**Unit - 2**

**Process Management**

**Process Concept**

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes jobs, whereas a timeshared system has user programs or tasks. Even on a single user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser and an e-mail package. And even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

The terms Job and process are used almost interchangeably in this text. Although we personally prefer the term process, much of operat1ng-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job.

**The Process**

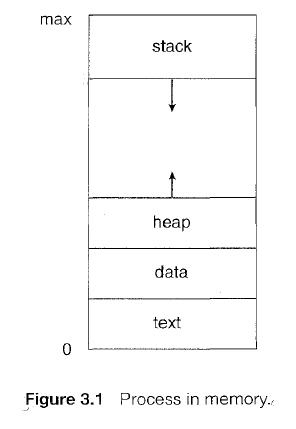
A process is a program at the time of execution.

## Differences between Process and Program

|  |  |
| --- | --- |
| **Process** | **Program** |
| Process is a dynamic object | Program is a static object |
| Process is sequence of instruction  Execution | Program is a sequence of instructions |
| Process loaded in to main memory | Program loaded into secondary storage  devices |
| Time span of process is limited | Time span of program is unlimited |
| Process is a active entity | Program is a passive entity |

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line.

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/process.JPG?attredirects=0)

**Process State**

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

**New**: The process is being created.

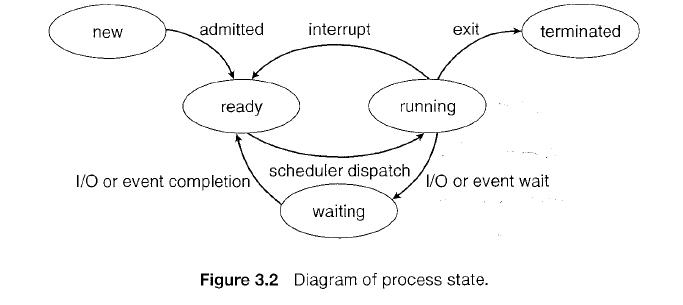
**Running**: Instructions are being executed.

**Waiting**: The process is waiting for some event to occur (such as an I/O Completion or reception of a signal).

**Ready**. The process is waiting to be assigned to a processor.

**Terminated**: The process has finished execution.

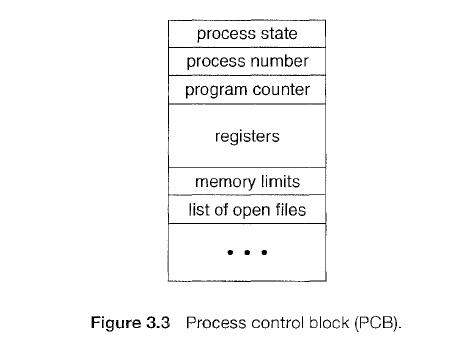
These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however.

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/process%20state.JPG?attredirects=0)

1. **New ->Ready** : OS creates process and prepares the process to be executed,thenOSmoved the process into readyqueue.
2. **Ready->Running** : OS selects one of the Jobs from ready Queue and move themfrom ready to Running.
3. **Running->Terminated** : When the Execution of a process has Completed, OSterminatesthatprocess from running state. Sometimes OS terminates the process for someother reasons including Time exceeded, memory unavailable, access violation, protection Error, I/O failure and soon.
4. **Running->Ready** : When the time slot of the processor expired (or) If the processorreceivedanyinterrupt signal, the OS shifted Running -> ReadyState.
5. **Running -> Waiting** : A process is put into the waiting state, if the process need an event occur (or) an I/O Devicerequire.
6. **Waiting->Ready** : A process in the waiting state is moved to ready state whenthe eventforwhichit has beenCompleted.

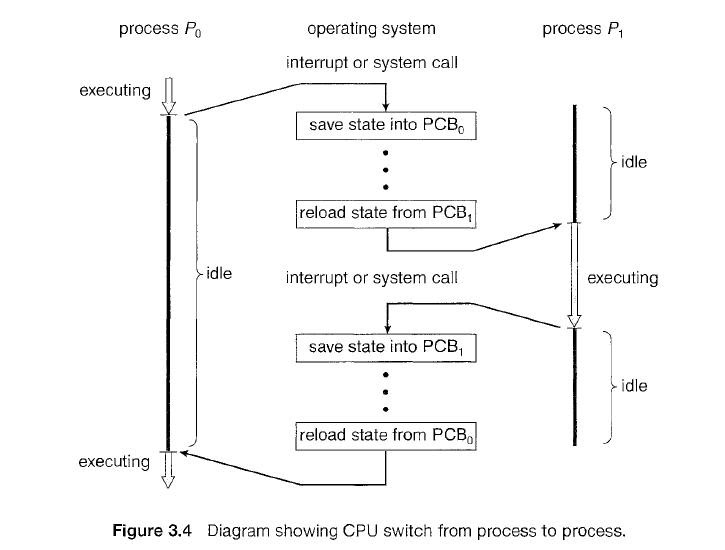
**Process Control Block**

Each process is represented in the operating system by a process control block (PCB) - also called a task control block. It contains many pieces of information associated with a specific process, including these:

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/PCB.JPG?attredirects=0)

* **Process state**: The state may be new, ready running, waiting, halted, and so on.
* **Program counter**: The counter indicates the address of the next instruction to be executed for this process.
* **CPU registers**: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
* **CPU-scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
* **Memory-management information**: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
* **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
* **I/O status information**: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief the PCB simply serves as the repository for any information that may vary from process to process.

**[](https://sites.google.com/site/btecnotes/os-unit---2-notes/threads.JPG?attredirects=0)**

**Threads**

The process model discussed so far has implied that a 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. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

A process is divide into number of light weight process, each light weight process is said to be a Thread. The Thread has a program counter (Keeps track of which instruction to execute next), registers (holds its current working variables), stack (execution History).

## Thread States:

1. bornState : A thread is justcreated.
2. readystate : The thread is waiting forCPU.
3. running : System assigns the processor to thethread.
4. sleep : A sleeping thread becomes ready after the designated sleep timeexpires**.**
5. dead : The Execution of the threadfinished.

## Eg: Word processor.

Typing, Formatting, Spell check, saving are threads.

## Differences between Process and Thread

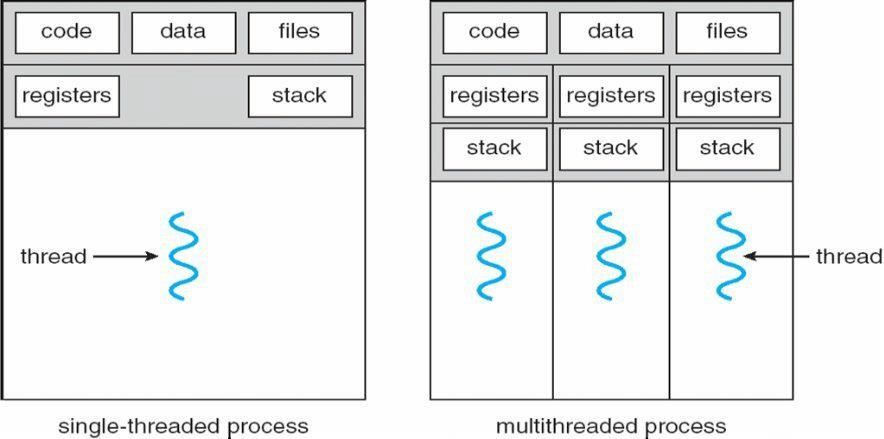
|  |  |
| --- | --- |
| **Process** | **Thread** |
| Process takes more time to create. | Thread takes less time to create. |
| it takes more time to complete execution &  terminate. | Less time to terminate. |
| Execution is very slow. | Execution is very fast. |
| It takes more time to switch b/w two  processes. | It takes less time to switch b/w two  threads. |
| Communication b/w two processes is difficult . | Communication b/w two threads is  easy. |
| Process can’t share the same memory area. | Threads can share same memory area. |
| System calls are requested to communicate  each other. | System calls are not required. |
| Process is loosely coupled. | Threads are tightly coupled. |
| It requires more resources to execute. | Requires few resources to execute. |

**Multithreading**

A process is divided into number of smaller tasks each task is called a Thread. Number of Threads with in a Process execute at a time is called Multithreading.

If a program, is multithreaded, even when some portion of it is blocked, the whole program is not blocked.The rest of the program continues working If multiple CPU’s are available.

Multithreading gives best performance.If we have only a single thread, number of CPU’s available, No performance benefits achieved.

* + Process creation is heavy-weight while thread creation is light-weight  Can simplify code, increase efficiency

Kernels are generally multithreaded

**CODE-** Contains instruction

**DATA-** holds global variable **FILES-**

opening and closing files

**REGISTER-** contain information about CPU state **STACK-**parameters, local variables, functions **Types Of Threads:**

1. **User Threads** : Thread creation, scheduling, management happen in user space by Thread Library. user threads are faster to create and manage. If a user thread performs a system call, which blocks it, all the other threads in that process one also automatically blocked, whole process is blocked

**Advantages**

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

**Disadvantages**

* In a typical operating system, most system calls areblocking.
* Multithreaded application cannot take advantage ofmultiprocessing.

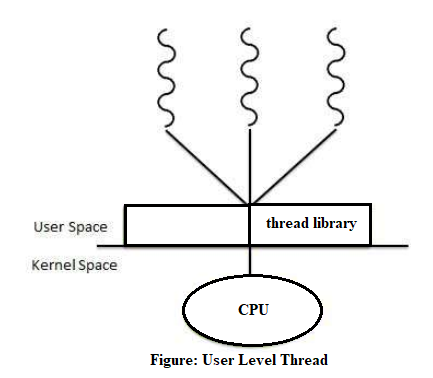
1. **Kernel Threads**: kernel creates, schedules, manages these threads .these threads are slower, manage. If one thread in a process blocked, over all process need not be blocked.

**Advantages**

* + Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
  + If one thread in a process is blocked, the Kernel can schedule another thread of the sameprocess.
  + Kernel routines themselves can multithreaded.

**Disadvantages**

* + Kernel threads are generally slower to create and manage than the userthreads.
  + Transfer of control from one thread to another within same process requires a mode switch to the Kernel.



**Multithreading Models**

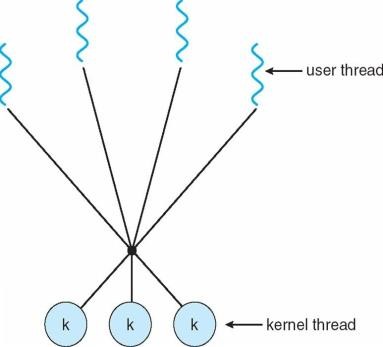
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

* Many to many relationship.
* Many to one relationship.
* One to one relationship.

**Many to Many Model**

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

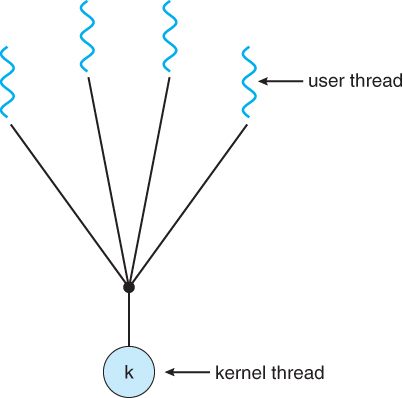
Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



**Many to One Model**

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

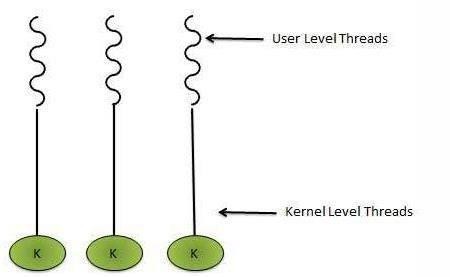
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.

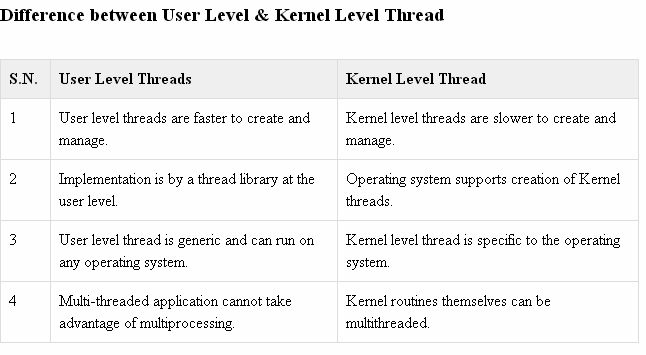


**One to One Model**

There is one to one relationship of user level thread to the kernel level thread.This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.





## PROCESS SCHEDULING:

CPU is always busy in **Multiprogramming**. Because CPU switches from one job to another job. But in

**simple computers** CPU sit idle until the I/O request granted.

**scheduling** is a important OS function. All resources are scheduled before use.(cpu, memory, devices…..)

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing

. **Scheduling Objectives**

Maximize throughput.

Maximize number of users receiving acceptable response times. Be predictable.

Balance resource use.

Avoid indefinite postponement. Enforce Priorities.

Give preference to processes holding key resources

**SCHEDULING QUEUES**: people live in rooms. Process are present in rooms knows as queues. There are 3types

1. **job queue**: when processes enter the system, they are put into a **job queue**, which consists all processes in the system. Processes in the job queue reside on mass storage and await the allocation of main memory.
2. **ready queue**: if a process is present in main memory and is ready to be allocated to cpu forexecution, is kept in **readyqueue.**
3. **device queue**: if a process is present in waiting state (or) waiting for an i/o event to complete is said to being device queue.(or)

The processes waiting for a particular I/O device is called device queue.

**Schedulers :** There are 3 schedulers

1. Long term scheduler.
2. Medium term scheduler
3. Short term scheduler.

Scheduler duties:

* + Maintains the queue.
  + Select the process from queues assign to CPU.

## Types of schedulers

1. **Long term scheduler:**

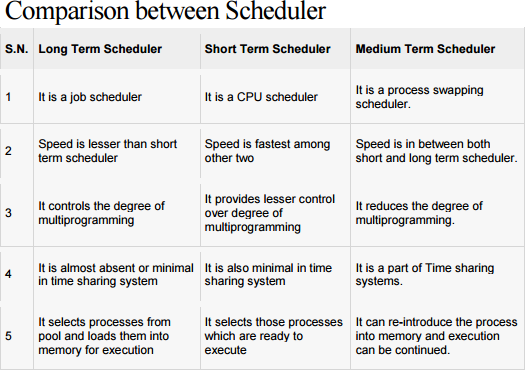
select the jobs from the job pool and loaded these jobs into main memory (ready queue). Long term scheduler is also called job scheduler.

## Short term scheduler:

select the process from ready queue, and allocates it to the cpu.

If a process requires an I/O device, which is not present available then process enters device queue. short term scheduler maintains ready queue, device queue. Also called as cpu scheduler.

1. **Medium term scheduler**: if process request an I/O device in the middle of the execution, then the process removed from the main memory and loaded into the waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations performed by medium term scheduler.



**Context Switch**: Assume, main memory contains more than one process. If cpu is executing a process, if time expires or if a high priority process enters into main memory, then the scheduler saves information about current process in the PCB and switches to execute the another process. The concept of moving CPU by scheduler from one process to other process is known as context switch.

**Non-Preemptive Scheduling**: **CPU** is assigned to one process, CPU do not release until the competition of that process. The CPU will assigned to some other process only after the previous process has finished.

**Preemptive scheduling**: here CPU can release the processes even in the middle of the execution. CPU received a signal from process p2. OS compares the priorities of p1 ,p2. If p1>p2, CPU continues the execution of p1. If p1<p2 CPU preempt p1 and assigned to p2.

**Dispatcher:** The main job of dispatcher is switching the cpu from one process to another process. Dispatcher connects the cpu to the process selected by the short term scheduler.

**Dispatcher latency**: The time it takes by the dispatcher to stop one process and start another process is known as dispatcher latency. If the dispatcher latency is increasing, then the degree of multiprogramming decreases.

**Scheduling Criteria**

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

* **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
* **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
* **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.
* **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the an1.ount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
* **Response time**: 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. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as timesharing systems), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

**Scheduling Algorithms**

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms. In this section, we describe several of them.

**First-Come, First-Served Scheduling**

By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

            On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

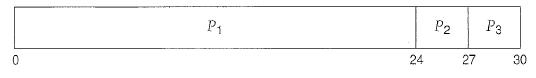
**Process           Burst Time**

     P1                     24

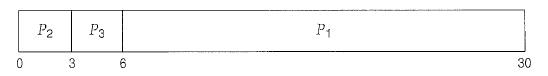
   P2                     3

      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, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/FCFS%201.JPG?attredirects=0)

The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/FCFS%202.JPG?attredirects=0)

The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the 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/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

**Shortest-Job-First Scheduling**

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. 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. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because m.ost people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

**Process           Burst Time**

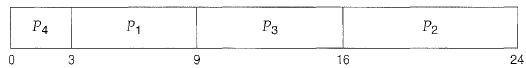
    P1                     6

    P2                     8

   P3                     7

   P3                     3

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

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/SJFS%201.JPG?attredirects=0)

The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0) /4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

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

The real difficulty with the SJF algorithm is, knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average**of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let tn be the length of the nth CPU burst, and let Tn+1 be our predicted value for the next CPU burst. Then, for α**, 0 ≤**α**≤ 1**, define

Tn+1=α tn+(1-α)Tn

The value of tn contains our most recent information; Tn stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If α = 0, then Tn+l = Tn, and recent history has no effect (current conditions are assumed to be transient). If α=1, then Tn+l = tn , and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, α= 1/2, so recent history and past history are equally weighted. The initial T0 can be defined as a constant or as an overall system average. Figure 5.3 shows an exponential average with α=1/2 and T0= 10.

To understand the behavior of the exponential average, we can expand the formula for Tn+l by substituting for Tn , to find,

Tn+1= αtn+(1- α) αtn-1+….+(1- α)j αtn-j+…+(1- α)n+1T0­

Since both α and (1- α) are less than or equal to 1, each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a 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.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

**Process           Arrival Time              Burst Time**

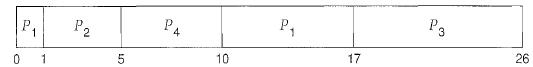
                    P1            0                                  8

                   P2                     1                                  4

                   P3                     2                                  9

                   P4                     3                                  5

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

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/SJFS%202.JPG?attredirects=0)

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is [(10- 1) + (1 - 1) + (17- 2) + (5-3)]/ 4 = 26/4 = 6.5 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

**Priority Scheduling**

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

Note that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2 … P5, with the length of the CPU burst given in milliseconds:

**Process           Burst Time                 Arrival**

     P1         10                                3

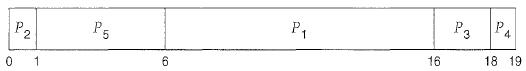
              P2                     1                                  1

                P3                     2                                  4

                P4                     1                                  5

               P5                     5                                  2

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

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/Priority%20schgeduling.JPG?attredirects=0)

The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes.

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 by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

1. **Shortest Remaining Time First ( SRTF );**

This is primitive scheduling algorithm.

Short term scheduler always chooses the process that has term shortest remaining time. When a new process joins the ready queue , short term scheduler compare the remaining time of executing process and new process. If the new process has the least CPU burst time, The scheduler selects that job and connect to CPU. Otherwise continue the old process.

|  |  |  |
| --- | --- | --- |
| **PROCESS** | **BURST TIME** | **ARRIVAL TIME** |
| P1 | 3 | 0 |
| P2 | 6 | 2 |
| P3 | 4 | 4 |
| P4 | 5 | 6 |
| P5 | 2 | 8 |



P1 arrives at time 0, P1 executing First , P2 arrives at time 2. Compare P1 remaining time and P2 ( 3-2 =

1) and 6. So, continue P1 after P1, executing P2, at time 4, P3 arrives, compare P2 remaining time (6-1=5

) and 4 ( 4<5 ) .So, executing P3 at time 6, P4 arrives. Compare P3 remaining time and P4 ( 4- 2=2 ) and 5 (2<5 ). So, continue P3 , after P3, ready queue consisting P5 is the least out of three. So execute P5, next P2, P4.

**FORMULA :** Finish time - Arrival Time Finish Time for P1 => 3-0 = 3 Finish Time for P2 => 15-2 = 13 Finish Time for P3 => 8-4 =4

Finish Time for P4 => 20-6 = 14 Finish Time for P5 => 10-8 = 2

Average Turn around time => 36/5 = 7.2 ms.

**Robin round 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.

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

            One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.

            The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

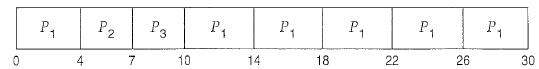
**Process           Burst Time**

                                                                                                                        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. The CPU is then given to the next process, process *P3.*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:

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/round%20robin.JPG?attredirects=0)

Let's calculate the average waiting time for the above schedule. P1 waits for 6 milliseconds (10- 4), *P2*waits for 4 milliseconds, and *P3*waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

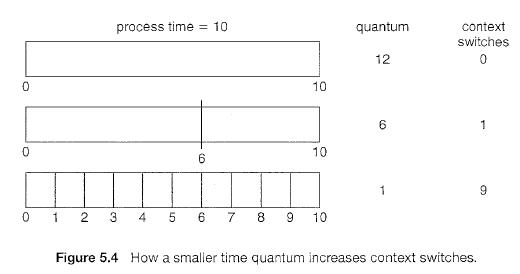
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n. processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n - 1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

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. This approach was used in Control Data Corporation (CDC) hardware to implement ten peripheral processors with only one set of hardware and ten sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in ten slow processors rather than one fast one. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than ten real processors would have been).

In software, we need also to consider the effect of context switching on the performance of RR scheduling. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in. less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.4).

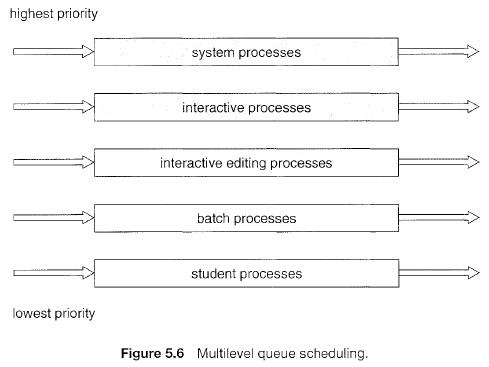
Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/proiority%20scheduling%202.JPG?attredirects=0)

**Multilevel Queue Scheduling**

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

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 5.6). 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.

[](https://sites.google.com/site/btecnotes/os-unit---2-notes/multi%20level%20schgeduling.JPG?attredirects=0)

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

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

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

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

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

**Multilevel Feedback Queue Scheduling**

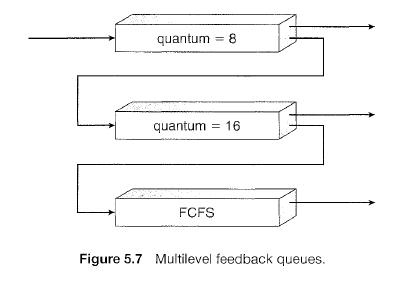
Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

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

            For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

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

**[](https://sites.google.com/site/btecnotes/os-unit---2-notes/multilevel%20feed%20back.JPG?attredirects=0)**

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

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

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

**Process synchronization** refers to the idea that multiple processes are to join up or [handshake](https://en.wikipedia.org/wiki/Handshaking) at a certain point, in order to reach an agreement or commit to a certain sequence of action. Coordination of simultaneous processes to complete a task is known as process synchronization.

## The critical section problem

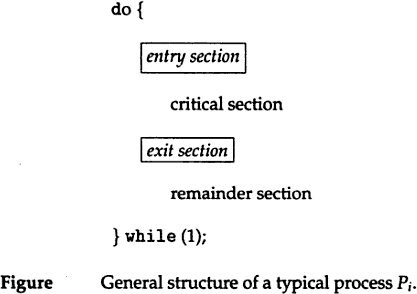
Consider a system , assume that it consisting of n processes. Each process having a segment of code. This segment of code is said to be critical section.

E.G: Railway Reservation System.

Two persons from different stations want to reserve their tickets, the train number, destination is common, the two persons try to get the reservation at the same time. Unfortunately, the available berths are only one; both are trying for that berth.

It is also called the critical section problem. Solution is when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

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



## A solution to the critical section problem must satisfy the following 3 requirements:

## 1.mutual exclusion:

Only one process can execute their critical section at any time.

## Progress:

When no process is executing a critical section for a data, one of the processes wishing to enter a critical section for data will be granted entry.

## Bounded wait:

No process should wait for a resource for infinite amount of time.

## Critical section:

The portion in any program that accesses a shared resource is called as critical section (or) critical region.

## Peterson’s solution:

Peterson solution is one of the solutions to critical section problem involving two processes. This solution states that when one process is executing its critical section then the other process executes the rest of the code and vice versa.

Peterson solution requires two shared data items:

1. **turn**: indicates whose turn it is to enter into the critical section. If turn == i ,then process i is allowed into their critical section.
2. **flag:** indicates when a process wants to enter into critical section. When process i wants to entertheir critical section,it sets flag[i] to true.

**do {**flag[i] = TRUE; turn = j;

**while (flag[j] && turn == j);**

**critical section**

flag[i] = FALSE;

remainder section

**} while (TRUE);**

**Synchronization hardware**

In a uniprocessor multiprogrammed system, mutual exclusion can be obtained by disabling the interrupts before the process enters its critical section and enabling them after it has exited the critical section**.**

Disable interrupts

Critical section

Enable interrupts

Once a process is in critical section it cannot be interrupted. This solution cannot be used in multiprocessor environment. since processes run independently on different processors.

In multiprocessor systems, **Testandset** instruction is provided,it completes execution without interruption. Each process when entering their critical section must set **lock**,to prevent other processes from entering their critical sections simultaneously and must release the lock when exiting their critical sections.

**Do**

**{**

**acquire**

**lock critical section**

**release lock**

**remainder section**

**} while (TRUE);**

A process wants to enter critical section and value of lock is false then **testandset** returns false and the value of lock becomes true. thus for other processes wanting to enter their critical sections **testandset** returns true and the processes do busy waiting until the process exits critical section and sets the value of lock to false.

## Definition:

boolean TestAndSet(boolean&lock){ boolean temp=lock;

Lock=true; return temp;

}

## Algorithm for TestAndSet

*do{*

*while testandset(&lock)*

*//do nothing*

*//critical section* lock*=false*

*remainder section*

*}while(TRUE);*

## Swap instruction can also be used for mutual exclusion Definition

Void swap(boolean &a, boolean &b)

{

boolean temp=a; a=b;

b=temp;

}

## Algorithm

do

{

key=true; while(key=true) swap(lock,key); critical section lock=false; remainder section

}while(1);

lock is global variable initialized to false.each process has a local variable key. A process wants to enter critical section,since the value of lock is false and key is true.

## lock=false key=true

after swap instruction,

## lock=true key=false

now key=false becomes true,process exits repeat-until,and enter into critical section. When process is in critical section (lock=true),so other processes wanting to enter critical section will have

## lock=true key=true

Hence they will do busy waiting in repeat-until loop until the process exits critical section and sets the value of lock to false.

## Semaphores

A semaphore is an integer variable.semaphore accesses only through two operations.

1. **wait:** wait operation decrements the count by1.

If the result value is negative,the process executing the wait operation is blocked.

## signaloperation:

Signal operation increments by 1,if the value is not positive then one of the process blocked in wait operation unblocked.

wait (S) {

while S <= 0 ; // no-op

S--;

}

signal (S)

{

S++;

}

In binary semaphore count can be 0 or 1. The value of semaphore is initialized to 1

do {

wait (mutex);

// Critical Section signal (mutex);

// remainder section

} while (TRUE);

First process that executes wait operation will be immediately granted sem.count to 0. If some other process wants critical section and executes wait() then it is blocked,since value becomes -1. If the process exits critical section it executes signal().sem.count is incremented by 1.blocked process is removed from queue and added to ready queue.

## Classic problems of synchronization

1. **Bounded-buffer problem**

Two processes share a common ,fixed –size buffer.

Producer puts information into the buffer, consumer takes it out.

The problem arise when the producer wants to put a new item in the buffer,but it is already full. The solution is for the producer has to wait until the consumer has consumed atleast one buffer. similarly if the consumer wants to remove an item from the buffer and sees that the buffer is empty,it goes to sleep until the producer puts something in the buffer and wakes it up.

## Problems:

1. **Deadlock**

Deadlock occurs when multiple processes are blocked.each waiting for a resource that can only be freed by one of the other blocked processes.

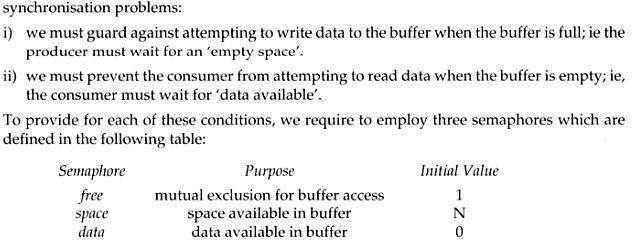
## Starvation

one or more processes gets blocked forever and never get a chance to take their turn in the critical section.

## Priority inversion

If low priority process is running ,medium priority processes are waiting for low priority process,high priority processes are waiting for medium priority processes.this is called Priority inversion.

The two most common kinds of semaphores are **counting semaphores** and **binary semaphores**. Counting semaphores represent multiple resources, while binary semaphores, as the name implies, represents two possible states (generally 0 or 1; locked or unlocked).



## The structure of the producer process

do {

// produce an item in nextp wait (empty);

wait (mutex);

// add the item to the buffer signal (mutex); signal (full);

} while (TRUE);

## The structure of the consumer process

do { wait (full); wait (mutex);

// remove an item from buffer to nextc signal (mutex);

signal (empty);

// consume the item in nextc

} while (TRUE);

## The readers-writers problem

A database is to be shared among several concurrent processes.some processes may want only to read the database,some may want to update the database.If two readers access the shared data simultaneously no problem.if a write,some other process access the database simultaneously problem arised.Writes have excusive access to the shared database while writing to the database.This problem is known as readers- writes problem.

## First readers-writers problem

No reader be kept waiting unless a writer has already obtained permission to use the shared resource.

## Second readers-writes problem:

Once writer is ready,that writer performs its write as soon as possible.

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. A reader writer lock in read mode. but only one process may acquire the lock for writing as exclusive access is required for writers.

Semaphore mutex initialized to 1

* Semaphore wrt initialized to 1
* Integer read count initialized to 0

## The structure of a writer process

do {

wait (wrt) ;

// writing is performed signal (wrt) ;

} while (TRUE);

## The structure of a reader process

do {

wait (mutex) ; readcount ++ ;

if (readcount == 1) wait (wrt) ;

signal (mutex)

// reading is performed wait (mutex) ; readcount

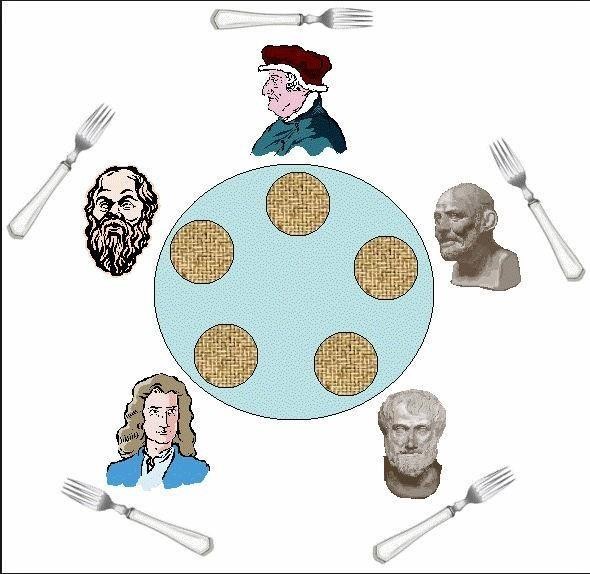
- - ;

if (readcount == 0) signal (wrt) ; signal (mutex) ;

} while (TRUE);

## Dining Philosophers problem

1. Five philosophers are seated on 5 chairs across a table. Each philosopher has a plate full of noodles. Each philosopher needs a pair of forks to eat it. There are only 5 forks available all together. There is only one fork between any two plates of noodles.
2. In order to eat, a philosopher lifts two forks, one to his left and the other to his right. if he is successful in obtaining two forks, he starts eating after some time, he stops eating and keeps both the forks down.



### What if all the 5 philosophers decide to eat at the same time ?

All the 5 philosophers would attempt to pick up two forks at the same time. So,none of them succeed.

One simple solution is to represent each fork with a semaphore.a philosopher tries to grab a fork by executing wait() operation on that semaphore.he releases his forks by executing the signal() operation.This solution guarantees that no two neighbours are eating simultaneously.

Suppose all 5 philosophers become hungry simultaneously and each grabs his left fork,he will be delayed forever.

## The structure of Philosopher *i*:

do{

wait ( chopstick[i] );

wait ( chopStick[ (i + 1) % 5] );

// eat

signal ( chopstick[i] );

signal (chopstick[ (i + 1) % 5] );

// think

} while (TRUE);

## Several remedies:

1. Allow at most 4 philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pickup his fork only if both forks are available.
3. An odd philosopher picks up first his left fork and then right fork. an even philosopher picks up his right fork and then his left fork.

## MONITORS

The disadvantage of semaphore is that it is unstructured construct. Wait and signal operations can be scattered in a program and hence debugging becomes difficult.

A monitor is an object that contains both the data and procedures needed to perform allocation of a shared resource. To accomplish resource allocation using monitors, a process must call a **monitor entry routine**. Many processes may want to enter the monitor at the same time. but only one process at a time is allowed to enter. Data inside a monitor may be either global to all routines within the monitor (or) local to a specific routine. Monitor data is accessible only within the monitor. There is no way for processes outside the monitor to access monitor data. This is a form of information hiding.

If a process calls a monitor entry routine while no other processes are executing inside the monitor, the process acquires a lock on the monitor and enters it. while a process is in the monitor, other processes may not enter the monitor to acquire the resource. If a process calls a monitor entry routine while the other monitor is locked the monitor makes the calling process wait outside the monitor until the lock on the monitor is released. The process that has the resource will call a monitor entry routine to release the resource. This routine could free the resource and wait for another requesting process to arrive monitor entry routine calls signal to allow one of the waiting processes to enter the monitor and acquire the resource. Monitor gives high priority to waiting processes than to newly arriving ones.

## Structure:

monitor monitor-name

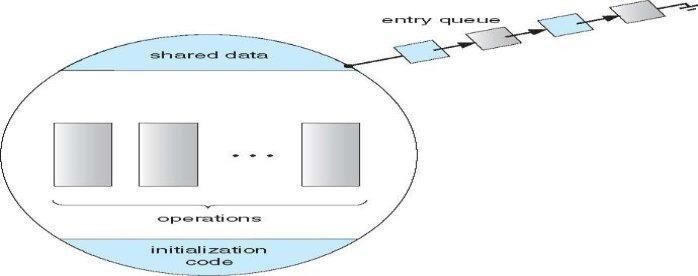
{

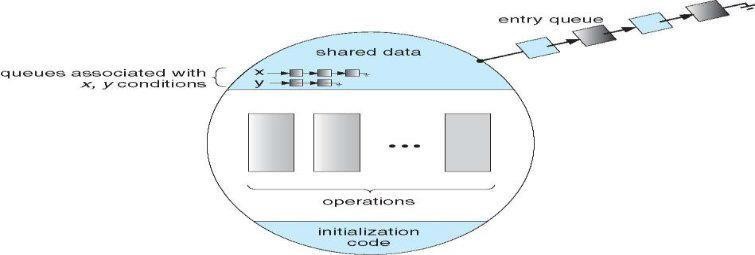
// shared variable declarations procedure P1 (…) { …. } procedurePn (…) {……} Initialization code (…) { … }

}

}

Processes can call procedures p1,p2,p3……They cannot access the local variables of the monitor

**Schematic view of a Monitor**

**Monitor with Condition Variables**

Monitor provides condition variables along with two operations on them i.e. wait and signal.

### wait(condition variable) signal(condition variable)

Every condition variable has an associated queue.A process calling wait on a particular condition variable is placed into the queue associated with that condition variable.A process calling signal on a particular condition variable causes a process waiting on that condition variable to be removed from the queue associated with it.

**Solution to Producer consumer problem using monitors:**

**monitor producerconsumer condition full,empty;**

**int count;**

**procedure insert(item)**

**{**

**if(count==MAX) wait(full) ; insert\_item(item); count=count+1; if(count==1) signal(empty);**

**}**

**procedure remove()**

**{**

**if(count==0) wait(empty); remove\_item(item); count=count-1; if(count==MAX-1) signal(full);**

**}**

**procedure producer()**

**{**

**producerconsumer.insert(item);**

**}**

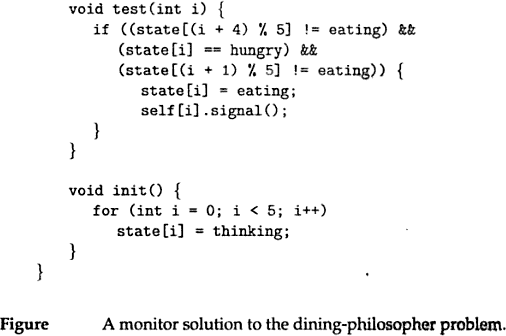
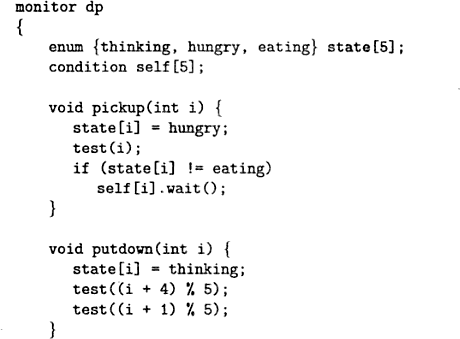
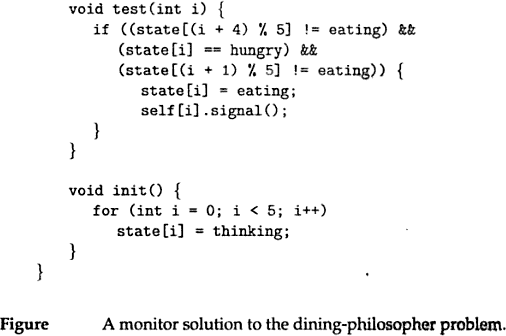
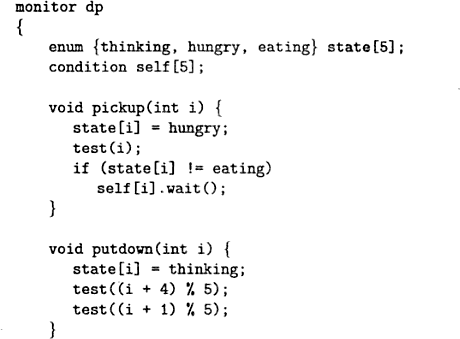
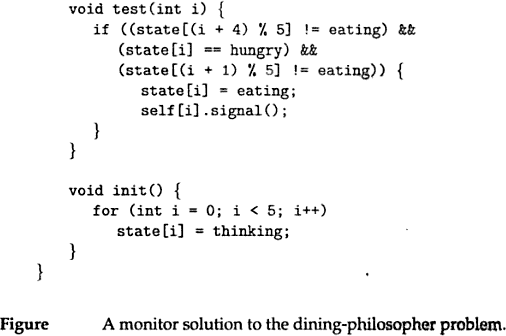
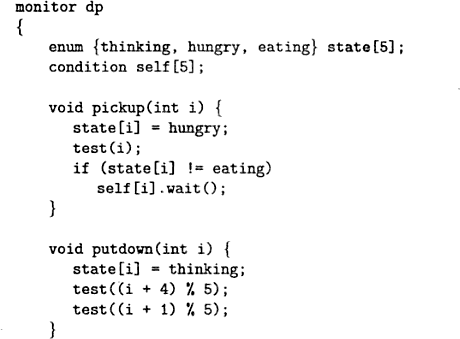
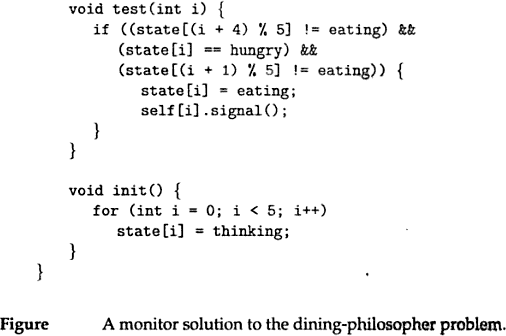
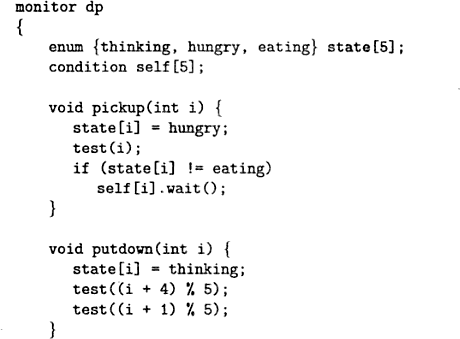
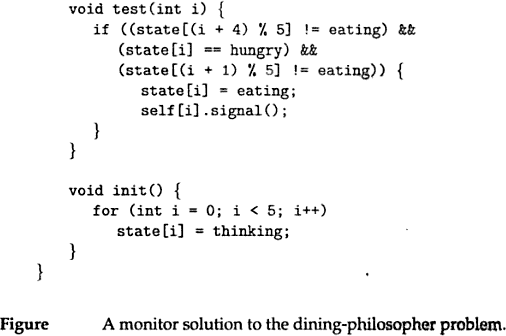
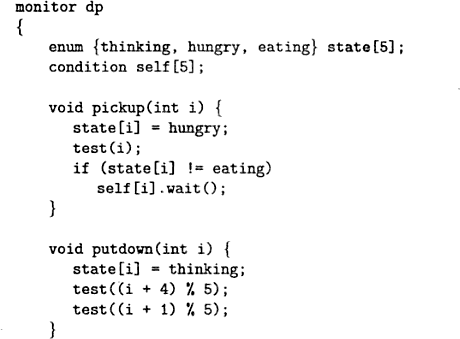
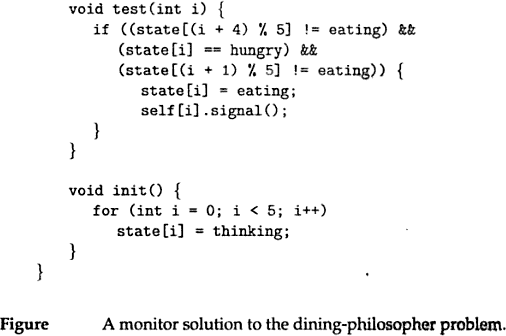
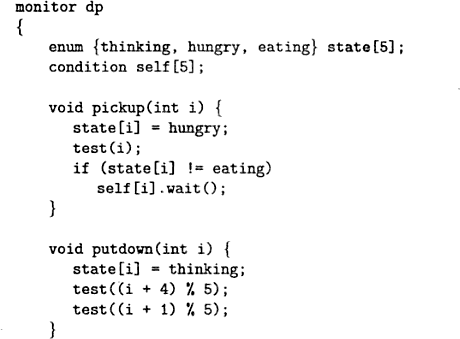
**procedure consumer()**

**{**

**producerconsumer.remove();**

**}**

**Solution to dining philosophers problem using monitors**



A philosopher may pickup his forks only if both of them are available.A philosopher can eat only if his two neighbours are not eating.some other philosopher can delay himself when he is hungry.

**Diningphilosophers.Take\_forks( ) :** acquires forks ,which may block the process.

## Eat noodles ( )

**Diningphilosophers.put\_forks( ):** releases the forks.

## Resuming processes within a monitor

If several processes are suspended on condion x and x.signal( ) is executed by some process. then

***how do we determine which of the suspended processes should be resumed next ?*** solution is FCFS(process that has been waiting the longest is resumed first).In many circumstances, such simple technique is not adequate. alternate solution is to assign priorities and wake up the process with the highest priority.

## Resource allocation using monitor boolean inuse=false; conditionavailable;

**//conditionvariable**

**monitorentry void get resource()**

**{**

**if(inuse) //is resource inuse**

**{**

**wait(available); wait until available issignaled**

**}**

**inuse=true; //indicate resource is now inuse**

**}**

**monitor entry void return resource()**

**{**

**inuse=false; //indicate resource is not in use signal(available); //signal a waiting process to proceed**

**}**