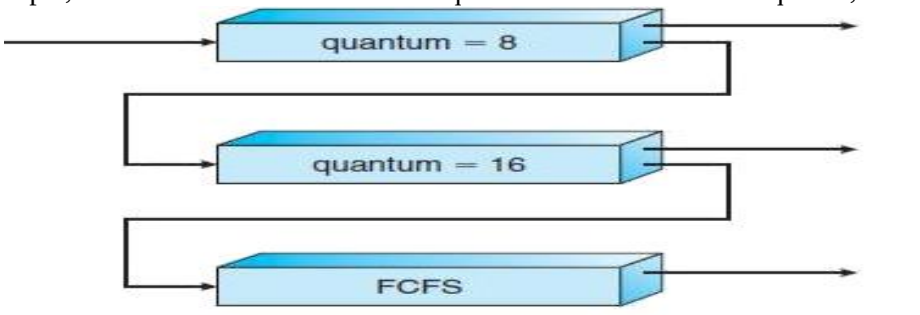
Example of a multilevel queue scheduling algorithm with five queues, in order of priority:
System processes
Interactive processes
Interactive editing processes
Batch processes
Student processes



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

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



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. a multilevel feedback queue

Downloaded from: annauniversityedu.blogspot.com

DEADLOCKS:

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

The resources of a computer system may be partitioned into several types such as CPU cycles, files, and I/O devices (such as printers and DVD drives)

A process must request a resource before using it and must release the resource after using it.

A process may utilize a resource in only the following sequence.

Request. The process requests the resource. If the request cannot be granted immediately then the requesting process must wait until it can get the resource.

Use. The process can operate on the resource

Release. The process releases the resource.

NECESSARY CONDITIONS FOR DEADLOCKS:

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.

Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait. A set *{P0, P1, ..., Pn}* of waiting processes must exist such that *P0* is waiting for a resource held by *P1*, *P1* is waiting for a resource held by *P2*, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P0*.

RESOURCE ALLOCATION GRAPH:

Deadlocks can be described more clearly in terms of a directed graph called a **system resource-allocation graph**.

This graph consists of a set of vertices *V* and a set of edges *E*.

The set of vertices *V* is partitioned into two different types of nodes *P = {P1, P2, ..., Pn}*, the set consisting of all the active processes in the system, and *R = {R1, R2, ..., Rm}*, the set consisting of all resource types in the system.

A directed edge *Pi → Rj* is called a **request edge**; a directed edge *Rj → Pi* is called an **assignment edge**

Represent each process *Pi* as a circle and each resource type *Rj* as a rectangle.

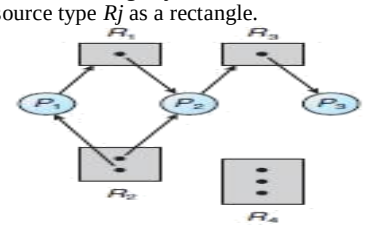


Figure 7.1 Resource-allocation graph.

The sets *P*, *R*, and *E*:

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E={P1→R1,P2→R3,R1→P2,R2→P2,R2→P1,R3→P3}

Process states:

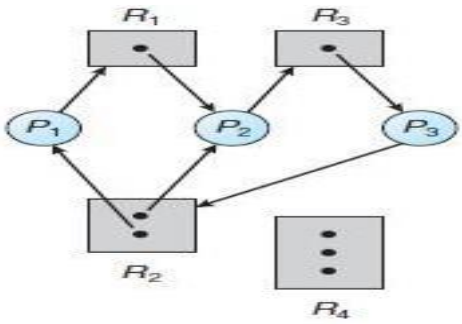
Process *P1* is holding an instance of resource type *R2* and is waiting for an instance of resource type *R1*.

Process *P2* is holding an instance of *R1* and an instance of *R2* and is waiting for an instance of *R3*.

Process *P3* is holding an instance of *R3*.

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist

Downloaded from: annauniversityedu.blogspot.com



Consider a process *P*3 requests an instance of resource type *R*2. Since no resource instance is currently available, we add a request edge *P*3→ *R*2 to the graph.

Two minimal cycles exist in the system:

***P*1→*R*1→*P*2→*R*3→*P*3→*R*2→*P*1**

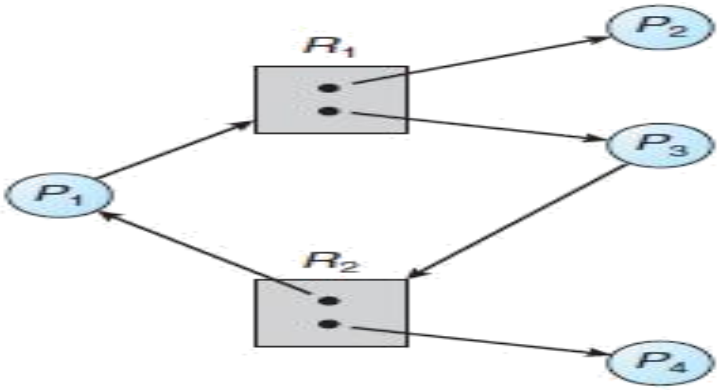
***P*2→*R*3→*P*3→*R*2→*P*2**

Processes *P*1, *P*2, and *P*3 are deadlocked.

Process *P*2 is waiting for the resource *R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process *P*2 to release resource *R*1.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

In the following diagram there is a cycle *P*1 → *R*1 → *P*3 → *R*2 → *P*1.But there is no deadlock.



Resource allocation graph with a cycle but no deadlock.

*P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

If a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

METHODS FOR HANDLING DEADLOCKS:

The method for handling deadlocks include

Deadlock avoidance or Deadlock Prevention.

Deadlock Detection

Deadlock Recovery.

DEADLOCK PREVENTION:

Deadlock prevention provides a set of methods to ensure that at least one of the four necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Downloaded from: annauniversityedu.blogspot.com

MUTUAL EXCLUSION:

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable

Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

We cannot prevent deadlocks by denying the mutual-exclusion condition for the non-sharable resources .

HOLD AND WAIT:

Whenever a process requests a resource, it does not hold any other resources.

This allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

NO PREEMPTION:

If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.

The preempted resources are added to the list of resources for which the process is waiting.

The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

CIRCULAR WAIT:

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R1, R2, ..., Rm\}$ be the set of resource types.

Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

If the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

***F*(tape drive) = 1**

***F*(disk drive) = 5**

***F*(printer) = 12**

Each process can request resources only in an increasing order of enumeration.

That is, a process can initially request any number of instances of a resource type —say, Ri .

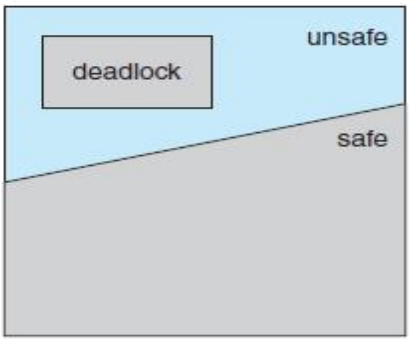
After that, the process can request instances of resource type Rj if and only if $F(Rj) > F(Ri)$.

Example, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

DEADLOCK AVOIDANCE:

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

SAFE STATE:



A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.

A system is in a safe state only if there exists a **safe sequence**.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.

Not all unsafe states are deadlocks; however an unsafe state ***may*** lead to a deadlock.

Downloaded from: annauniversityedu.blogspot.com

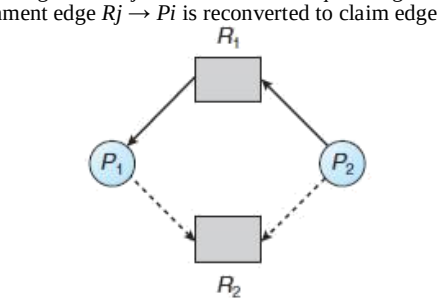
A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if,
the resource requests that *Pi* make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j* < *i*.
Example: consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2
Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.
Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives .

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition.
Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives);
Then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

RESOURCE ALLOCATION GRAPH:

A new type of edge, called a **claim edge** is used in resource allocation graph.
A claim edge *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future.
When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge.
When a resource *Rj* is released by *Pi*, the assignment edge *Rj* → *Pi* is reconverted to claim edge Pi ->Rj



Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.
If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

BANKERS ALGORITHM:

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
This number may not exceed the total number of resources in the system.
When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
The following data structures are needed to implement bankers algorithm, where *n* is the number of processes in the system and *m* is the number of resource types:

Downloaded from: annauniversityedu.blogspot.com

Available: Length *m* indicates the number of available resources of each type.
Max. An *n* × *m* matrix defines the maximum demand of each process.
Allocation. An *n* × *m* matrix defines the number of resources of each type currently allocated.
Need. An *n* × *m* matrix indicates the remaining resource need of each process.

SAFETY ALGORITHM:

It is an algorithm for finding out whether or not a system is in a safe state

Let***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.

Find an index *i* such that both a. ***Finish***[*i*] == ***false***
Need*i* ≤***Work***

Work =***Work*** + ***Allocation****i*
Finish[*i*] = ***true***
Go to step 2.

If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.
This algorithm may require an order of *m* × *n*2 operations to determine whether a state is safe.

RESOURCE REQUEST ALGORITHM: Let
Request*i* be the request vector for process *Pi*.

If ***Request****i* ≤***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

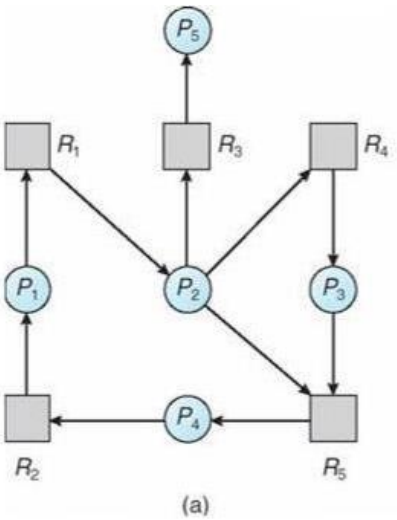
Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows: ***Available*** = ***Available***–***Request****i* ;
Allocation*i* = ***Allocation****i* + ***Request****i*
; ***Need****i* = ***Need****i* –***Request****i* ;
If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Request****i* **,** and the old resource-allocation state is restored.

DEADLOCK DETECTION:

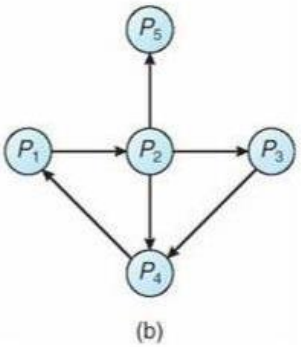
A Deadlock detection algorithm examines the state of the system to determine whether a deadlock has occurred

SINGLE INSTANCE OF EACH RESOURCE TYPE:

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.



Resource-Allocation Graph



Corresponding wait-for graph

Downloaded from: annauniversityedu.blogspot.com

This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
A deadlock exists in the system if and only if the wait-for graph contains a cycle.
To detect deadlocks, the system needs to ***maintain*** the wait for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph.

SEVERAL INSTANCE OF RESOURCE TYPE:

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type
This algorithm uses the following data structure,

Available. A vector of length *m* indicates the number of available resources of each type.

Allocation. An *n* × *m* matrix defines the number of resources currently allocated to each process.

Request. An *n* × *m* matrix indicates the current request of each process.

Let***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available.*** For *i* = 0, 1, ..., *n–1*, if ***Allocation****i* _= 0, then ***Finish***[*i*] = ***false.*** Otherwise, ***Finish***[*i*] = ***true.***

Find an index *i* such that both

Finish[*i*] == ***false***

Request*i* ≤***Work***

Work =***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

If ***Finish***[*i*] ==***false*** for some *i*, 0≤*i*<*n*, then the system is in a deadlocked state.

Moreover, if ***Finish***[*i*] == ***false,*** then process *Pi* is deadlocked.

This algorithm requires an order of *m* × *n*2 operations to detect whether the system is in a deadlocked state.

DEADLOCK RECOVERY:

When a detection algorithm determines that a deadlock exists ,the possibility is to let the system **recover** from the deadlock automatically

There are two options for breaking a deadlock.

1. PROCESS TERMINATION:

It is the process of eliminating the deadlock by aborting a process. It involves two methods

Abort all deadlocked process: This method breaks the deadlock cycle but with a greater expense.

Abort one process at a time until the deadlock cycle is eliminated: This method aborts the deadlocked process one by one and after each process is aborted, it checks whether any process are still deadlocked.

2. RESOURCE PREEMPTION:

To eliminate deadlocks Preempt some resources from the process and give the resource to other process until the deadlock cycle is broken.

Selecting an victim: It is the process of selecting which resource and which process are to be preempted.

Rollback: If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Starvation: The resources will not always be preempted from the same process, else it leads to starvation.

Downloaded from: annauniversityedu.blogspot.com