Introduction of Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process**: Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition

counter++ could be implemented as:

register1 = counter

register1 = register1 + 1

counter = register1

counter-- could be implemented as:

register2 = counter

register2 = register2 – 1

counter = register2

Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = counter	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = counter	{register2 = 5}
S3: consumer execute register2 = register2 – 1	{register2 = 4}
S4: producer execute counter = register1	{counter = 6 }
S5: consumer execute counter = register2	{counter = 4}

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the

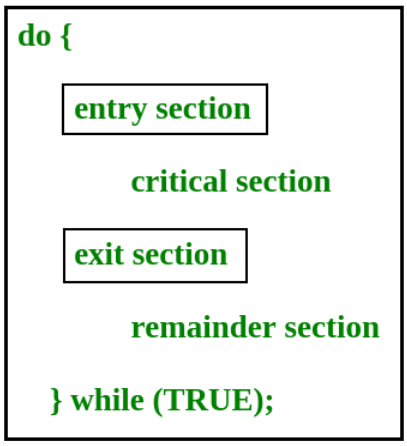
shared variable is wrong so for that all the processes doing race to say that my output is correct this condition known as race condition.

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Critical Section Problem

Consider system of n processes {p0, p1, … pn-1}

- Each process has critical section segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder 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.



In the entry section, the process requests for entry in the **Critical Section.**

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

- **Progress** : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection cannot be postponed indefinitely.
- **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.

Two process solution for critical section:

First Solution: **Boolean turn=0;**

	P_0	P_1

do {
entry section
critical section
exit section
remainder section
} while (TRUE);
	while(1)
	{
	While(turn!=0);
	Critical Section
	Turn=1;
	Remainder Section
	}
		while(1)
		{
		While(turn!=1);
		Critical Section
		Turn=0;
		Remainder Section
		}

Solution first is following Mutual Exclusion and fails to follow Progress as both process spin in some order that is both process can be executed one by one. Any Process P0 or P1 cannot be executed two times sequentially.

Second Solution:

Boolean flag[2] =

F	F
0	**1**

	P_0	P_1

do {
entry section
critical section
exit section
remainder section
} while (TRUE);
	while(1)
	{
	Flag[0]= T;
	while(flag[1]);
	Critical Section
	flag[0]=F;
	Remainder Section
	}
		while(1)
		{
		Flag[1]= T;
		while(flag[0]);
		Critical Section
		flag[1]=F;
		Remainder Section
		}

Solution second Follows mutual Exclusion but fails to follow progress.

Peterson's Solution

Peterson's Solution is a classical software based solution to the critical section problem.
In Peterson's solution, we have two shared variables:

- boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
- int turn : The process whose turn is to enter the critical section.

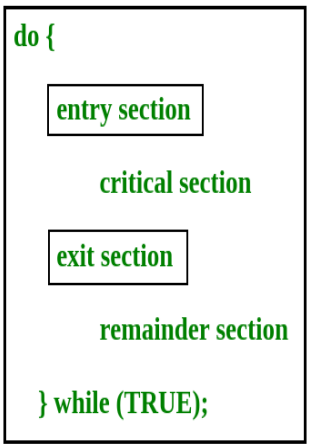
```
do {

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

        critial section

    flag[i] = FALSE ;

        remainder section

}  while (TRUE) ;
```

Second Solution:

Turn =

Flag [2]=

F	F
0	1



P_0

```
while(1)
{
Flag[0]= T;
Turn=1;
while(turn==1 && flag[1]==T);

        Critical Section

flag[0]=F;

        Remainder Section

}
```

P_1

```
while(1)
{
Flag[1]= T;
Turn=0;
while(turn==0 && flag[0]==T);

        Critical Section

flag[1]=F;

        Remainder Section

}
```

- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Peterson's Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

Semaphore

Semaphores in Process Synchronization

- Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- Semaphore is simply an integer variable which is non-negative and shared between threads.
- This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- It uses two basic function wait(S) and Signal(S).

wait(S)
{
While(S<=0); // NO OPERATION
(Busy Waiting)
S=S-1;
}
	signal(S)
	{
	S=S+1;
	}

Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Producer Consumer Problem using Semaphores (Bounded Buffer):

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 –

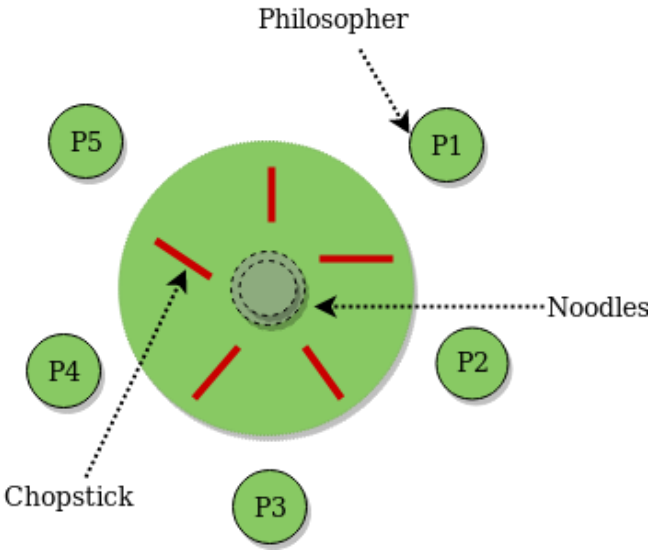
```
do{

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.

Dining Philosopher Problem

The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Semaphore Solution to Dining Philosopher –

Each philosopher is represented by the following pseudo code for Philosopher P[i]:

```
do
{
    THINK;
     PICKUP(CHOPSTICK[i]);
PICKUP(CHOPSTICK[i+1 mod 5]);
     EAT;
     PUTDOWN(CHOPSTICK[i]);
   PUTDOWN(CHOPSTICK[i+1 mod 5]);
   } While( TRUE)
```

```
do
  {
    THINK;
     wait(CHOPSTICK[i]);
    wait(CHOPSTICK[i+1 mod 5]);
     EAT;
     Signal(CHOPSTICK[i]);
    Signal(CHOPSTICK[i+1 mod 5]);
} While( TRUE)
```

There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher.

Readers-Writers Problem

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :
– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:
1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {
    // writer requests for critical section
    wait(wrt);


    // performs the write


    // leaves the critical section
    signal(wrt);
} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.
    - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

```
do {
      // Reader wants to enter the critical section
   wait(mutex);


   // The number of readers has now increased by 1
   readcnt++;


   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even one reader
   // thus we give preference to readers here
   if (readcnt==1)
     wait(wrt);

   // other readers can enter while this current reader is inside
   // the critical section
```

```
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);         // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.