Multiprogramming os - more than one process at a time in memory -

Processes = p1, p2, p3… pn

Resources = r1, r2;

Some resources are there which are shareable but in mutual exclusion fashion. If all processes try to access such resource at a time this will cause inconsistency.

So we need some policy about which process will use the resource.

When order of execution can change result that is called race condition.

That area where we access our shared resources that is critical section.

If we didn’t work properly system can give inconsistent result.

Characteristics of solution of critical section:-

1. Mutual exclusion ( Mandatory criteria ): Only one process will access critical section at a time.
2. Progress ( Mandatory criteria): only those process should compete or race which want to access critical section.
3. Bounded Wait ( Optional criteria ): there must be a maximum bound upto which a process need to wait before it can access critical section.

**Process synchronization**

* Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled, and we are able to minimizing chances of inconsistent data.
* Maintaining data consistency demands mechanisms to ensure synchronized execution of processes.
* Process Synchronization was introduced to handle problems that arose while multiple process executions.

**Race Condition:**

* A race condition is a situation that may occur inside a critical section.
* This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.
* Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction.
* Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

**Critical section:**

* Critical section is a code segment that can be accessed by only one process at a time.
* Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

The critical section is given as follows:

do

{

Entry Section

Critical Section

Exit Section

Remainder Section

}

While (TRUE);

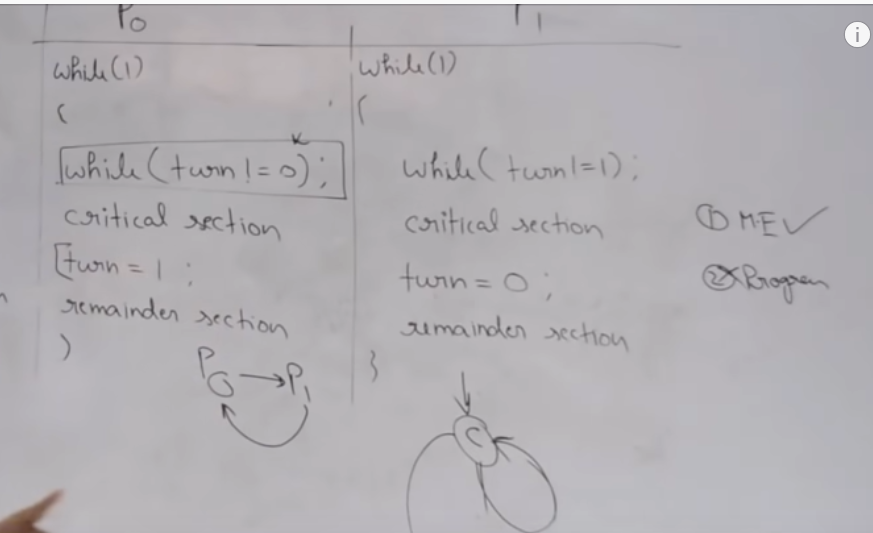
In the above diagram, the entry sections handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that critical section is free.

**Any solution to the critical section problem must satisfy three conditions:**

* **Mutual Exclusion** : If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
* **Progress** : Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
* **Bounded Waiting** : A bound must exist 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.
* Or, Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

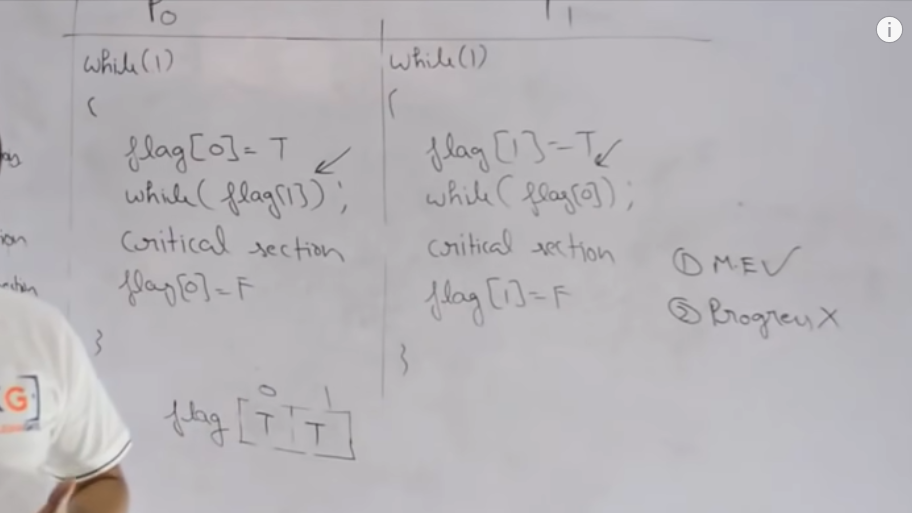
**Algo 1 ( Using Turn Variable ):**

* Satisfying mutual exclusion but don’t satisfy progress.

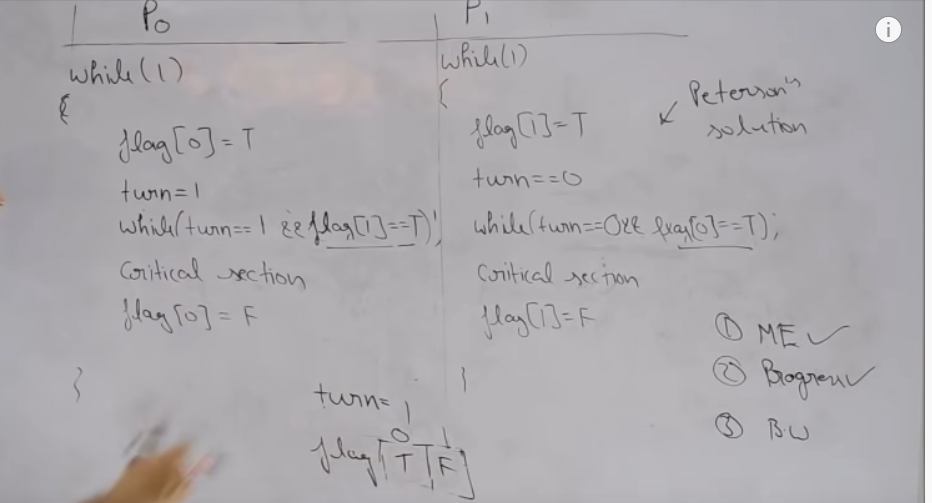


**Algo 2 ( Using Flag variables ):**

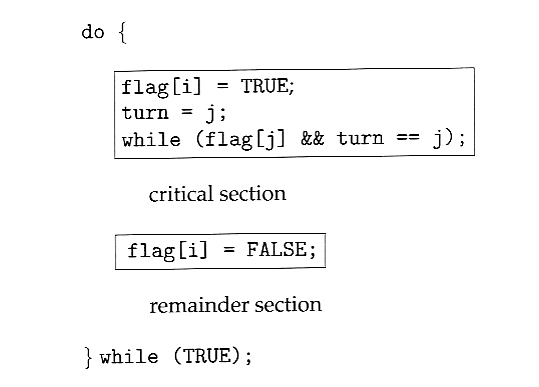
* Satisfy Mutual Exclusion and somewhat satisfy progress but in one case put the system in deadlock.

****

**Algo 3 ( Peterson’s Solution ):**



* Peterson's Solution is a classic software-based solution to the critical section problem.
* It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
* Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections.
* For convenience of discussion, "this" process is Pi, and the "other" process is Pj. ( I.e. j = 1 - i )
* Peterson's solution requires two shared data items:
* int turn - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
* boolean flag[ 2 ] - Indicates when a process wants to enter into their critical section. When process i wants to enter their critical section, it sets flag[ i ] to true.
* In the following diagram, the entry and exit sections are enclosed in boxes.
* In the entry section, process i first raises a flag indicating a desire to enter the critical section.
* Then turn is set to j to allow the other process to enter their critical section if process j so desires.
* The while loop is a busy loop ( notice the semicolon at the end ), which makes process i wait as long as process j has the turn and wants to enter the critical section.
* Process i lowers the flag[ i ] in the exit section, allowing process j to continue if it has been waiting.



**Producer Consumer Problem:**

**Problem Statement** – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

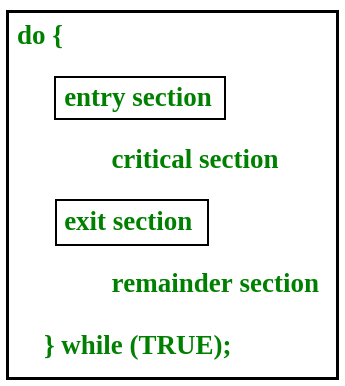
**To solve this problem, we need two counting semaphores** – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Initialization of semaphores –  
mutex = 1  
Full = 0 // Initially, all slots are empty. Thus full slots are 0  
Empty = n // All slots are empty initially

**Solution for Producer –**

*do{*

*//produce an item*

*wait(empty);*

*wait(mutex);*

*//place in buffer*

*signal(mutex);*

*signal(full);*

*}while(true)*

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

**Solution for Consumer –**

*do{*

*wait(full);*

*wait(mutex);*

*// remove item from buffer*

*signal(mutex);*

*signal(empty);*

*// consumes item*

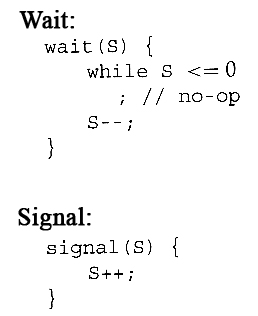
*}while(true)*

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

**Semaphores:**

* A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal().

The definition of wait () and signal are given below:



when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

**There are two types of semaphores :**

1. Binary Semaphores
2. Counting Semaphores

**Binary Semaphores:**

* They can only be either 0 or 1.
* They are also known as mutex locks, as the locks can provide mutual exclusion.
* All the processes can share the same mutex semaphore that is initialized to 1.
* Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section.
* When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.