MODULE -2

MULTI-THREADED PROGRAMMING

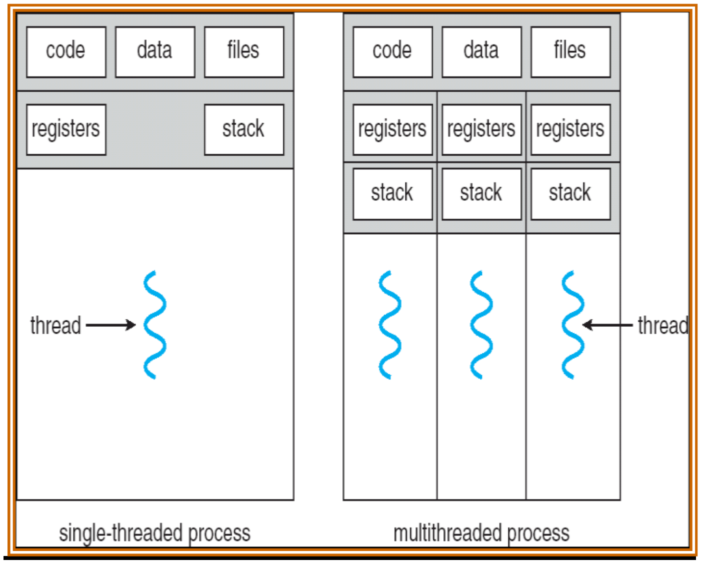
Contents:

Process Synchronization:

2.1 Overview

➢ Threads

- A thread is a basic unit of CPU utilization.
- **It comprises a thread ID, a program counter, a register set, and a stack**. It shares its **code section, data section**, and other operating-system resources, such as **open files and signals** with **other threads** belonging to the same process.
- 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.
- The below **figure** illustrates the difference between a **traditional single threaded** process and a **multithreaded** process.



- **Motivation**

  - Many software packages that run on modern desktop PCs are multithreaded.
  - An application is implemented as a separate process with several threads of control. Eg: A Web browser might have one thread to display images or text while another thread retrieves data from the network.
  - As **process creation** takes **more time** than **thread creation** it is more efficient to use process that contains multiple threads. So, that the amount of time that a client have to wait for its request to be serviced from the web server will be less.
  - Threads also play an important role in **remote procedure call.**

- **Benefits**

  The benefits of multithreaded programming

  1. **Responsiveness:** Multithreading allows program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

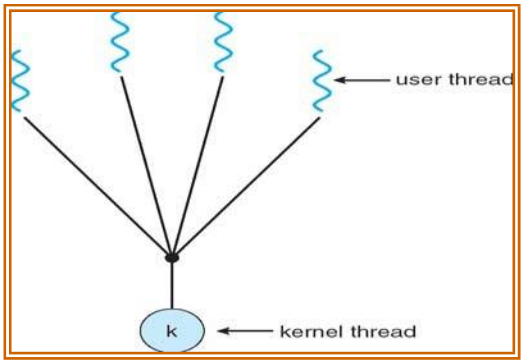
2. **Resource Sharing:** Threads share the memory and resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy:** Because of resource sharing context switching and thread creation are fast when working with threads.
4. **Utilization of multiprocessor architectures:** Threads can run in parallel on different processors. Multithreading on multi-CPU machine increases concurrency.

Multithreading Models

- Support for threads may be provided either at user level for **user threads** or by the kernel for **kernel threads**.
- User threads are created and managed by Thread libraries without the kernel support.
- Kernel threads are directly managed by OS.
- There **must be a relationship** between user threads and kernel threads. **Three** common ways of establishing this relationship are **:**
  1. **Many-to-One**
  2. **One-to-One**
  3. **Many-to-Many**

- **Many-to-One**

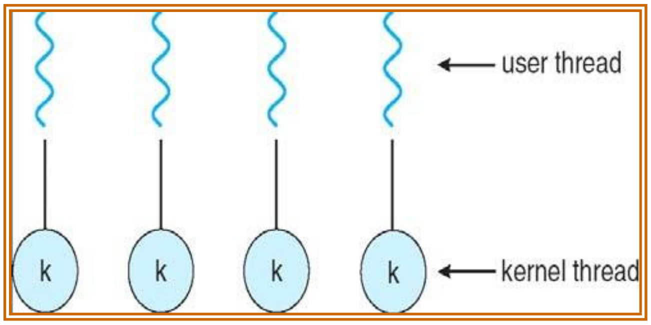
  - Many user-level threads are mapped to single kernel thread as shown in below **figure**.
  - This model is efficient as the thread management is done by the thread library in user space, but the entire process will block if a thread makes a blocking system call.
  - As only one thread can access the kernel thread at a time, multiple threads are unable to run in parallel on multiprocessors.
  - **Examples:** Solaris Green Threads, GNU Portable Threads



- **One-to-One**

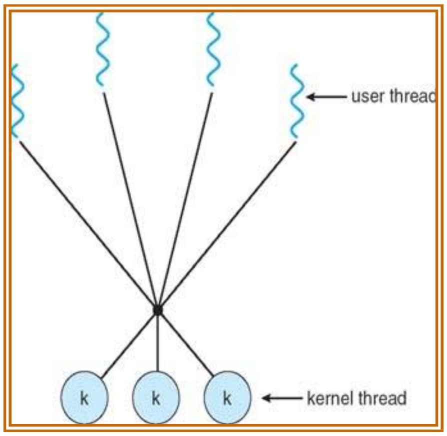
  - Each user-level thread maps to kernel thread as shown in **figure 4.3**
  - It provides more concurrency than Many-to-One model by allowing thread to run when a thread makes a blocking system call.
  - It allows multiple threads to run in parallel on multiprocessors.

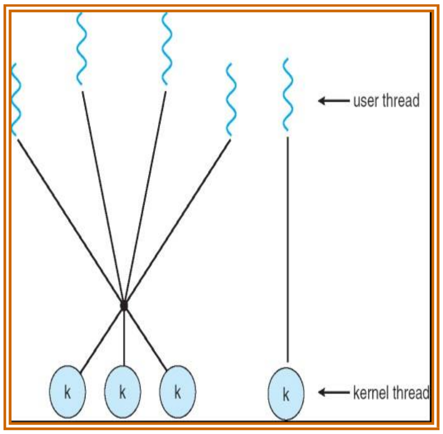
- The only **drawback** is, creating a user thread requires creating the corresponding kernel thread and it burdens performance of an application.
- **Examples:** Windows NT/XP/2000, Linux



- **Many-to-Many Model**

  - **One-to-One** model **restricts** creating more user threads and **Many-to-One** model allows creating more user threads but kernel can schedule only one thread at a time. These drawbacks can be **overcome** by Many-to-Many model as shown in below **figure 4.4. (Left side)**
  - Many-to-Many model allows many user level threads to be mapped to many kernel threads.
  - It allows the operating system to create a sufficient number of kernel threads.
  - When thread performs a blocking system call, the kernel can schedule another thread for execution.
  - It allows user-level thread to be bound to a kernel thread and this is referred as **two- level model** as shown in below **figure 4.5**. **(Right side)**
  - **Examples:** IRIX, HP-UX, Solaris OS.





➢ Thread Libraries

- A thread library provides the programmer an **API** for creating and managing threads.
- There are **two primary ways** of implementing a thread library.
- The first approach is to provide a **library entirely in user space** with **no kernel support.** All code and data structures for the library exist in user space.

- The second approach is to implement a **kernel-level library supported directly by the operating system.**
- Three primary **thread libraries** are
  - **POSIX Pthreads:** extension of posix standard, they may be provided as either a user bor kernel library.
  - **Win32 threads:** is a kernel level library available on windows systems.
  - **Java threads:** API allows creation and management directly in Java programs. However, on windows java threads are implemented using win32 and on UNIX and Linux using Pthreads

- **Pthreads**

  - Pthreads, the threads extension of the POSIX standard, may be provided as either a user or kernel-level library.
  - Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
  - This is a specification for thread behavior, not an implementation.
  - Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX.
  - **Shareware** implementations are available in the public domain for the various Windows operating systems as well.
  -

Multithreaded C program using the Pthreads API

```
#include <pthread.h>
#include <stdio.h>
int sum;                          /* this data is shared by the thread(s) */
void *runner(void *param);        /* the thread */
int main(int argc, char *argv[])
{
pthread_t  tid;                   /* the thread identifier */
pthread_attr_t attr;              /* set of thread attributes */

if (argc != 2)
{
fprintf(stderr,"usage: a.out <integer value>\n");
return -1;
}
if (atoi(argv[1]) < 0)
{
fprintf(stderr,"%d must be>= 0\n",atoi(argv[1]));
return -1;
}
pthread_attr_init(&attr);                    /* get the default attributes */
pthread_create(&tid,&attr,runner,argv[1]);   /* create the thread */
pthread_join(tid,NULL);                      /* wait for the thread to exit */
printf("sum = %d\n",sum);
}

void *runner(void *param)                /* The thread will begin control in this function */
{
int i, upper= atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
    sum += i;
pthread_exit(0) ;
}
```

- **Win32 Threads**

  - The Win32 thread library is a kernel-level library available on Windows systems.
  - The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. We must include the windows.h header file when using the Win32 API.
  - Threads are created in the Win32 API using the CreateThread() function and a set of attributes for the thread is passed to this function.
  - These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.
  - The parent thread waits for the child thread using the WaitForSingleObject() function, which causes the creating thread to block until the summation thread has exited.

Multithreaded C program using the Win32 API

```
#include <Windows.h>
#include <stdio.h>
DWORD Sum;                                    /* data is shared by the thread(s) */
                                              /* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
DWORD Upper = *(DWORD*)Param;
for (DWORD i = 0; i <= Upper; i++)
    Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
DWORD ThreadId;
HANDLE ThreadHandle;
int Param;

if (argc != 2)                     /* perform some basic error checking */
{
    fprintf(stderr," An integer parameter is required\n");
```

```
     return -1;
}
Param = atoi(argv[1]);
if (Param < 0)
{
    fprintf(stderr,"An integer>= 0 is required\n");
    return -1;
}
                                                                      /*create the thread*/
ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);

                                                               // NULL: default security attributes
                                                               // 0: default stack size
                                                               // Summation: thread function
                                                               // &Param: parameter to thread function
                                                               // 0: default creation flags
                                                               //ThreadId: returns the thread identifier

if (ThreadHandle != NULL)
 {
  WaitForSingleObject(ThreadHandle,INFINITE);   // now wait for the thread to finish
  CloseHandle(ThreadHandle);                    // close the thread handle
  printf("surn = %d\n" ,Sum);
}
}
```

- **Java Threads**
  - The Java thread API allows thread creation and management directly in Java programs.
  - Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads.
  - All Java programs comprise at least a single thread of control and even a simple Java program consisting of only a **main()** method runs as a single thread in the JVM.
  - There are **two techniques** for creating threads in a Java program. One approach is to create a new class that is derived from the **Thread class** and to override its **run()** method. An alternative and more commonly used technique is to define a class that implements the **Runnable interface**. The **Runnable interface** is defined as follows:

```
public interface Runnable
{
        public abstract void run () ;
}
```

  - When a class implements Runnable, it must define a **run()** method. The code implementing the run() method runs as a separate thread.

- Creating a Thread object does not specifically create the new thread but it is the **start()** method that actually creates the new thread. Calling the **start()** method for the new object does two things:
  - It allocates memory and initializes a new thread in the JVM.
  - It calls the **run()** method, making the thread eligible to be run by the JVM.
- As Java is a **pure object-oriented** language, it has **no notion of global data.** If two or more threads have to share data means then the sharing occurs by passing reference to the shared object to the appropriate threads.
- This shared object is referenced through the appropriate **getSum() and setSum()** methods.
- As the Integer class is **immutable**, that is, once its value is set it cannot change, a new **sum** class is designed.
- The parent threads in Java uses join() method to wait for the child threads to finish before proceeding.

Java program for the summation of a non-negative integer.

```
class Sum
{
private int sum;
        public int getSum()
            {
               return sum;
             }

       public void setSum(int sum)
           {
              this.sum = sum;
            }
}


class Summation implements Runnable
{
private int upper;
private Sum  sumValue;

  public Summation(int upper, Sum sumValue)
      {
         this.upper = upper;
         this.sumValue = sumValue;
      }

 public void run()
     {
         int sum = 0;
         for (int i = 0; i <= upper; i++)
```

```
          sum += i;
          sumValue.setSum(sum);
      }
}

public class Driver
 {
       public static void main(String[]  args)
        {
            if (args.length > 0)
               {
                  if (Integer.parseint(args[O]) < 0)
                           System.err.println(args[O] + "must be>= 0.");
                  else
                      {
                         Sum sumObject = new Sum();      //create the object to be shared
                         int upper= Integer.parseint(args[O]);
                         Thread thrd =new Thread(new Summation(upper, sumObject));
                         thrd.start();
                         try
                        {
                          thrd. join () ;
                          System.out.println("The sum of "+upper+" is "+sumObject.getSum());
                        }
                         catch (InterruptedException ie) { }
                      }
              }
              else
              System.err.println("Usage: Summation <integer value>");
 }
```

Threading Issues

- **The fork() and exec() System Calls**

  - ✓ The semantics of the **fork() and exec()** system calls change in a multithreaded program. Some UNIX systems have chosen to have **two versions** of **fork(), one** that duplicates all threads and **another** that duplicates only the thread that invoked the fork() system call.
  - ✓ If a thread invokes the **exec()** system call, the program specified in the parameter to **exec()** will replace the entire process including all threads.
  - ✓ Which of the two versions of **fork()** to use depends on the application. If **exec()** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.

- **Thread Cancellation**

  - ✓ Thread cancellation is the task of terminating a thread before it has completed. **For example**, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.
  - ✓ A thread that is to be cancelled is often referred to as the **target thread**. Cancellation of a target thread may occur in **two different** scenarios:
    - o **Asynchronous cancellation:** One thread immediately terminates the target thread.
    - o **Deferred cancellation:** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

  - ✓ The **difficulty** with asynchronous cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a cancelled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.
  - ✓ With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. This allows a thread to check whether it should be cancelled at a point when it can be cancelled safely. Pthreads refers to such points as **cancellation points.**

- **Signal Handling**

  - ✓ A signal is used in UNIX systems to notify a process that a particular event has occurred.
  - ✓ A signal may be received either synchronously or asynchronously, depending on the **source of** and **the reason** for the event being signaled.
  - ✓ All signals, whether synchronous or asynchronous, follow the same pattern:
    - o A signal is generated by the occurrence of a particular event.
    - o A generated signal is delivered to a process.
    - o Once delivered, the signal must be handled.
  - ✓ **Examples** of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal.
  - ✓ When a signal is generated by an event external to a running process, that process receives the signal **asynchronously. Examples** of such signals include terminating a process with specific keystrokes (such as <control><C>) and having **a timer expires**. An asynchronous signal is sent to another process.

  - ✓ Every signal may be handled by one of **two possible handlers**,
    - o A default signal handler
    - o A user-defined signal handler
  - ✓ Every signal has a **default signal handler** that is run by the kernel when handling that signal.
  - ✓ This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.

- ✓ Signals may be handled in different ways. Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program.
- ✓ Delivering signals is more complicated in multithreaded programs. The following **options exist** to deliver a signal:
    - o Deliver the signal to the thread to which the signal applies.
    - o Deliver the signal to every thread in the process.
    - o Deliver the signal to certain threads in the process.
    - o Assign a specific thread to receive all signals for the process.

- **Thread Pools**

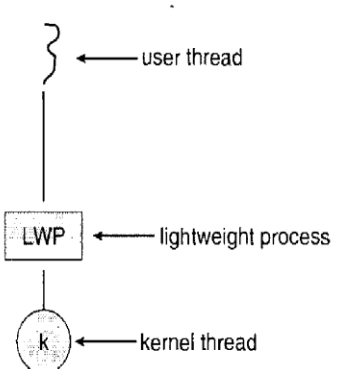
    - ✓ The idea behind a thread pool is to **create a number of threads** at process startup and place them into a pool, where they sit and wait for work.
    - ✓ When a server receives a request, it awakens a thread from this pool and passes the request to it to service.
    - ✓ Once the thread completes its service, it returns to the pool and waits for more work.
    - ✓ If the pool contains no available thread, the server waits until one becomes free.
    - ✓ The **benefits** of Thread pools are,
        - o Servicing a request with an existing thread is usually faster than waiting to create a thread.
        - o A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
    - ✓ The **number of threads** in the pool can be set based on **factors** such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.

- **Thread-Specific Data**

    - ✓ Threads belonging to a process share the data of the process. This sharing of data provides one of the benefits of multithreaded programming. But, in some circumstances, each thread might need its own copy of certain data. Such data is called as **thread-specific data**. **For example**, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier.
    - ✓ Most thread libraries including Win32 and Pthreads provide support for thread-specific data.

- **Scheduler Activations**

    - ✓ Many systems implementing either the many-to-many or two-level model place an **intermediate data structure** between the user and kernel threads. This data structure is known as a lightweight process, or LWP as shown in the following figure 4.6



- ✓ An application may require any number of LWPs to run efficiently.
- ✓ In a CPU-bound application running on a single processor only one thread can run at once, so one LWP is sufficient. An application that is I/O- intensive may require multiple LWPs to execute.
- ✓ One scheme fo communication between the user-thread library and the kernel is known as

  scheduler activation. It **works** as follows: The kernel provides an application with a set of **virtual processors (LWPs)**, and the application can schedule user threads onto an available virtual processor. The kernel must inform an application about certain events. This procedure is known as an **upcall.**
- ✓ Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.
- ✓ One event that triggers an upcall occurs when an application thread is about to block. In this

  situation, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the

  application.
- ✓ The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and gives up the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
- ✓ When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run.
- ✓ The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.

- ✓ After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

Differences

- ✓ The differences between process and thread are,

	Process	Thread
1.	It is called heavyweight process.	It is called lightweight process.
2.	Process switching needs interface with OS.	Thread switching does not need interface with OS.
3.	Multiple processes use more resources than multiple threads.	Multiple threaded processes use fewer resources than multiple processes.
4.	In multiple process implementations each process executes same code but has its own memory and file resources.	All threads can share same set of open files.
5.	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.
6.	In multiple processes each process operates independently of others.	One thread can read, write or even completely wipeout another threads stack.

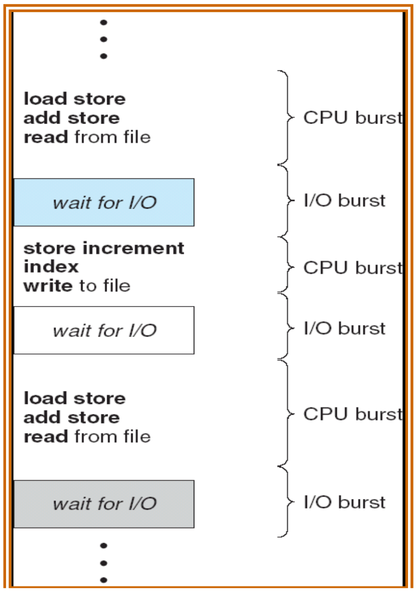
PROCESS SHEDULING

Basic Concepts

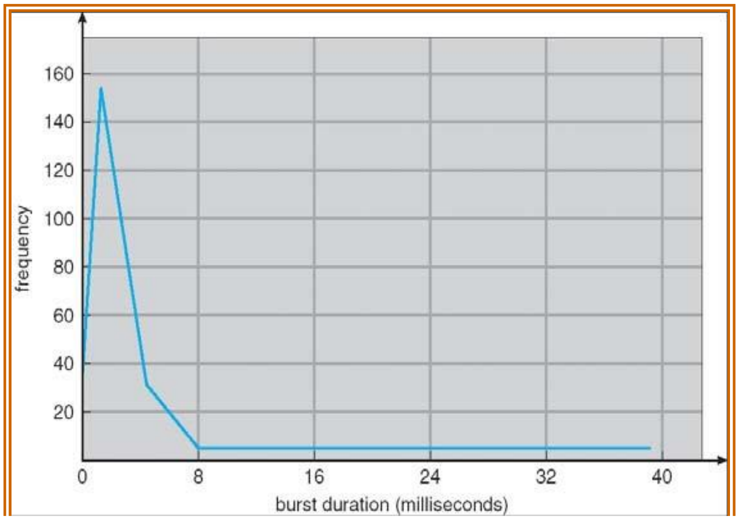
- ✓ In a single-processor system, only one process can run at a time and others must wait until the CPU is free and can be rescheduled.
- ✓ The objective of **multiprogramming** is to have some process running at all times, to **maximize CPU utilization**.
- ✓ With multiprogramming, several processes 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.
- ✓ The CPU is one of the primary computer resources. Thus, its scheduling is central to
- ✓ operating-system design.

- **CPU–I/O Burst Cycle**

  - ✓ Process execution consists of a **cycle** of CPU execution and I/O wait
  - ✓ Process execution starts with **CPU burst** and this is followed by **I/O burst** as shown in below **figure 5.1.**
  - ✓ The final CPU burst ends with a system request to terminate execution.



- ✓ The duration of CPU bursts vary from process to process and from computer to computer.
- ✓ The **frequency curve** is as shown below **figure 5.2.**



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

- ✓ A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. But all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The **records** in the queues are **Process Control Blocks (PCBs)** of the processes.

- **Preemptive scheduling**
  - ✓ **CPU-scheduling decisions** may take place under the following **four circumstances**.
    1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
    2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
    3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
    4. When a process terminates

  - ✓ When scheduling takes place only under circumstances **1 and 4**, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it is **preemptive**.
  - ✓ Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. A scheduling algorithm is preemptive if, once a process has been given the CPU and it can be taken away.

- **Dispatcher**

  - ✓ Another component involved in the CPU-scheduling function is the dispatcher.
  - ✓ The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
  - ✓ This function involves the following:
    - o Switching context
    - o Switching to user mode
    - o Jumping to the proper location in the user program to restart that program

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

➢ Scheduling Criteria

- ✓ Many criteria have been suggested for comparing CPU scheduling algorithms. The **criteria** include the following:

  - o **CPU utilization:** CPU must be kept as busy as possible. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

- **Throughput**: If the CPU is busy executing processes, then work is being done. One **measure of work** is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time**: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** The measure of the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

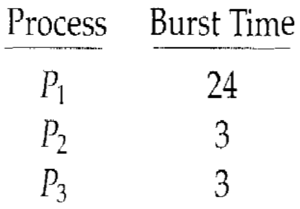
Scheduling Algorithms

CPU Scheduling deals with the problem of **deciding** which of the processes in the ready queue is to be allocated the CPU. Following are some **scheduling algorithms,**

- FCFS Scheduling.
- Round Robin Scheduling.
- SJF Scheduling.
- Priority Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback Queue Scheduling.

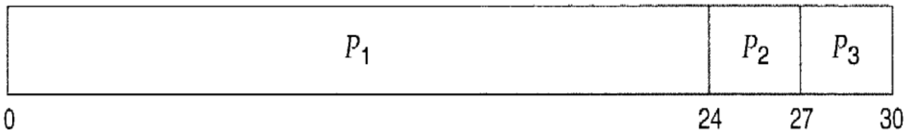
First-Come-First-Served (FCFS) Scheduling

- ✓ The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm.
- ✓ With this scheme, the process that requests the CPU first is allocated the CPU first.
- ✓ The implementation of the FCFS policy is easily managed with a FIFO queue.
- ✓ When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- ✓ When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- ✓ The average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:



Process	Burst Time
P_1	24
P_2	3
P_3	3

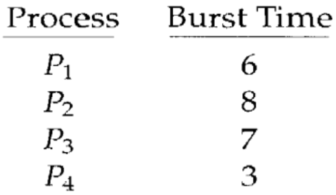
- ✓ If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart:



- ✓ The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
- ✓ The FCFS scheduling algorithm is **nonpreemptive**. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

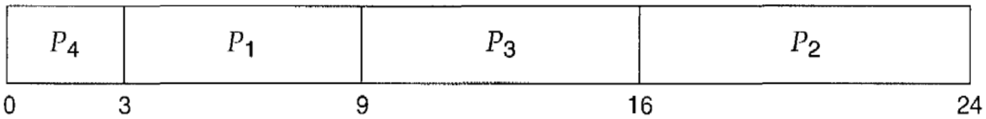
- **Shortest-Job-First Scheduling**

  - ✓ This algorithm associates with each process the length of the process's next CPU burst.
  - ✓ When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
  - ✓ If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
  - ✓ As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:



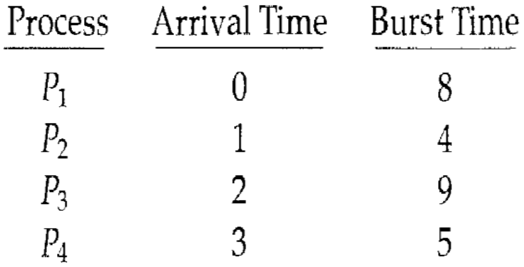
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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



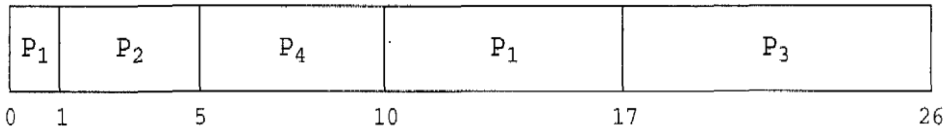
  - ✓ The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.
  - ✓ The SJF scheduling algorithm is **optimal**, it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one, decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

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



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

- ✓ If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:

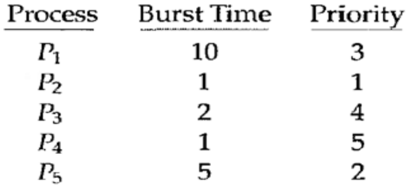


- ✓ Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1.
- ✓ The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 -1) + (1-1) + (17 -2) + (5- 3))/4 = 26/4 = 6.5 milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

- **Priority Scheduling**

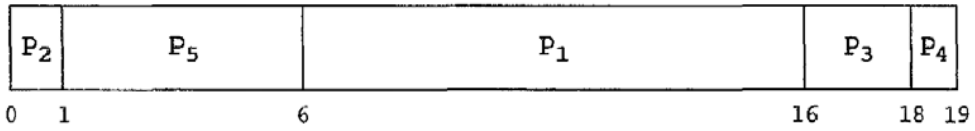
- ✓ The SJF algorithm is a special case of the general priority scheduling algorithm.
- ✓ A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- ✓ Equal-priority processes are scheduled in FCFS order.
- ✓ An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the

  (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- ✓ As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2, … , P_5, with the length of the CPU burst given in milliseconds:



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

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

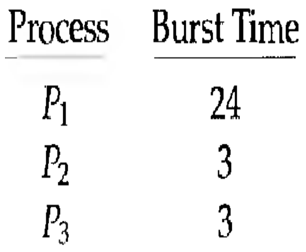


- ✓ The average waiting time is 8.2 milliseconds.

- Priority scheduling can be either **preemptive or nonpreemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A **major problem** with priority scheduling algorithms is **indefinite blocking, or starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- A **solution** to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

- **Round-Robin Scheduling**

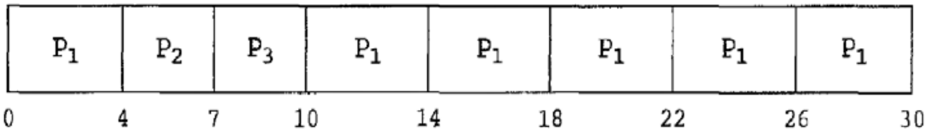
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a **time quantum or time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds.
- The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen.
  - The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
  - Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:



Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue ie. process P_2. Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires.

✓ The CPU is then given to the next process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



✓ The average waiting time is 17/3 = 5.66 milliseconds.

✓ The RR scheduling algorithm is thus preemptive.

✓ 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. Each process must wait no longer than (n-1) * q time units until its next time quantum. **For example**, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

- **Multilevel Queue Scheduling**

✓ Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground (interactive) processes and background (batch) processes**.

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

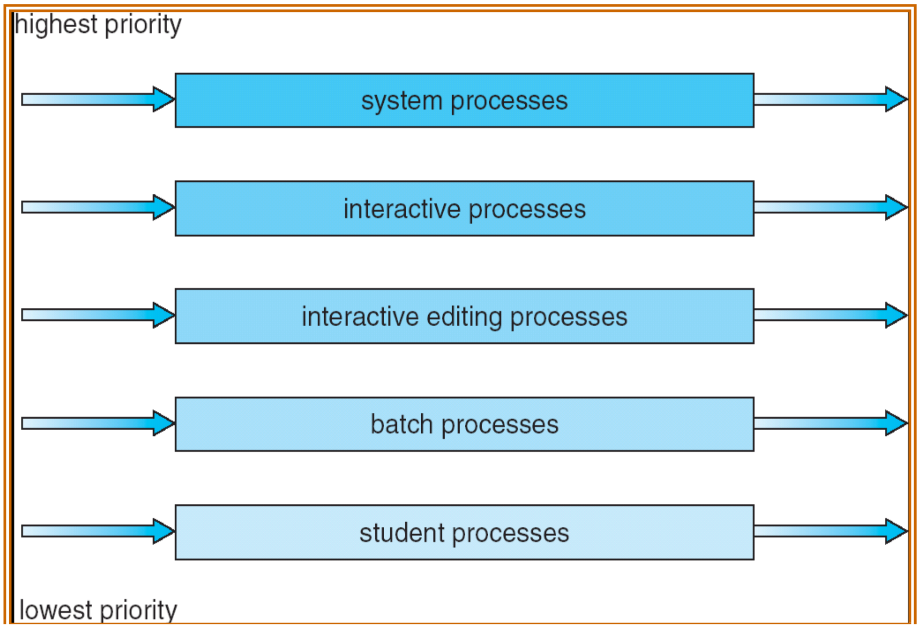
✓ Foreground processes have priority over background processes.

✓ A multilevel queue scheduling algorithm partitions the ready queue into several separate queues as shown in **figure 5.3**.

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

✓ Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.

✓ The foreground queue might be scheduled by an **RR algorithm**, while the background queue is scheduled by an **FCFS algorithm**.

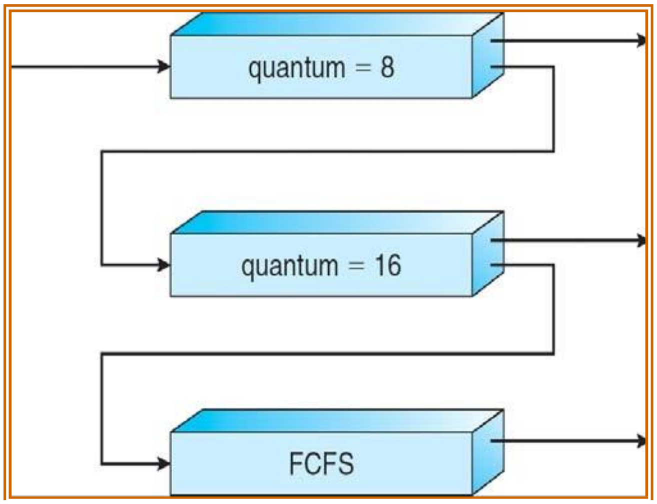


- ✓ There must be scheduling among the queues, which is commonly implemented as **fixed-priority preemptive** scheduling. For example, the foreground queue may have absolute priority over the background queue.
- ✓ An example of a multilevel queue scheduling algorithm with **five queues**, listed below in order of priority:
  1. System processes
  2. Interactive processes
  3. Interactive editing processes
  4. Batch processes
  5. Student processes
- ✓ Each queue has absolute priority over lower-priority queues. No process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- ✓ If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- ✓ Another possibility is to **time-slice** among the queues. So, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For example, the foreground queue can be given **80 percent** of the CPU time for RR scheduling among its processes, whereas the background queue receives **20 percent** of the CPU to give to its processes on an FCFS basis.

- **Multilevel Feedback-Queue Scheduling**

  - ✓ When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
  - ✓ The multilevel feedback-queue scheduling algorithm allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts.

- ✓ If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- ✓ A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
- ✓ For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 as shown in below **figure 5.4.**
- ✓ The scheduler first executes all processes in queue 0. Only when queue 0 is empty it will execute processes in queue 1.
- ✓ Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2.
- ✓ A process in queue 1 will in turn be preempted by a process arriving for queue 0.



- ✓ A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are scheduled on an FCFS basis but they run only when queue 0 and 1 are empty.
- ✓ This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
- ✓ A multilevel feedback-queue scheduler is defined by the following **parameters:**
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The method used to determine when to upgrade a process to a higher-priority queue.
  - The method used to determine when to demote a process to a lower-priority queue.
  - The method used to determine which queue a process will enter when that process needs service.

➢ Multiple-Processor Scheduling

- **Approaches to Multiple-Processor Scheduling**

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

  ✓ A second approach uses **symmetric multiprocessing (SMP),** where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

- Some of the **issues** related to SMP are,

a. Processor Affinity

✓ The data most recently accessed by the process is populated in the **cache** for the processor and successive memory accesses by the process are often satisfied in cache memory.

✓ If the process **migrates** to another processor, the contents of cache memory must be invalidated for the processor being **migrated from**, and the cache for the processor being **migrated to** must be re-populated. Because of the **high cost** of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead tries to keep a process running on the same processor. This is known as **processor affinity**, i.e., a process has an affinity for the processor on which it is currently running.

✓ Processor affinity takes **several forms**. When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so, a situation is known as **soft affinity**. Here, it is possible for a process to migrate between processors.

✓ Some systems such as Linux provide system calls that support **hard affinity**, thereby allowing a process to specify that it must not migrate to other processors.

b. Load Balancing

✓ On SMP systems, it is important to keep the workload balanced among all processors to utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU.

✓ Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

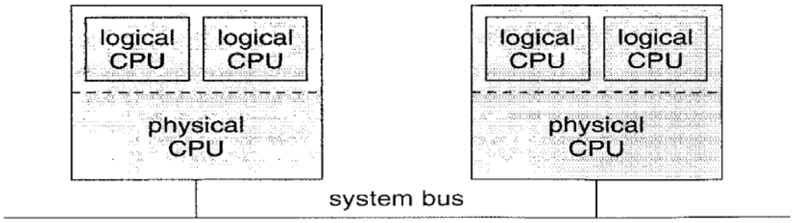
✓ There are two general approaches to load balancing: **push migration** and **pull migration**.

✓ With push migration, a specific task periodically checks the load on each processor and if it finds an imbalance it evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

✓ Pull migration occurs when an idle processor pulls a waiting task from a busy processor.

c. Symmetric Multithreading

- ✓ SMP systems allow several threads to run concurrently by providing multiple physical processors.
- ✓ An alternative strategy is to provide **multiple logical processors** rather than physical processors. Such a strategy is known as symmetric multithreading (or SMT).
- ✓ The idea behind SMT is to create multiple logical processors on the same physical processor, presenting a view of several logical processors to the operating system, even on a system with only a single physical processor.
- ✓ Each logical processor has its own architecture state, which includes general-purpose and machine-state registers and is responsible for its own interrupt handling, meaning that interrupts are delivered to and handled by logical processors rather than physical ones. Otherwise, each logical processor shares the resources of its physical processor, such as cache memory and buses.
- ✓ The following **figure 5.5** illustrates a typical **SMT architecture** with two physical processors, each housing two logical processors. From the operating system's perspective, four processors are available for work on this system.



➢ Thread Scheduling

- ✓ On operating systems that support user-level and kernel-level threads, the kernel-level threads are being scheduled by the operating system.
- ✓ User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
- ✓ One **distinction** between user-level and kernel-level threads lies in how they are **scheduled.** On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS),** since competition for the CPU takes place among threads belonging to the same process.
- ✓ To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS).** Competition for the CPU with SCS scheduling takes place among all threads in the system.
- ✓ PCS is done according to priority. The scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer. PCS will preempt the currently running thread in favour of a higher-priority thread.

- **Pthread Scheduling**

  - ✓ Pthreads identifies the following contention scope values:

    - o PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
    - o PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

  - ✓ On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs.
  - ✓ The number of LWPs is maintained by the thread library using scheduler activations. The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user- level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.
  - ✓ The Pthread IPC provides the following **two functions** for getting and setting the contention scope policy;

    - o pthread_attr_setscope (pthread_attr_t *attr, int scope)
    - o pthread_attr_getscope (pthread_attr_t *attr, int *scope)

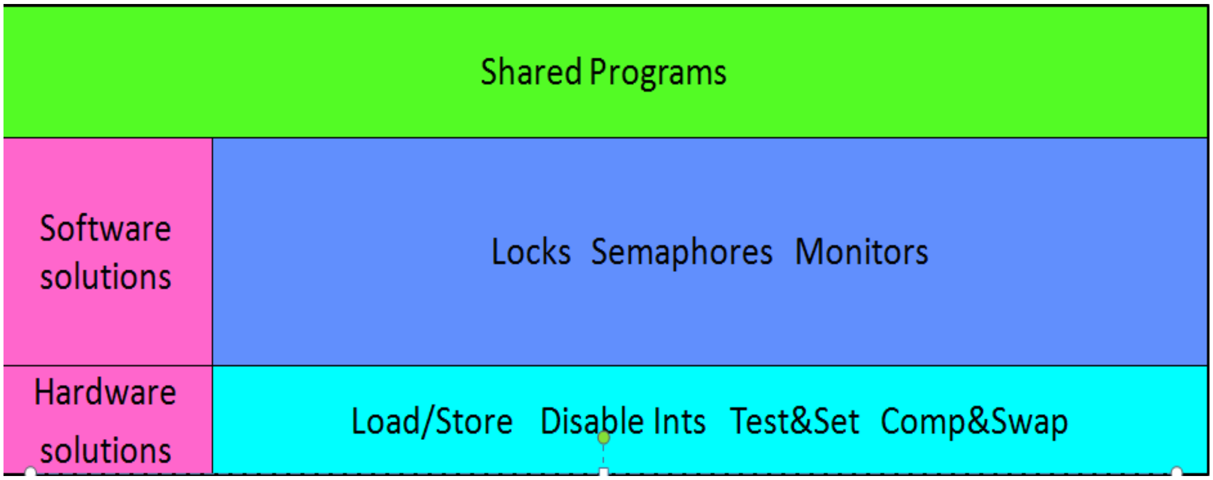
  - ✓ The first parameter for both functions contains a pointer to the attribute set for the thread.
  - ✓ The second parameter for the pthread_attr_setscope () function is passed either the THREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of pthread_attr_getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns non-zero values.

Module 2 (Continued)
Process Synchronization

- The critical section problem;
- Peterson's solution;
- Synchronization hardware;
- Semaphores;
- Classical problems of synchronization;
- Monitors.

2.3.1 Introduction:

A co-operating process is one that can affect or be affected by other process executing in the system. It can use shared memory or message passing for communication. If shared memory is implemented, the access to the share data must be taken care and will be discussed in detail in this chapter.


Shared Programs
Software solutions
Locks Semaphores Monitors
Hardware solutions
Load/Store Disable Ints Test&Set Comp&Swap

2.3.2 Race Condition:

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against race condition, we require that the processes must be synchronized in some way.

Consider an example:

```
a=10
{
        Read a
        a++
        write a
}
```

If P1 and P2 are different process and sharing the above code section, then P1 & P2 executes one after other P1 prints **a** as **11**and P2 prints **a** as **12**. Assume if P1 and P2 are executing concurrently, then outcome of execution depends on order in which the access take place, which is a race condition and leads to data inconsistency.

2.3.3 Critical-section problem:

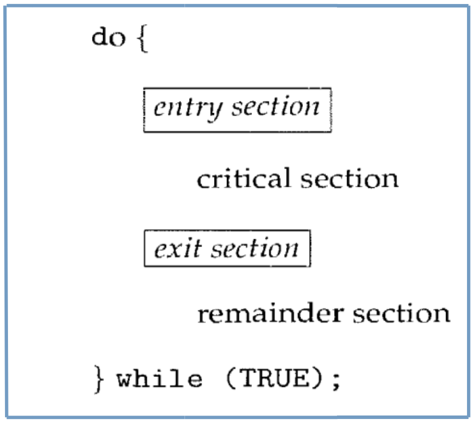
Critical section is a code /part of a program which contain access to shared variables and has to be executed as an atomic action.

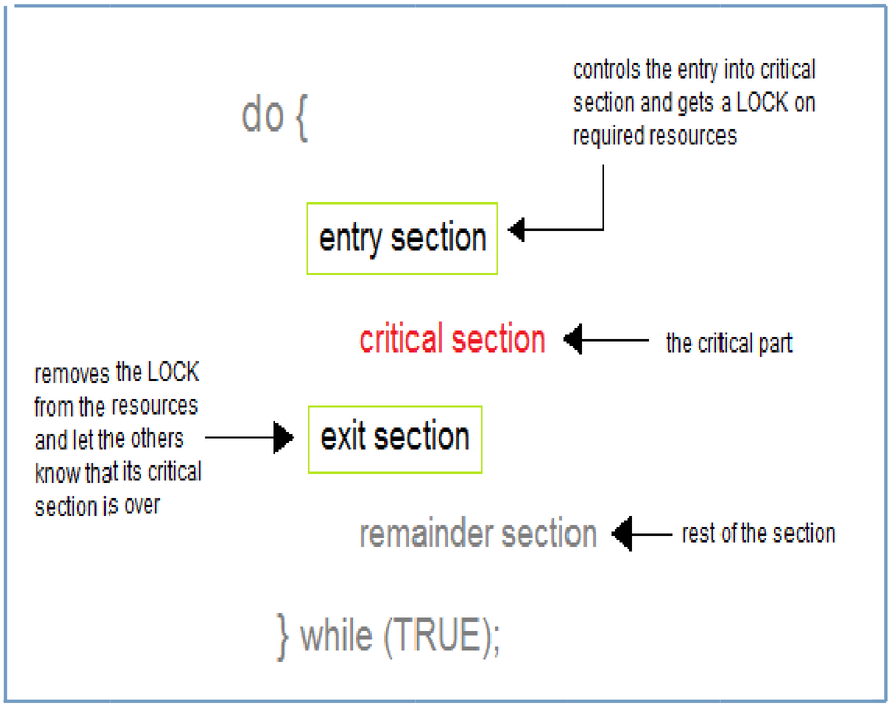
Example: Consider the air line ticket reservation system, if only one sit is available and if 3 persons are booking a ticket at a time, only one should be allowed to block a sit, else it leads to data inconsistency. So the code which allows for updation of the seat variable(shared data) is the critical section.

Consider a system consisting of n processes {P_0, P_1... P_{n-1}}. Each process has a segment of code, called a **critical section**, in which the process may be changing the 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 critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The **critical-section problem** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its **critical section**. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

The **general structure** of a typical process P_i, is shown in the following figure




do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

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

1. **Mutual exclusion.** If process P, 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 the decision on 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.

Two general approaches are used to **handle** critical sections in operating systems.

1. Pre-emptive kernel 2. Non preemptive kernel.

A **pre-emptive kernel** allows a process to be pre-empted while it is running in kernel mode. It is **difficult to design** for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors. It is more **suitable** for **real-time programming**, as it will allow a real-time process to pre-empt a process currently running in the kernel. Also, pre-emptive kernel may be **more responsive**, since there is less risk that a kernel-mode process will run for an arbitrarily long period before giving up the processor to waiting processes.

A **Nonpreemptive kernel** does not allow a process running in kernel mode to be pre-empted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. It is essentially free from **race conditions** on kernel data structures, as only one process is active in the kernel at a time.

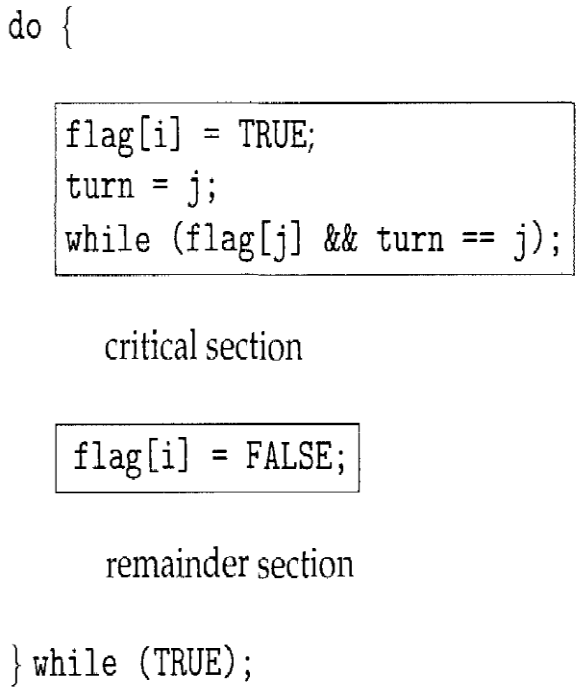
2.3.4 Peterson's solution

- A classic **software-based solution** to the critical-section problem is known as **Peterson's solution.** Peterson's solution is restricted to **two processes** that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_0 and P_1 **or** P_i and P_j where j=1-i.

Peterson's solution requires **two data items** to be shared between the two processes,

- 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 P_i, 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 P_i is ready to enter its critical section.
- To enter the critical section, process P_i 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 decides which of the two processes is allowed to enter its critical section first.
- The following algorithm describes the structure of P_i in Peterson's solution.



```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

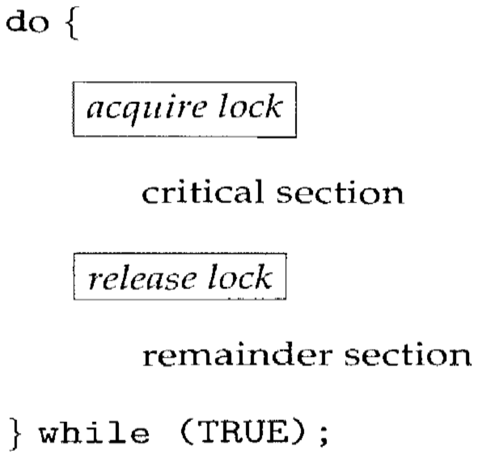
- To prove that this solution is correct, we need to show that,
  1. Mutual exclusion is preserved.
  2. The progress requirement is satisfied.
  3. The bounded-waiting requirement is met.

- To prove **property 1**, we note that each P_i enters its critical section only if **either flag[j] = = false or turn = = i.** For P_0 to enter, turn must be equal to 0 and for P_1 to enter, turn must be equal to 1 because flag[0] = = flag[1] = = true. Since the value of turn can be either 0 or 1 but cannot be both, hence P_0 and P_1cannot enter into critical section simultaneously.
- To prove **properties 2 and 3**, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = = true and turn = = j. If P_j is not ready to enter the critical section, then flag [j] = = false, and P_i can enter its critical section. If turn = = j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag[j] to true, it must also set turn to i. Thus, P_i will enter the critical section (progress) after at most one entry by P_i (bounded waiting).

2.3.5 Synchronization hardware:

- Any solution to the critical-section problem requires a simple tool called **a lock**. **Race conditions** are prevented by protecting the critical regions by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for a long time

This is illustrated below,



```
do {

    acquire lock

        critical section

    release lock

        remainder section

} while (TRUE);
```

- Hardware features can make any programming task easier and improve **system efficiency.**
- The critical-section problem can be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. This solution is **not feasible** in a multiprocessor environment. Disabling interrupts on a multiprocessor can be **time consuming**, as the message is passed to all the processors. This message passing delays entry into each critical section, and **system efficiency decreases.**
- Many modern computer systems therefore provide special hardware instructions that allow us either to **test** and **modify** the content of a word or to **swap** the contents of two words **atomically** that is, as one uninterruptible unit.
- We can use these special instructions to solve the critical-section problem in a relatively simple manner.The TestAndSet() instruction can be defined as below,

```
boolean TestAndSet(boolean *target)     // This code returns false for first time i,e rv
{                                       //value and sets lock as true for next
        boolean rv = *target;           //execution
        *target = TRUE;
        return rv;

}
```

- The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- The structure of process P_i is shown below,

```
do {
        while (TestAndSetLock(&lock))
            ; // do nothing

        // critical section
```

```
                lock = FALSE;        //sets lock as False so allows next  to
                                     //take chance
            // remainder section

          }while (TRUE);
```

- The Swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words as shown below,

```
        void Swap(boolean *a, boolean *b)
                    {
  boolean temp = *a;
  *a = *b;
     *b = temp;
                    }
```

- It is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable **lock** is declared and is initialized to false. In addition, each process has a local Boolean variable **key**. The structure of process P_i is shown below,

```
do {
    key = TRUE;
      while (key = = TRUE)
            Swap(&lock, &key
    };

            // critical section

        lock = FALSE;

            // remainder section
 }while (TRUE);
```

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.
- Another algorithm is given below using the TestAndSet() instruction that satisfies all the critical-section requirements. The common **data structures** are,

```
boolean waiting[n];
boolean lock;
```

- These data structures are initialized to **false**.

```
do {
    waiting[i] = TRUE;
     key = TRUE;
    while (waiting[i] && key)
            key= TestAndSet(&lock);
    waiting[i] = FALSE;
```

```
                // critical section

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;

        if (j = = i)
                lock = FALSE;
        else
                waiting[j] = FALSE;

                // remainder section
        } while (TRUE);
```

2.3.6 Semaphores:

The **hardware-based solutions** to the critical-section problem are **complicated** for application programmers to use. To **overcome this difficulty**, we can use a **synchronization tool** called a **semaphore.**

Dijkstra's worked on semaphores established over 30 years ago the foundation of modern techniques for accomplishing synchronization

- A semaphore, S, is a integer variable that is changed or tested only by one of the two following indivisible operations

  P(S): while(S<=0)
          Do*no-operation*;
      S--;
  V(S): S++;

- In Dijkstra's original paper, the P operation was an abbreviation for the Dutch word Proberen, meaning "to test" and the V operation was an abbreviation for the word Verhogen, meaning "to increment"

Now, P () and V() is normally called wait() and signal()

A **semaphore S** is an integer variable it is accessed only through **two standard atomic operations: wait()** and **signal()**. The wait() operation was termed as **P**; signal() was called **V**. The definition of **wait()** is as follows,

```
wait(S)
    {
            (while S<= 0) ; // no-operation
            S--;
    }
```

- ✓ The definition of **signal()** is as follows,

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

- ✓ The wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Usage:

Operating systems often distinguish between **counting and binary** semaphores.The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1.

Binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion. We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, **mutex**, initialized to 1. Each process P_i is organized as shown,

```
do {
        waiting(mutex);

                // critical section

        signal (mutex) ,

                // remainder section
    }while (TRUE);
```

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). When count=0, all resources are being used. Then the 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: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that P_2 be executed only after S_1 has completed. We can implement this by letting P_1 and P_2 to share a common semaphore **synch**, initialized to 0, and by inserting the statements

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

in process P_1, and the statements

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

in process P_2. Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal (synch), which is after statement S_1 has been executed.

Disadvatage:

- The main **disadvantage** of the semaphore definition 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 entry code.

- This type of semaphore is also called a **spin lock** because the process "spins" while waiting for the lock.

<u>Implementation :</u>

To overcome the need for busy waiting, we can **modify** the definition of the **wait() and signal()** semaphore operations. When a process executes the wait() operation and finds that the semaphore value is not positive, then 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.**

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. To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
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 now be defined as,

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

The signal () semaphore operation can now 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. These two operations are provided by the operating system as basic system calls. This implementation, semaphore values may be negative, if a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a **deadlock** 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, we consider a system consisting of two processes, P_0 and P_1, 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 P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.
- 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:

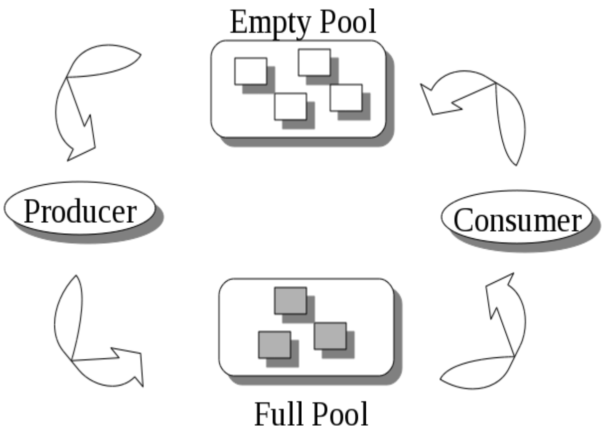
A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for 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.

As an example, assume we have three processes—*L*, *M*, and *H*—whose priorities follow the order $L < M < H$. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*. 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. That is insufficient for most general-purpose operating systems, however. 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 *H*, thereby preventing process *M* from preempting its execution. When process *L* had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H*—not *M*—would run next.

2.3.7 Classic problems of Synchronization

- Bounded-buffer problem
- The Readers-Writers Problem
- The Dining-Philosophers Problem

The Bounded-Buffer Problem:



The pool consists of **n** buffers is considered, 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 code for the producer process is shown:

```
do{
            // produce an item in nextp
      ...

      wait(empty); // checks for empty buffer, if no then waits
      wait (mutex); // if empty buffer is available, then will fill it
      ...
      // add nextp to buffer
      ...
      signal(mutex);    //releases the buffer
      signal (full);       // increments full array
   }while (TRUE);
```

The code for the consumer process is shown:

```
do
{
            wait (full);    // if full is 0, it waits
            wait(mutex); //if full is non –zero , then reads
...
                  //remove an item from buffer to nextc
...
```

```
                signal(mutex);
                signal(empty);
                //consume the item in nextc
        ...

        }while (TRUE);
```

The Readers-Writers Problem:

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.

- ✓ **Readers** - processes that only read the database.
- ✓ **Writers** - processes performing both read and write (update).
- ✓ **Problem**: If two readers access the shared data simultaneously, no problem will result. But if a writer and some other thread (either a reader or a writer) access the database simultaneously, problem arises.

This synchronization problem is referred to as the readers-writers problem.

The readers-writers problem has several variations. The **first** readers-writers problem, which requires that no reader must be kept waiting unless a writer has already obtained permission to use the shared object. The **second** readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible.

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 mutex, wrt;
int readcount;
```

- ✓ The semaphores **mutex** and **wrt** are initialized to 1and **readcount** is initialized to 0.
- ✓ The semaphore wrt is common to both reader and writer processes.
- ✓ The **mutex** semaphore is used to ensure **mutual exclusion** when the variable **readcount** is updated.
- ✓ The **readcount** variable keeps track of how many processes are currently reading the object.
- ✓ The semaphore **wrt** functions as a mutual-exclusion semaphore for the writers.
- ✓ The code for a writer process is,

```
do {
        wait(wrt);
          …..
        // writing is performed
        signal(wrt);
} while (TRUE);
```

- ✓ The code for a reader process is shown,

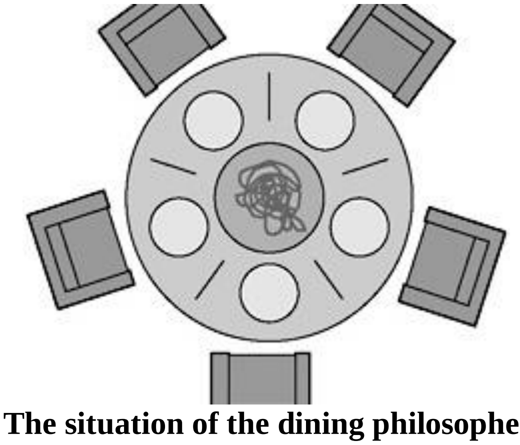
```
do {
        wait (mutex);
        readcount++;
        if (readcount = = 1)        // First reader must
```

```
            wait (wrt);          // block the writers
       signal(mutex);
            ……..
            // reading is performed
            ……..
      wait(mutex);
      readcount--;
      if (readcount= = 0)          //Last reader must
            signal(wrt);           //unblock the writers
      signal(mutex);
   } while (TRUE);
```

- ✓ If a writer is in the critical section and n readers are waiting, then one reader is queued on **wrt**, and n-1 readers are queued on **mutex**.
- ✓ Also, when a writer executes **signal(wrt)**, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

<u>The Dining-Philosophers Problem:</u>

- ✓ 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 centre of the table is a bowl of noodles, and the table is laid with five single chopsticks (below figure).
- ✓ When a philosopher is thinking, he does not interact with her colleagues.
- ✓ When a philosopher gets hungry, he tries to pick up the two chopsticks that are closest to him (the chopsticks that are between his left and right neighbors).
- ✓ 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.



The situation of the dining philosophers

- ✓ 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 is shown below,

```
do {
        wait(chopstick[i]);
       k[(i+l) % 5]);
          ……
                // eat
         ........

        signal(chopstick[i]);
       signal(chopstick[(i+l) % 5]);
        ………
        // think
       ..........
   } while (TRUE);
```

- The disadvantage is it can create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab right chopstick, he will be delayed forever.
- Several possible remedies to the deadlock problem are available.
    - Allow at most four philosophers to be sitting simultaneously at the table.
    - Allow a philosopher to pick up chopsticks only if both chopsticks are available (to do this, he must pick them up in a critical section).
    - Use an asymmetric solution; that is, an odd philosopher picks up first left chopstick and then right chopstick, whereas an even philosopher picks up right chopstick and then left chopstick.

2.3.8 Monitors:

- All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.
- This may result in various difficulties.
    - 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 may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.
    - Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
        ……
```

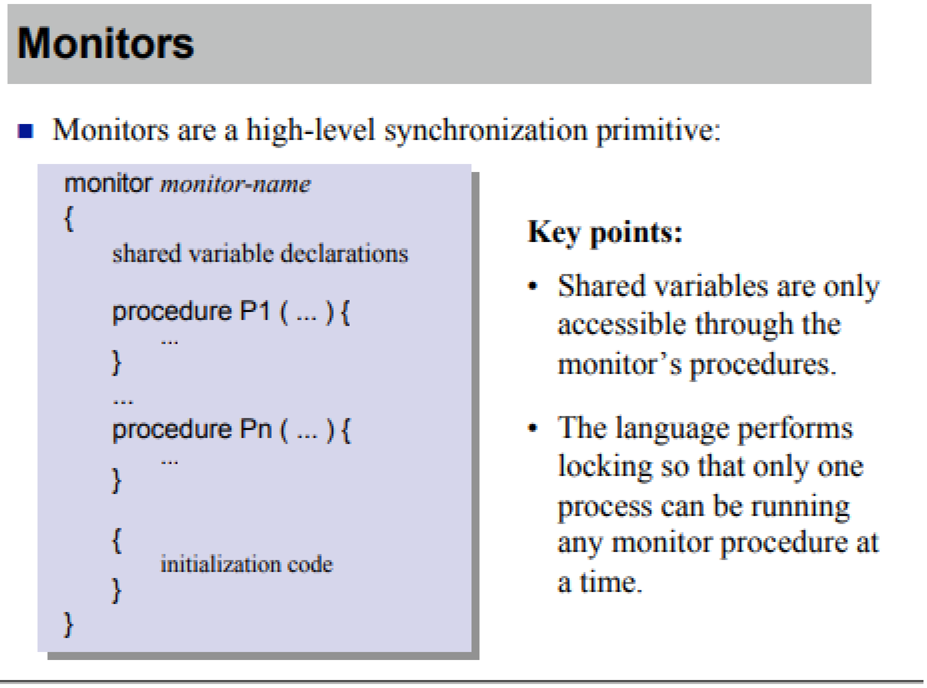
```
                                   critical section
                                   ……
                               wait(mutex);
```

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.

✓ These **examples** illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, one fundamental high-level synchronization construct-the **monitor** is used.

Usage:



Monitors

- ■ Monitors are a high-level synchronization primitive:

```
monitor monitor-name
{
        shared variable declarations

        procedure P1 ( ... ) {
                ...
        }
        ...
        procedure Pn ( ... ) {
                ...
        }

        {
                initialization code
        }
}
```

Key points:

- Shared variables are only accessible through the monitor's procedures.
- The language performs locking so that only one process can be running any monitor procedure at a time.

✓ Monitors encapsulate data structures that are not externally accessible ♦Mutual exclusive access to data structure enforced by compiler or language run-time.

✓ An abstract data type encapsulates data with the set of functions to operate on that data are independent of any specific implementation of ADT.

✓ A monitor type is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within a monitor.It declares the variables whose values define the state of an instance of that type as shown below:

```
monitor monitor name
{
   // shared variable declarations
```

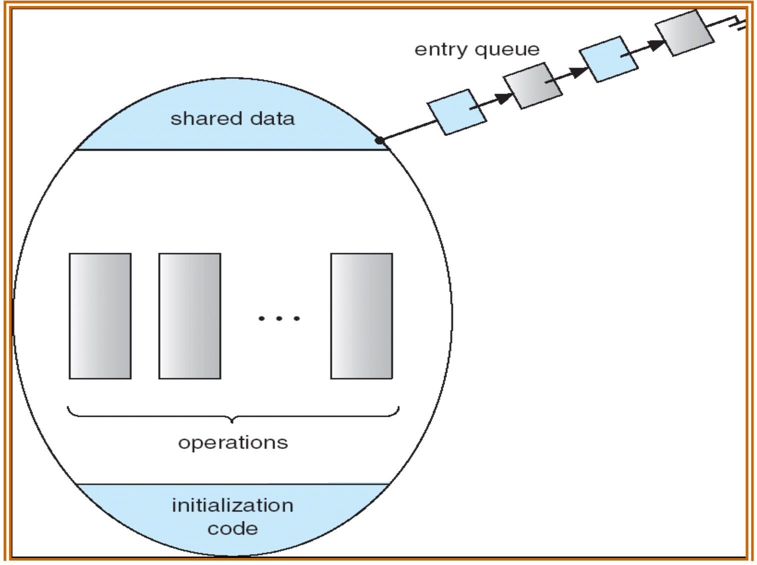
```
                    procedure P1 ( . . . ) {
                    ………
                    }
                    procedure P2 ( . . . ) {
                    ……….
                    }
                              .
                              .
                    procedure Pn ( . . . ) {
                    …………
                    }
                    initialization code ( . . . ) {
                    …………
                    }
          }
```

- ✓ The functions defined within a monitor can access only those variables declared locally within the monitor and its formal parameters and the local variables of a monitor can be accessed by only the local functions.
- ✓ **Consider an example:**

```
Monitor bank
{
          Condition balance;

          Withdraw(amount)
          {
                    balance=balance-amount;
          }
}
```

- ✓ Balance variable can be updated only by calling withdraw method and monitors allows only one process to enter and ensures synchronization.
- ✓ The monitor construct ensures that only one process at a time is active within the monitor. The programmer does not need to code this synchronization constraint explicitly.

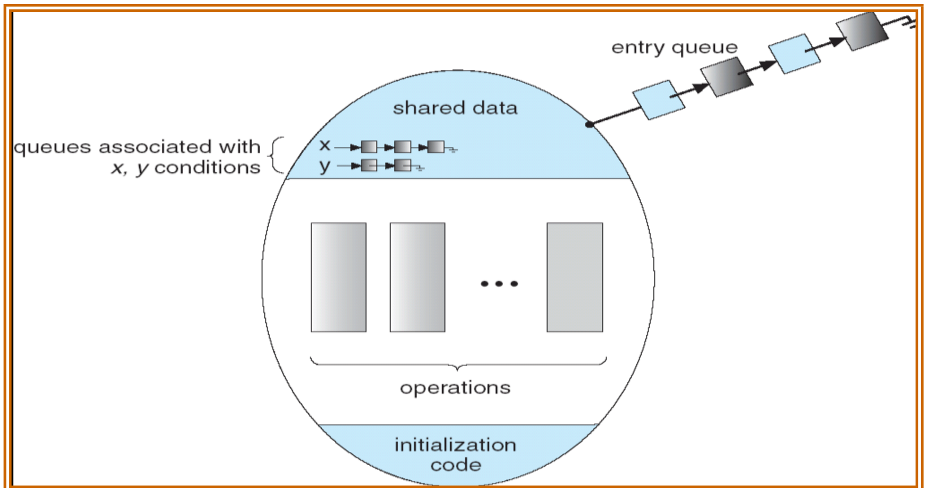


Schematic view of a monitor

- ✓ The monitor construct, as defined so far 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. A programmer can define one or more variables of the type **condition**,

condition x, y;

- ✓ For example in producer and consumer problem, Monitors allows only one process to enter ,but it will not check whether it is producer or consumer. If consumer enters and if there are empty buffers then , either it can read nor it allows other process (not even producer) to enter the monitor. So condition variables are used.
- ✓ 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.
- ✓ If no process is suspended, then the signal () operation has no effect; that is, the state of x is the same as if the operation has never been executed. It is shown in below figure.



Monitor with condition variables

- ✓ Condition Variables need wait and wakeup as in semaphores . Monitor uses Condition Variables Conceptually associated with some conditions Operations on condition variables:
  - o wait(): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true
  - o signal(): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread
- ✓ Suppose when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x. If the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. However both processes can continue with their execution. Two possibilities exist,
  - o **Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.
  - o **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

Dining-Philosophers Solution Using Monitors:

✓ It is 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.

✓ We use the following data structure to distinguish among three states in which we may find a philosopher.

```
enum {thinking, hungry, eating}state [5];
```

✓ Philosopher i can set the variable **state[i] = eating** only if her two neighbors are not eating.

✓ We also need to declare

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

where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

✓ The distribution of the chopsticks is controlled by the monitor **dp**, whose definition is shown below.

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

        void pickup(int i)
        {
                state[i] =HUNGRY;              //set philosopher as hungry
                test(i);                   //whether he gets both the chopsticks

            if (state [i] ! = EATING)          //  after eating set philosopher  to
                                           //wait  until  he  is  hungry  &  chopsticks
                                           //are available
                    self [i] . wait() ;
        }

        void putdown(int i)
        {
                state[i] =THINKING;            //set philosopher to thinking
                test((i + 4) % 5);     //Test whether the neighbor philosophers
                                       //are hungry and waiting for chopsticks
                test((i + 1) % 5);
        }
```

//test function first checks whether both the chopsticks are available for the philosopher. If yes allows him //to eat by setting self[i].signal[self is a condition variable which allows to continue eating].

```
        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;
            }
        }
```

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

```
dp.pickup(i);
        ...
        eat
        ...
dp.putdown(i);
```

Implementing a Monitor 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** (initialized to 0) is introduced, on which the signaling processes may suspend themselves.
- ✓ An integer variable **next_count** is also provided to count the number of processes suspended on **next**. Thus, each external procedure **F** is replaced by ,

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

Mutual exclusion within a monitor is ensured.

- ✓ Condition variables are implemented as follows. For each condition x, semaphore **x_sem** and an integer variable **x_count**, both initialized to 0 are introduced.
- ✓ 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--;
}
```

<u>Resuming Processes within a Monitor</u>

- ✓ If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then the problem is how to determine which of the suspended processes should be resumed next.
- ✓ One simple solution is to use an FCFS ordering, so that the process that has been waiting for the longest time is resumed first.
- ✓ In many circumstances, such simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it 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.
- ✓ To illustrate this, consider the **ResourceAllocator** monitor shown below, which controls the **allocation of a single resource** among competing processes.
- ✓

```
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;
                                        }
                                  }
```

- ✓ Each process, when requesting an allocation of this resource, it specifies the maximum time it plans to use the resource.
- ✓ The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

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

where R is an instance of type ResourceAllocator.

- ✓ The monitor concept cannot guarantee that the preceding access sequence will be observed and the following problems can occur,
    - o A process might access a resource without first gaining access permission to the resource.
    - o A process ntight never release a resource once it has been granted access to the resource.
    - o A process might attempt to release a resource that it never requested.
    - o A process might request the same resource twice (without first releasing the resource).
- ✓ The same difficulties are encountered with the use of semaphores.

<u>Solution to the Producer-Consumer problem using Monitors</u>

Monitors make solving the producer-consumer a little easier. Mutual exclusion is achieved by placing the critical section of a program inside a monitor. In the code below, the critical sections of the producer and consumer are inside the monitor *Producer Consumer*. Once inside the monitor, a process is blocked by the **Wait** and **Signal** primitives if it cannot continue.

Two condition variables

- has_empty: buffer has at least one empty slot
- has_full: buffer has at least one full slot

nfull: number of filled slots need to do our own counting for condition variables

monitor ProducerConsumer

```
{
        int nfull = 0;
        condition has_empty, has_full;
        producer()
         {
                if (nfull == N)
                        wait (has_empty); … // fill a slot
                         ++ nfull;
                        signal (has_full);
        }
        consumer()
         {
                if (nfull == 0)
                         wait (has_full); … // empty a slot
                        -- nfull;
                        signal (has_empty);
         }
}
```