Operating System

Week 7 - Long Descriptive Questions

1.Discuss in detail the usage of semaphores. Write a program in C and explain binary semaphores along with its methods.

Ans.

A critical section execution is handled by a **semaphore**. A semaphore is simply a variable that stores an integer value. This integer can be accessed by two operations: wait() and signal(). When a process enters the critical section, P(s) is invoked and the semaphore s is set to 1. After the process exits the critical section, s is re-initialized to 0. An example of how Process P is executed inside the critical section is shown below:

```
//Some Code

P(s);

//critical section(cs)

//exit from cs

V(s);

//remaining code
```

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that is called the wait function.
A semaphore uses two atomic operations, wait and signal for process synchronization.
A Semaphore is an integer variable, which can be accessed only through two operations wait() and signal().
There are two types of semaphores: Binary Semaphores and 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 the mutex semaphore to 0 and some other process can enter its critical section.

Wait and Signal Operations in Semaphores

Both of these operations are used to implement process synchronization. The goal of this semaphore operation is to get mutual exclusion.

Wait for Operation

This type of semaphore operation helps you to control the entry of a task into the critical section. However, If the value of wait is positive, then the value of the wait argument X is decremented. In the case of negative or zero value, no operation is executed. It is also called P(S) operation.
After the semaphore value is decreased, which becomes negative, the command is held up until the required conditions are satisfied.

```
Copy CodeP(S)
{
   while (S<=0);
   S--;
}
```

Signal operation

This type of Semaphore operation is used to control the exit of a task from a critical section. It helps to increase the value of the argument by 1, which is denoted as V(S).

```
Copy CodeP(S)
{
   while (S>=0);
   S++;
}
```

Example :

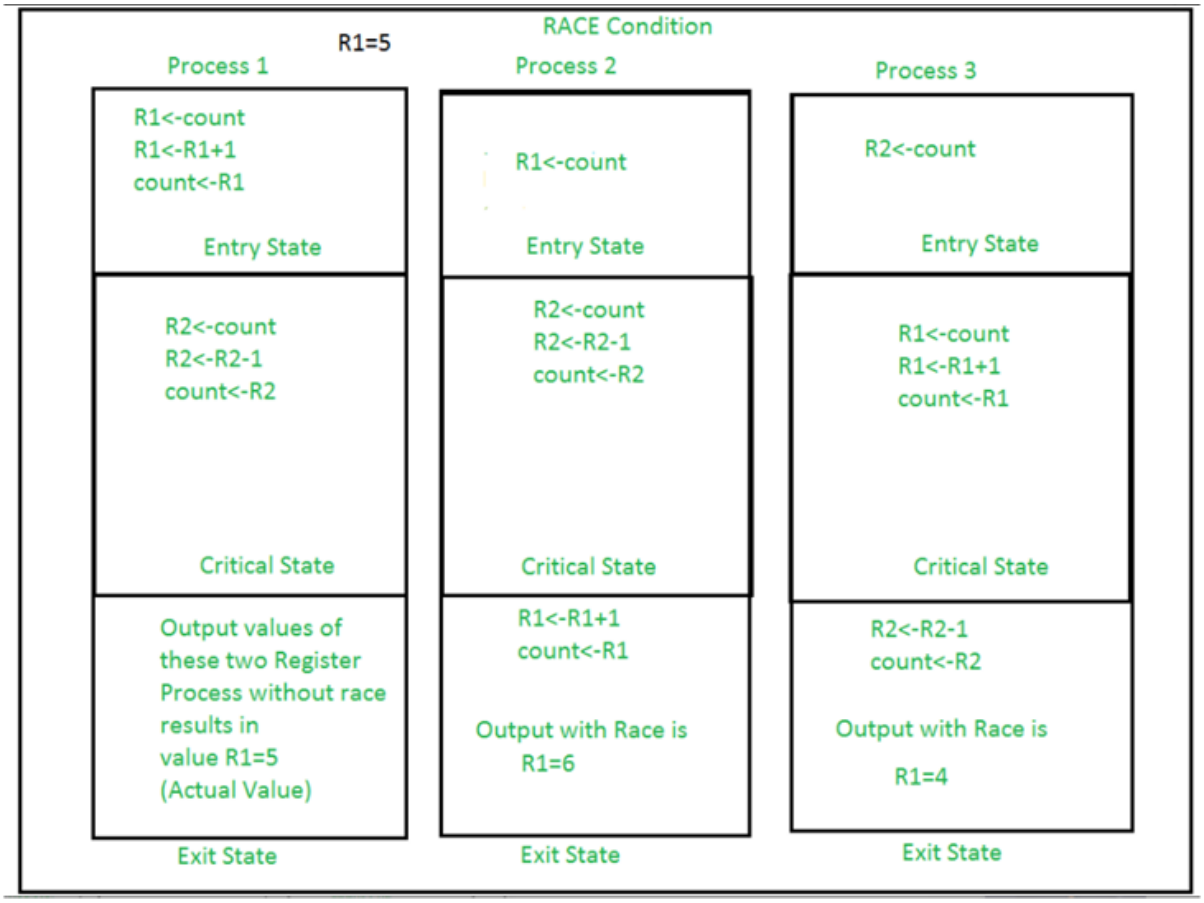
```
Shared var mutex: semaphore = 1;
Process i
   begin
   .
   .
   P(mutex);
   execute CS;
   V(mutex);
   .
   .
   End;
```

- Counting Semaphores: They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

2.With the help of diagram, explain the Race Condition.

Ans.

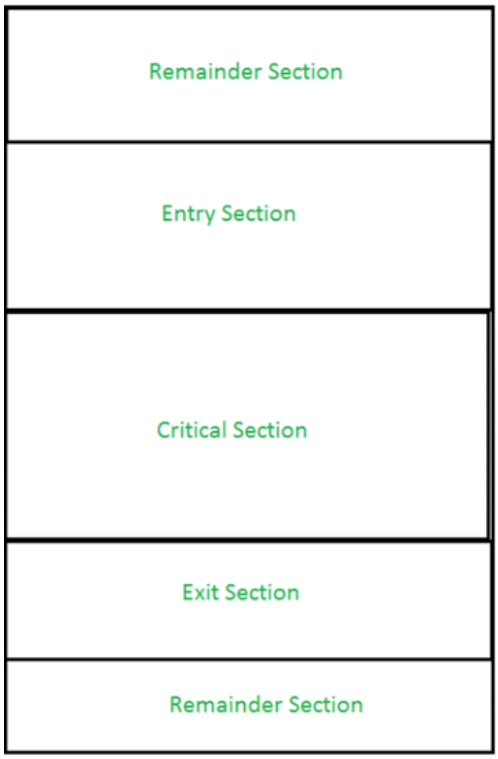
Process Synchronization is mainly used for Cooperating Process that shares the resources. Let us consider an example of racing condition image.



When race conditions occur

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

This condition can be avoided using the technique called Synchronization or Process Synchronization, in which we allow only one process to enter and manipulates the shared data in Critical Section as shown in the diagram of the view of CS.



3. Explain hardware approach in mutual exclusion.

Ans.

There are three algorithms in the hardware approach of solving Process Synchronization problem:

1. Test and Set
2. Swap
3. Unlock and Lock

1. Test and Set:
Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured. Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

Test and Set Pseudocode –
//Shared variable lock initialized to false
boolean lock;

```
boolean TestAndSet (boolean &target){
    boolean rv = target;
    target = true;
    return rv;
}

while(1){
    while (TestAndSet(lock));
    critical section
    lock = false;
    remainder section
}
```

2. Swap:

Swap algorithm is a lot like the TestAndSet algorithm. Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section. Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section. Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured. However, again bounded waiting is not ensured for the very same reason.

Swap Pseudocode –

```
// Shared variable lock initialized to false
// and individual key initialized to false;

boolean lock;
Individual key;

void swap(boolean &a, boolean &b){
    boolean temp = a;
    a = b;
    b = temp;
}

while (1){
    key = true;
    while(key)
        swap(lock,key);
    critical section
    lock = false;
    remainder section
}
```

3. Unlock and Lock :

Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting. A ready queue is maintained with respect to the process in the critical section. All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially. Once the ith process gets out of the critical section, it

does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms. Instead, it checks if there is any process waiting in the queue. The queue is taken to be a circular queue. j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section. If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section. If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section. This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm.
Unlock and Lock Pseudocode –

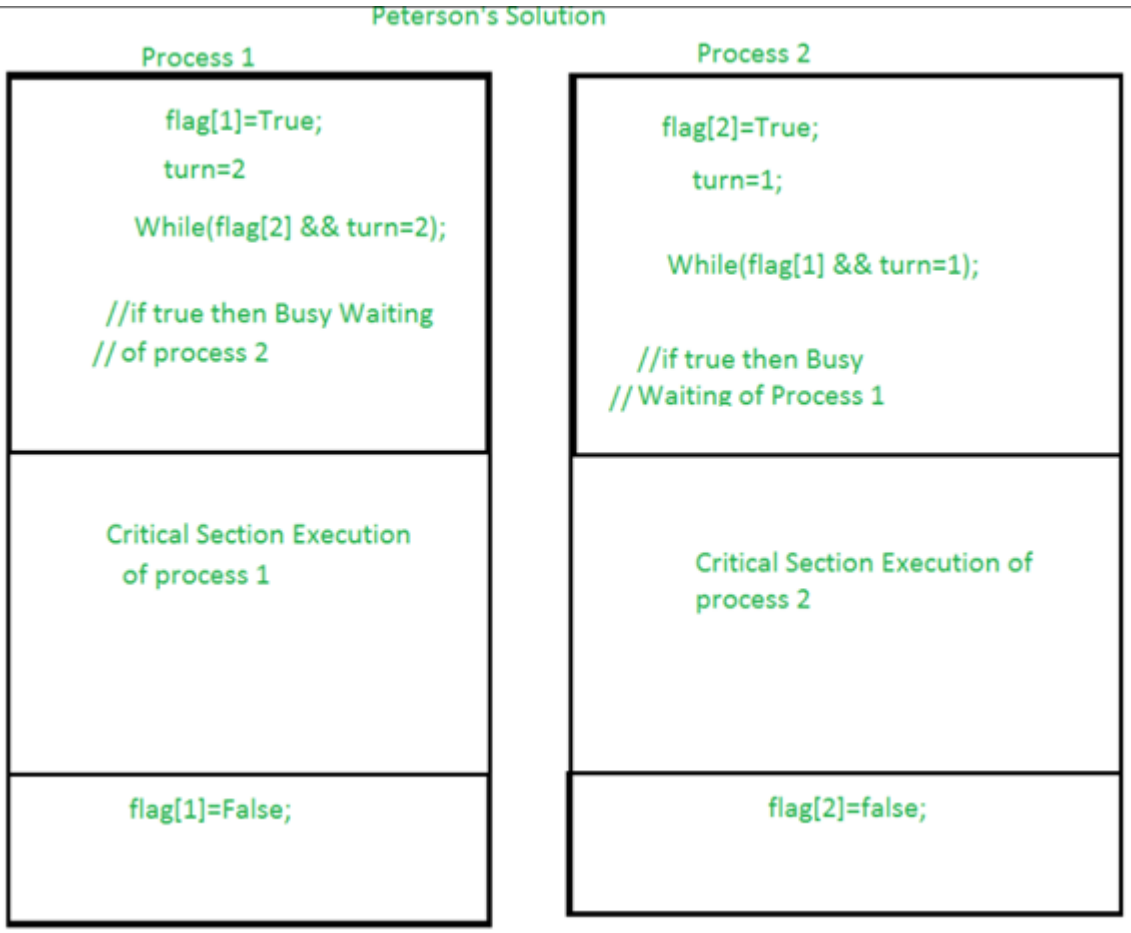
```
// Shared variable lock initialized to false
// and individual key initialized to false

boolean lock;
Individual key;
Individual waiting[i];

while(1){
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = TestAndSet(lock);
    critical section
    j = (i+1) % n;
    while(j != i && !waiting[j])
        j = (j+1) % n;
    if(j == i)
        lock = false;
    else
        waiting[j] = false;
    remainder section
}
```

4.Discuss Peterson's solution.

Ans.
In operating systems, there may be a need for more than one process to access a shared resource such as memory or CPU. In shared memory, if more than one process is accessing a variable, then the value of that variable is determined by the last process to modify it, and the last modified value overwrites the first modified value. This may result in losing important information written during the first process. The location in which these processes are occurring is called the critical section. These critical sections prevent information loss by preventing two processes from being in the same critical region at the same time or updating the same variable simultaneously. This problem is called the Critical-Section problem and one of the solutions to this problem is the Peterson's solution.



Peterson's solution is a classic solution to the critical section problem. The critical section problem ensures that no two process change or modify the value of a resource at the same time.
For example, let int a=5 and there are two processes p1 and p2 that can modify the value of a. p1 adds 2 to a a=a+2 and p2 multiplies a with 2, a=a*2. If both processes modify the value of a at the same time, then the value of a depends on the order of execution of the process. If p1 executes first, a will be 14 and if p2 executes first, a will be 12. This change of values due to access by two processes at a time is the cause of the critical-section problem.
The section in which the values are being modified is called the critical section. There are another three sections except for the critical sections such as entry section,exit section, and the reminder section.

- The process entering the critical region must pass the entry region in which they request entry to the critical section.
- The process exiting the critical section must pass the exit region.
- The remaining code that is left after execution is in the remainder section.

Peterson's solution provides a solution to the following problems,

- It ensures that if a process is in the critical section then no other process must be allowed to enter the critical section. This property is termed mutual exclusion.
- In the event that more than one process wants to enter the critical section, then the process that should enter the critical region first must be established. This is termed progress.

- There is a limit to the number of requests that can be made by processors to enter the critical region, provided that a process has already requested to enter and is waiting. This is termed bounding.
- It provides platform neutrality as this solution is developed to run in user mode, which doesn't require any permission from the kernel.