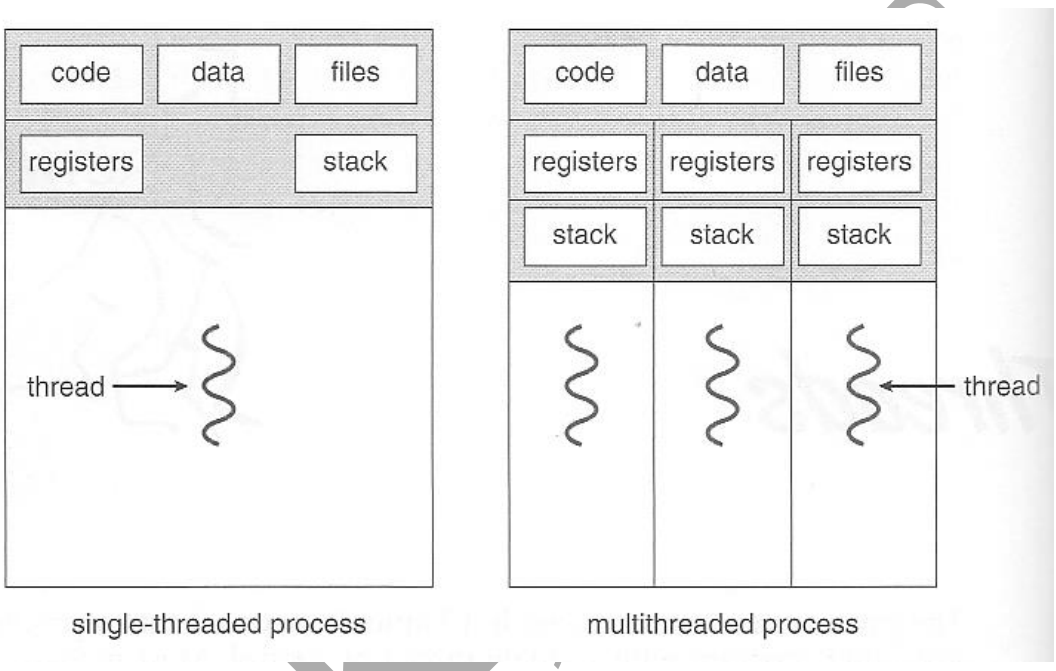
Multi threaded programming

- A ***thread*** is a basic unit of CPU utilization. It consists of a thread ID, program counter, a stack, and a set of registers.
- Traditional processes have a single thread of control. It is also called as **heavyweight process**. There is one program counter, and one sequence of instructions that can be carried out at any given time.
- A multi-threaded application have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files. Such process are called as **lightweight process**.



Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- In a web server - Multiple threads allow for multiple requests to be served simultaneously. A thread is created to service each request; meanwhile another thread listens for more client request.
- In a web browser – one thread is used to display the images and another thread is used to retrieve data from the network.

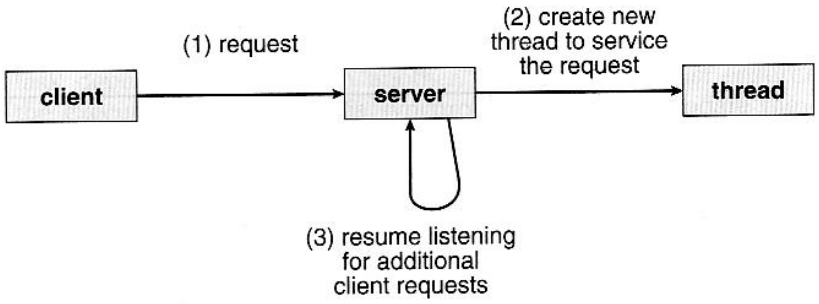


Figure 4.2 Multithreaded server architecture.

Benefits

The four major benefits of multi-threading are:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
   Multi threading allows a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. **Economy** - Creating and managing threads is much faster than performing the same tasks for processes. Context switching between threads takes less time.
4. **Scalability**, i.e. **Utilization of multiprocessor architectures** – Multithreading can be greatly utilized in a multiprocessor architecture. A single threaded process can make use of only one CPU, whereas the execution of a multi-threaded application may be split among the available processors. Multithreading on a multi-CPU machine increases concurrency. In a single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip, would have to execute the threads one after another. On a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing.

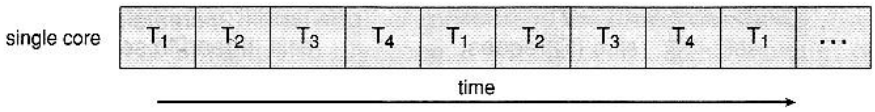


Figure 4.3 Concurrent execution on a single-core system.

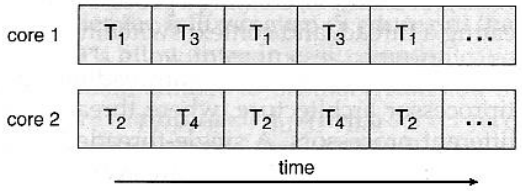


Figure 4.4 Parallel execution on a multicore system.

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- For application programmers, there are five areas where multi-core chips present new challenges:
    1. Dividing activities - Examining applications to find activities that can be performed concurrently.
    2. Balance - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
    3. Data splitting - To prevent the threads from interfering with one another.
    4. Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
    5. Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Multithreading Models

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- User threads are the threads that application programmers would put into their programs. They are supported above the kernel, without kernel support.
- Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple tasks simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following models.

a) Many-To-One Model

In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

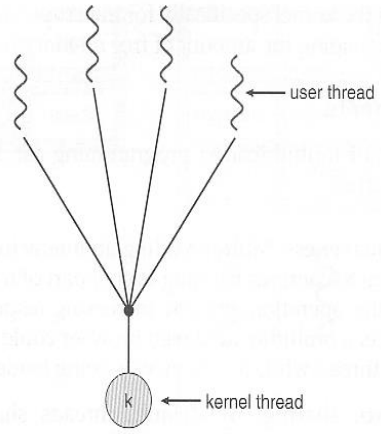


Figure 4.2 Many-to-one model.

- Thread management is handled by the thread library in user space, which is very efficient.
- If a blocking system call is made by one of the threads, then the entire process blocks. Thus blocking the other user threads from continuing the execution.
- Only one user thread can access the kernel at a time, as there is only one kernel thread. Thus the threads are unable to run in parallel on multiprocessors.
- Green threads of Solaris and GNU Portable Threads implement the many-to-one model.

b) One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- This model places a limit on the number of threads created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

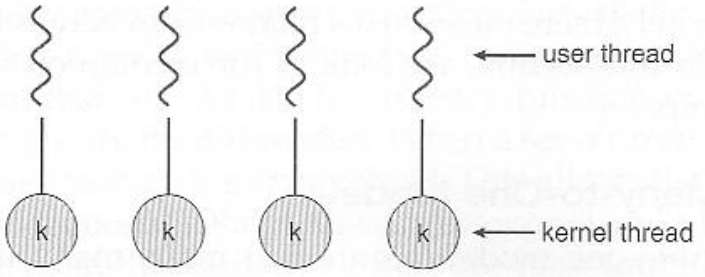


Figure 4.3 One-to-one model.

c) Many-To-Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

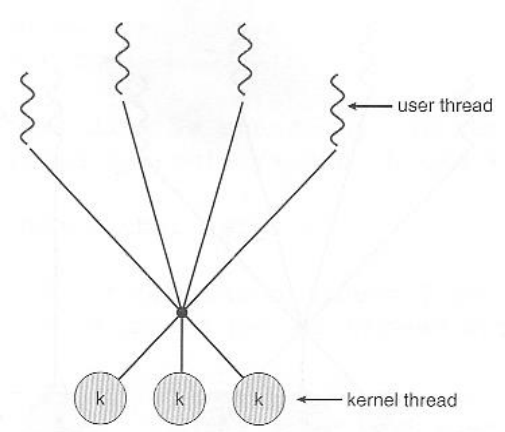


Figure 4.4 Many-to-many model.

- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.
- This model is also called as two-tier model.
- It is supported by operating system such as IRIX, HP-UX, and Tru64 UNIX.

Threading Issues

a) The fork( ) and exec( ) System Calls

The fork( ) system call is used to create a separate, duplicate process.
When a thread program calls fork( ),

- The new process can be a copy of the parent, with all the threads
- The new process is a copy of the single thread only (that invoked the process)

If the thread invokes the exec( ) system call, the program specified in the parameter to exec( ) will be executed by the thread created.

b) Cancellation

Terminating the thread before it has completed its task is called thread cancellation. The thread to be cancelled is called **target thread**.

Example : Multiple threads required in loading a webpage is suddenly cancelled, if the browser window is closed.

Threads that are no longer needed may be cancelled in one of two ways:

1. **Asynchronous Cancellation -** cancels the thread immediately.
2. **Deferred Cancellation** – the target thread periodically check whether it has to terminate, thus gives an opportunity to the thread, to terminate itself in an orderly fashion.

   In this method, the operating system will reclaim all the resources before cancellation.

c) Signal Handling

A signal is used to notify a process that a particular event has occurred.

All signals follow same path-

1) A signal is generated by the occurrence of a particular event.
2) A generated signal is delivered to a process.
3) Once delivered, the signal must be handled.

A signal can be invoked in 2 ways : synchronous or asynchronous.

Synchronous signal – signal delivered to the same program. Eg – illegal memory access, divide by zero error.

Asynchronous signal – signal is sent to another program. Eg – Ctrl C

In a single-threaded program, the signal is sent to the same thread. But, in multi-threaded environment, the signal is delivered in variety of ways, depending on the type of signal –

- Deliver the signal to the thread, to which the signal applies.
- Deliver the signal to every threads in the process.
- Deliver the signal to certain threads in the process.
- Deliver the signal to specific thread, which receive all the signals.

A signal can be handled by one of the two ways –

Default signal handler - signal is handled by OS.

User-defined signal handler - User overwrites the OS handler.

d) Thread Pools

In multithreading process, thread is created for every service. Eg – In web server, thread is created to service every client request.

Creating new threads every time, when thread is needed and then deleting it when it is done can be inefficient, as –

Time is consumed in creation of the thread.

A limit has to be placed on the number of active threads in the system. Unlimited thread creation may exhaust system resources.

An alternative solution is to create a number of threads when the process first starts, and put those threads into a ***thread pool***.

- Threads are allocated from the pool when a request comes, and returned to the pool when no longer needed(after the completion of request).
- When no threads are available in the pool, the process may have to wait until one becomes available.

Benefits of Thread pool –

- Thread creation time is not taken. The service is done by the thread existing in the pool. Servicing a request with an existing thread is faster than waiting to create a thread.
- The thread pool limits the number of threads in the system. This is important on systems that cannot support a large number of concurrent threads.

The (maximum) number of threads available in a thread pool may be determined by parameters like the number of CPUs in the system, the amount of memory and the expected number of client request.

e) Thread-Specific Data

- Data of a thread, which is not shared with other threads is called thread specific data.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data.

Example – if threads are used for transactions and each transaction has an ID. This unique ID is a specific data of the thread.

f) Scheduler Activations

Scheduler Activation is the technique **used** for communication between the user-thread library and the kernel.
It works as follows:

— 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.

Example - The kernel triggers an upcall occurs when an application thread is about to block. The kernel makes an upcall to the thread library informing that a thread is about to block and also informs the specific ID of the thread.

The upcall handler handles this thread, by saving the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.

The upcall handler then schedules another thread that is eligible to run on the 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. Thus assigns the thread to the available virtual processor.

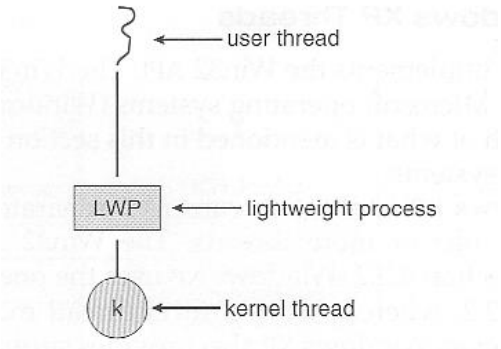


Figure 4.9 Lightweight process (LWP.)

Thread Libraries

- Thread libraries provide an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space.
- There are three main thread libraries in use –
    1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. Win32 threads - provided as a kernel-level library on Windows systems.
    3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.
- The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

4.3.1 Pthreads

- The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function.
- Pthread_create() function is used to create a thread.

```
#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;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Figure 4.9

4.3.2 Win32 Threads

- Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature.
- Here summation() function is used to perform the separate thread function.
- CreateThread() is the function to create a thread.

```
#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;
   /* perform some basic error checking */
   if (argc != 2) {
      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, // 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) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Figure 4.7 Multithreaded C program using the Win32 API.

4.3.3 Java Threads

- ALL Java programs use Threads.

```
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[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        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>"); }
}
```

Figure 4.8 Java program for the summation of a non-negative integer.

- The creation of new Threads requires to implement the Runnable Interface, which contains a built-in method "public void run( )" . The Thread class will have to overwrite the built-in function run( ), in which the thread code should be written.
- Creating a Thread Object does not start the thread running - To start the thread, the built-in start( ) method should be invoked, which in turn call the run() method(where statements to be executed by thread are written).

CPU SCHEDULING

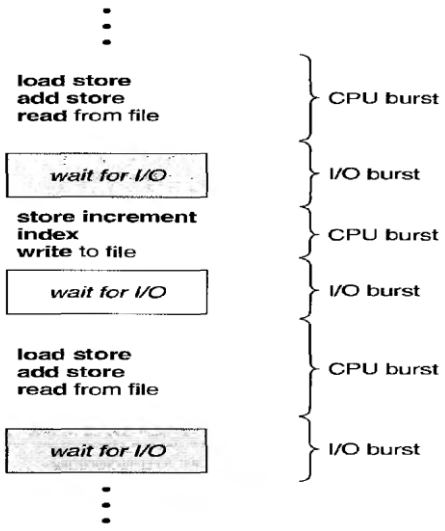
3.1 BASIC CONCEPTS

In a single-processor system, only one process can run at a time; other processes must wait until the CPU is free. The objective of multiprogramming is to have some process running at all times in processor, to maximize CPU utilization.

In 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. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is 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. The state of process under execution is called **CPU burst** and the state of process under I/O request & its handling is called **I/O burst**. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution as shown in the following figure:



CPU Scheduler

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

A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. All the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

Non - Preemptive Scheduling – once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Preemptive Scheduling – The process under execution, may be released from the CPU, in the middle of execution due to some inconsistent state of the process.

Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

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

SCHEDULING CRITERIA

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization** - The CPU must be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent .
- **Throughput** - If the CPU is busy executing processes, then work is done fast. One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time -** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

  Time spent waiting (to get into memory + ready queue + execution + I/O)
- **Waiting time -** The total amount of time the process spends waiting in the ready queue.
- **Response time -** The time taken from the submission of a request until the first response is produced is called the response time. It is the time taken to start responding. In interactive system, response time is given criterion.

It is desirable to **maximize** CPU utilization and throughput and to **minimize** turnaround time, waiting time, and response time.

SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

1. First-Come, First-Served Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. This algorithm is always nonpreemptive, once a process is assigned to CPU, it runs to completion.

Advantages :

- more predictable than other schemes since it offers time
- code for FCFS scheduling is simple to write and understand

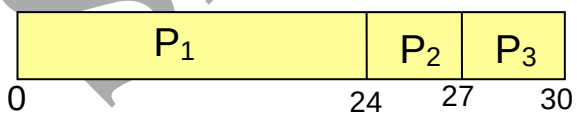
Disadvantages:

- Short jobs(process) may have to wait for long time
- Important jobs (with higher priority) have to wait
- cannot guarantee good response time
- average waiting time and turn around time is often quite long
- lower CPU and device utilization.

Example:-

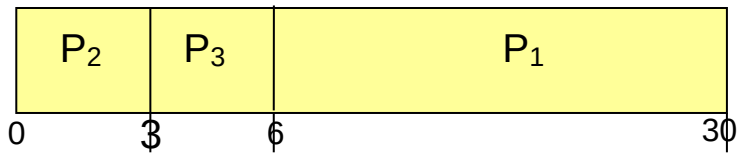
Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1*, *P2* , *P3*
The Gantt Chart for the schedule is:



Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order *P2* , *P3* , *P1*
The Gantt chart for the schedule is:



Waiting time for *P1* = 6*; P2* = 0*; P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3
Much better than previous case

Here, there is a ***Convoy effect***, as all the short processes wait for the completion of one big process. Resulting in lower CPU and device utilization.

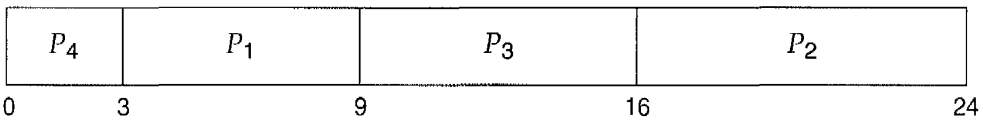
3.3.2 Shortest-Job-First Scheduling

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 provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst, and let t_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

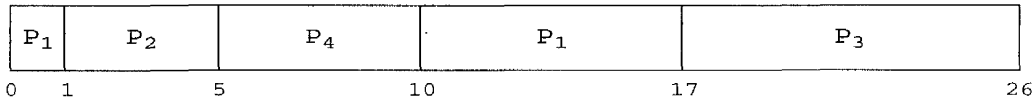
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

The 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

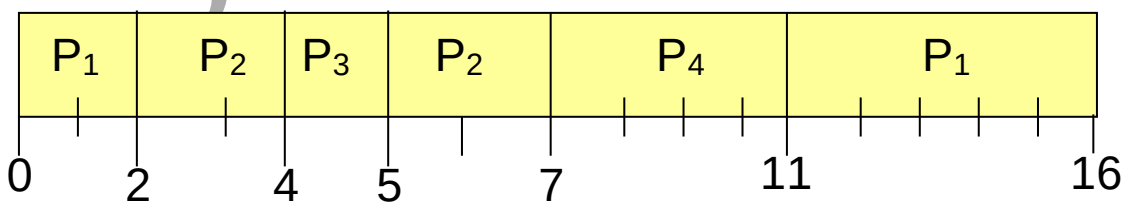
If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



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.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

->SJF (preemptive)



->Average waiting time = (9 + 1 + 0 +2)/4 = 3

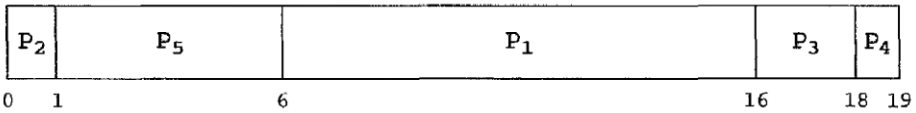
3.3.3 Priority Scheduling

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.

3.3.4 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.

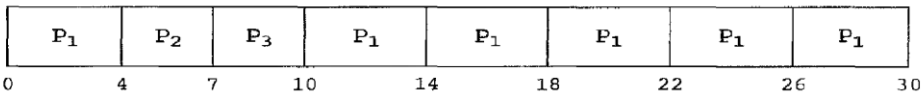
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

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, 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, 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.

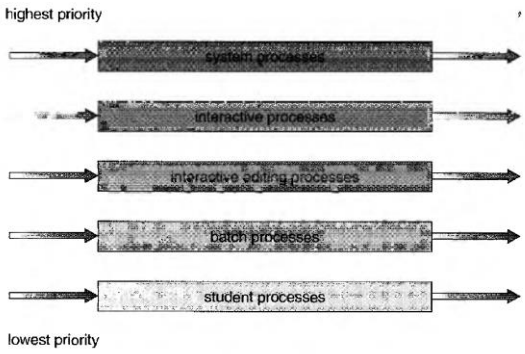
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row . If a process's CPU burst exceeds 1 time quantum, that process is preempted arid is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

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) x 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.

3.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.



In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

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

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

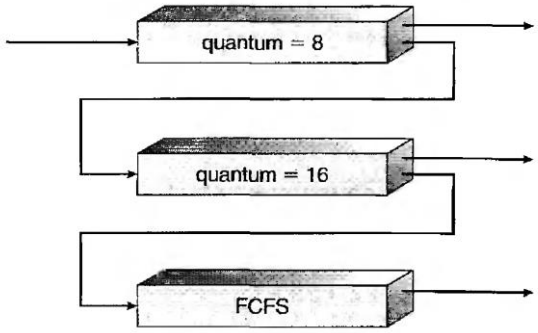
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process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

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

3.3.6 Multilevel Feedback-Queue Scheduling

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

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will 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 run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

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

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

3.4 MULTIPLE-PROCESSOR SCHEDULING

3.4.1 Approaches to Multiple-Processor Scheduling

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

A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

3.4.2 Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor: The data most recently accessed by the process populates the cache for the

processor; and as a result, successive memory accesses by the process are often satisfied in cache memory. Now, 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 attempt to keep a process running on the same processor. This is known as **processor affinity**, meaning that a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so— we have a situation known as soft affinity. Here, it is possible for a process to migrate between processors. Some systems —such as Linux—also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.

3.4.3 Load Balancing

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

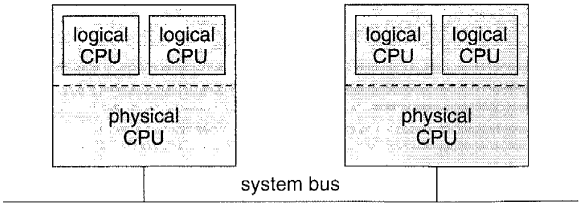
Load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

There are two general approaches to load balancing: push migration and pull migration. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

3.4.4 Symmetric Multithreading

SMP systems allow several threads to run concurrently by providing multiple physical processors. An alternative strategy is to provide multiple logical—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. Furthermore, each logical processor 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 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.



3.5 THREAD SCHEDULING

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

3.5.1 Contention Scope

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.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer. PCS will typically preempt the thread currently running in favor of a higher-priority thread.

3.5.2 Pthread Scheduling

Pthreads identifies the following contention scope values:

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

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations. 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;

• pthread_attr_setscope (pthread_attr_t *attr, int scope)
• pthread_attr_getscope (pthread_attr_t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread_attr_setscope () function is passed either the 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.

Process Synchronization

5.1 Background

Since processes frequently needs to communicate with other processes therefore, there is a need for a well- structured communication, without using interrupts, among processes.

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 the race condition ensure only one process at a time can be manipulating the variable or data. To make such a guarantee processes need to be synchronized in some way

5.2 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. when one process is executing in its critical section, no other process is allowed to execute in its critical section. 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 Pi is shown in Figure

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

General structure of a typical process P_i.

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

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

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

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

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

- **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- **Nonpreemptive kernels**.. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

5.3 Peterson's Solution

A classic software-based solution to the critical-section problem known as **Peterson's solution**. It addresses the requirements of mutual exclusion, progress, and bounded waiting.

- It Is two process solution.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

  int turn;

  Boolean flag[2]

  The variable turn indicates whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready.
- The structure of process *Pi* in Peterson's solution:

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- It proves that
    1. Mutual exclusion is preserved
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

5.4 Synchronization Hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Simple hardware instructions can be used effectively in solving the critical-section problem. These solutions are based on the **locking** —that is, protecting critical regions through the use of locks.

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

Solution to Critical Section problem using locks

- Modern machines provide special atomic hardware instructions

  Atomic = non-interruptable
- Either test memory word and set value(TestAndSet( )) Or swap contents of two memory words(Swap( )).
- The definition of the test and set() instruction

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

- Using test and set() instruction, mutual exclusion can be implemented by declaring a boolean variable lock, initialized to false. The structure of process *Pi* is shown in Figure:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test and set().

- Using Swap() instruction, mutual exclusion can be provided as : A global Boolean variable lock is declared and is initialized to *false* and each process has a local Boolean variable *key*.

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

The definition of the Swap() instruction

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

               //    critical section

      lock = FALSE;

               //      remainder section

} while (TRUE);
```

Mutual exclusion implementation with Swap() instruction

- Test and Set() instruction & Swap() Instruction do not satisfy the bounded-waiting requirement.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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);
```

. Bounded-waiting mutual exclusion with test and set()

5.5 Semaphores

The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. So operating-systems designers build software tools to solve the critical-section problem, and this synchronization tool called as Semaphore.

- Semaphore *S is an* integer variable
- Two standard operations modify S: wait() and signal()
  Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
   while S <= 0
         ; // no-op
    S--;
}
signal (S) {
  S++;
}
```

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.

Usage:

Semaphore classified into:

- Counting semaphore: Value can range over an unrestricted domain.

- Binary semaphore(Mutex locks): Value can range only between from 0 & 1. It provides mutual exclusion.

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
         // remainder section
} while (TRUE);
```

- Consider 2 concurrently running processes:

  *S*1;

  signal(synch);

  In process *P*1, and the statements

  wait(synch);

  *S*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.

Implementation:

The disadvantage of the semaphore is **busy waiting** i.e While a process is in critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process spins while waiting for the lock.

Solution for Busy Waiting problem:

Modify the definition of the wait() and signal()operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which

changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, define a semaphore as follows:

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

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

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

and the signal() semaphore operation can be defined as

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

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

5.6.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes, these processes are said to be deadlocked.

Consider below example: 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**.

5.7 Classic Problems of Synchronization

5.7.1 The Bounded-Buffer Problem:

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Code for producer is given below:

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

Code for consumer is given below:

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

5.7.2 The Readers–Writers Problem

- A data set is shared among a number of concurrent processes
  - ✓ Readers – only read the data set; they do **not** perform any updates
  - ✓ Writers– can both read and write
- Problem – allow multiple readers to read at the same time
  - ✓ Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities.
  - ✓ *First* variation – no reader kept waiting unless writer has permission to use shared object
  - ✓ *Second* variation- Once writer is ready, it performs asap.
- Shared Data
  - ✓ Data set
  - ✓ Semaphore mutex initialized to 1
  - ✓ Semaphore wrt initialized to 1
  - ✓ Integer readcount initialized to 0

The structure of writer process:

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

The structure of reader process:

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

5.7.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Solution: 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 in Figure

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

Several possible remedies to the deadlock problem are replaced by:

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

5.8 Monitors

Incorrect use of semaphore operations:

- 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);
```

- 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.

Solution:

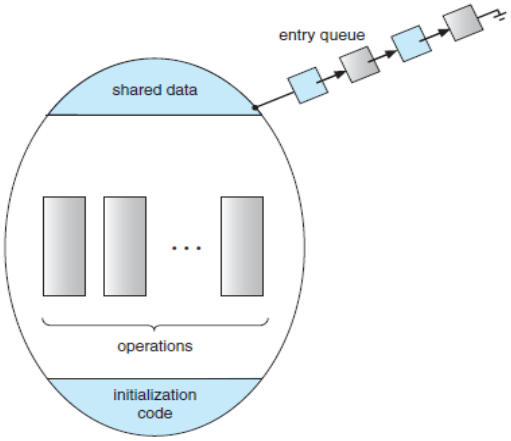
Monitor: An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

The syntax of a monitor type is shown in Figure:

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

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Schematic view of a monitor:



To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition.*

```
condition x, y;
```

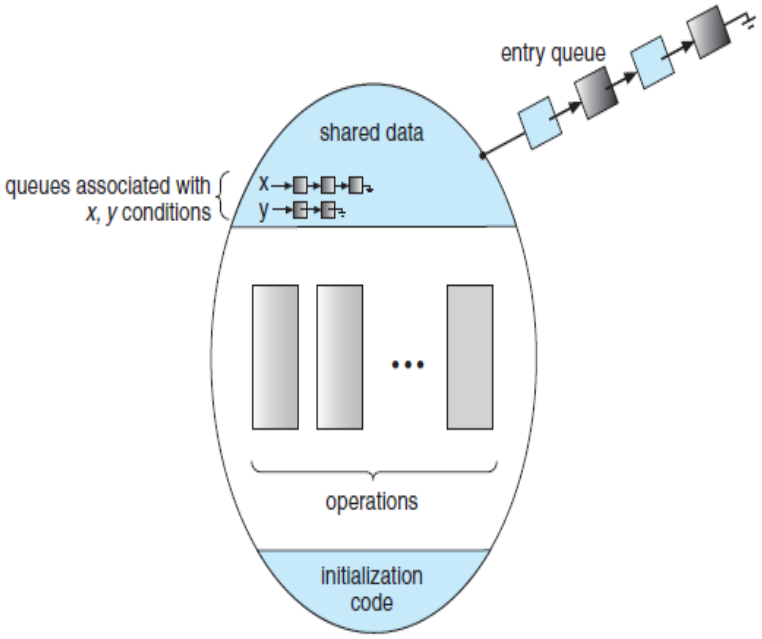
The only operations that can be invoked on a condition variable are wait() and signal(). The operation

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

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

```
x.signal();
```

The x.signal() operation resumes exactly one suspended process. 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 had never been executed. Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.



5.8.2 Dining-Philosophers Solution Using Monitors

A deadlock-free solution to the dining-philosophers problem using monitor concepts. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Consider following data structure:

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

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

And also declare:

```
Condition self[5];
```

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

A monitor solution to the dining-philosopher problem:

```
monitor DiningPhilosophers
  {
     enum { THINKING; HUNGRY, EATING) state [5] ;
     condition self [5];

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

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

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

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

5.8.3 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, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

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

Mutual exclusion within a monitor is ensured.

For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

5.8.4 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has the form

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

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

```
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;
   }
}
```

The ResourceAllocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.

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. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).