Unit -2 |Concurrent Processes

**LECTURE 1**

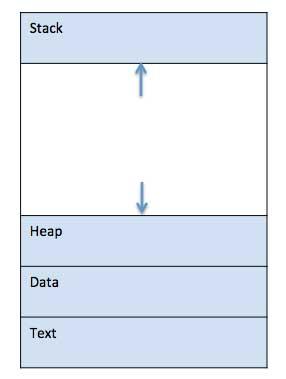
## Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

*A process is defined as an entity which represents the basic unit of work to be implemented in the system.*

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −



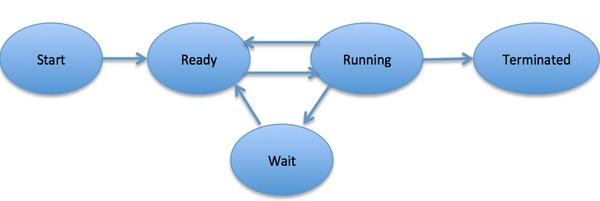
| **S.No** | **Component & Description** |
| --- | --- |
| 1 | **Stack**  The process Stack contains the temporary data such as method/function parameters, return address and local variables. |
| 2 | **Heap**  This is dynamically allocated memory to a process during its run time. |
| 3 | **Text**  This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. |
| 4 | **Data**  This section contains the global and static variables. |

## Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

| **S.No** | **State & Description** |
| --- | --- |
| 1 | **Start**  This is the initial state when a process is first started/created. |
| 2 | **Ready**  The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process. |
| 3 | **Running**  Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions. |
| 4 | **Waiting**  Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available. |
| 5 | **Terminated or Exit**  Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory. |



## Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

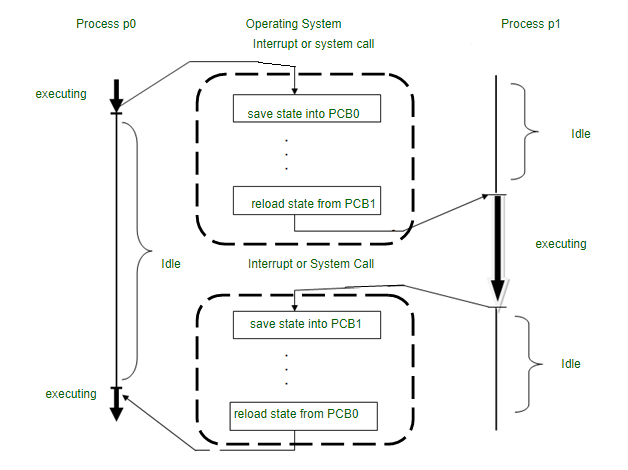
| **S.N.** | **Information & Description** |
| --- | --- |
| 1 | **Process State**  The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**  This is required to allow/disallow access to system resources. |
| 3 | **Process ID**  Unique identification for each of the process in the operating system. |
| 4 | **Pointer**  A pointer to parent process. |
| 5 | **Program Counter**  Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**  Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**  Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**  This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**  This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**  This includes a list of I/O devices allocated to the process. |

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

CPU Switches From Process To Process



**Lectrure 2**

# **Operations on Processes**

There are many operations that can be performed on processes. Some of these are process creation, process preemption, process blocking, and process termination. These are given in detail as follows −

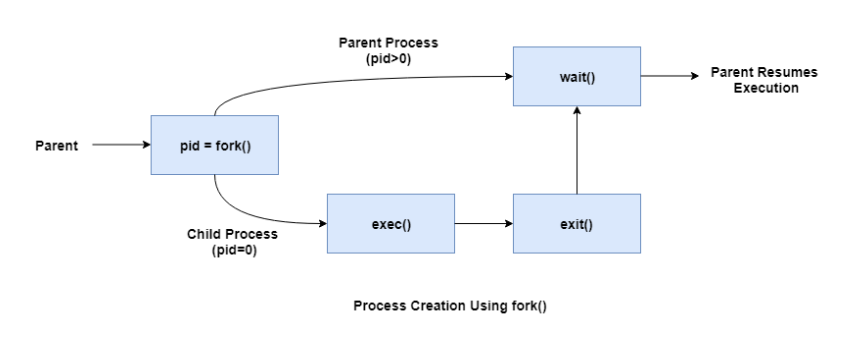
**Process Creation**

Processes need to be created in the system for different operations. This can be done by the following events −

* User request for process creation
* System initialization
* Execution of a process creation system call by a running process
* Batch job initialization

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows −



## Process Termination

Whenever the process finishes executing its final statement and asks the operating system to delete it by using exit() system call.

At that point of time the process may return the status value to its parent process with the help of wait() system call.

All the resources of the process including physical and virtual memory, open files, I/O buffers are deallocated by the operating system.

### **Reasons for process termination**

The reasons that the process may terminate the execution of one of its children are as follows −

* The child exceeds its usage of resources that it has been allocated.
* The task that is assigned to the child is no longer required.
* The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates either normally or abnormally, then all its children have to be terminated. This concept is referred to as cascading termination.

**Inter-process Communication:**

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. So, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

∙ Information sharing

∙ Computation speedup

∙ Modularity

∙ Convenience

Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. These are some fundamental models of inter-process communication:

1) shared memory

2) message passing

3) Naming

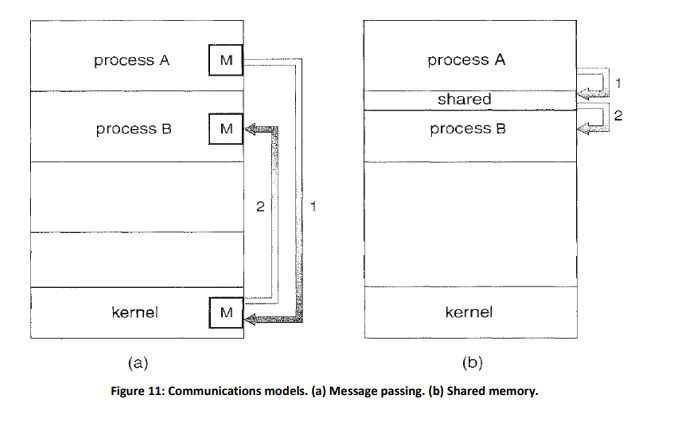
4) Synchronization

5) Buffering

**Shared Memory:**

In the shared-memory model, a region of memory that is shared by cooperating processes is established. A shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Processes can then exchange information by reading and writing data to the shared region. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Shared memory is faster than message passing, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. Shared memory allows maximum speed and convenience of communication.

**  
Message passing:**

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a chat program.

Message passing is slower than Shared memory, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

Message passing is useful for exchanging smaller amounts of data and is also easier to implement than shared memory.

The actual function of message-passing is normally provided in the form of a pair of primitives:

∙ Send(message)

∙ Receive(message)

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. Here are several methods for logically implementing a link and the send () / receive () operations:

∙ Direct or indirect communication

∙ Synchronous or asynchronous communication

∙ Automatic or explicit buffering

**Addressing (Naming):**

Processes that want to communicate must have a way to refer to each other. The various schemes for specifying processes in send and receive primitives are of two types:

1. Direct communication

2. Indirect communication

**Direct Communication:** In direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

∙ send (P, message) - Send a message to process P.

∙ receive (Q, message)- Receive a message from process Q.

A communication link in this scheme has the following properties:

∙ A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

∙ A link is associated with exactly two processes.

∙ Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing, that is, both the sender process and the receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

∙ send(P, message) - Send a message to process P.

∙ receive (id, message) -Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

**Indirect Communication:** In indirect communication, the messages are sent to and received from mailboxes, or ports. Each mailbox has a unique identification. Two processes can communicate only if the processes have a shared mailbox. The send() and receive() primitives are defined as follows:

∙ send (A, message) -Send a message to mailbox A.

∙ receive (A, message)-Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

∙ A link is established between a pair of processes only if both members of the pair have a shared mailbox.

∙ A link may be associated with more than two processes.

∙ Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

A mailbox may be owned either by a process or by the operating system. When a process that owns a mailbox terminates, the mailbox disappears. If mailbox is owned by the operating system, then it must provide a mechanism that allows a process to: Create a new mailbox, Send and receive messages through the mailbox, Delete a mailbox.

**Synchronization:**

Communication between processes takes place through calls to send () and receive () primitives. Message passing may be either blocking or non-blocking also known as synchronous and asynchronous.

**∙ Blocking send** - The sending process is blocked until the message is received by the receiving process or by the mailbox.

**∙ Non-blocking send** - The sending process sends the message and resumes operation.

**∙ Blocking receive** - The receiver blocks until a message is available.

**∙ Non-blocking receive** - The receiver retrieves either a valid message or a null.

Different combinations of send () and receive () are possible. When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver. This combination allows for tight synchronization between processes.

**Buffering:**

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

**∙ Zero capacity** - The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**∙ Bounded capacity** - The queue has finite length. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

**∙ Unbounded capacity** - The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

**Lecture 3**

**Principles of concurrency:**

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency. To achieve the consistency of shared data we need to understand the principles of concurrency which are given below:

**Race Condition:**

∙ A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples:

− Suppose that two processes, P1 and P2, share the global variable X. At some point in its execution, P1 updates X to the value 1, and at some point in its execution, P2 updates X to the value 2. Thus, the two tasks are in a race to write variable X. In this example the “loser” of the race (the process that updates last) determines the final value of X.

− Consider two process, P3 and P4, that share global variables b and c, with initial values b = 1 and c = 2. At some point in its execution, P3 executes the assignment b = b + c, and at some point in its execution, P4 executes the assignment c = b + c. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments. If P3 executes its assignment statement first, then the final values are b = 3 and c = 5. If P4 executes its assignment statement first, then the final values are b = 4 and c = 3.

**Operating System Concerns:**

Design and management issues for concurrency are as follows:

1. The OS must be able to keep track of the various processes. This is done with the use of process control blocks.

2. The OS must allocate and de-allocate various resources for each active process.

3. The OS must protect the data and physical resources of each process against unintended interference by other processes.

4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

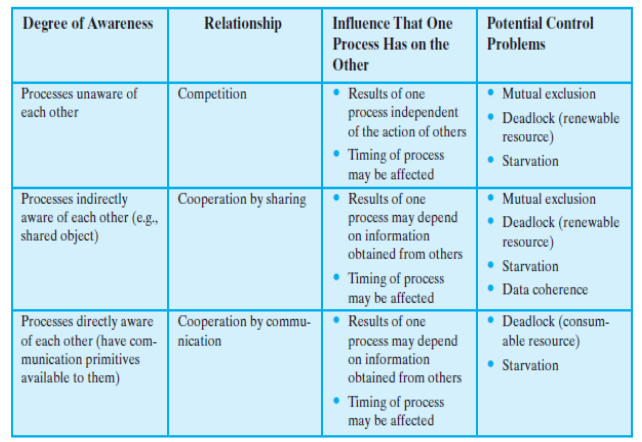
**Process Interaction:**

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other’s existence. Following are the degrees of awareness of process:

1. Processes unaware of each other

2. Processes indirectly aware of each other

3. Processes directly aware of each other



**Requirements for Mutual Exclusion:**

Mutual exclusion should meet the following requirements:

Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.

∙ A process that halts in its noncritical section must do so without interfering with other processes.

∙ It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.

∙ When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

∙ No assumptions are made about relative process speeds or number of processors.

∙ A process remains inside its critical section for a finite time only.

**The Critical-Section Problem:**

Consider a system consisting of n processes {Po, P1 , ... , Pn - 1}. Each process has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file, and so on. “When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.”

In Critical-section problem:

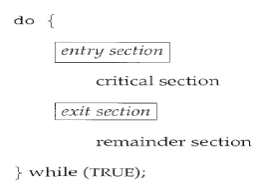
∙ Each process must request permission to enter its critical section.

∙ 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 as:



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.

**Lecture 4**

# **Bakery Algorithm**

Lamport proposed a bakery algorithm, a software solution, for the n process mutual exclusion problem. This algorithm solves a critical problem, following the fairest, first come, first serve principle.

This algorithm is known as the bakery algorithm as this type of scheduling is adopted in bakeries where token numbers are issued to set the order of customers. When a customer enters a bakery store, he gets a unique token number on its entry. The global counter displays the number of customers currently being served, and all other customers must wait at that time. Once the baker finishes serving the current customer, the next number is displayed. The customer with the next token is now being served.

Similarly, in Lamport's bakery algorithm, processes are treated as customers. In this, each process waiting to enter its critical section gets a token number, and the process with the lowest number enters the critical section. If two processes have the same token number, the process with a lower process ID enters its critical section.

### **Algorithm of Lamport's bakery algorithm**

do

{

    entering[i] :=  true; // show interest in critical section

   // get a token number

    number[i] := 1 + max(number[0],  number[1], ..., number[n - 1]);

    entering [i] :=  false;

    for ( j :=  0 ;  j**<n**; j++)

    {

 // busy wait until process Pj receives its token number

        while (entering [j])

        { ; } // do nothing

       while (number[j] !=  0 &&  (number[j], j) **<** (number[i], i)) // token comparison

         { ; } // do nothing

       }

        // critical section

         number[i] :=  0;  // Exit section

        // remainder section

    } while(1);

Structure of process Pi for Lamport's bakery algorithm to critical section problem.

**Explanation -**

**This algorithm uses the following two Boolean variables.**

1. boolean entering[n];
2. int number[n];

All entering variables are initialized to false, and n integer variables numbers are all initialized to 0. The value of integer variables is used to form token numbers.

When a process wishes to enter a critical section, it chooses a greater token number than any earlier number.

Consider a process Pi wishes to enter a critical section, it sets

**entering[i]** to true to make other processes aware that it is choosing a token number. It then chooses a token number greater than those held by other processes and writes its token number. Then it sets **entering[i]** to false after reading them. Then It enters a loop to evaluate the status of other processes. It waits until some other process Pj is choosing its token number.

Pi then waits until all processes with smaller token numbers or the same token number but with higher priority are served fast.

When the process has finished with its critical section execution, it resets its number variable to 0.

**The Bakery algorithm meets all the requirements of the critical section problem.**

**LECTURE 5**

## Mutual Exclusion

**Mutual Exclusion** is a property of [process synchronization](https://www.geeksforgeeks.org/introduction-of-process-synchronization/) that states that “no two processes can exist in the critical section at any given point of time”. The term was first coined by Dijkstra. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

The need for mutual exclusion comes with concurrency. There are several kinds of concurrent execution:

1. Interrupt handlers
2. Interleaved, preemptively scheduled processes/threads
3. Multiprocessor clusters, with shared memory
4. Distributed systems

Mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

The requirement of mutual exclusion is that when process P1 is accessing a shared resource R1, another process should not be able to access resource R1 until process P1 has finished its operation with resource R1.

Examples of such resources include files, I/O devices such as printers, and shared data structures.

Software Based Solution to Critical Section Problem:

**1. Peterson’s Solution:** Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 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:

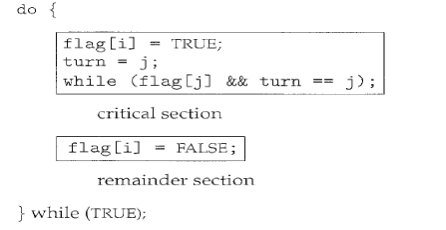
**int turn;**

**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. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.



In this solution following conditions are fulfilled:

∙ 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 note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that Po and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes ---say, Pi---must have successfully executed the while statement, whereas P; had to execute at least one additional statement ("turn== j"). However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section; as a result, mutual exclusion is preserved. To prove properties 2 and 3, we note that a process P; 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 Pi is not ready to enter the critical section, then flag [j] ==false, and P; 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 P; will enter the critical section. If turn== j, then Pi will enter the critical section. However, once Pi exits its critical section, it will reset flag [j] to false, allowing P; to enter its critical section. If Pi resets flag [j] to true, it must also set turn to i. Thus, since P; does not change the value of the variable turn while executing the while statement, P; will enter the critical section (progress) after at most one entry by P1 (bounded waiting).

**LECTURE 6**

**2. Dekker’s Algorithm**

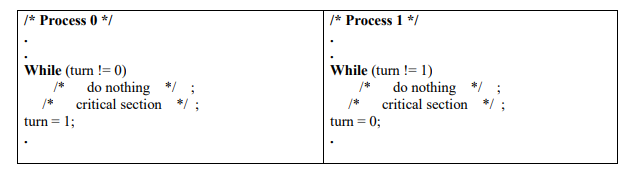
It is a simple and efficient algorithm that allows only one process to access a shared resource at a time. The algorithm achieves mutual exclusion by using two flags that indicate each process's intent to enter the critical section. By alternating the flags' use and checking if the other process's flag is set, the algorithm ensures that only one process enters the critical section at a time.

Algorithm

The algorithm uses flags to indicate the intention of each process to enter a critical section, and a turn variable to determine which process is allowed to enter the critical section first.

1st Attempt

A process wishing to execute its critical section first examines the contents of turn (a global memory location). If the value of turn is equal to the number of the process, then the process may proceed to its critical section. Otherwise, it is forced to wait. Waiting process repeatedly reads the value of turn until it is allowed to enter its critical section. This procedure is known as busy waiting or spin waiting, because the waiting process do nothing productive and consumes processor time while waiting for its chance.



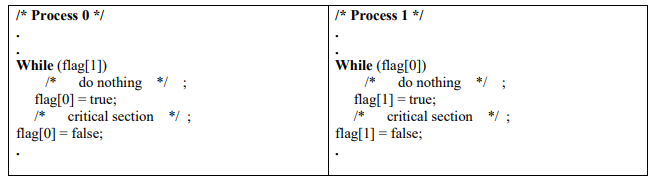
This solution guarantees the mutual exclusion property but has drawbacks:

− If one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

2nd Attempt

The flaw in the first attempt is that it stores the name of the process that may enter its critical section and if one process fails, the other process is permanently blocked. To overcome this problem a Boolean vector flag is defined.

If one process wants to enter its critical section it first checks the other process flag until that flag has the value false, indicating that the other process is not in its critical section. The checking process immediately sets its own flag to true and proceeds to its critical section. When it leaves its critical section it sets its flag to false.



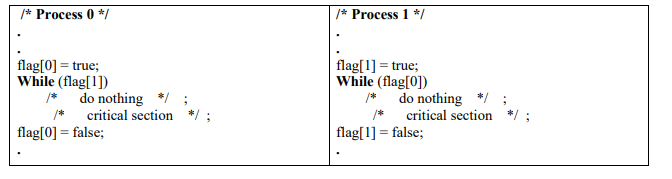
In this solution if one process fails outside the critical section including the flag setting code then the other process is not blocked because in this condition flag of the other process is always false. However, this solution has two drawbacks:

− If one process fails inside its critical section or after setting its flag to true just before entering its critical section, then the other process is permanently blocked.

− It does not guarantee mutual exclusion.

3rd Attempt

Because a process can change its state after the other process has checked it before the other process can enter its critical section, the second attempt failed. Perhaps we can fix this problem with a simple interchange of two statements as:



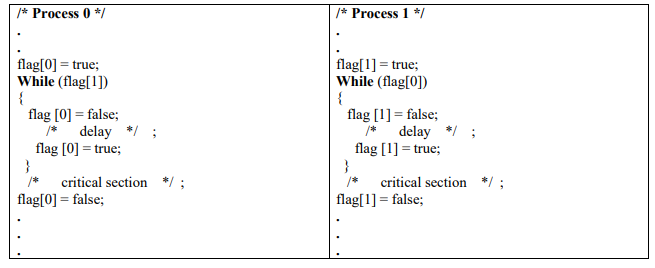
This solution guarantees mutual exclusion for example consider, if P0 sets flag [0] to true, P1 cannot enter its critical section. If P1 already in its critical section P0 sets its flag then P0 will be blocked by the while statement.

Problem:

− If both process set their flag to true before while statement, then each will think that other one is in its critical section, causing deadlock.

4th Attempt

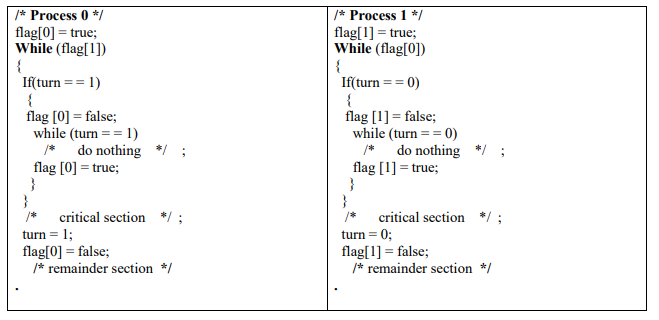
In the third attempt, a process sets its state without knowing the state of the other process. We can fix this in a way that makes each process more deferential: each process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag to defer to the other process.



This solution guarantees mutual exclusion but is still flawed. Consider the following sequence of events: P0 sets flag [0] to true. P1 sets flag [1] to true. P0 checks flag [1]. P1 checks flag [0]. P0 sets flag [0] to false. P1 sets flag [1] to false. P0 sets flag [0] to true. P1 sets flag [1] to true.

This sequence could be extended indefinitely, and neither process could enter its critical section.

**A Correct Solution**



When P0 wants to enter its critical section, it sets its flag to true. It then checks the flag of P1. If that is false, P0 may immediately enter its critical section. Otherwise, P0 consults turn. If it finds that turn = 0, then it knows that it is its turn to insist and periodically checks P1’s flag. P1 will at some point note that it is its turn to defer and set its flag to false, allowing P0 to proceed. After P0 has used its critical section, it sets its flag to false to free the critical section and sets turn to 1 to transfer the right to insist to P1.

**LECTURE 7**

**Semaphores:**

Controlling synchronization by using an abstract data type, called a semaphore, was proposed by Dijkstra in 1965.

∙ Semaphores are easily implemented in OS and provide a general purpose solution to controlling access to critical section.

∙ A semaphore is an integer value used for signaling among processes.

∙ A semaphore (S) is an integer variable that can perform following atomic operations:

1) Initialization

2) Decrement

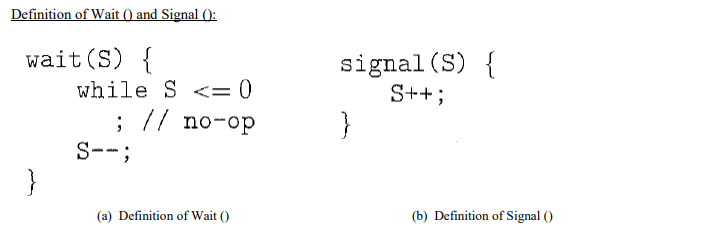
3) Increment

Decrement results in the blocking of a process and increment results in the unlocking of a process.

∙ Apart from initialization a semaphore (S) is accessed only through two standard atomic operations:

1) Wait() or Down() 🡪 P (from the Dutch proberen, “To Test”)

2) Signal() or Up() 🡪 V (from verhogen, “To Increment” )



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.

This situation is a critical section problem and can be resolved in either of two ways:

1) By using Counting Semaphore

2) By using Binary Semaphore

1) Counting Semaphore

∙ The value of a counting semaphore can range over an unrestricted domain.

∙ It is also known as general semaphore.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

2)Binary Semaphore

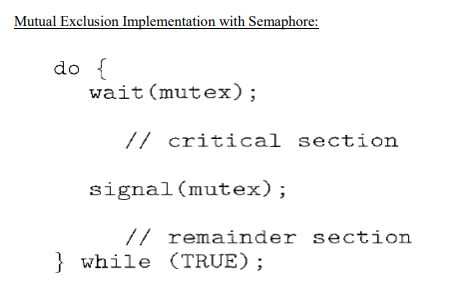
∙ The value of a binary semaphore can range only between 0 and 1.

∙ binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

∙ In this, queue is used to hold the processes waiting on the semaphore.

∙ FIFO, policy is used to remove the processes from the queue.

∙ binary semaphores are used to deal with the critical-section problem for multiple processes. The n processes share a semaphore, mutex, initialized to 1.



## Mutex

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for [Mutual Exclusion](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/) Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

Mutex uses a priority inheritance mechanism to avoid [priority inversion](https://www.geeksforgeeks.org/priority-inversion-what-the-heck/) issues. The priority inheritance mechanism keeps higher-priority processes in the blocked state for the minimum possible time. However, this cannot avoid the priority inversion problem, but it can reduce its effect up to an extent.

### **Advantages of Mutex**

* No race condition arises, as only one process is in the critical section at a time.
* Data remains consistent and it helps in maintaining integrity.
* It’s a simple locking mechanism that can beobtained by a process before entering into a critical section and released while leaving the critical section.

### **Disadvantages of Mutex**

* If after entering into the critical section, the thread sleeps or gets preempted by a high-priority process, no other thread can enter into the critical section. This can lead to starvation.
* When the previous thread leaves the critical section, then only other processes can enter into it, there is no other mechanism to lock or unlock the critical section.
* Implementation of mutex can lead to busy waiting that leads to the wastage of the CPU cycle.

**Lecture 8**

**Producer-Consumer Problem:**

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

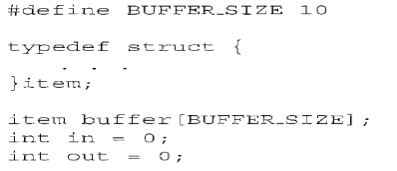
Below are a few points that considered as the problems occur in Producer-Consumer:

* The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
* Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
* Accessing memory buffer should not be allowed to producer and consumer at the same time.

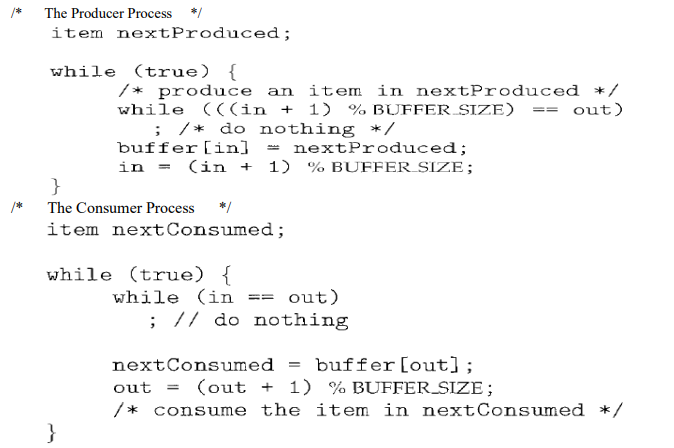
To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. To solve this problem let’s take the bounded buffer. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The following variables reside in a region of memory shared by the producer and consumer processes:

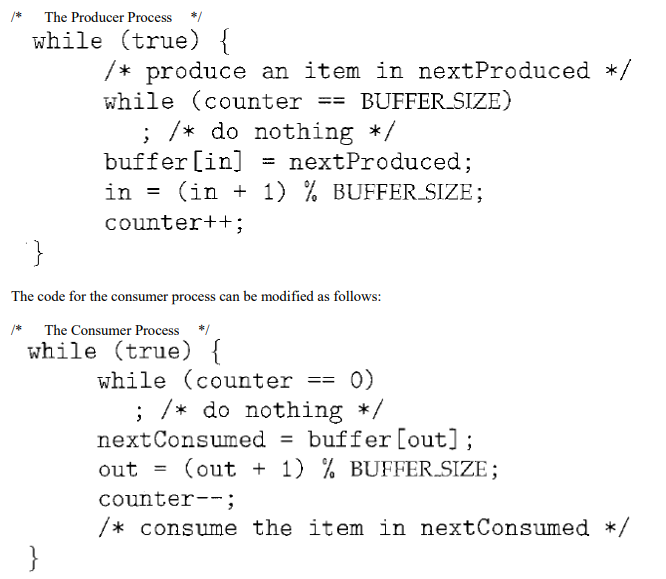


The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer and the variable out points to the first full position in the buffer. The buffer is empty when in = = out and the buffer is full when ((in+ 1) % BUFFER\_SIZE) = = out.



This scheme allows at most BUFFER\_SIZE - 1 items in the buffer at the same time.

To overcome the limitation of BUFFER\_SIZE – 1, we add an integer variable counter, which is initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:



**LECTURE 9**

**TestAndSet() Instruction**

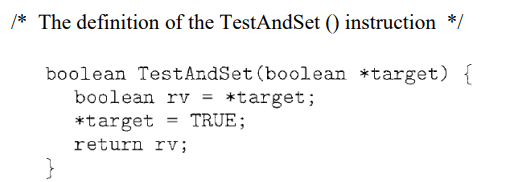
∙ Test-and-set instructions are offered by RAM.

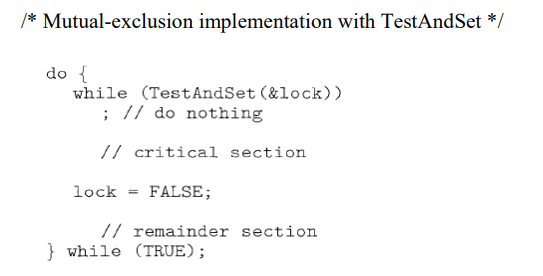
∙ Test-and-set instructions are used to write to a memory location and return its old value as a single atomic (non-interruptible) operation.

∙ 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.

A lock can be built using a TestAndSet() instruction as follows:





Problem:

No bounded waiting.

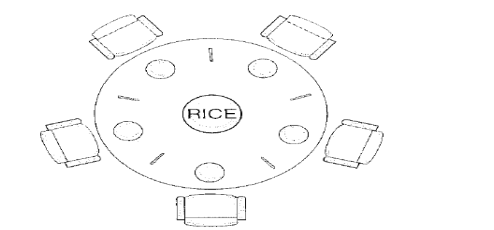
**LECTURE 10**

**Classical Problems of Synchronization:**

**The Dining-Philosophers problem:**

∙ Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of the table is a large plate of food. The philosophers can only afford five chopsticks. One chopstick is placed between each pair of philosophers and they agree that each will only use the chopstick to his immediate right and left.

∙ The problem is to design a set of processes for philosophers such that each philosopher can eat periodically and none dies for hunger. A philosopher to the left or right of a dining philosophers cannot eat while the dining philosopher is eating, since forks are shared resources.



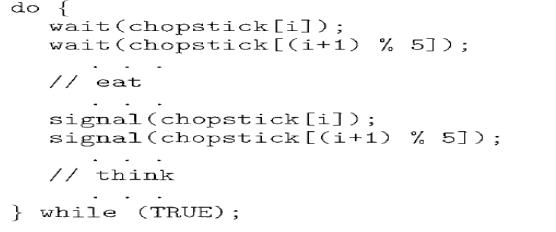
Solution to this problem using semaphore can be implemented as:

Each philosopher picks up his right chopstick before he tried to pick up his left chopstick. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; he releases his chopstick by executing the signal operation on the appropriate semaphore.

**/\* Shared Data \*/**

Semaphore chopstick[5]; // All the elements of chopstick are initialized to 1.

**/\* Structure of philosopher i \*/**

****

Although this solution guarantees that no two neighbours are eating simultaneously, but it could create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Some possible solutions to overcome the deadlock problem are:

1. Allow at most four philosophers to be sitting simultaneously at the table.

2. Allow a philosopher to pick up his chopsticks only if both chopsticks are available.

3. Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

**LECTURE 11**

**Sleeping Barber Problem:**

Consider a barber’s shop where there is only one barber, one barber chair and a number of waiting chairs for the customers. When there are no customers the barber sits on the barber chair and sleeps. When a customer arrives he awakes the barber or waits in one of the vacant chairs if the barber is cutting someone else’s hair. When all the chairs are full, the newly arrived customer simply leaves.

**Problems**

∙ There might be a scenario in which the customer ends up waiting on a barber and a barber waiting on the customer, which would result to a deadlock.

∙ Then there might also be a case of starvation when the customers don’t follow an order to cut hair, as some people won’t get a haircut even though they had been waiting long.

The solution to these problems involves the use of three semaphores out of which one is a mutex (binary semaphore). They are:

∙ Customers: Helps count the waiting customers.

∙ Barber: To check the status of the barber, if he is idle or not.

∙ accessSeats: A mutex which allows the customers to get exclusive access to the number of free seats and allows them to increase or decrease the number.

∙ NumberOfFreeSeats: To keep the count of the available seats, so that the customer can either decide to wait if there is a seat free or leave if there are none.

**The Procedure**

∙ When the barber first comes to the shop, he looks out for any customers i.e. calls P(Customers), if there are none he goes to sleep.

∙ Now when a customer arrives, the customer tries to get access to the accessSeats mutex i.e. he calls P(accessSeats), thereby setting a lock.

∙ If no free seat (barber chair and waiting chairs) is available he releases the lock i.e. does a V(accessSeats) and leaves the shop.

∙ If there is a free seat he first decreases the number of free seats by one and he calls V(Customers) to notify the barber that he wants to cut.

∙ Then the customer releases the lock on the accessSeats mutex by calling V(accessSeats).

∙ Meanwhile when V(Customers) was called the barber awakes.

∙ The barber locks the accessSeats mutex as he wants to increase the number of free seats available, as the just arrived customer may sit on the barber’s chair if that is free.

∙ Now the barber releases the lock on the accessSeats mutex so that other customers can access it to the see the status of the free seats.

∙ The barber now calls a V(Barber), i.e. he tells the customer that he is available to cut.

∙ The customer now calls a P(Barber), i.e. he tries to get exclusive access of the barber to cut his hair.

∙ The customer gets a haircut from the barber and as soon as he is done, the barber goes back to sleep if there are no customers or waits for the next customer to alert him.

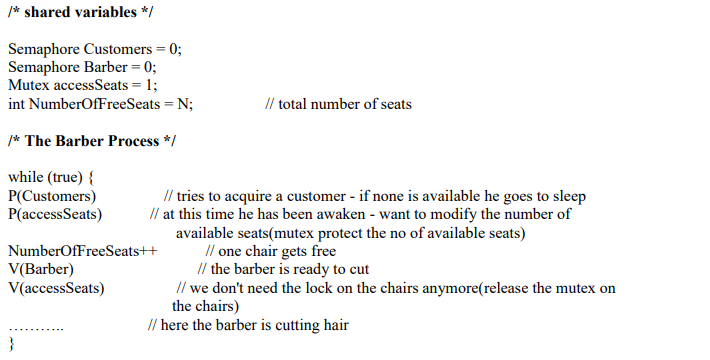
∙ When another customer arrives, he repeats the above procedure again.

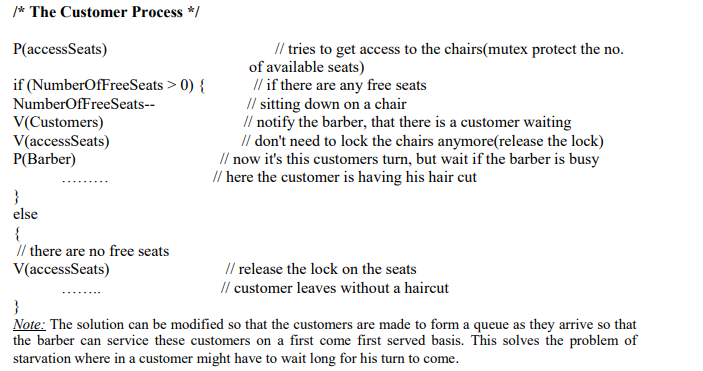
∙ If the barber is busy then the customer decides to wait on one of the free waiting seats.

∙ If there are no customers, then the barber goes back to sleep.

**Implementation**

The following pseudo-code guarantees synchronization between barber and customer and is deadlock free, but may lead to starvation of a customer.





Important Questions:

1. What do you understand by Critical Section? (AKTU 2015-16)
2. What do you mean by Concurrent Processes? (AKTU 2021-22)
3. Explain in detail about the Mutual Exclusion and Critical Section Problem. (AKTU 2021-22)
4. Explain in detail about the Dining Philosopher Problem. (AKTU 2021-22)
5. Explain in detail about the Inter Process Communication models and Schemes. (AKTU 2021-22)
6. What is a Critical Section problem? Give the conditions that a solution to the critical section problem must satisfy. (AKTU 2018-19)
7. Explain what semaphores are, their usage, implementation given to avoid busy waiting and binary semaphores. (AKTU 2018-19)
8. What is Producer Consumer problem? How it can illustrate the classical problem of synchronization? Explain. (AKTU 2018-19)
9. What is the use of IPC? (AKTU 2017-18)
10. Discuss mutual exclusion using test and set() instruction. (AKTU 2017-18)
11. Give the principles, mutual exclusion in critical section problem. Also discuss how well these principles are followed in Dekker’s solution. (AKTU 2016-17)