Operating Systems

www.BrainKart.com

Click Here !!! for Operating Systems full study material.

Click Here !!! for other subjects (Anna University)

Click Here !!! for Anna University Notes Android App.

Click Here !!! for BrainKart Android App.

Click Here for Operating Systems full study material.

UNIT - II PROCESS MANAGEMENT

Processes

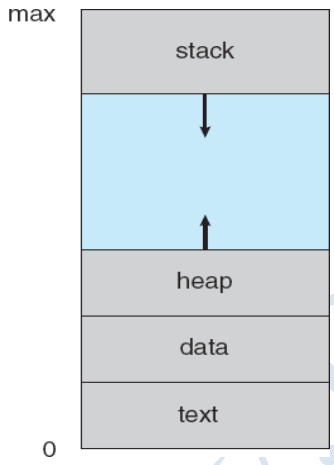
Process Concept

- A process is an instance of a program in execution.
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**

The Process

- Process memory is divided into four sections.
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

A Process in memory



- A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk
- (often called an **executable file**).
- A process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.

- Two common techniques for loading executable files are
  - double-clicking an icon representing the executable file
  - entering the name of the executable file on the command line

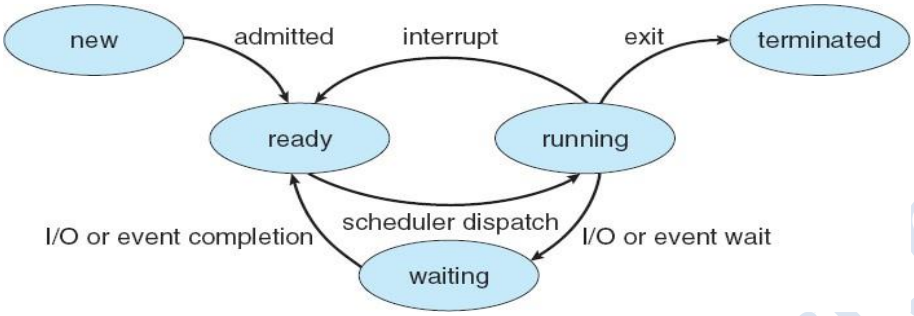
Process State

- 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:

www.BrainKart.com

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

Process State

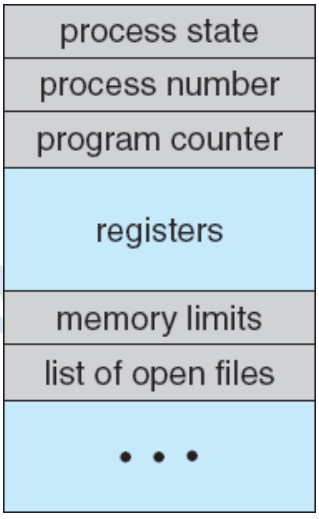


- It is important that only one process can be *running* on any processor at any instant.
- Many processes may be *ready* and *waiting*.

Process Control Block

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
- The PCB simply serves as the repository for any information that may vary from process to process.
- It contains many pieces of information associated with a specific process.

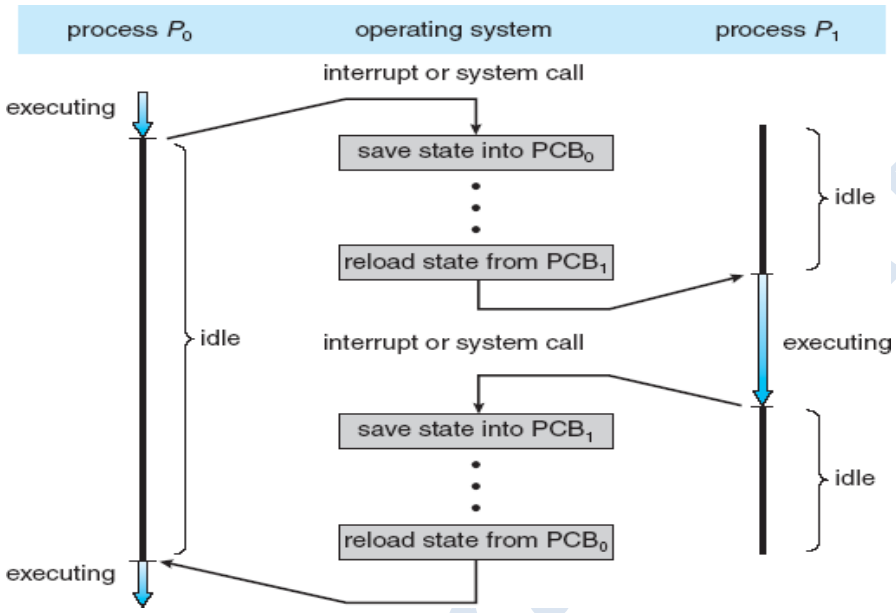
Process Control Block



BrainKart.com

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

CPU switches from process to process



- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Threads

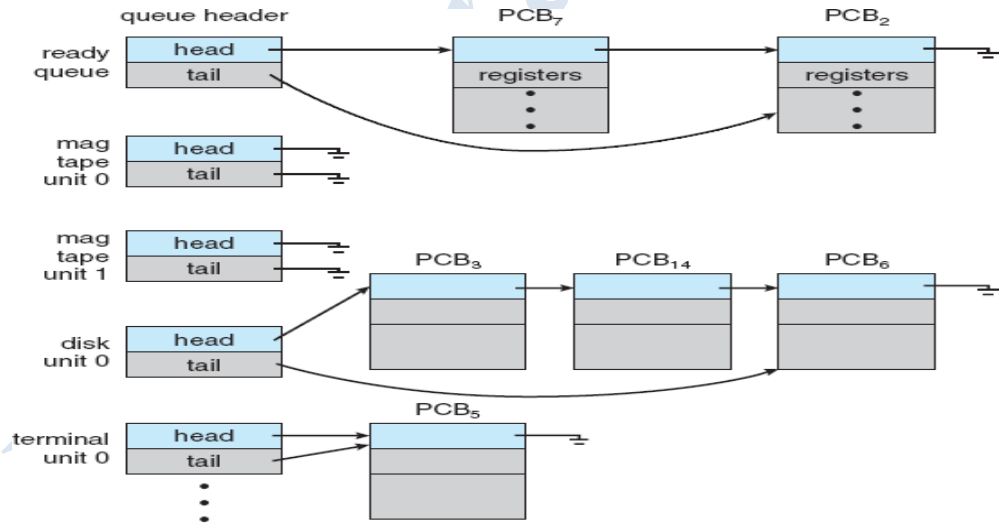
- A process is a program that performs a single **thread** of execution. This single thread of control allows the process to perform only one task at a time.
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

- On a system that supports threads, the PCB is expanded to include information for each thread.

Process Scheduling

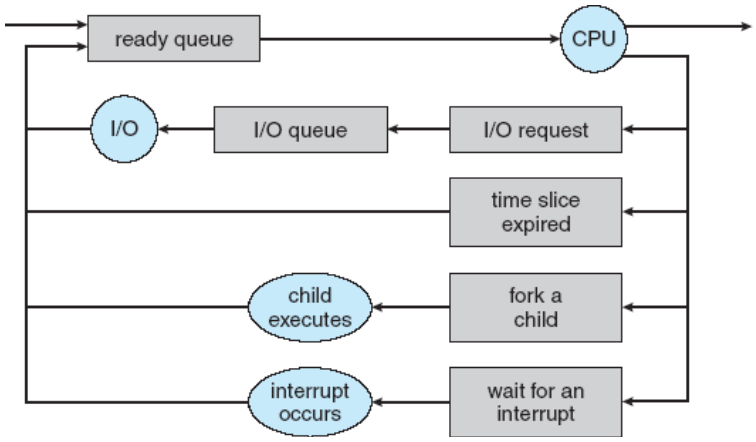
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- **Process scheduler** – selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- **Scheduling Queues**
  - **Job queue –** set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
  - **Device queues –** set of processes waiting for an I/O device. Each device has its own device queue

    Ready Queue and various I/O device Queues



- Processes migrate among the various queues

Queueing-diagram representation of process scheduling.



A common representation of process scheduling is a **queueing diagram.**

- Each rectangular box represents a queue.
- Two types of queues are present:
    - The ready queue and a set of device queues.
    - The circles represent the resources that serve the queues,
    - The arrows indicate the flow of processes in the system
    - A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**.
    - Once the process is allocated the CPU and is executing, one of several events could occur:
        - The process could issue an I/O request and then be placed in an I/O queue.
        - The process could create a new child process and wait for the child's termination.
        - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

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

Schedulers

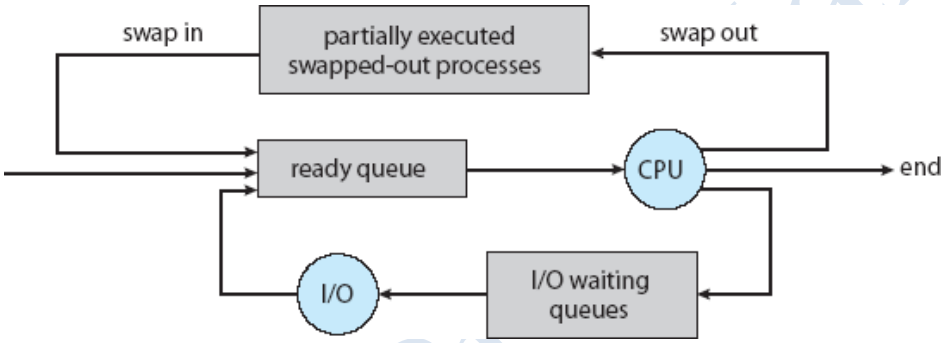
- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate **scheduler**.
- **Long-term scheduler (or job scheduler) –** selects which processes should be brought into the ready queue from the pool
    - Long-term scheduler is invoked infrequently (seconds, minutes) - (may be slow)
    - The long-term scheduler controls the degree of multiprogramming (the number of processes in memory)
    - long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.

www.BrainKart.com

- An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.
- A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes.

- **Short-term scheduler (or CPU scheduler) –** selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) - (must be fast)
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

Addition of medium-term scheduling to the queueing diagram.



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context Switch** - Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Context-switch times are highly dependent on hardware support.
  - some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set.

Operations on Processes

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

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

- The creating process is called a parent process, and the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a **tree** of processes.
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

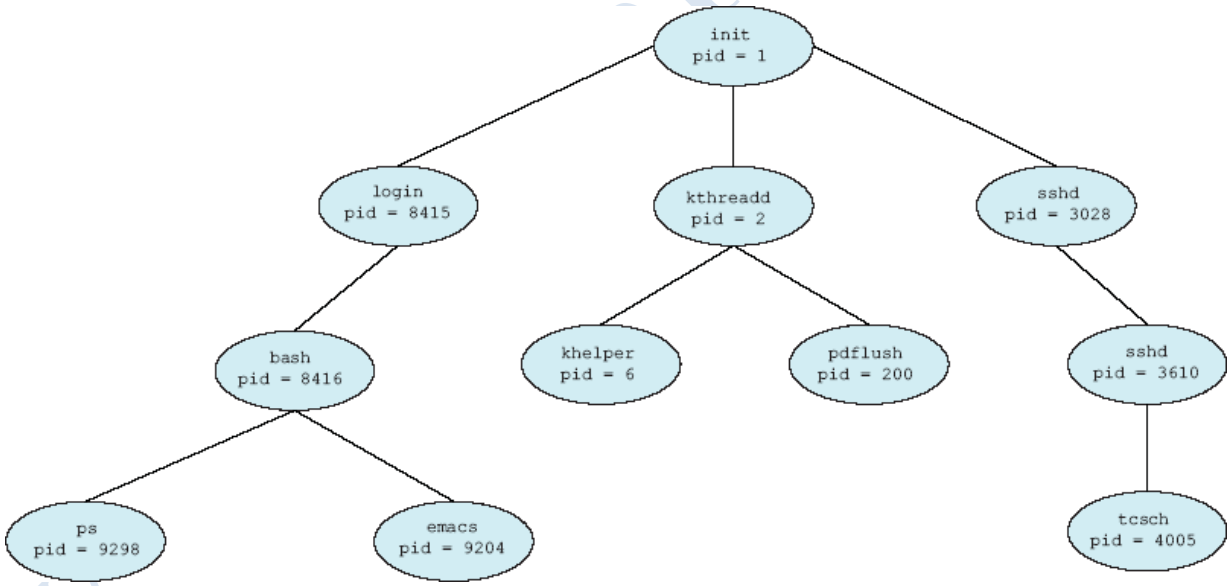
Resource sharing options

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

Execution options

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

A tree of processes on a typical Linux system.



Process Creation

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

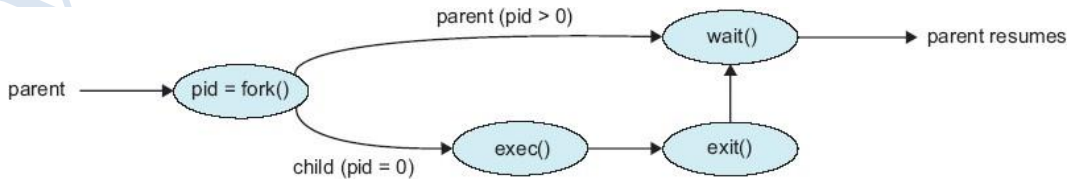
- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.

- Each process is identified by its process identifier, which is a unique integer.
- A new process is created by the fork() system call.
- The new process consists of a copy of the address space of the original process.
- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.

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

```
#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 creation using the fork() system call.



www.BrainKart.com

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
- **cascading termination.** All children, grandchildren, etc. are terminated.
- The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process
  - **pid = wait(&status);**
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

Inter process communication

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

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

Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

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

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

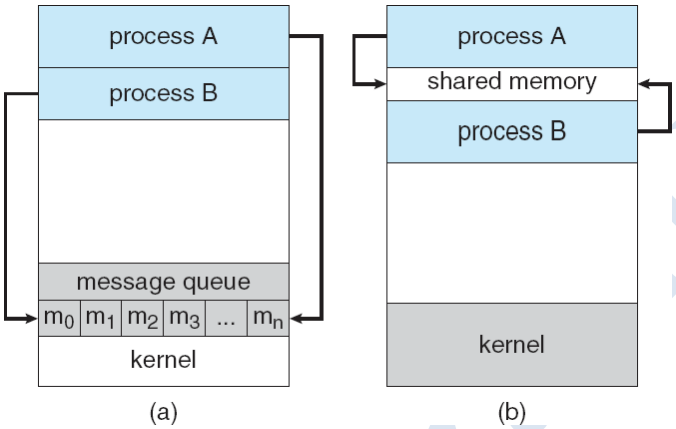
- Cooperating processes require an **interprocess communication (IPC)** mechanism

that will allow them to exchange data and information.

- There are two fundamental models of interprocess communication:
  - **shared memory**
  - **message passing**.

- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

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.
- A shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

Producer–Consumer problem

- Paradigm for cooperating processes, *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.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used.
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

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

The shared buffer is implemented as a circular array with two logical pointers: in and out.

- The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.
- The buffer is empty when in == out;
- The buffer is full when ((in + 1) % BUFFER SIZE) == out.

Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

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

Message Passing System

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)

www.BrainKart.com

- **receive**(*message*)
- The *message* size is either fixed or variable
- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Naming

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

Direct Communication

Each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

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

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

Properties of Communication Link

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

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

A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

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

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

receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
- send(*A, message*) – send a message to mailbox A
- receive(*A, message*) – receive a message from mailbox A
- Mailbox sharing
  - *P1*, *P2*, and *P3* share mailbox A
  - *P1*, sends; *P2* and *P3* receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

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

Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

- Different combinations of send() and receive() are possible.When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.
- The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements.
- The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. when the consumer invokes receive(), it blocks until a message is available.

The producer process using message passing.

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

The consumer process using message passing.

```
message next_consumed;
while (true) {
  receive(next_consumed);
   /* consume the item in next consumed */
}
```

Buffering

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

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

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

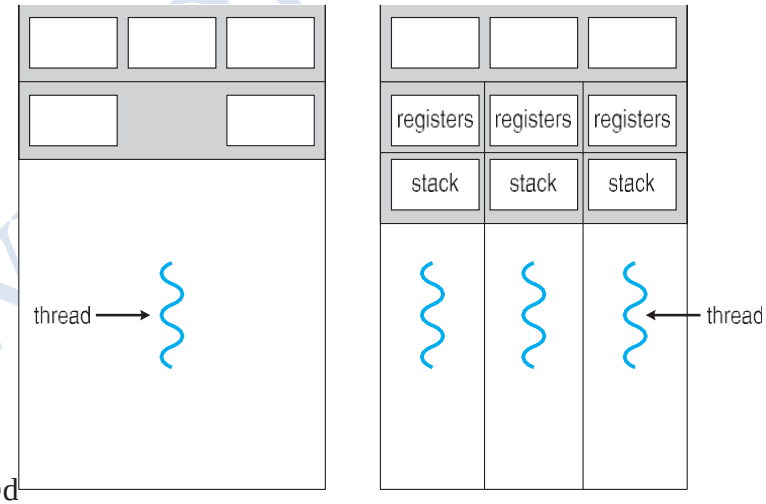
• **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Threads

Overview

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

Single and Multithreaded Processes

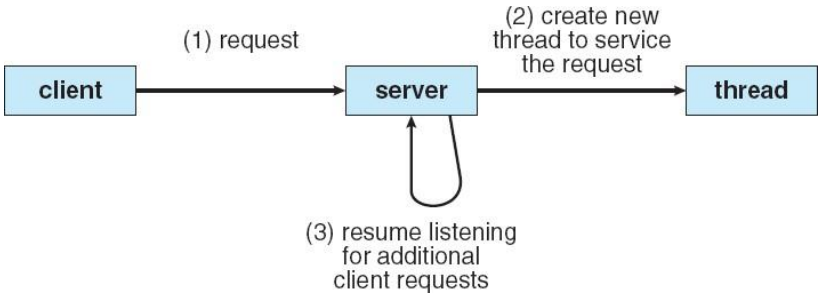


single-threaded process multithreaded process

Motivation

- Most mod
- Threads r
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
- Answer a network request

- Process creation is heavy-weight while thread creation is light-weight. 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.
- Can simplify code, increase efficiency
- Most operating system kernels are generally multithreaded.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**.

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

2. **Resource sharing**.

Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer.

However, threads share the memory and the resources of the process to which they belong by default.

The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**.

Allocating memory and resources for process creation is costly.

4. **Scalability.**

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

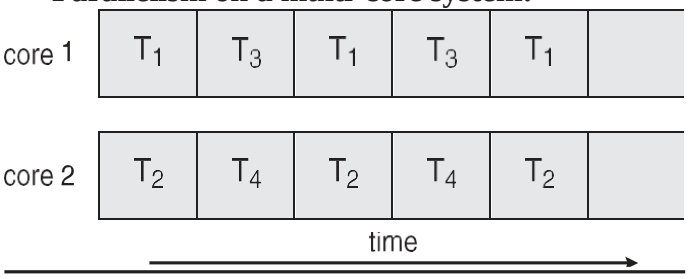
Multicore Programming

Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems.

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

Concurrent execution on single-core system:

Parallelism on a multi-core system:



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.

Parallelism implies a system can perform more than one task simultaneously

Concurrency supports more than one task making progress

Programming Challenges

Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution.

For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

Five areas present challenges in programming for multicore systems:

1. Identifying tasks. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

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

3. Data splitting. The data accessed and manipulated by the tasks must be divided to run on separate cores.

4. 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.

5. 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 than testing and debugging single-threaded applicatio**ns.**

Types of parallelism

- **Data parallelism –** distributes subsets of the same data across multiple cores, same operation on each

- **Task parallelism –** distributing threads across cores, each thread performing unique operation

Multithreading models

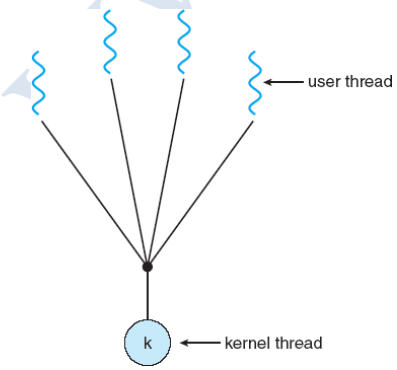
- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

User threads	**Kernel threads**
User threads are supported above the Kernel and are implemented by a thread library at the user level	Kernel threads are supported directly by the operating system
Thread creation & scheduling are done in the user space, without kernel. Therefore they are fast to create and manage	Thread creation, scheduling and management are done by the operating system.
Blocking system call will cause the entire process to block	If the thread performs a blocking system call, the kernel can schedule another thread in the application for execution

- Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.

- **Many-to-One**
- **One-to-One**
- **Many-to-Many**

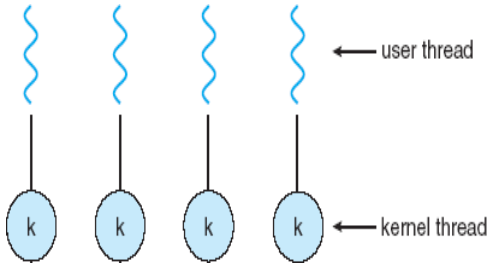
Many-to-One Model



www.BrainKart.com

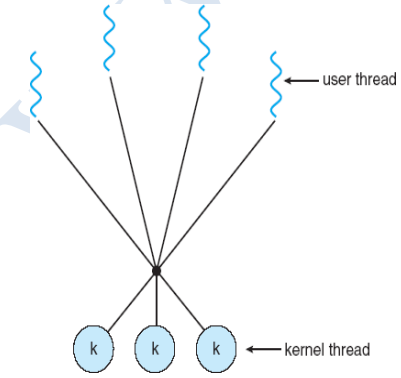
- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient
- The entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time,multiple threads are unable to run in parallel on multicore systems.
- **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.
- However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

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.
- **Drawback** - creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.

Many-to-Many Model

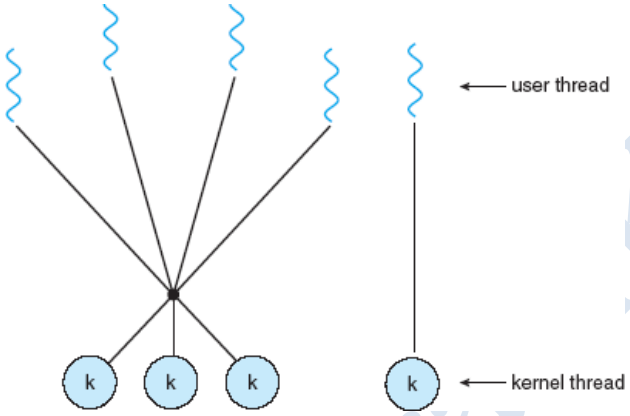


- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine
- Many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.

Drawback

Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still 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 sometimes referred to as the **two-level model.**



Windows 7 – Thread and SMP Management

The native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality.

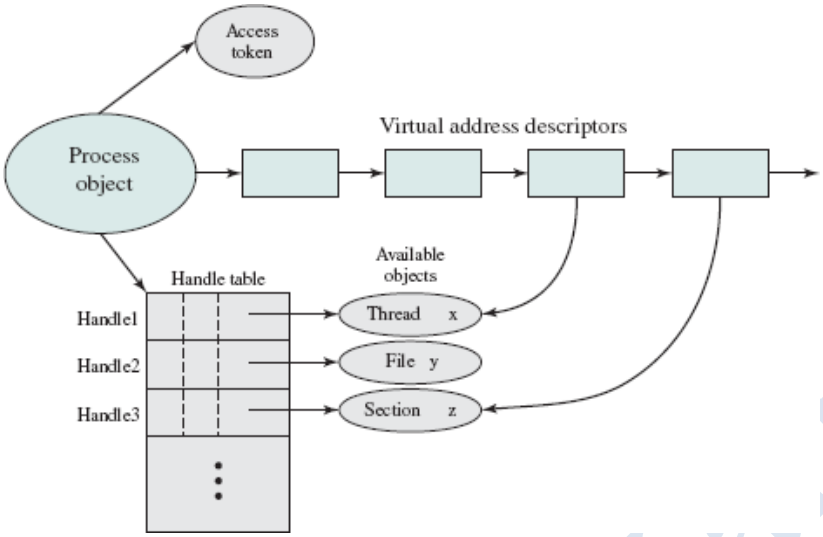
Characteristics of Windows processes:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

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 and on secured objects. The access token controls whether the process can change its own attributes.



- Related to the process is a series of blocks that define the virtual address space currently assigned to this process.
- The process cannot directly modify these structures but must rely on the virtual memory manager, which provides a memory allocation service for the process.
- The process includes an object table, with handles to other objects known to this process. Figure The process has access to a file object and to a section object that defines a section of shared memory.

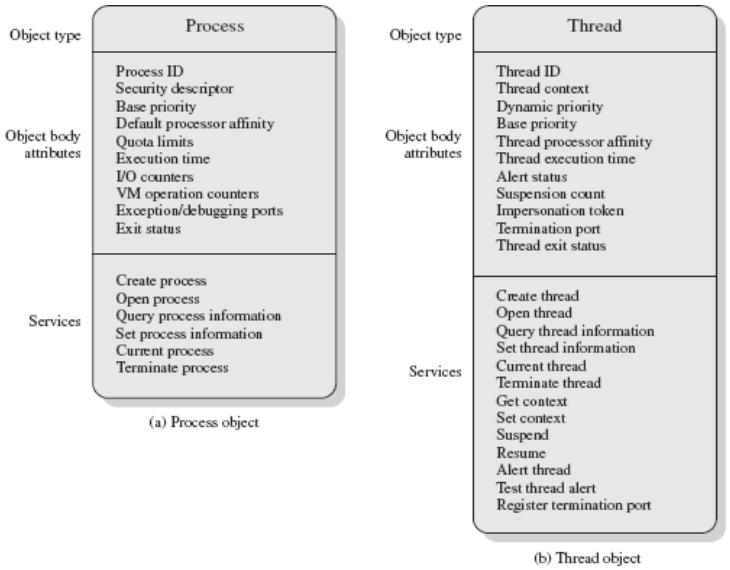
Process and Thread Objects

- The object-oriented structure of Windows facilitates the development of a general- purpose process facility.
- Windows makes use of two types of process-related objects: processes and threads.
- A process is an entity corresponding to a user job or application that owns resources, such as memory and open files.
- A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Windows Process and Thread Objects

www.BrainKart.com

Click Here for Operating Systems full study material.



(a) Process object

(b) Thread object

Windows Process Object Attributes

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 the 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.

Windows Thread Object Attributes

www.BrainKart.com

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

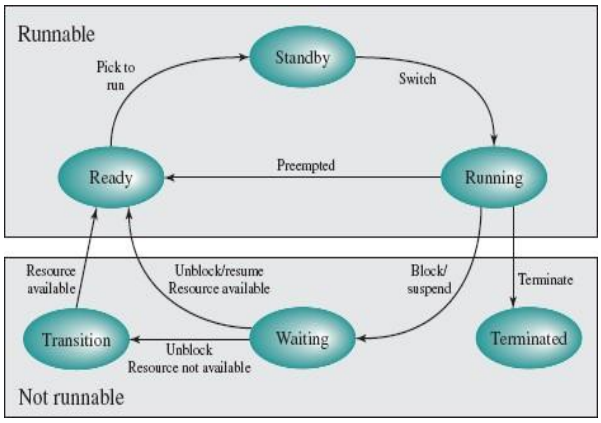
The *thread processor affinity* is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the *process processor affinity*.

Multithreading

- Windows supports concurrency among processes because threads in different processes may execute concurrently (appear to run at the same time).
- Multiple threads within the same process may be allocated to separate processors and
- execute simultaneously (actually run at the same time).
- A multithreaded process achieves concurrency without the overhead of using multiple processes.
- 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

www.BrainKart.com



An existing Windows thread is in one of six states.

- **Ready:** A ready thread 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. The thread waits in this state until that processor is made available.
- **Running:** 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, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state.
- **Waiting:** A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.
- **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.

Symmetric Multiprocessing Support

Windows supports SMP hardware configurations.

The kernel dispatcher uses the policy of **soft affinity** in assigning threads to processors:

Soft affinity: 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.

Hard affinity: It is possible for an application to restrict its thread execution only to certain processors

CPU Scheduling

Basic Concepts:

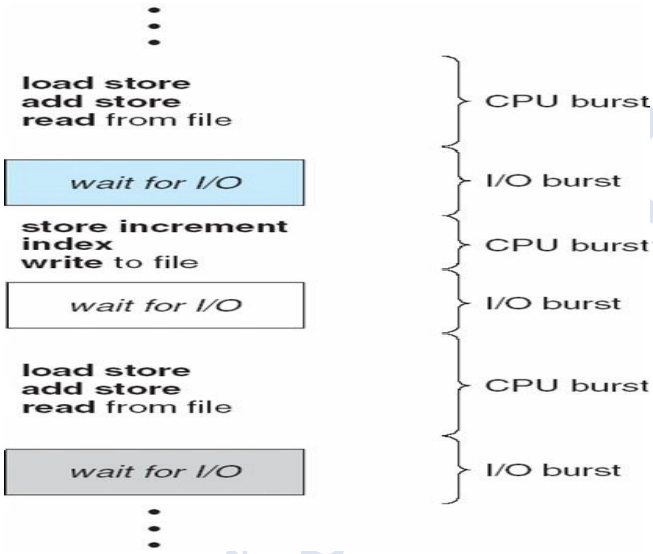
- CPU scheduling is the basis for multi-programmed operating systems.
- By switching the CPU among processes, the OS can make the computer work more productive.

- The main objective of multiprogramming is to have some process running at all times in order to maximize CPU utilization.
- Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

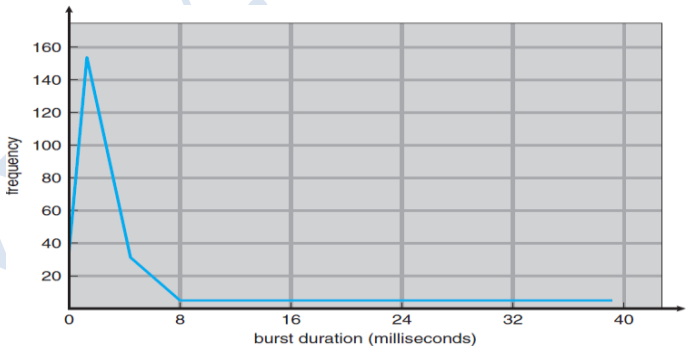
CPU–I/O Burst Cycle

- Process execution consists of a *cycle* of CPU execution and I/O wait i.e CPU–I/O Burst Cycle –**CPU burst d**istribution followed by I/O burst distribution.

The alternating sequence of CPU And I/O Bursts is as shown below.



The Histogram of CPU-burst Times is as shown below:



- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
The selection process is carried out by the **short-term scheduler**, or CPU scheduler.
The scheduler selects a single process from the memory among various processes that are ready to execute, and allocates the CPU to one of them.

CPU scheduling decisions may take place when a process.

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is **nonpreemptive.**
All other scheduling is **preemptive.**

Non-Preemptive: Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor).

Preemptive: Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters the ready list, the process on the processor should be removed and returned to the ready list.

Dispatcher

Dispatcher is a module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- switching context.
- switching to user mode.
- jumping to the proper location in the user program to restart that program.

Dispatch latency – The time taken for the dispatcher to stop one process and start another running.

Scheduling Criteria

Many criteria are there for comparing CPU scheduling algorithm. Some criteria required for determining the best algorithm are given below.

- **CPU utilization** – keep the CPU as busy as possible. The range is about 40% for lightly loaded system and about 90% for heavily loaded system.
- **Throughput** – The number of processes that complete their execution per time unit.
- **Turnaround time** – The interval from the time of submission of a process to the time of completion is the Turnaround time.

- **Waiting time** – amount of time a process has been waiting in the ready queue (or) the sum of the periods spent waiting in the ready queue.
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not the output (for time-sharing environment) is the response time.

Best Algorithm consider following:

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Formulas to calculate Turn-around time & waiting time is:

Waiting time = Finishing Time – (CPU Burst time + Arrival Time)

Turnaround time = Waiting Time + Burst Time

First-Come, First-Served (FCFS) Scheduling algorithm.

- This is the simplest CPU-scheduling algorithm.
- According to this algorithm, the process that requests the CPU first is allocated the CPU first.
- The implementation of FCFS is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- The average waiting time, under the FCFS policy is quite long.

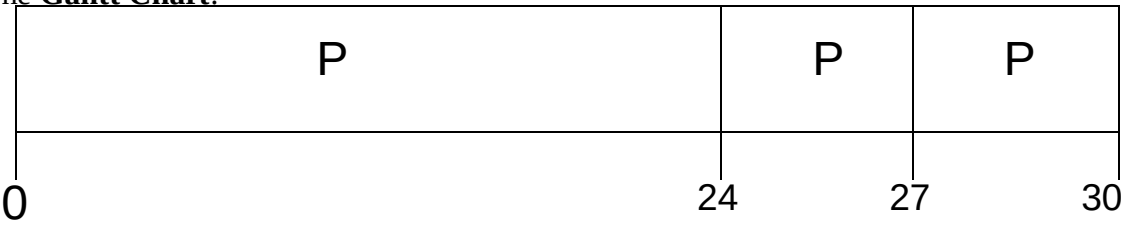
Example Problem

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

Process	Burst Time(ms)
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

The **Gantt Chart**:



Waiting time

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17 ms.

Turnaround Time = Waiting Time + Burst Time

- Turnaround Time for $P_1 = (0+24)=24$; $P_2 = (24+3)=27$; $P_3 = (27+3)=30$
- Average Turnaround Time = (24+27+30)/3 = 27 ms

Convoy effect

All other processes wait for the one big process to get off the CPU. This results in lower CPU and device utilization which could be overcome if shorter processes were allowed to go first.

Shortest-Job-First (SJF) Scheduling

- This algorithm associates with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. It is also called as **shortest next CPU burst**.
- If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.
- SJF is optimal – gives minimum average waiting time for a given set of processes (by moving the short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process, therefore the average waiting time decreases.
- The difficulty with the SJF algorithm is knowing the length of the next CPU request.

  Example - Nonpreemptive

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

- Waiting time for $P_1 = 3$; $P_2 = 16$; $P_3 = 9$; $P_4 = 0$
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- Turnaround Time P1=9; P2=24; P3=16; P4=3
- ATT = (9+24+16+3)/4 = 27 ms

- The SJF algorithm may be either "**preemptive**" or "**non-preemptive**". The choice arises when a new process arrives at the ready queue while a previous process is executing.
- The new process may have a shorter next CPU burst than what is left of the currently executing process.
- Then a preemptive SJF algorithm will preempt the currently executing process, where as a non-preemptive will allow the currently running process to finish its CPU burst.
- The "Preemptive" SJF scheduling is sometimes called as "**Shortest-remaining-time-first**" scheduling.

➢ Now we add the concepts of varying arrival times and preemption to the analysis

Example for preemptive

Process	***Arrival* Time**	**Burst Time**
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Waiting Time = Finishing Time-(Arrival time + Burst Time)

➢ WT of P_1 = *17-(0+8) = 9; P2=0; P3=15; P4=2*

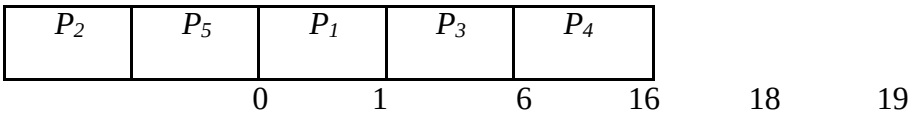
➢ Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- The SJF algorithm is a special case of the general priority-scheduling algorithm.
- A priority number (integer) 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.
- The CPU is allocated to the process with the highest priority (smallest integer ° highest priority)
- There are two types of priority scheduling.
  - Preemptive priority scheduling.
  - Non-preemptive priority scheduling.
- SJF is a priority scheduling where priority is the predicted next CPU burst time.

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

Priority scheduling Gantt Chart



Waiting Time for $P_1 = 6; P_2 = 0; P_3 = 16; P_4 = 18; P_5 = 1$
Average waiting time =(6+0+16+18+1)/5= 8.2 msec

Calculate Turnaround Time and Average Turnaround Time.

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantities to compute priority of a process. For example, time limits, memory requirements, the number of open files etc.
- External priorities are set by criteria that are external to the operating system, such as importance of the process and other political factors.

Preemptive priority scheduling: 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.
Non-premptive priority scheduling: A "non-preemptive priority" scheduling algorithm will simply put the new process at the head of the ready queue.

- Problem " **Starvation"** – low priority processes may never execute
- Solution "**Aging"** – as time progresses increase the priority of the process

Round Robin (RR)

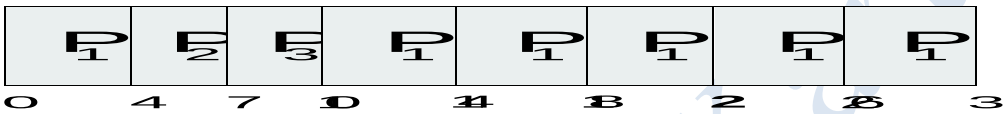
- Round robin scheduling is designed especially for time-sharing systems.
- It is similar to FCFS, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time called a *time quantum or time slice.*
- A time quantum is generally from 10-100 milliseconds.
- A ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- To implement RR scheduling, the ready queue is kept as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process.
- If the CPU burst time is less than the time quantum, the process itself will release the CPU voluntarily. Otherwise, if the CPU burst of the currently running process is longer than the time quantum a context switch will be executed and the process will be put at the tail of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

- The performance of RR completely depends on the size of the time quantum.
- If the time quantum is very large then FCFS policy is followed.
- If time quantum is small then this approach is called processor sharing.
- The time quantum must be large with respect to context switch, otherwise overhead is too high

Example

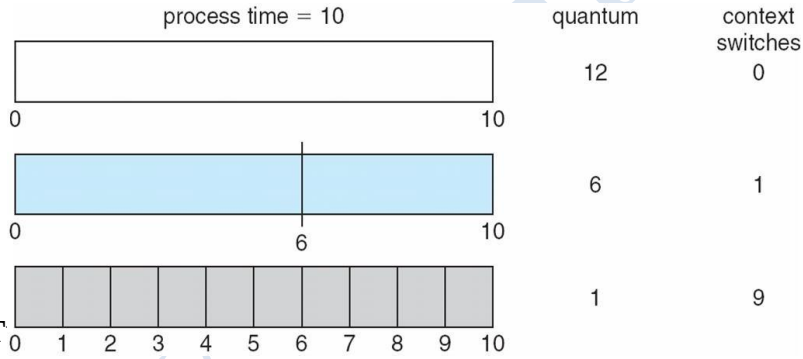
Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



Waiting Time for P_1 = (10-4) = 6 (or) 30-(0+24) = 6; P_2 = 4; P_3 = 7
Average Waiting Time = (6+4+7)/3=5.6 ms

Time Quantum and Context Switch Time



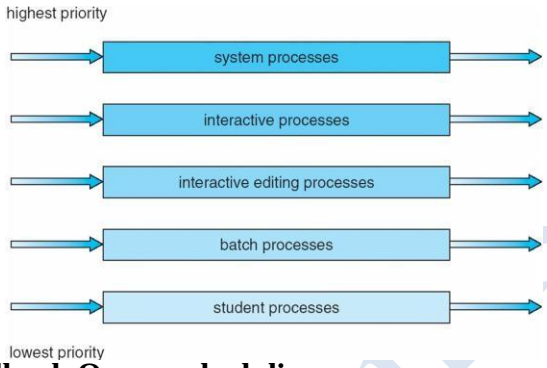
Multilevel Queue Scheduling

- This scheduling algorithm is created for situations in which processes are easily classified into different groups.
- There are two types of processes
  - Foreground or interactive processes.
  - Background or batch processes.
- This algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue based on properties like process size, process priority or process type.

- Each queue has its own algorithm for example the foreground queue might be scheduled by an RR algorithm and the background queue is scheduled by an FCFS algorithm.
- The foreground queue have absolute priority over background queue.
- Scheduling must be done between the queues
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Due to fixed priority scheduling there are possibility of starvation.
- The solution to this problem is : "Time slice" – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS.

No process in the batch queue could run unless the queues for system processes, interactive processes and interactive editing processes were empty.

Example of Multilevel Queue Scheduling



Multilevel Feedback Queue scheduling

- In multilevel queue scheduling the processes are fixed with respect to each queues and they cannot navigate between the queues.
- Multilevel Feedback Queue scheduling however allows a process to move between queues.
- A process can move between the various queues; aging can be implemented this way.
- If a process uses too much of CPU time, it will be moved to a low priority queue. If a process waits too long in a lower priority queue it may be moved to a higher priority queue. This form aging prevents starvation.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues.
  - scheduling algorithms for each queue.
  - method used to determine when to upgrade a process to a higher priority queue.
  - method used to determine when to demote a process to a lower priority queue.
  - The method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

Three queues:

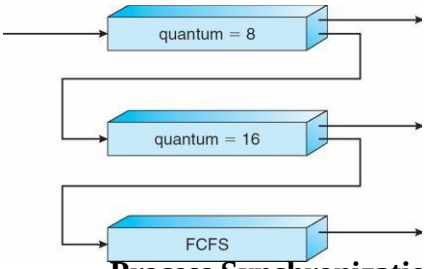
- $Q0$ – RR with time quantum 8 milliseconds
- $Q1$ – RR time quantum 16 milliseconds

- *Q*2 – FCFS

Scheduling

- A new job enters queue *Q0* which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue *Q*1.
- At *Q*1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue *Q*2.

Multilevel Feedback Queues



Process Synchronization

- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Race Condition

- Producer-consumer problem. It is described that how a bounded buffer could be used to enable processes to share memory
  - Bounded buffer problem. The solution allows at most BUFFERSIZE-1 items in the buffer at the same time.
  - An integer variable counter, initialized to 0. 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:

```
while (true)
 {
        /* produce an item in next Produced */
       while (counter == BUFFER_SIZE)
                    ; /* do nothing */
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
 }
```

- The code for the consumer process:

```
while (true)
 {
     while (counter == 0)
```

```
    ; /* do nothing */
   nextConsumed = buffer [out] ;
   out = (out + 1) % BUFFER_SIZE;
   counter--;
  /* consume the item in nextConsumed */
}
```

- 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.
- We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as follows:
    - *register*1 = counter
    - *register*1 = *register*1 + 1
    - counter = *register*1
- where *register*1 is one of the local CPU registers. Similarly, the statement "counter--" is implemented as follows:
    - *register*2 = counter
    - *register*2 = *register*2 − 1
    - counter = *register*2
- where again *register*2 is one of the local CPU registers. Even though *register*1 and *register*2 may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler.
- The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order One such interleaving is the following:
    - *T*0: *producer* execute *register*1 = counter {*register*1 = 5}
    - *T*1: *producer* execute *register*1 = *register*1 + 1 {*register*1 = 6}
    - *T*2: *consumer* execute *register*2 = counter {*register*2 = 5}
    - *T*3: *consumer* execute *register*2 = *register*2 − 1 {*register*2 = 4}
    - *T*4: *producer* execute counter = *register*1 {*counter* = 6}
    - *T*5: *consumer* execute counter = *register*2 {*counter* = 4}
- Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at *T*4 and *T*5, we would arrive at the incorrect state "counter == 6".
- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- We would arrive at incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A **race condition** is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing (race to access and modify data)

- **Definition** - To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter and this is referred as process synchronization.

The Critical-Section Problem

Definition: Each process has a segment of code, called a critical section (CS), in which the process may be changing common variables, updating a table, writing a file, and so on.

- The important feature of the system is that, when one process is executing in its CS, no other process is to be allowed to execute in its CS.
- That is, no two processes are executing in their CSs at the same time.
- Each process must request permission to enter its CS. The section of code implementing this request is the entry section.
- The CS may be followed by an exit section.
- The remaining code is the remainder section.

General structure of a typical process *Pi*

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

A solution to the CS problem must satisfy the following requirements:

1. **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **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, and this selection cannot be postponed indefinitely.
3. **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.

- Atomic operation. Atomic means either an operation happens in its entirely or NOT at all (it cannot be interrupted in the middle). Atomic operations are used to ensure that cooperating processes execute correctly.
- Machine instructions are atomic, high level instructions are not.

Peterson's Solution

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Does not require strict alternation.

- Peterson's solution is restricted to two processes that alternate execution between their CSs and remainder sections. The processes are numbered P0 and P1.
- Peterson's solution requires two data items to be shared between the two processes:

```
int turn;
boolean flag[2];
```

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1. For convenience, when presenting *Pi*, we use *Pj* to denote the other process; that is, j equals 1 − i.

Algorithm

The structure of process *Pi* in Peterson's solution.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- 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; that is, j equals 1 − i. 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. With an explanation of these data structures complete.
- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

- To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.
- Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.
- These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time.

Mutex Locks

- Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the mutex lock.
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock.

Solution to the critical-section problem using mutex locks.

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

- 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;;
}
```

- The definition of release() is as follows:

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

- Calls to either acquire() or release() must be performed atomically. Thus, mutex locks are often implemented using one of the hardware mechanisms.

www.BrainKart.com

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().
- In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:
  wait() and
  signal().
- The wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "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++;
}
```

Semaphore Usage

- 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.
- Binary semaphores behave similarly to mutex locks.
- On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- 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 (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).

www.BrainKart.com

- 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
- We can also use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In **process P1**, we insert the statements

```
S1;
signal(synch);
```

  In **process P2**, we insert the statements

```
wait(synch);
S2;
```

- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- 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, and the state of the process is switched to the waiting state.
- 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 from the waiting state to the ready state.
- The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
- 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() operation suspends the process that invokes it.
  - The wakeup(P) operation resumes the execution of a blocked process P.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- When such a state is reached, these processes are said to be deadlocked
- To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
. .	..
. .	..
. .	..
signal(S);	signal(Q);
signal(Q);	signal(S);

- Suppose that P0 executes wait(S) and then 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).

- Since these signal() operations cannot be executed, P0 and P1 are deadlocked.
- We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Another problem related to deadlocks is **indefinite blocking or starvation**, a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

- A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes.
- Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.
- The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.
- This problem is known as priority inversion. It occurs only in systems with more than two priorities, so one solution is to have only two priorities.
- Typically these systems solve the problem by implementing a **priority-inheritance protocol.** 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 in question.
- When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process

Classic Problems of Synchronization

1. Bounded Buffer Problem
2. Reader Writer Problem
3. Dining Philosopher's Problem

The Bounded-Buffer Problem

- We 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.

The producer and consumer processes share the following data structures:

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

The structure of the 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);
```

The structure of the 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);
```

- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

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 (readers), whereas others may want to update (that is, to read and write) the database (writers).
- If two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, there could be some synchronization issues.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the readers-writers problem.

www.BrainKart.com

- The readers-writers problem has several variations, all involving priorities.
    - The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
    - The **second readers-writers problem** requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation.
    - In the first case, writers may starve.
    - In the second case, readers may starve.

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

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

- The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0.
- The semaphore rw mutex is common to both reader and writer processes.
- 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 rw mutex functions as a mutual exclusion semaphore for the writers.

The structure of a writer process.

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

The structure of a reader process.

```
do {
        wait(mutex);
        readcount++;
        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);
```

The Dining-Philosophers Problem

- The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices.
- 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 thinks, she does not interact with her colleagues.

- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- 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 her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.

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.
- Thus, the shared data are

  semaphore chopstick[5] ;

where all the elements of chopstick are initialized to 1.

The structure of philosopher *i*.

```
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);
```

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

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.

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, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

- The semaphore solution to the CS problem.
    - All processes share a semaphore variable mutex, which is initialized to 1.
    - Each process must execute wait(mutex) before entering the CS and signal(mutex)afterward.
    - If this sequence is not observed,two processes may be in their CSs simultaneously.

- 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);
```

  - In this situation, several processes maybe executing in their CSs simultaneously, violating the mutual-exclusion requirement.
  - This error may be discovered only if several processes are simultaneously active in their CSs. Note that this situation may not always be reproducible.
- 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.
- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Definition: Monitor is a high-level language construct with a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.

- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.

Monitor Usage

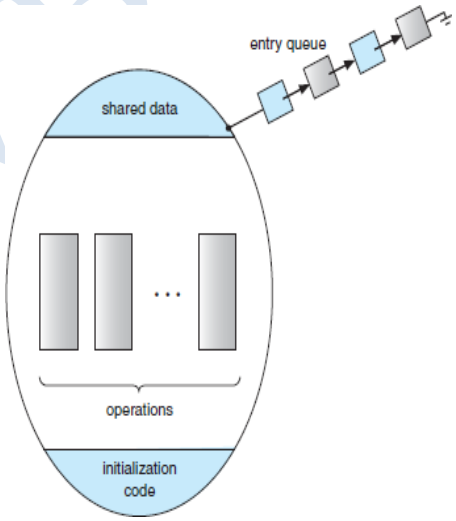
- A type, or abstract data type, encapsulates private data with public methods to operate on that data. A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

Syntax of a monitor.

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

- The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure 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 procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Schematic view of a Monitor



- The monitor construct is not sufficiently powerful for modeling some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). The operation

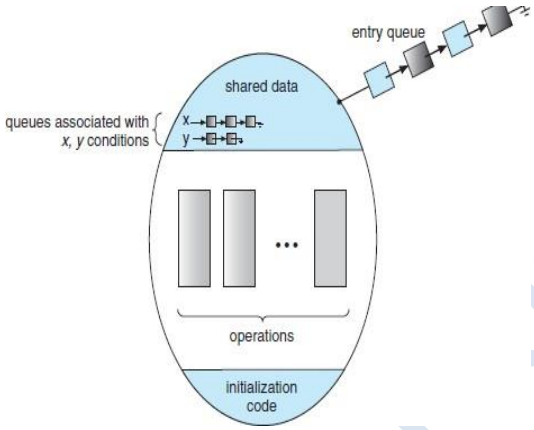
x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

The x.signal() operation resumes exactly one suspended process.

Monitor with condition variables.



Dining-Philosophers Solution Using Monitors

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem.

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher.

For this purpose, the following data structure is used:

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).

A monitor solution to the dining-philosopher problem.

```
monitor DiningPhilosophers
{
        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;
        }
}
```

We also need to declare

```
condition self[5];
```

This 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 act 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);
```

Implementing a Monitor Using Semaphores

- Now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore mutex (initialized to 1) is provided.
- A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0.

- An integer variable
- next count is also provided to count the number of processes suspended on next

Thus, each external function F is replaced by

```
wait(mutex);
...
body of F
...
if (next count > 0)
        signal(next);
else
        signal(mutex);
```

We can now describe how condition variables are implemented as well. For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0.

The operation x.wait() can now be implemented as

```
x count++;
if (next count > 0)
        signal(next);
else
        signal(mutex);
wait(x sem);
x count--;
```

The operation x.signal() can be implemented as

```
if (x count > 0)
{
        next count++;
        signal(x sem);
        wait(next);
        next count--;
}
```

Resuming Processes within a Monitor

- If several processes are suspended on condition x, and an x.signal() operation

is executed by some process, then how do we determine which of the suspended processes should be resumed next?

- One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate.
- For this purpose, the conditional-wait construct can be used. This construct has the form

```
x.wait(c);
```

- where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended.

- When x.signal() is executed, the process with the smallest priority number is resumed next.
- A process that needs to access the resource must observe the following sequence:

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type ResourceAllocator.

A monitor to allocate a single resource.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time)
    {
        if (busy)
        x.wait(time);
        busy = true;
    }
    void release()
    {
        busy = false;
        x.signal();
    }
    initialization code()
     {
        busy = false;
    }
}
```

Dead lock

Definition:

A process request resources, if the resources are not available at that time, the process enters in to a wait state. It may happen that waiting processes will never again change the state, because the resources they have requested are held by other waiting processes. *This situation is called as* ***dead lock.***

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.
- CPUcycles, files,and I/O devices (such as printers and DVD drives) are examples of resource types.

- If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.
- If a process requests an instance of a resource type, the allocation of *any* instance of the type should satisfy the request.
- A process must request a resource before using it and must release the resource after using it.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

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

2. Use. The process can operate on the resource

3. Release. The process releases the resource.

Deadlock Characterizations:-

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

Necessary Conditions for Deadlock:-

A dead lock situation can arise if the following four conditions hold simultaneously in a system.

1) MUTUAL EXCLUSION:- At least one resource must be held in a on-sharable mode. i.e only one process can hold this resource at a time . other requesting processes should wait till it is released.

2) HOLD & WAIT:- there must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3) NO PREEMPTION:- Resources cannot be preempted, that is a resource can be released voluntarily by the process holding it, after that process has completed its task.

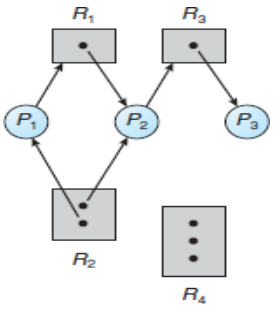
4) CIRCULAR WAIT:- There must exist a set {p0,p1,p1….pn} of waiting processes such that p0 is waiting for a resource that is held by the p1, p1 is waiting for the resource that is held by the p2…. And so on. pn is waiting for a resource that is held by the p0.

Resource-Allocation Graph

- Deadlocks can be described more precisely 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 = {P*1, *P*2, ...,*Pn}*, the set consisting of all the active processes in the system, and *R = {R*1, *R*2, ..., *Rm}*, the set consisting of all resource types in the system.
- Adirected edge fromprocess *Pi* to resource type *Rj* is denoted by *Pi* → *Rj* ; it signifies that process *Pi* has requested an instance of resource type *Rj* andis currently waiting for that resource.
- A directed edge from resource type *Rj*to process *Pi* is denoted by *Rj* → *Pi* ; it signifies that an instance of resourcetype *Rj* has been allocated to process *Pi* .

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

Resource Allocation Graph



The resource-allocation graph depicts the following situation.

• The sets *P, R,* and *E*:

- *P = {P*1, *P*2, *P*3*}*
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

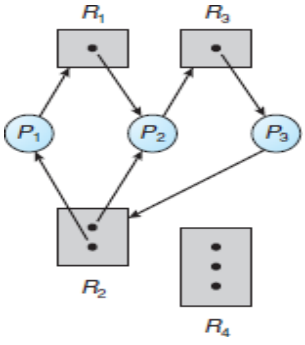
Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

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.*

Resource-allocation graph with a deadlock.

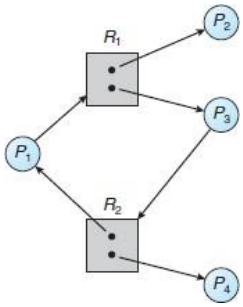


- Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.
- we also have a cycle: P1 → R1 → P3 → R2 → P1
- if the graph contains no cycles, then no process in the system is deadlocked.

www.BrainKart.com

- If the graph does contain a cycle, then a deadlock may exist.

Resource-allocation graph with a cycle but no deadlock.



Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows.

- **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
- **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.

Deadlock Prevention

- For a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

Mutual Exclusion

- The mutual exclusion condition must hold. That is, at least one resource must be non sharable.
- Sharable resources 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.
- A process never needs to wait for a sharable resource.
- We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non sharable.

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol 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, however, it must release all the resources that it is currently allocated.
- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- Both these protocols have two main disadvantages.
    - First, resource utilization may be low, since resources may be allocated but unused for a long period.
    - Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, we can use the following protocol.
    - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held 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.
- 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.
- Each process can request resources only in an increasing order of enumeration.
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- Each request requires that in making this decision the system consider
    - the resources currently available,

  - the resources currently allocated to each process,
  - the future requests and releases of each process.
- The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- **The resource-allocation state** is defined by the number of available and allocated resources and the maximum demands of the processes.

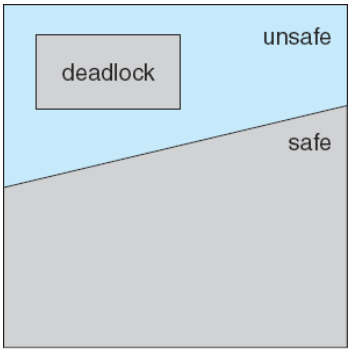
Safe State

- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a safe sequence.

Safe sequence

  - A sequence of processes <p1.p2…pn> is a safe sequence for the current allocation state if, for each Pi the resource requests that Pj can still make can be satisfied by the currently available resources plus the resources held by all Pj , with j<i.
  - In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished.
  - When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
  - When terminates, P i+1 can obtain its needed resources, and so on.
  - If no such sequence exists, then the system state is said to be **unsafe**.
- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks

Safe, unsafe, and deadlocked state spaces



- An unsafe state may lead to a deadlock.
  - As long as the state is safe, the OS can avoid unsafe (and deadlocked) states.

- In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

Example

*P*0, *P*1, and *P*2. Process *P*0 requires ten tape drives, process *P*1 may need as any 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

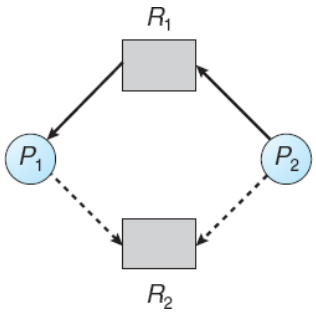
- At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Process *P*1 can immediately be allocated all its tape drives and then return them then process *P*0 can get all its tape drives and return them and finally process *P*2 can get all its tape drives and return them
- A system can go from a safe state to an unsafe state.
- Suppose that, at time *t*1, process *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process *P*1 can be allocated all its tape drives.
- When it returns them, the system will have only four available tape drives. Since process *P*0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives.
- If it does so, it will have to wait, because they are unavailable. Similarly, process *P*2 may request six additional tape drives and have to wait, resulting in a deadlock.

Resource-Allocation-Graph

- A new type of edge, called a **claim edge is** introduced.
- A **claim edge** *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi* , the assignment edge *Rj* → *Pi* is reconverted to a claim edge *Pi* → *Rj* .
- That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.

Resource-allocation graph for deadlock avoidance.

www.BrainKart.com



- 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. In that case, process *Pi* will have to wait for its requests to be satisfied.

Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- 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.

Data structures

- **Available**.Avector of length*m*indicates the number of available resources of each type. If ***Available***[*j*] equals *k*, then *k* instances of resource type *Rj* are available.
- **Max**. An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k*, then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj* .
- **Need**. An *n* × *m* matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task. Note that ***Need***[*i*][*j*] equals***Max***[*i*][*j*] − ***Allocation***[*i*][*j*].

Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state.

This algorithm can be described as follows:

1. 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.

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

   If no such *i* exists, go to step 4.
3. ***Work*** =***Work*** + ***Allocation****i*
   Finish[*i*] = ***true***
   Go to step 2.
4. 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* [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj* . When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Request****i* ≤***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

4. 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

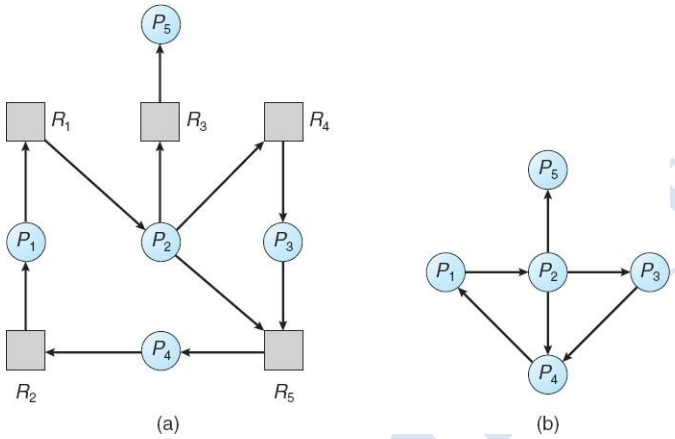
- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm then a deadlock situation may occur.
- In this environment, the system must provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - An algorithm to recover from the deadlock.

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.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

- An edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs.
- An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq* .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
-

(a) Resource-allocation graph. (b) Corresponding wait-for graph



- 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.
- An algorithm to detect a cycle in a graph requires an order of *n*2 operations, where *n* is the number of vertices in the graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

Data Structures

- **Available**.Avector of length*m*indicates the number of available resources of each type.
- **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An *n* × *m* matrix indicates the current request of each process. If ***Request***[*i*][*j*] equals *k*, then process *Pi* is requesting *k* more instances of resource type *Rj* .

Algorithm:

1. 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.***
2. Find an index *i* such that both
   a. ***Finish***[*i*] == ***false***

b. ***Request****i* ≤***Work***
If no such *i* exists, go to step 4.

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

4. 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.

Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How ***often*** is a deadlock likely to occur?
2. How ***many*** processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available.

1. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
2. Another possibility is to let the system **recover** from the deadlock automatically.

There are two options for breaking a deadlock.

1. One is simply to abort one or more processes to break the circular wait.
2. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

• **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

• **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Many factors may affect which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used(for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Issues

1. **Selecting a victim**.
   - Which resources and which processes are to be preempted?
   - As in process termination, we must determine the order of preemption to minimize cost.
   - Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback**.
   - If we preempt a resource from a process, what should be done with that process?
   - We must roll back the process to some safe state and restart it from that state.
   - Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
3. **Starvation**.
   - How do we ensure that starvation will not occur?
   - How can we guarantee that resources will not always be preempted from the same process?

www.BrainKart.com