MODULE – II
Multi-Threading Programming & Process Synchronization

Threads

- Process is a program that performs a single thread of execution.
- For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

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

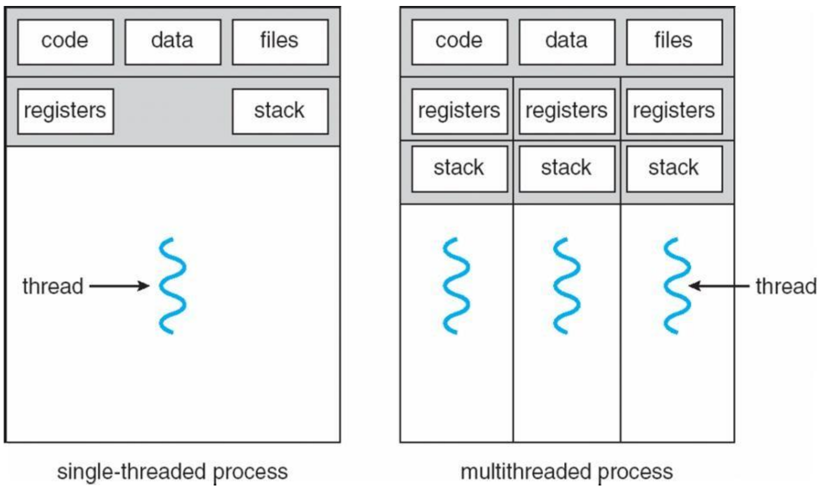


Figure 2.14 Single-threaded and multithreaded processes.

Benefits of multithreaded programming

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

- **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance of a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
- **Resource sharing**. Processes may only share resources through techniques such as shared memory or message passing. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability**. The benefits of multithreading can be greatly increased in a multiprocessor

architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multi CPU machine increases parallelism.

Multithreading Models

Support for threads may be provided either at the user level, for **user threads** or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Ultimately, a relationship must exist between user threads and kernel threads.

1. Many to One Model

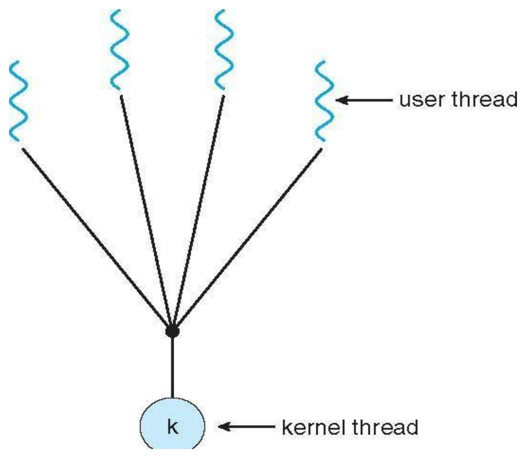


Figure 2.15Many-to-one model.

- The many-to-one model (Figure 2.15) maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user One to One Model space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to rum in parallel on multiprocessors.
- Example: Solaris

2. One to One Model

- The one-to-one model (Figure 2.16) maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.

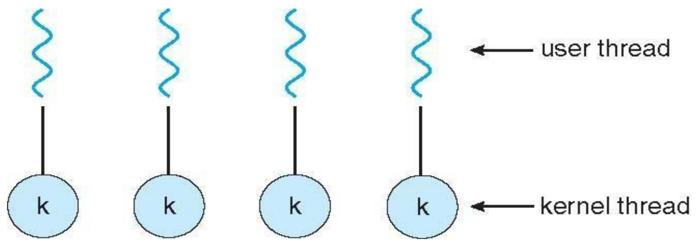


Figure 2.16 One-to-one model

3. **Many to Many Model**
    - The many-to-many model (Figure 2.17) multiplexes many user-level threads to smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.
    - Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
    - The many-to-many model **suffers** from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

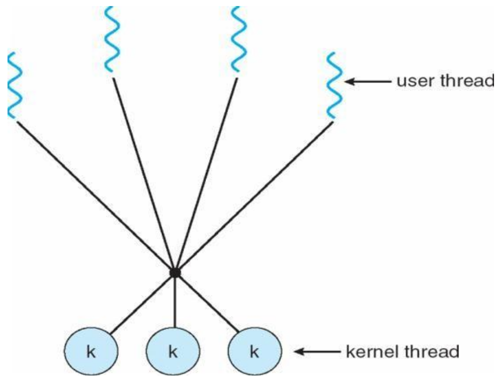


Figure 2.17 Many-to-many model.

Threading Issues

1. **The fork() and exec() System Calls**
    - If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
    - If a thread invokes the exec() system call, the program specified in the parameter to exec () will replace the entire process-including all threads.

2. **Cancellation**
    - Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
    - A thread that is to be canceled is often referred to as the **target thread**.
    - Cancellation of a target thread may occur in two different scenarios:
        - **Asynchronous cancellation**. One thread immediately terminates the target thread.

  - **Deferred cancellation.** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.

3. **Signal Handling**
   - A signal is used in UNIX systems to notify a process that a particular event has occurred. All signals, whether synchronous or asynchronous, follow the same pattern:
     - A signal is generated by the occurrence of a particular event.
     - A generated signal is delivered to a process.
     - Once delivered, the signal must be handled.
   - Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated.
   - Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action can be overridden by a **user defined signal handler** that is called to handle the signal.
   - Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?
   - In general the following options exist:
     - Deliver the signal to the thread to which the signal applies.
     - Deliver the signal to every thread in the process.
     - Deliver the signal to certain threads in the process.
     - Assign a specific thread to receive all signals for the process.

4. **Thread Pools**
   - The first **issue** concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.
   - The second **issue** is more troublesome: if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.
   - The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work.
   - Thread pools offer these benefits:
     - Servicing a request with an existing thread is usually faster than waiting to create a thread.
     - A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

5. **Thread-Specific Data**
   - Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming.
   - However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread specific data**.
   - For example, in a transaction-processing system, we might service each transaction in a

separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

6. Scheduler Activations

- *A* final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models.
- Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.
- This can be achieved with the help of scheduler activations. Many systems implementing either the many-to-many or two-level model place an intermediate data structure between the user and kernel threads.
- This data structure—typically known as a lightweight process, or LWP.
- The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
- Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall.**
- Upcalls are handled by the thread library with an **upcall handler,** and upcall handlers must run on a virtual processor.
- One event that triggers an upcall occurs when an application thread is about to block.

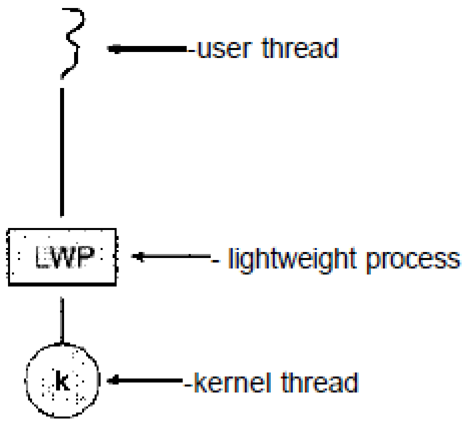
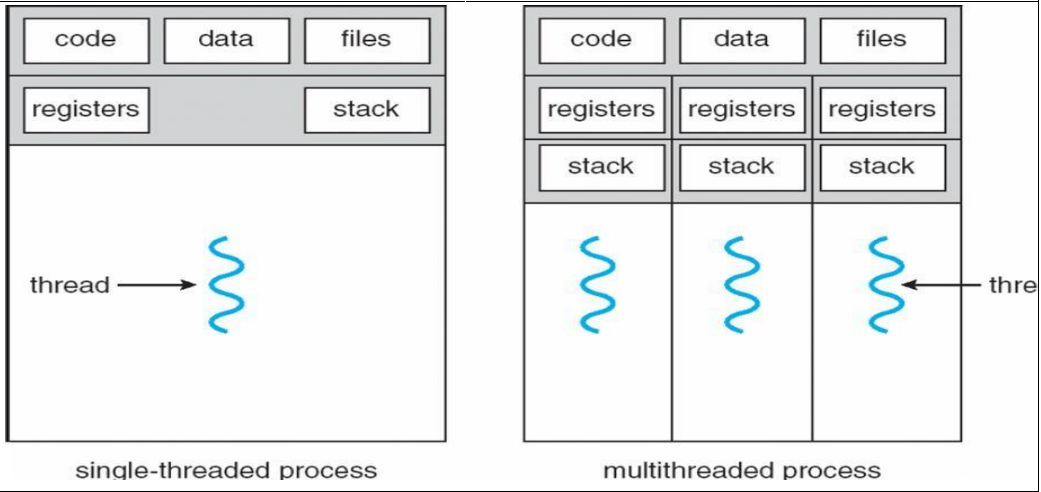


Figure 4.9 Lightweight process (LWP.)

Difference between Process and Thread

S. N.	Pro cess	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.
	 code data files registers stack thread single-threaded process code data files registers registers registers stack stack stack thre multithreaded process	

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Process Scheduling

5.1.1 CPU-I/O Burst Cycle:

The success of CPU scheduling depends on an observed property of processes: Process execution consists of a **cycle** of CPU execution and I/O wait.
Processes alternate between these two states.
Process execution begins with a CPU **burst.**
That is followed by an **I/O burst,** which is followed by another CPU burst, then another I/O burst, and so on.
Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

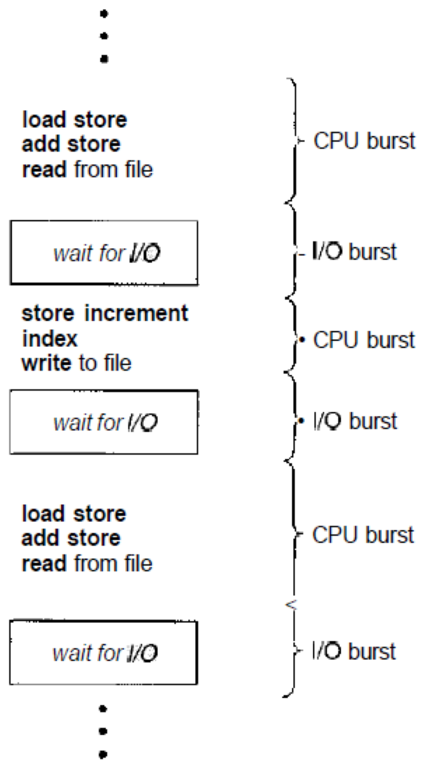


Figure 5.1 Alternating sequence of CPU and I/O bursts.

5.1.2 CPU Scheduler

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

5.1.3 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/0 request or an invocation of wait for the termination of one of the child processes)
    2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
    3. When a process switches from the waiting state to the ready state (for example, at

completion of I/0)

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**.

5.1.4 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.

5.2 Scheduling Criteria

Many criteria have been suggested for comparing CPU-scheduling algorithms.

- **CPU utilization**. We want to keep the CPU as busy as possible. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time**. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. *Waiting time* is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Hence another measure response time used, which is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

5.3 Scheduling Algorithms

1. **First-Come, First-Served Scheduling**

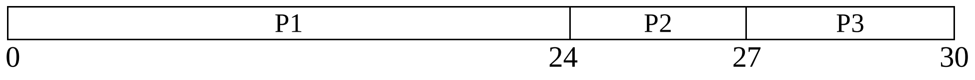
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- On the **negative side**, the average waiting time under the FCFS policy is often quite long.
- FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is

important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

- Example:

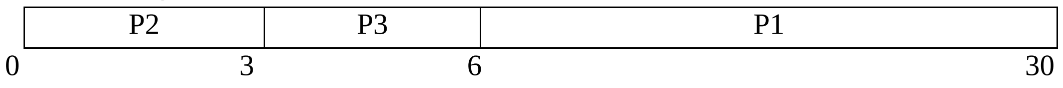
Process	Burst Time
P1	24
P2	3
P3	3

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



Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	0	0 + 24 = 24
P2	24	24 + 3 = 27
P3	27	27 + 3 = 30

- Average Waiting Time = (0 + 24 + 27) / 3 = 17 milliseconds.
- Average Turnaround Time = (24 + 27 + 30) / 3 = 27 milliseconds.
- If the processes arrive in the order P2, *P3* , P1, however, the results will be as shown in the following Gantt chart:



Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	6	6 + 24 = 30
P2	0	0 + 3 = 3
P3	3	3 + 3 = 6

- Average Waiting Time = (6 + 0 + 3) / 3 = 3 milliseconds.
- Average Turnaround Time = (24 + 27 + 30) / 3 = 27 milliseconds.

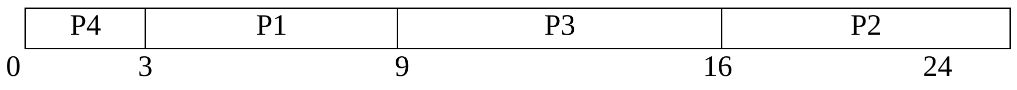
2. Shortest-Job-First Scheduling (Non Preemptive SJF)

- 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.
- The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

- Example:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

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



Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	3	3 + 6 = 9
P2	16	16 + 8 = 24
P3	9	9 + 7 = 16
P4	0	0 + 3 = 3

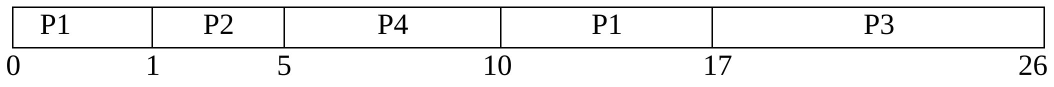
- Average Waiting Time = (3 + 16 + 9 + 0) /4 = 7 milliseconds.
- Average Turnaround Time = (9 + 24 + 16 + 3)/ 4 = 13 milliseconds.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

3. Shortest-Remaining-Time-First scheduling - SRTF (Preemptive SJF)

- A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. **Preemptive SJF** scheduling is sometimes called **shortest-remaining-time-first scheduling**.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- Preemptive SJF schedule is as depicted in the following Gantt chart:



Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	10 – 1 - 0 = 9	9 + 8 = 17
P2	1 - 1 = 0	0 + 4 = 4
P3	17 - 2 = 15	15 + 9 = 24
P4	5 - 3 = 2	2 + 5 = 7

- The average waiting time (9 + 0 + 15 + 2) / 4 = 26 / 4 = 6.5 milliseconds.
- The average turnaround time is (17 + 4 + 24 + 7) / 4 = 52 / 4 = 13 milliseconds.

- Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

4. 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.
- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, P3, P4, P5with the length of the CPU burst given in milliseconds:

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

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

P2	P5	P1	P3	P4

0 1 6 16 18 19

Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	6	6 + 10 =16
P2	0	0 + 1 = 1
P3	16	16 + 2 = 18
P4	18	18 + 1 = 19
P5	1	1 + 5 = 6

- The average waiting time is (6+0+16+18+1) / 5 = 8.2 milliseconds.
- The Average Turnaround Time = (16 + 1 + 18 + 19 + 6) / 5 =
- Priority scheduling can be either preemptive or non-preemptive.
- 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.
- 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. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by1 every 15 minutes.

5. 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 enable the system 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 in length.
- 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.
- The average waiting time under the RR policy is often long.
- Example:

Process	Burst Time
P1	24
P2	3
P3	3

- If we use a time quantum of 4 milliseconds, then process P1 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 *P2.* Process *P2* does not need 4 milliseconds, so it quits before its time quantum expires and so on.
- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum.
- The resulting RR schedule is as follows:

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

Process	Waiting Time	Turnaround Time = (Waiting Time + Burst Time)
P1	10 - 4 = 6	6 + 24 =30
P2	4	4 + 3 = 7
P3	7	7 + 3 = 10

- P1 waits for 6 milliseconds (10- 4), *P2* waits for 4 milliseconds, and *P3* waits for 7 milliseconds.
- The average waiting time is (6+4+7) = 17/3 = 5.66 milliseconds.
- The average turnaround time is now (30 + 7 + 10)/3 = 15.66 milliseconds.
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.
- The performance of the RR algorithm depends heavily on the size of the time quantum.
- At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1 / n the speed of the real processor.

6. Multilevel Queue Scheduling

- *A* multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 2.19).

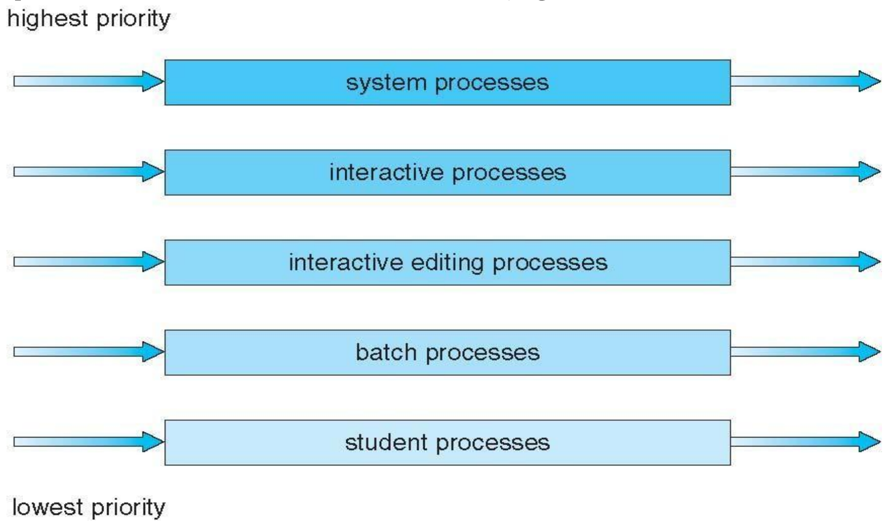


Figure 2.19 Multilevel Queue Schedule

- 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.
- 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.

7. Multilevel Feedback Queue Scheduling

- The multilevel feedback queue scheduling algorithm allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue.
- 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 2.20).

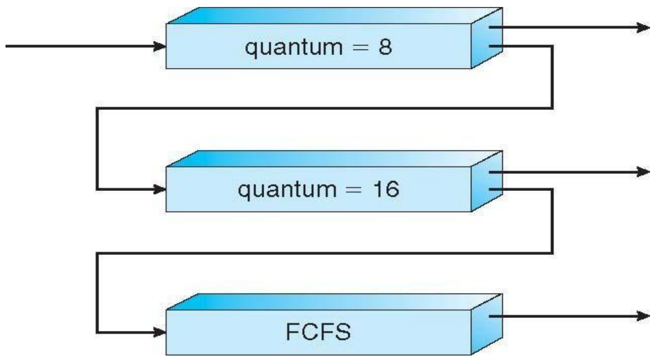


Figure 2.20 Multilevel Feedback Queue

- 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.

Process Synchronization

3.1 Introduction

The original pseudo code for producer consumer problem is,

```
while (true) {
  /* Produce an item */
      while (((in + 1) % BUFFER SIZE)  == out)
             ;   /* do nothing -- no free buffers */
            buffer[in] = item;
            in = (in + 1) % BUFFER SIZE;
}
```

Figure 3.1 Producer Process code

```
while (true) {
       while (in == out)
            ; // do nothing -- nothing to consume
             // remove an item from the buffer
             item = buffer[out];
             out = (out + 1) % BUFFER SIZE;
         return item;
}
```

Figure 3.2 Consumer Process code

This solution allows at most BUFFER_SIZE-1 items in the buffer at the same time. To remove this deficiency we can add an integer variable counter initialized to 0.Counter is incremented every time we add (produce) new item into buffer and decremented every time we remove (consume) an item from buffer. The modified code is as shown below.

```
while (true) {
  /* Produce an item */
      while (counter == BUFFER_SIZE)
             ;   /* do nothing -- no free buffers */
            buffer[in] = item;
            in = (in + 1) % BUFFER SIZE;
            counter++ ;
}
```

Figure 3.3 Producer Process code

```
while (true) {
       while (counter == 0)
            ; // do nothing -- nothing to consume
             // remove an item from the buffer
             item = buffer[out];
             out = (out + 1) % BUFFER SIZE;
             counter-- ;
         return item;
}
```

Figure 3.4 Consumer Process code

The above codes work correct if executed separately, but may not work correct if executed concurrently.

Assume the current counter value is 5 and the producer and consumer executes the statements counter++ and counter-- concurrently.

The statement counter ++ is implemented in machine level language as follows,

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter= $register_1$

And statement counter-- is implemented in machine level language as follows,

$register_2$ = counter
$register_2$ = $register_2$ -1
counter= $register_2$

Where register1 and register2 are local CPU registers.

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is,

T0: producer execute	$register_1$=counter	{ $register_1$= 5}
T1: producer execute	$register_1$= $register_1$+ 1	{ $register_1$= 6}
T2: consumer execute	$register_2$ = counter	{ $register_2$ = 5}
T3: consumer execute	$register_2$ = $register_2$ - 1	{ $register_2$ = 4}
T4: producer execute	counter= $register_1$	{ counter = 6}
T5: consumer execute	counter = register2	{ counter = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter== 6".

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 r**ace condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

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.
- 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 3.5.

```
do{
        Entry section
                    Critical section
        Exit section
                    Remainder section
}while (TRUE);
```

Figure 3.5 General structure of Process Pi

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.

3.3 Peterson's Solution

- Peterson's solution is a classic **software-based** solution to the critical-section problem.
- Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are **no guarantees** that Peterson's solution will work correctly on such architectures.
- Peterson's solution is restricted to **two** processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0and P1. For convenience, when presenting Pi, we use Pjto denote the other process; that is, j equals 1-i.
- Peterson's solution requires the two processes to share two data items:

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

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.
- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

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

                Critical section

        flag[i] = FALSE;

                Remainder section
} while (TRUE);
```

Figure 3.6 The structure of process A in Peterson's solution.

- We need to show that:
  - Mutual exclusion is preserved.
  - The progress requirement is satisfied.
  - The bounded-waiting requirement is met.
- To prove property 1, we note that each P; enters its critical section only if either flag [j] == false or turn == i. Also we note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true. Since the value of turn can be either 0 or 1

but cannot be **both**, p0 and p1 cannot execute there while loop successfully. Hence one process enter the critical section other is waiting in the while loop.

- To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn == j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag [j] ==false, and Pi can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn== j, then Pj will enter the critical section. However, once Pj exits its critical section, it will reset flag [j] to false, allowing Pi to enter its critical section. If Pj resets flag [j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (**progress**) after at most one entry by Pj (**bounded waiting**).

3.4 Synchronization Hardware

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical-section problem requires a simple tool-**a lock.**
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in Figure 3.7.

```
do{
    Acquire Lock

                Critical section

    Release Lock

                Remainder section

}while (TRUE);
```

Figure 3.7 Solution to critical section problem using Locks

- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- Unfortunately, this solution is not as feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.
- Many modern computer systems therefore provide special hardware instructions such as **TestAndSet ()** and **Swap()**, that allow us either to test and modify the content of a word or to swap the contents of two words automatically.
- The TestAndSet () instruction can be defined as shown in Figure 3.8.

```
booleanTestAndSet(boolean *target) {
        booleanrv = *target;
        *target = TRUE;
        returnrv;
    }
```

Figure 3.8 The definition of the TestAndSet () instruction.

- The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown in Figure 3.9.

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

                //critical section
        lock = FALSE;

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

Figure 3.9Mutual-exclusion implementation with TestAndSet ().

- The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown in Figure 3.10.

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

Figure 3.10 The definition of the Swap () instruction.

- If the machine supports the swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable **lock** is declared and is initialized to false and each process has a local Boolean variable **key**. The structure of process Pi is shown in Figure 3.11.

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

                //critical section
        lock = FALSE;
                //remainder section
} while (TRUE);
```

Figure 3.11 Mutual-exclusion implementation with the Swap() instruction.

- Figure 3.12, shows the algorithm using TestAndSet () instruction that **satisfies all the critical-section requirements**.
- The common data structures are,

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

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

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

            //critical section

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

            //remainder section

    } while (TRUE);
```

Figure 3.12 Bounded-waiting mutual exclusion with TestAndSet ().

- To prove that the **mutual exclusion** requirement is met, we note that process Pi can enter its critical section only if either waiting [i] == false or key == false. The value of key can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet () will find key== false; all others must wait.
- The **progress** requirement is met, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.
- To prove that the **bounded-waiting** requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i+ 2, ...,n-1, 0, ..., i -1). It designates the first process in this ordering that is in the entry section (waiting[j]==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

- Unfortunately for hardware designers, implementing atomic TestAndSet() instructions on multiprocessors is **not a trivial task**.

3.5 Semaphores

- The hardware-based solutions to the critical-section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called semaphore.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () or P( ) and signal( ) or V( ).

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

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

Figure 3.13 wait( ) and signal( ) method definition.

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait (S), the testing of the integer value of S (S <= 0), as well as its possible modification (S--), must be executed without interruption.

3.5.1 Semaphores Usage

- Operating systems often distinguish between **counting** and **binary** semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- On some systems, binary semaphores are known as **mutex** locks, as they are locks that provide mutual exclusion.
- We can use binary semaphores to deal with the critical-section problem £or multiple processes. Then processes share a semaphore, mutex, initialized to 1.
- Each process *Pi* is organized as shown in Figure 3.14.

```
do {
        wait (mutex) ;
                //critical section
        signal(mutex);
                //remainder section
} while (TRUE);
```

Figure 3.14 Mutual-exclusion implementation with semaphores.

3.5.2 Semaphores Implementation

- The main **disadvantage** of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **Spinlock**.
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can **block**itself using **block()** operation.
- 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.
- To implement semaphores under this definition, we define a semaphore asa "C' struct:

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

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

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

- The signal () semaphore operation can now be defined as

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

- The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- Note that in this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait () operation.

Advantages of Semaphores

- Simple to implement
- Machine independent.
- Correctness cane be determined easily

3.6 Classical Problems of Synchronization

3.6.1. Bounded Buffer (Producer Consumer) Problem

- The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We assume that the pool consists of *n* buffers, each capable of holding one item.
- The **mutex** semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The **empty** and **full** semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- The code for the producer process is shown in Figure 3.15; the code for the consumer process is shown in Figure 3.16.

```
do {
        //produce an item in nextp
        …
        wait(empty);
        wait(mutex);
        …
        //add nextp to buffer
        …
        signal(mutex);
        signal(full);
} while (TRUE);
```

Figure 3.15 producer process

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

Figure 3.15 Consumer process

3.6.2. Readers Writers Problem

- Suppose that a database is to be **shared** among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.
- Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these **difficulties** do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers-writers problem**.
- The readers-writers problem has several variations, all involving priorities.
- The **first** readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The **second** readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in **starvation**.
- In the solution to the **first** readers-writers problem, the reader processes share the following data structures:

```
Semaphore mutex, wrt;
int readcount;
```

- The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object.

- The code for a writer process is shown in Figure 3.16; the code for a reader process is shown in Figure 3.17.

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

Figure 3.16The structure of a writer process.

```
do {
        wait (mutex);
        readcount++;
        if (readcount 1)
        wait (wrt);
        signal(mutex);
                //reading is performed
        wait(mutex);
        readcount--;
        if (readcount 0)
        signal(wrt);
        signal(mutex);
} while (TRUE);
```

Figure 3.17The structure of a reader process.

- Acquiring a reader-writer lock requires specifying the mode of the lock either *read* or *write* access. When a process wishes only to read shared data, it requests the reader-writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

3.6.3. 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 (Figure 3.18).

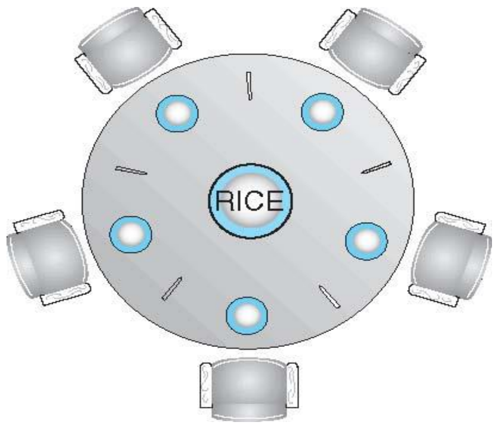


Figure 3.18 The situation of the dining philosophers.

- When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing await () 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 3.18.

```
do {
        wait(chopstick[i]);
        wait(chopstick[(i+l) % 5]);
        //eat
        signal(chopstick[i]);
        signal(chopstick[(i+l) % 5]);
        //think
while (TRUE);
```

Figure 3.18 The structure of philosopher *i.*

Monitors

- An abstract data type-or ADT- encapsulates private data with public methods to operate on that data. A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.
- The syntax of a monitor type is shown in Figure 3.19.

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

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

        Initialization code (…) { … }
}
```

- A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

```
condition x, y;
```

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

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

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

signal();

The

x. signal()

operation resumes exactly one suspended process. 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.

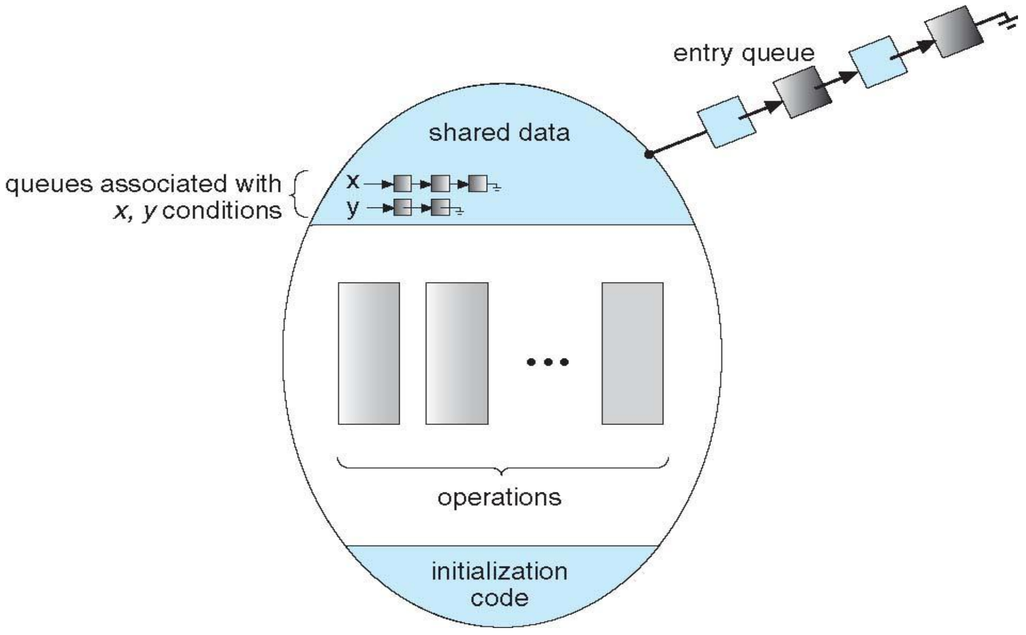


Figure 3.19Monitor with condition variables.

Dining-Philosophers Solution Using Monitors

- Philosopher may in one of the state thinking, hungry or waiting. For this purpose, we introduce the following data structure:

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

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

  conditionself[5];

  in which philosopher i can delay herself when she is hungry but is unable toobtain the chopsticks she needs.
- The solution to the dining-philosophers is shown in Figure 3.20.

```
monitor dp
{
        enum {THINKING, HUNGRY, EATING} state[5];
        condition self[5];
        void pickup(inti)
        {
                state[i] =HUNGRY;
                test(i);
                if (state [i] ! = EATING)
                        self [i] . wait() ;
        }
        void putdown(inti)
        {
                state[i] =THINKING;
                test((i + 4) % 5);
```

```
            test((i + 1) % 5);
        }
        void test(inti)
        {
            if ((state[(i + 4) % 5] !=EATING) &&(state[i] ==HUNGRY) &&(state[(i + 1)% 5]
            !=EATING))
            {
                state[i] =EATING;
                self[i] .signal();
            }
        }
        initialization_code()
        {
            for (inti = 0; i< 5; i++)
            state[i] =THINKING;
        }
}
```

Figure 3.20 A monitor solution to the dining-philosopher problem.

- Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the put down() operation.
- Thus, philosopher i must invoke the operations pickup() and put down() in the following sequence:

```
DiningPhilosophers.pickup(i);
        Eat
DiningPhilosophers.putdown(i);
        Think
```

- It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Questions from recent VTU papers

1. Define race condition. Explain reader's Writer's problem with semaphores. (Dec 13)
2. Define race condition. List the requirements that a solution to the critical problem must satisfy. (Dec 08 / June 10 / June 13)
3. What is critical section problem? What are the three requirements to be met by a solution to the critical section problem? (June09 / Dec 09)
4. What is Peterson's solution to the critical section problem?
5. Define mutual exclusion and critical section. Write software solution for 2 process synchronization. (Dec 12)
6. Define an algorithms TestAndSet( ) & Swap( ). Show that they satisfy mutual exclusion. (Dec 08 / June10 / Dec 11)
7. What is synchronization? Explain synchronization hardware. (June11)
8. What is busy waiting in critical section problem? How semaphore is used to solve this problem. (Dec 12)

9. What are semaphores? Explain two preemptive semaphore operations. What are the advantages of semaphores? (June10)
10. What are semaphores? Explain the solution to producer-consumer problem using semaphores. (Jun 11 / Dec 12)
11. Describe the bounded buffer problem & give the solution for the same using semaphores. Write the structure of producer & consumer processes. (June09)
12. Describe how the dining philosopher problem brings out the need for synchronization & avoids deadlocks. (Dec 09)
13. What do you mean by binary and counting semaphore? Explain the implementation of wait() and signal() semaphore operation. (June 13)
14. What is monitor? Write the monitor solution for the dining philosopher problem. (Dec 09 / 12 / 13)
15. What are monitors? Explain it. (June11)
16. What are monitors? Explain with an example of program request. (June 12)
17. Describe the following: i) Semaphore ii) wait( ) operation iii) signal ( ) operation. (June09)

Multi Threading & Scheduling Algorithms

1. What is multithreading? What are the benefits of multithreaded programming? (June 10 / Dec 11)
2. Write a note on multithreaded models. (June 09 / Dec 09)
3. Consider 4 jobs with (arrival time, burst time) as (0, 5) (0.2, 2) (0.6, 8) (1.2, 4). Find the average turnaround time and waiting time for the jobs using FCFS, SJF and RR(q=1) scheduling algorithms. (June11)
4. Consider the following set of processes. (June10)

Process	Arrival Time	Burst Time
P1	0	1
P2	1	9
P3	2	1
P4	3	9

   a) Draw Gantt charts showing the execution of these processes using FCFS, preemptive SJF, non-Pre-emptive SJF and RR (Quantum 1) scheduling schemes.
   b) Compute the turnaround time and waiting time for each process for each of the schemes above.
   c) Compute the average turnaround time and average waiting time in each scheme and thus find the best scheme in this particular case.
5. Why thread is called LWP? Describe any one threading model. (Dec 08)
6. Suppose the following jobs arrive for processing at the times indicated. Each job will run the listed amount of time.

Job	1	2	3
Arrival Time	0.0	0.4	1.0
Burst Time	8	4	1

   a) Give Gantt chart illustrating the execution of these jobs using the non-preemptive FCFS and SJF scheduling algorithms
   b) What is turnaround time and waiting time of each job for the above algorithms?
   c) Compute average turnaround time if CPU is left idle for 1 unit and then SJF is used.(Job1 and Job2 will wait during this time) (Dec.09)
7. Consider the following set of processor with a length of CPU burst time given in milliseconds.

Process	Arrival time	Burst time	Priority
P1	0	7	4
P2	3	2	2
P3	4	3	1
P4	4	1	4
P5	5	3	3

Find the average waiting time &average turnaround time using Gantt chart for the following scheduling algorithms.

i. Preemptive SJF

ii. Preemptive Priority(Smaller number represents high priority)

iii. Round-Robin(time slice =1ms) (Dec 08)