**CPU Scheduling:**

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated first to the CPU. CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state. (I/O or other event)
2. Switches from running to ready state. (Interrupt)
3. Switches from waiting to ready. (I/O or event completion)
4. Terminates  
   Scheduling under 1 and 4 is **non-preemptive**All other scheduling is **preemptive**

Dispatcher module gives control of the CPU to the process selected by the short term scheduler; this involves: **switching context switching to user mode  
jumping to the proper location in the user program to restart that program**  
• **Dispatch latency** – time it takes for the dispatcher to stop one process and start  
another running.

**Scheduling Criteria:**

* **CPU utilization** – keep the CPU as busy as possible
* **Throughput** – # of processes that complete their execution per time unit
* **Turnaround time** – amount of time to execute a particular process
* **Waiting time** – amount of time a process has been waiting in the ready queue
* **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

**First Come, First Served Scheduling (FCFS) Algorithm:** This is the simplest CPU scheduling algorithm. In this scheme, the process which requests the CPU first, that is allocated to the CPU first. The implementation of the FCFS algorithm is easily managed with a FIFO queue. When a process enters the ready queue its PCB is linked onto the rear of the queue. The average waiting time under FCFS policy is quiet long. Consider the following example:

|  |  |
| --- | --- |
| **Process** | **CPU time** |
| P1 | 3 |
| P2 | 5 |
| P3 | 2 |
| P4 | 4 |

Using FCFS algorithm find the average waiting time and average turnaround time if the order is P1, P2, P3, P4.

**Solution:** If the process arrived in the order P1, P2, P3, P4 then according to the FCFS the Gantt chart will be:

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P3 | P4 |

0 3 8 10 14

The waiting time for process P1 = 0, P2 = 3, P3 = 8, P4 = 10 then the turnaround time for process P1 = 0 + 3 = 3, P2 = 3 + 5 = 8, P3 = 8 + 2 = 10, P4 = 10 + 4 =14.

Then average waiting time = (0 + 3 + 8 + 10)/4 = 21/4 = 5.25 Average turnaround time = (3 + 8 + 10 + 14)/4 = 35/4 = 8.75

The FCFS algorithm is non preemptive means once the CPU has been allocated to a process then the process keeps the CPU until the release the CPU either by terminating or requesting I/O.

The disadvantage in the FCFS algorithm is all process wait for the big process (longest CPU burst) to release the CPU is called **convoy effect.**

The average waiting time in the FCFS scheduling algorithm is not minimal, vary based on the process order.

**Shortest Job First Scheduling (SJF) Algorithm:** This algorithm associates with each process if the CPU is available. This scheduling is also known as shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of the process rather than its total length. Consider the following example:

|  |  |
| --- | --- |
| **Process** | **CPU time** |
| P1 | 3 |
| P2 | 5 |
| P3 | 2 |
| P4 | 4 |

**Solution:** According to the SJF the Gantt chart will be

|  |  |  |  |
| --- | --- | --- | --- |
| P3 | P1 | P2 | P4 |

0 2 5 9 14

The waiting time for process P1 = 0, P2 = 2, P3 = 5, P4 = 9 then the turnaround time for process P3 = 0 + 2 = 2, P1 = 2 + 3 = 5, P4 = 5 + 4 = 9, P2 = 9 + 5 =14.

Then average waiting time = (0 + 2 + 5 + 9)/4 = 16/4 = 4 Average turnaround time = (2 + 5 + 9 + 14)/4 = 30/4 = 7.5

The SJF algorithm may be either preemptive or non preemptive algorithm. The preemptive SJF is also known as shortest remaining time first.

Consider the following example.

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **CPU time** |
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |

In this case the Gantt chart will be

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P2 | P4 | P1 | P3 |

0 1 5 10 17 26

The waiting time for process P1 = 10 - 1 = 9

P2 = 1 – 1 = 0

P3 = 17 – 2 = 15

P4 = 5 – 3 = 2

The average waiting time = (9 + 0 + 15 + 2)/4 = 26/4 = 6.5

**Priority Scheduling Algorithm:** In this scheduling 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 manner. Consider the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process** | | **Arrival Time** | | **CPU time** | |
| P1 | | 10 | | 3 | |
| P2 | | 1 | | 1 | |
| P3 | | 2 | | 3 | |
| P4 | | 1 | 4 |
| P5 | | 5 | 2 |

According to the priority scheduling the Gantt chart will be

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P2 | P5 | P1 | P3 | P4 |

0 1 6 16 18 19

The waiting time for process P1 = 6

P2 = 0

P3 = 16

P4 = 18

P4 = 1

The average waiting time = (0 + 1 + 6 + 16 + 18)/5 = 41/5 = 8.2

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 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.

**Round Robin Scheduling Algorithm:** This type of algorithm is designed only for the time sharing system. It is similar to FCFS scheduling with pre-emption condition to switch between processes. A small unit of time called quantum time or time slice is used to switch between the processes. The average waiting time under the round robin policy is quiet long. Consider the following example:

|  |  |
| --- | --- |
| **Process** | **CPU time** |
| P1 | 3 |
| P2 | 5 |
| P3 | 2 |
| P4 | 4 |

Time Slice = 1 millisecond.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P1 | P2 | P3 | P4 | P1 | P2 | P4 | P2 | P4 | P2 |

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

The waiting time for process

P1 = 6

P2 = 10

P3 = 5

P4 = 9

The average waiting time = (6 + 10 + 5 + 9)/4 = 7.5

**Process Synchronization:**

A co-operation process is one that can affect or be affected by other processes executing in the system. Co-operating process may either directly share a logical address space or be allotted to the shared data only through files. In case of producer-consumer problem bounded buffer used to enable the processes to share the memory and counter variable is used, which is initialized to 0.

The counter value is incremented by 1 when new item is added to the buffer and decremented 1 when item is consumed or removed from buffer.

**Producer:** **consumer:**

While(true) While(true)

{ {

------- -------

------- -------

Counter++; counter--;

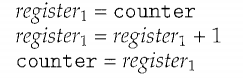
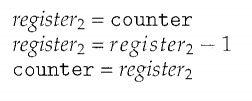
------- ---------

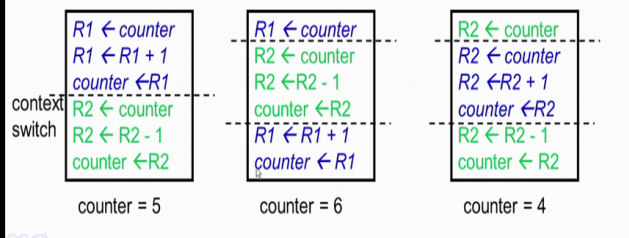
-------- ---------

} }

When both producer and consumer code executed one after another returns correct value for count, but when they execute concurrently, they may not function correctly. Suppose the value of counter is initially 5, and producer, consumer executing the instructions counter++, counter—concurrently.

Note that the statements may be implemented as follows in machine language:

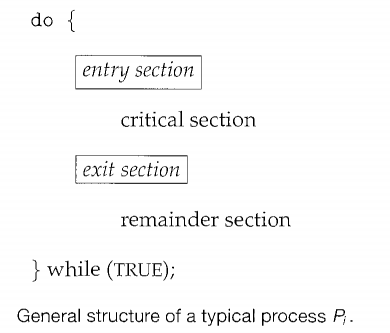


So, this was an example of the issues that would occur when we have a shared memory.  
So even though there was a very simple operation of incrementing a counter in one place  
while decrementing the counter in another place we had seen that the result could be  
different, depending on how the instructions get executed and how the context switch is  
occurred. So we would define this scenario more formally by what is known as Race  
Conditions.

A **Race Condition** is a situation where several processes access and manipulate the same  
data. The part of the process which accesses common or the shared data is known as a  
**Critical Section**. The outcome of a race condition would depend on the order in which  
the accesses to that data take place. As we have seen in the previous example, depending  
on which program executes first as well as depending on how the context switch is occur  
the result would vary. The race conditions could be prevented by what is known as  
synchronization. Essentially, with synchronization we would ensure that only one  
process at a time would manipulate the critical data.

**Critical-Section Problem:**

Consider a system consisting of n processes (P0, P1, ………Pn -1) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on. The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section. The execution of critical sections by the processes is a mutually exclusive. The critical section problem is to design a protocol that the process 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 is followed on exit section. The remaining code is the remainder section as shown below:



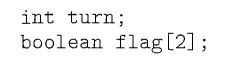
A solution to the critical-section problem must satisfy the following three requirements:

* **Mutual Exclusion:** In Mutual Exclusion, the critical section solution should ensure that not more than one process is in the critical section at a given time.
* **Progress:** the processes that are not executing in their remainder sections (executing or interested in critical section) can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
* **Bounded waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request.

**Peterson’s Solution:**

Peterson’s solution is the classic software-based solution for critical-section problem. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *Po* and P1. For convenience, when presenting *Pi,* we use *Pj* to denote the other process; that is, j equals 1 – i.

Peterson's solution requires the two processes to share two data items:



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.

If any process wants to enter into it’s critical section it needs to set its flag value to true and turn value to other process id,.i.e if Pi wants to enter into it’s 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. If both processes try to enter at the same time, turn will be set to both i and j at the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

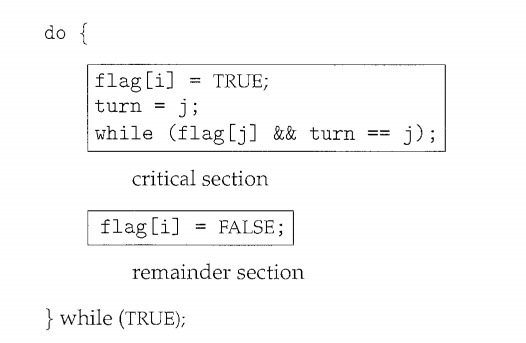


Fig:The structure of process Pi in Peterson's solution

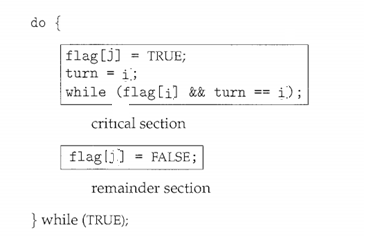


Fig: Structure of process Pj in Peterson’s solution

We now prove that this solution is correct. We need to show that:

* Mutual exclusion is preserved.
* The progress requirement is satisfied.
* The bounded-waiting requirement is met.
* If both processes are trying to enter into their critical section, both processes Pi, Pj simultaneous sets the corresponding flag value to TRUE; followed by the turn variable will be set to both i and j at roughly same time. But one of these assignments will last; the other will be overwritten immediately. Hence only one process get the chance to enter into their critical section; as a result mutual exclusion is preserved.
* Assume only Pi wants to execute it’s critical section. The flag values are -flag[i]=TRUE and flag[j]=FALSE. The while statement in the Pi is FALSE, successfully enter into it’s critical section. Hence progress requirement also satisfied.
* Consider both processes Pi and Pj are interested in critical section the corresponding flag values are-flag[i]=flag[j]=TRUE. But turn variable can be either i or j but can not both. If turn=i ,Pi will enter the critical section and Pj stuck in the while loop . Once Pi exit the critical section it will reset flag[i]=FALSE , allowing Pj to enter into it’s critical section. Hence Bounded waiting requirement also met.

The disadvantage in software-based solution like Peterson’s solutions are not guaranteed to work on modern computer architectures

**Synchronization Hardware:**

The 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 section can 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.

In this we will use hardware instructions like TestAndSet and Swap (to acquire lock)to control the processes which are going to execute the critical section.

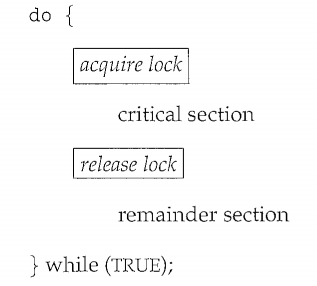


Fig: Solution to critical-section using locks

**Test and Set Lock:( TSL)**

TestAndSet is machine instruction which is atomic (all the instructions are executed in one machine cycle). The definition of test and set is as follows: it will return the Boolean value (TRUE or FALSE).

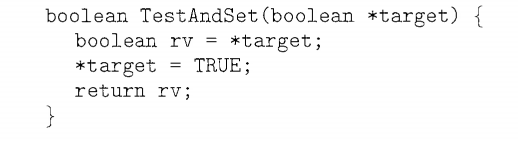


Fig: The definition of the TestAndSet () instruction

The TestAndSet instruction returns the current value of the target and sets the target to always TRUE.

To solve the critical section problem by using TestAndSet instruction we will use on shared variable Lock which is shared among different processes. The value of Lock is either of two values0(FALSE) or 1(TRUE). The value is 0 means it was unlocked, whereas value is 1 means it is locked. Before entering into critical section, the process checks the Lock variable. It will see whether the Lock variable value is 0(unlocked) or 1(locked). In case it is locked it keeps on waiting till it is unlocked. Initially the Lock variable is initialized to FALSE.

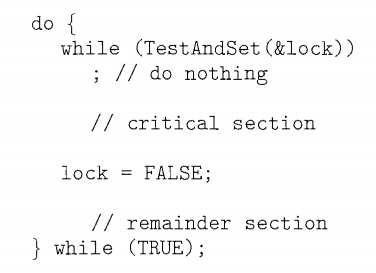


Fig: Mutual Exclusion implementation using TestAndSet()

The TestAndSet structure preserves mutual exclusion but, does not assure bounded waiting condition as it may happen that process Pi may keep on executing its critical section again and again and other processes has to keep on waiting.

**Swap Hardware Instruction:**

Like TestAndSet() instruction the swap() hardware instruction is also a atomic instruction. The only difference is it operates on two variables provided as the parameters. The structure of swap instruction is:

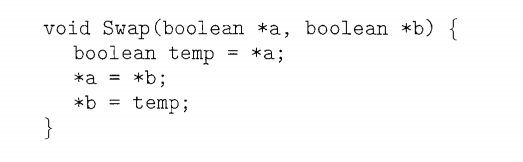
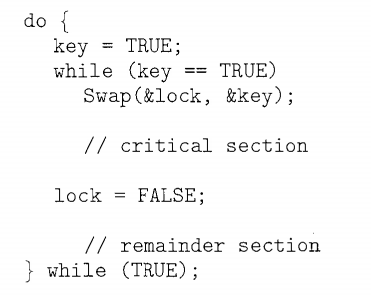
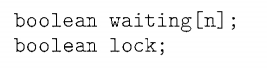


Fig : The definition of swap instruction

The structure of process Pi is as follows. The structure shown operates on one global bool variable lock and another local variable key which are initially set to FALSE. The process interested in critical section execute the code and set the lock to TRUE .



Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. We present another algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements. The common data structures are:



These data structures are initialized to FALSE.

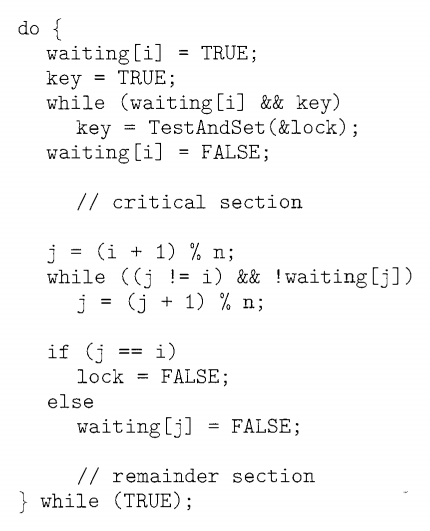


Fig: Bounded waiting and Mutual exclusion with TestAndSet()

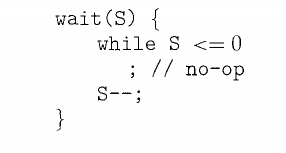
To prove that the mutual exclusion requirement is met, we note that process *P;* can enter its critical section only if either waiting [i] == FLASE or key == FALSE. The value of key can become false only if the TestAndSet () is executed. The first process to execute the TestAndSet () will find key== FALSE; all others must wait. The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, 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.

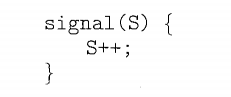
**Semaphores:**

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

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



The definition of signal () is as follows:



Based on the range of values semaphore is classified into two types:

* Binary semaphore : can range only between 0 and 1
* Counting/General Semaphore: can range over an unrestricted domain.

Generally in counting semaphores , the semaphore is initialized to the number of resources available. If any process want to use a resource performs a wait() operation and upon release a resource, it performs a signal() operation.

When semaphore count reaches to 0 , processes that wish to use a resource will block until the count becomes greater than 0.