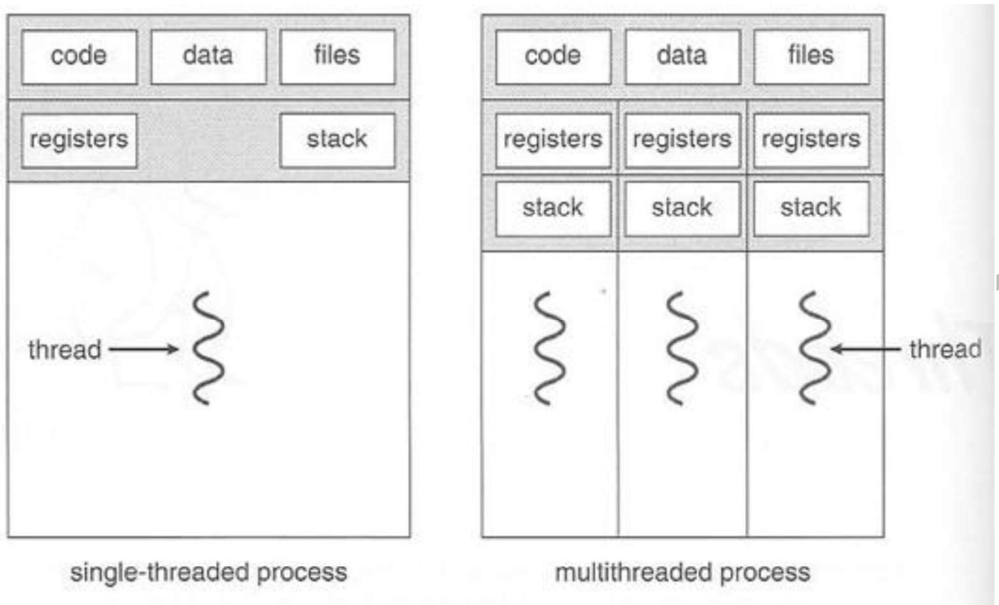
MODULE 2

Chapter 1: Multi threaded programming

1.1 Introduction:

- 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**.



1.1.1 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.

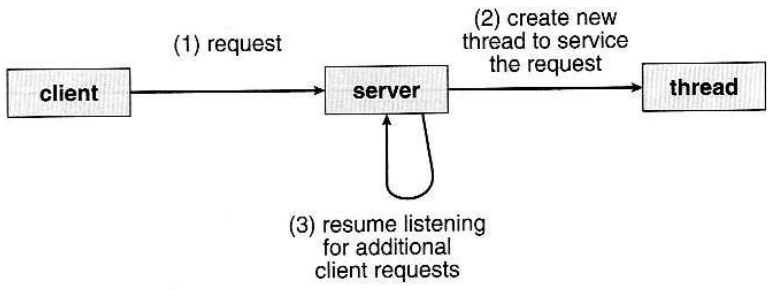

(1) request
client
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Figure 4.2 Multithreaded server architecture.

1.1.2 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.

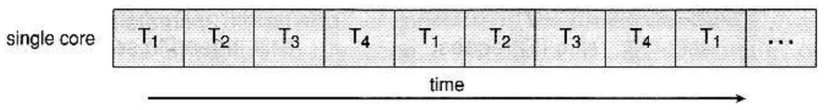


Figure 4.3 Concurrent execution on a single-core system.

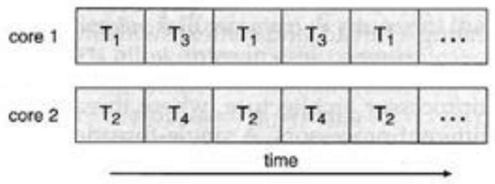

core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

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.

1.2 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.

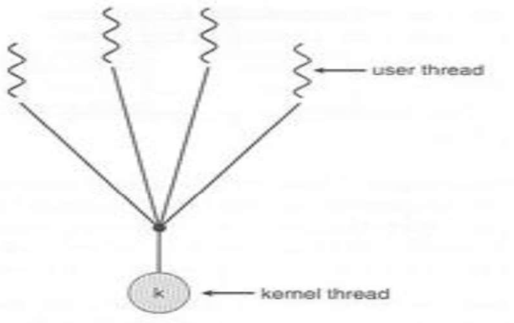


Figure 4.2 Many-to-one model.

- 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.

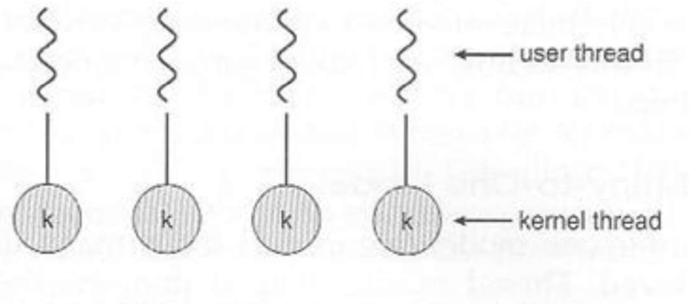


Figure 4.3 One-to-one model.

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.

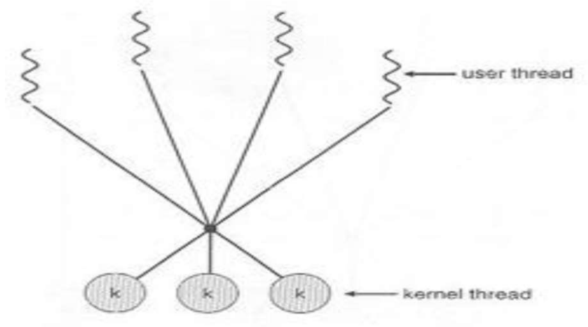


Figure 4.4 Many-to-many model.

- 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.

1.3 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.

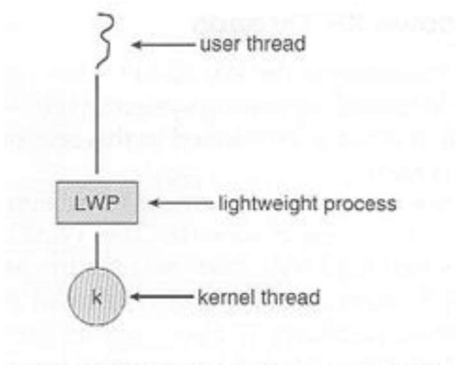


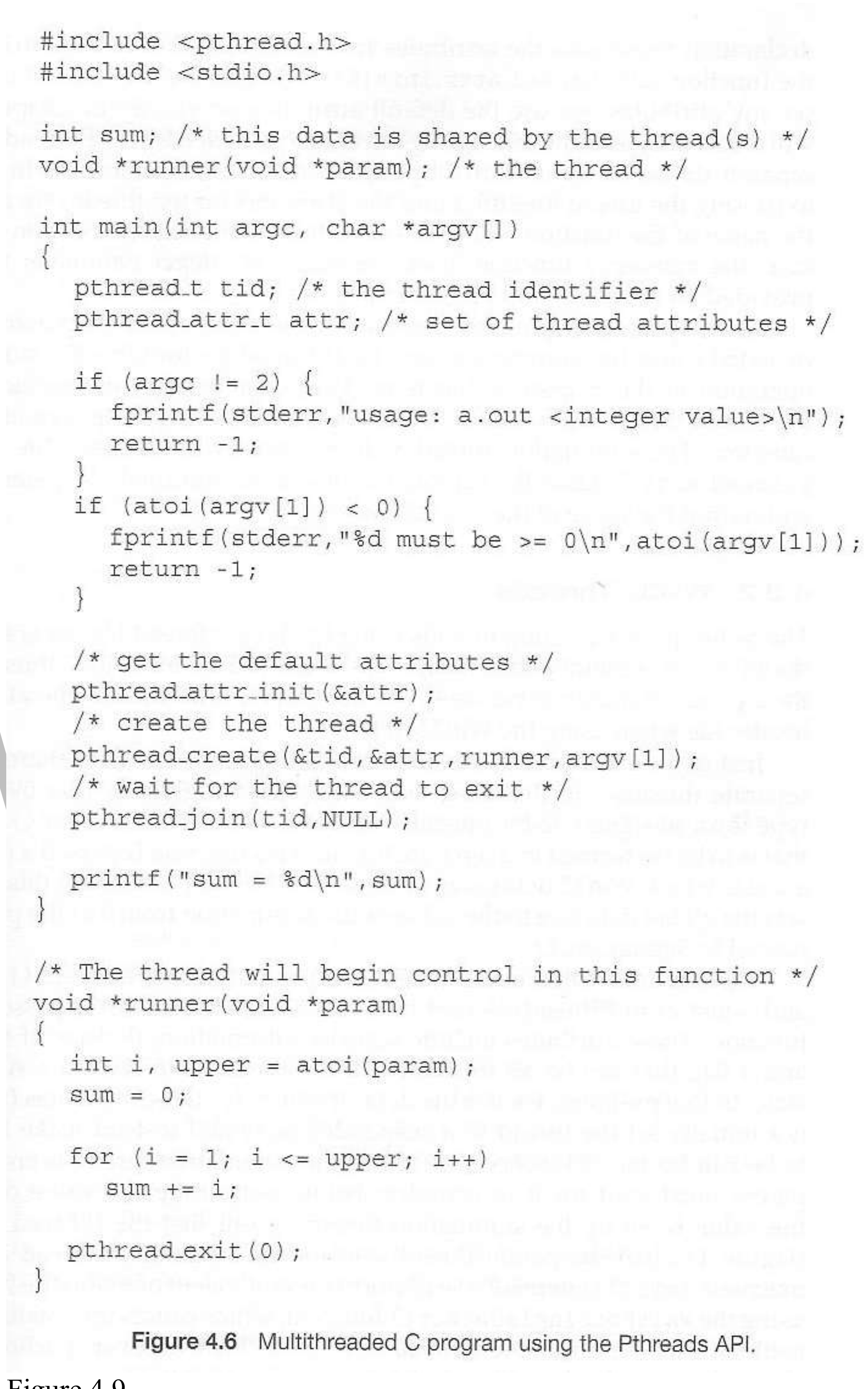
Figure 4.9 Lightweight process (LWP.)

1.4 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".

1.4.1 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

1.4.2 Win32 Threads

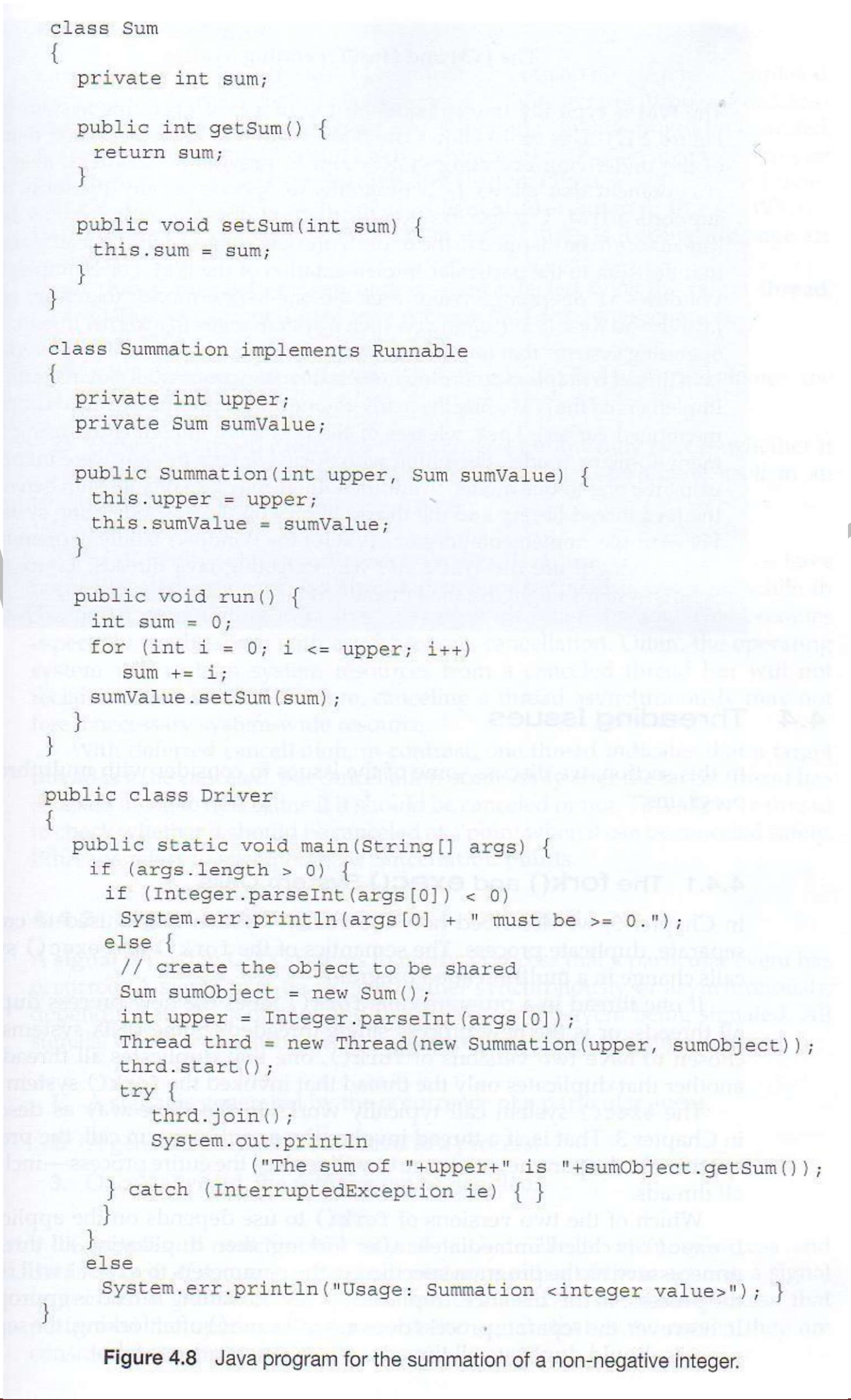
- Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature.
- Here summation() function is used to perform the separate thread function.
- CreateThread() is the function to create a thread.

```
#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.

1.4.3 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).

Chapter 2: CPU SCHEDULING

2.1 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.

2.1.1 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:

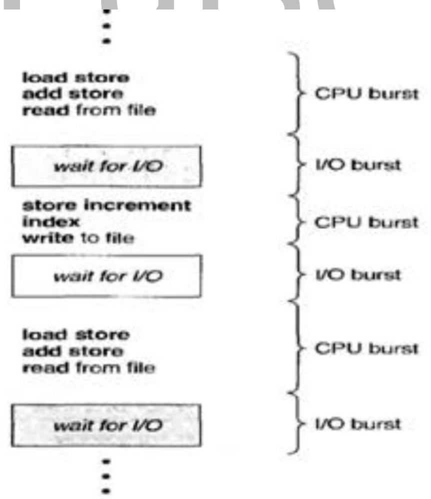


Figure 2.6 Alternating sequence of CPU and I/O bursts

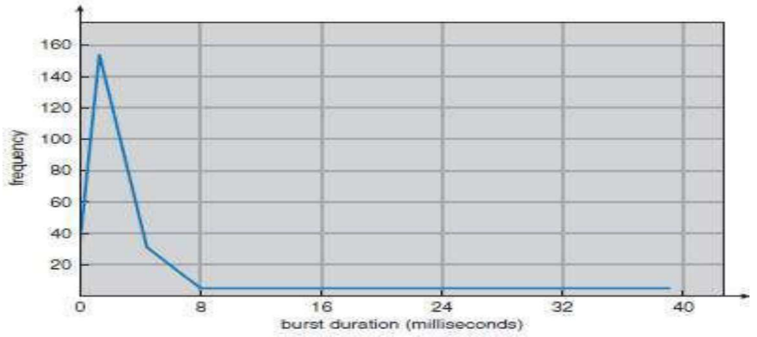


Figure 2.7 Histogram of CPU-burst durations

2.1.2 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.

2.1.3 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 (ioi example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non preemptive or cooperative; otherwise, it is preemptive

2.1.4 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**.

2.1.5 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.

2.3 Scheduling Algorithms

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

2.3.1 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

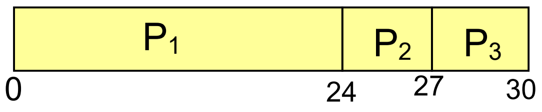
Disadvantages:

- 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:-

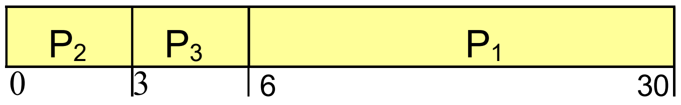
Process	Burst Time
P1	24
P2	3
P3	3

- 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 =

- 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.

2.3.2 Shortest-Job-First Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user,

as the „length'

- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
- Advantage:
  1) The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.
- Disadvantage:
  1) Determining the length of the next CPU burst.
- SJF algorithm may be either 1) non-preemptive or
  2) preemptive.

1) Non preemptive SJF

➢ The current process is allowed to finish its CPU burst.

2) Preemptive SJF

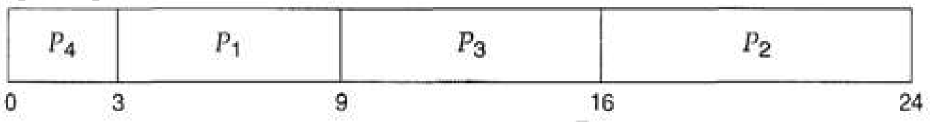
➢ If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted.

➢ It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).

- Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

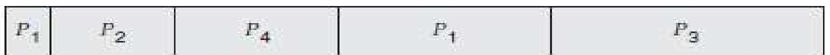
- For non-preemptive SJF, the Gantt Chart is as follows:



- Waiting time for P1 = 3; P2 = 16; P3 = 9; P4=0
  Average waiting time: (3 + 16 + 9 + 0)/4 = 7
- Example (preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

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

- For preemptive SJF, the Gantt Chart is as follows:



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.

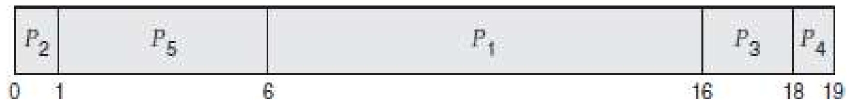
2.3.3 Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.
  1) **Internally-defined** priorities.
     - ➢ Use some measurable quantity to compute the priority of a process.
     - ➢ For example: time limits, memory requirements, no. of open files.
  2) **Externally-defined** priorities.
     - ➢ Set by criteria that are external to the OS
     - ➢ For example:
       - → importance of the process
       - → political factors
- Priority scheduling can be either preemptive or nonpreemptive.
  1) Preemptive
  ➢ The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.
  2) Non Preemptive
  ➢ The new process is put at the head of the ready-queue
- Advantage:
  1) Higher priority processes can be executed first.
- Disadvantage:
  1) Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU.

  Solution: **Aging** is a technique of increasing priority of processes that wait in system for a long time.
- Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time 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

- The Gantt chart for the schedule is as follows:



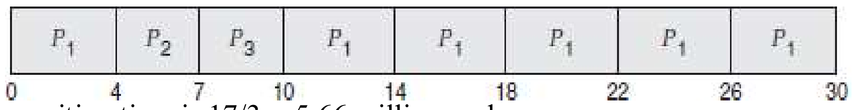
The average waiting time is 8.2 milliseconds.

2.3.4 Round-Robin Scheduling

- Designed especially for timesharing systems.
- It is similar to FCFS scheduling, but with preemption.
- A small unit of time is called a **time quantum** (or time slice).
- Time quantum is ranges from 10 to 100 ms.
- The ready-queue is treated as a **circular queue**.
- The CPU scheduler
  → goes around the ready-queue and
  → allocates the CPU to each process for a time interval of up to 1 time quantum.
- To implement:
  The ready-queue is kept as a FIFO queue of processes
- CPU scheduler
  3) Picks the first process from the ready-queue.
  4) Sets a timer to interrupt after 1 time quantum and
  5) Dispatches the process.
- One of two things will then happen.
  1) The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
  2) 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 OS.
     The process will be put at the tail of the ready-queue.
- Advantage:
  1) Higher average turnaround than SJF.
- Disadvantage:
  1) Better response time than SJF.
- Example: Consider the following set of processes that arrive at time 0, with the length of the CPU- burst time given in milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is as follows:



- The average waiting time is 17/3 = 5.66 milliseconds.
- The RR scheduling algorithm is preemptive.
  No process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready-queue..
- The performance of algorithm depends heavily on the size of the time quantum (Figure 2.8 & 2.9).
  1) If time quantum=very large, RR policy is the same as the FCFS policy.
  2) If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1) Larger the time quantum for a specific process time, less time is spend on context switching.
  2) The smaller the time quantum, more overhead is added for the purpose of context-

switching.

2.3.5 Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 2.10).
- 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.

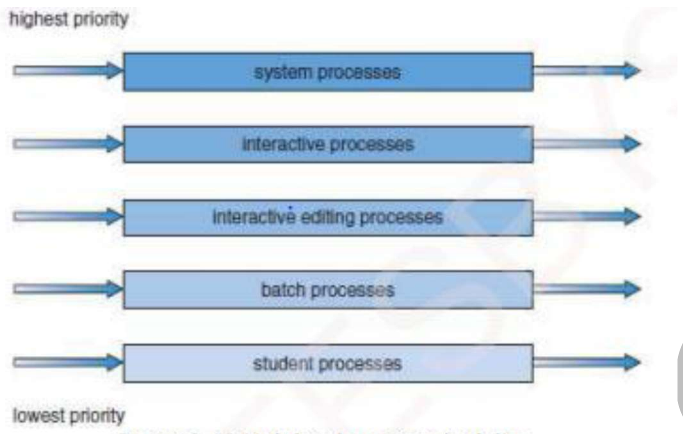


Figure 2.10 Multilevel queue scheduling

- 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.
- **Time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

  20% to background in FCFS
- 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

2.3.6 Multilevel Feedback-Queue Scheduling

- A process may move between queues (Figure 2.11).
- The basic idea:

  Separate processes according to the features of their CPU bursts. For example
  1) 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.
  2) If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue
     ¤ This form of aging prevents starvation.

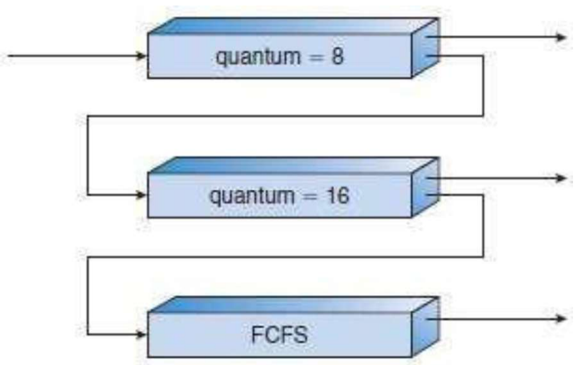


Figure 2.11 Multilevel feedback queues.

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  1) The number of queues.
  2) The scheduling algorithm for each queue.
  3) The method used to determine when to upgrade a process to a higher priority queue.
  4) The method used to determine when to demote a process to a lower priority queue.
  5) The method used to determine which queue a process will enter when that process needs service.

2.4 MULTIPLE-PROCESSOR SCHEDULING

If multiple CPUs are available, the scheduling problem becomes more complex.

2.4.1 Approaches to Multiple-Processor Scheduling

1) **Asymmetric Multiprocessing**
   - ➢ The basic idea is:
     - i) A master server is a single processor responsible for all scheduling decisions, I/O processing and other system activities.
     - ii) The other processors execute only user code.
   - ➢ Advantage:
     - i) This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

2) **Symmetric Multiprocessing**
   - ➢ The basic idea is:
     - i) Each processor is self-scheduling.
     - ii) To do scheduling, the scheduler for each processor
       - i. Examines the ready-queue and
       - ii. Selects a process to execute.
   - ➢ Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

2.4.2 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.

- Two forms:
  - **1) Soft Affinity**
    - ➢ When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.
    - ➢ It is possible for a process to migrate between processors.
  - **2) Hard Affinity**
    - ➢ When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

2.4.3 Load Balancing

- This attempts to keep the workload evenly distributed across all processors in an SMP system.
- Two approaches:
  - **1)Push Migration**
    - ➢ A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.
  - **2)Pull Migration**
    - ➢ An idle processor pulls a waiting task from a busy processor.

2.4.4 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). It has termed hyperthreading technology.

- The basic idea:
    1) Create multiple logical processors on the same physical processor.
    2) Present a view of several logical processors to the OS.
- Each logical processor has its own architecture state, which includes general-purpose and machine- state registers.
- Each logical processor is responsible for its own interrupt handling.
- SMT is a feature provided in hardware, not software.
- 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.

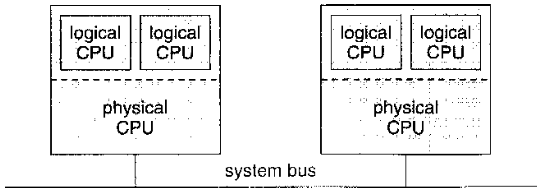


Fig: A typical SMT architecture

2.5 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).

2.5.1 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).
- 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.

2.5.2 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.

Chapter 3: Process Synchronization

3.1Background

- Co-operating process is one that can affect or be affected by other processes.
- Co-operating processes may either
  - share a logical address-space (i.e. code & data) or
  - share data through files or
  - messages through threads.
- Concurrent-access to shared-data may result in data-inconsistency.
- To maintain data-consistency:
- The orderly execution of co-operating processes is necessary.
- Suppose that we wanted to provide a solution to **producer-consumer problem** that fills all buffers. We can do so by having an variable counter that keeps track of the no. of full buffers.

  Initially, counter=0.
  - counter is incremented by the producer after it produces a new buffer.
  - counter is decremented by the consumer after it consumes a buffer.
- **Shared-data:**

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process:

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer Process:

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

- A situation where several processes access & manipulate same data concurrently and the outcome of the execution depends on particular order in which the access takes place, is called a **race condition**.
- Example:

  counter++ could be implemented as: counter— may be implemented as:

$register_1 = \text{counter}$
$register_1 = register_1 + 1$
$\text{counter} = register_1$

$register_2 = \text{counter}$
$register_2 = register_2 - 1$
$\text{counter} = register_2$

- Consider this execution interleaving with counter = 5 initially:

T_0:	*producer*	execute	$register_1 = \text{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \text{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\text{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\text{counter} = register_2$	$\{counter = 4\}$

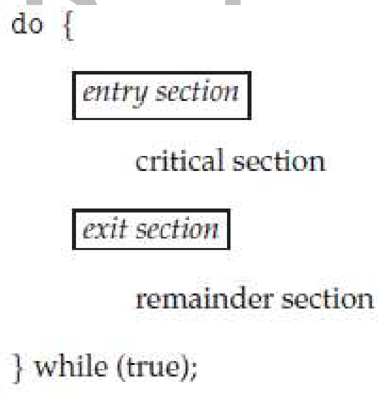
- The value of counter may be either 4 or 6, where the correct result should be 5. This is an example for race condition.

- To prevent race conditions, concurrent-processes must be synchronized.

3.2 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.
- Critical section is a segment-of-code in which a process may be
  → changing common variables
  → updating a table or
  → writing a file.
- Each process has a critical-section in which the shared-data is accessed.
- General structure of a typical process has following (Figure 2.12):
  1) Entry-section
  ➢ Requests permission to enter the critical-section.
  2) Critical-section
  ➢ Mutually exclusive in time i.e. no other process can execute in its critical-section.
  3) Exit-section
  ➢ Follows the critical-section.
  4) 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.

<u>A solution to the critical-section problem must satisfy the following three requirements:</u>

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.

6.3 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.

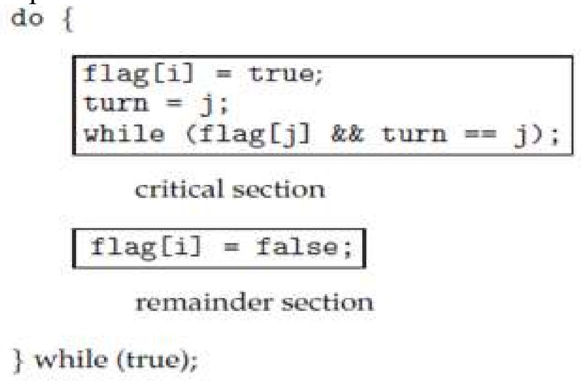
- It Is two process solution.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

  int turn;

  Boolean flag[2];

  where turn = indicates whose turn it is to enter its critical-section.
  (i.e., if turn==i, then process Pi is allowed to execute in its critical-section). flag = used to indicate if a process is ready to enter its critical-section.
  (i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

- 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);
```

- To enter the critical-section,
  - firstly process Pi sets flag[i] to be true and
  - then sets turn to the value j.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- The final value of turn determines which of the 2 processes is allowed to enter its critical-section first.

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

3.4 Synchronization Hardware

Hardware based Solution for Critical-section Problem

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

Hardware instructions for solving critical-section problem

- Modern systems provide special hardware instructions
  → to test & modify the content of a word atomically or
  → to swap the contents of 2 words atomically.
- Atomic-operation means an operation that completes in its entirety without interruption.

TestAndSet()

- 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);
```

Fig: Bounded-waiting mutual exclusion with test and set()

3.5 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.

3.5.1 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.

1) Solution for Critical-section Problem using Binary Semaphores

• Binary semaphores can be used to solve the critical-section problem for multiple processes.
• The „n' processes share a semaphore mutex initialized to 1 (Figure 2.20).

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

Figure 2.20 Mutual-exclusion implementation with semaphores

2) Use of Counting Semaphores

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ 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 the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving Synchronization Problems
- Semaphores can also be used to solve synchronization problems.
- For example, consider 2 concurrently running-processes:

P1 with a statement
S1 and P2 with a
statement S2.

- Suppose we require that S2 be executed only after S1 has completed.
- We can implement this scheme readily

→ by letting P1 and P2 share a common semaphore synch initialized to 0, and
→ by inserting the following statements in

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

process P1 and the following statements in

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

process P2

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

3.5.2 Implementation:
- Main disadvantage of semaphore: Busy waiting.
- **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.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock).
- To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:

1) When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

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

- We assume 2 simple operations:

1) **block()** suspends the process that invokes it.
2) **wakeup(P)** resumes the execution of a blocked process P.

- We define a semaphore as follows:

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

- **Definition of wait():**

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

Definition of signal():

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

- This (critical-section) problem can be solved in two ways:
    1) In a **uni-processor** environment
        ¤ Inhibit interrupts when the wait and signal operations execute.
        ¤ Only current process executes, until interrupts are re-enabled & the scheduler regains control.
    2) In a **multiprocessor** environment
        ¤ Inhibiting interrupts doesn't work.
        ¤ Use the hardware / software solutions described above.

3.5.3.Deadlocks & Starvation

- Deadlock occurs when 2 or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- The event in question is the execution of a signal() operation.
- To illustrate this, consider 2 processes, Po and P1, each accessing 2 semaphores, S and Q. Let S and Q be initialized to 1.

```
   P0           P1

wait(S);      wait(Q);
wait(Q);      wait(S);
   .             .
   .             .
   .             .
signal(S);    signal(Q);
signal(Q);    signal(S);
```

- Suppose that Po executes wait(S) and then P1 executes wait(Q).
  When Po executes wait(Q), it must wait until P1 executes signal(Q).
  Similarly, when P1 executes wait(S), it must wait until Po executes signal(S).
  Since these signal() operations cannot be executed, Po & P1 are deadlocked.
- Starvation (indefinite blocking) is another problem related to deadlocks.
- **Starvation** is 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.

3.6 Classic Problems of Synchronization

1) Bounded-Buffer Problem
2) Readers and Writers Problem
3) Dining-Philosophers Problem

3.6.1The Bounded-Buffer Problem

- The bounded-buffer problem is related to the producer consumer problem.
- There is a pool of n buffers, each capable of holding one item.
- **Shared-data**

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

where,
¤ mutex provides mutual-exclusion for accesses to the buffer-pool.
¤ empty counts the number of empty buffers.
¤ full counts the number of full buffers.

- The symmetry between the producer and the consumer.
  - ¤ The producer produces full buffers for the consumer.
  - ¤ The consumer produces empty buffers for the producer.
- **Producer Process:**

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

Consumer Process:

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

3.6.2 The Readers–Writers Problem

- A data set is shared among a number of concurrent processes.
- **Readers** are processes which want to only read the database (DB).
  Writers are processes which want to update (i.e. to read & write) the DB.
- Problem:
  - ➢ Obviously, if 2 readers can access the shared-DB simultaneously without any problems.
  - ➢ However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.

  Solution:
  - ➢ The writers must have exclusive access to the shared-DB while writing to the DB.
- **Shared-data**

```
semaphore mutex, wrt;
int readcount;
```

where,
- ¤ mutex is used to ensure mutual-exclusion when the variable readcount is updated.
- ¤ wrt is common to both reader and writer processes.
  wrt is used as a mutual-exclusion semaphore for the writers.
  wrt is also used by the first/last reader that enters/exits the critical-section.
- ¤ readcount counts no. of processes currently reading the object.

Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process:

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

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);
```

- The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
- The mode of lock needs to be specified:
  1) read mode

➢ When a process wishes to read shared-data, it requests the lock in read mode.

2) write mode

➢ When a process wishes to modify shared-data, it requests the lock in write mode.

- Multiple processes are permitted to concurrently acquire a lock in read mode, but only one process may acquire the lock for writing.
- These locks are most useful in the following situations:
  1) In applications where it is easy to identify
     → which processes only read shared-data and
     → which threads only write shared-data.
  2) In applications that have more readers than writers.

3.6.3 The Dining-Philosophers Problem

- Problem statement:
  ➢ There are 5 philosophers with 5 chopsticks (semaphores).
  ➢ A philosopher is either eating (with two chopsticks) or thinking.
  ➢ The philosophers share a circular table (Figure 2.21).
  ➢ The table has
     → a bowl of rice in the center and
     → 5 single chopsticks.
  ➢ From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
  ➢ A philosopher may pick up only one chopstick at a time.
  ➢ Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
  ➢ When 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.
- Problem objective:
  To allocate several resources among several processes in a deadlock-free & starvation-free manner.
- Solution:
  ➢ Represent each chopstick with a semaphore (Figure 2.22).
  ➢ A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
  ➢ The philosopher releases her chopsticks by executing the signal() on the semaphores.
  ➢ This solution guarantees that no two neighbors are eating simultaneously.
  ➢ **Shared-data**
     semaphore chopstick[5];
  Initialization
     chopstick[5]={1,1,1,1,1}.

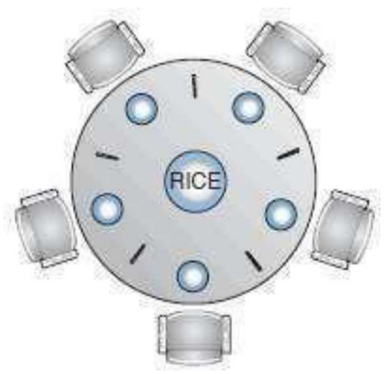
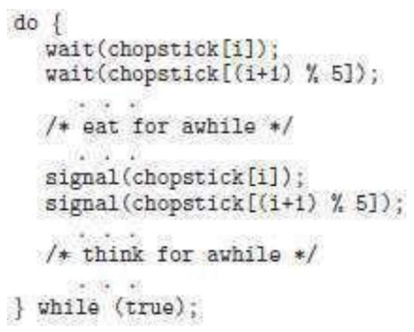


Figure 2.21 Situation of dining philosophers



```
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);
```

Figure 2.22 The structure of philosopher

- Disadvantage:

- Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Three possible remedies to the deadlock problem:
  1) Allow **at most 4** philosophers to be sitting simultaneously at the table.
  2) Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
  3) Use an **asymmetric solution**; i.e. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

3.7 Monitors

- **Monitor** is a high-level synchronization construct.
- It provides a convenient and effective mechanism for process synchronization.

Need for Monitors

- When programmers use semaphores incorrectly, following types of errors may occur:
  1) 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.
  2) 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.
  3) 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.

3.7.1 Usage

- A **monitor type** presents a set of programmer-defined operations that are provided to ensure mutual-exclusion within the monitor.
- It also contains (Figure 2.23):
  → declaration of variables
  → bodies of procedures(or functions).
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal-parameters.
  Similarly, the local-variables of a monitor can be accessed by only the local-procedures.

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

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

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

         .
         .
         .

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

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

Figure 2.23 Syntax of a monitor

- Only one process at a time is active within the monitor (Figure 2.24).
- To allow a process to wait within the monitor, a condition variable must be declared, as `condition x, y;`
- Condition variable can only be used with the following 2 operations (Figure 2.25):

1) x.signal()

➢ This operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

2) x.wait()

➢ The process invoking this operation is suspended until another process invokes x.signal().

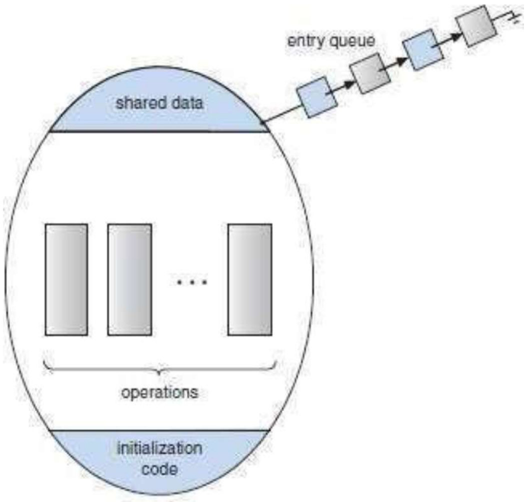


Figure 2.24 Schematic view of a monitor

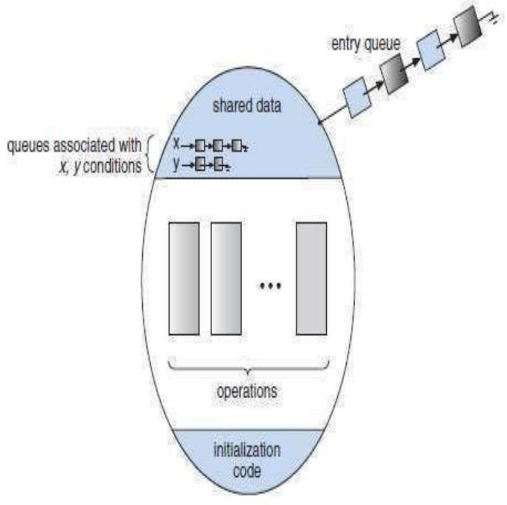


Figure 2.25 Monitor with condition variables

- Suppose when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Both processes can conceptually continue with their execution. Two possibilities exist:

1) Signal and wait

➢ P either waits until Q leaves the monitor or waits for another condition.

2) Signal and continue

➢ Q either waits until P leaves the monitor or waits for another condition.

3.7.2 Dining-Philosophers Solution Using Monitors

- The restriction is
  - ➢ A philosopher may pick up her chopsticks only if both of them are available.
- Description of the solution:
  1. The distribution of the chopsticks is controlled by the monitor dp (Figure 2.26).
  2. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process.
  3. After the successful completion of the operation, the philosopher may eat.
  4. Following this, the philosopher invokes the putdown() operation.
  5. Thus, philosopher i must invoke the operations pickup() and putdown() in the

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

following sequence:

```
monitor dp
{
   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 til  = THINKING;
      test((i + 4)  % 5} ;
      test( (i + 1)  % 5) ;
   }

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

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

Figure 2.26 A monitor solution to the dining-philosopher problem

3.7.3 Implementing a Monitor using Semaphores

- A process
  - → must execute wait(mutex) before entering the monitor and
  - → must execute signal(mutex) after leaving the monitor.
- Variables used:

  semaphore mutex; //
  (initially = 1) semaphore
  next; //
  (initially = 0) int next-count

= 0;
where
¤ mutex is provided for each monitor.
¤ next is used a signaling process to wait until the resumed process either leaves or waits
¤ next-count is used to count the number of processes suspended

- 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.

How condition variables are implemented ?
For each condition variable x, we have: semaphore x-sem; // (initially = 0) int x-count = 0;

- **Definition of x.wait()**

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

Definition of x.signal()

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

3.7.4 Resuming Processes within a Monitor

- Problem:
  If several processes are suspended, then how to determine which of the suspended processes should be resumed next?

Solution-1: Use an FCFS ordering i.e. the process that has been waiting the longest is resumed first. Solution-2: Use conditional–wait construct i.e. x.wait(c)
¤ c is a integer expression evaluated when the wait operation is executed (Figure 2.27).
¤ Value of c (a priority number) is then stored with the name of the process that is suspended.
¤ When x.signal is executed, process with smallest associated 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;
   }
}
```

Figure 2.27 A monitor to allocate a single resource

- ResourceAllocator monitor controls the allocation of a single resource among competing processes.

- Each process, when requesting an allocation of the resource, 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.
- 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.