NAME - SUSNATO BARUA
DEPT - IT ROLL - 1854044
Paper - Operating Systems Concepts (INFO3102)

1. Define and explain with proper example. what is Race Condition? How can this be solved?

Ans 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. It occurs when a software program depends on the timing of one or more processes to function correctly. If a thread runs or finishes at an unexpected time, it may cause unpredictable ~~sequence, a race~~

~~condition may occur~~ behaviour such as incorrect output or a program deadlock. If a program relies on threads that run in an unpredictable sequence, a race condition may occur.

Example- Suppose two processes need to perform a bit flip on a specific memory location. Under normal circumstances the operation should work like this:

Process 1	Process 2	Memory Value
Read value		0
Flip value		1
	Read value	1
	Flip value	0

Process 1 performs a bit flip, changing the memory value from 0 to 1. Process 2 then performs a bit ~~flop~~ flip and changes the memory value from 1 to 0.

If a race condition occured causing these two processes to overlap, the sequence could look like:

Process 1	Process 2	Memory Value
Read Value		0
	Read value	0
Flip Value		1
	Flip Value	1

The bit has an ending value of 1 when its value should be 0. This occurs because process 2 is unaware that process 1 is performing a simultaneous bit flip.

Race conditions can be prevented by serialization of memory or storage access. This means if read and write commands are received close together, the read command is executed and completed first by default.

2. Explain the necessary conditions of critical section. What do you mean by entry section and exit section?

Ans. The necessary conditions of critical section are:-

a) Mutual exclusion - If process P_i is executing in its critical section, then no other process can execute in its critical section.

b) Progress - If no progress is executing in its critical section and there exist some processes that wish to enter ~~and there~~ their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.

c) Bounded waiting - There exist a bound on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that request is granted. The solution should be such that every process gets a chance to be executed.

Entry section is part of the process which decides the entry of a particular process into the critical section.
Exit section allows the other process that are waiting in the entry section to enter into the critical sections.

3. What are the different software and hardware oriented solutions for critical section?

Ans The software solutions for two processes is algorithm1, algorithm2, Dekker's algorithm and Peterson algorithm. The software solution for multiprocess is Lamport-

Bakery algorithm. The hardware oriented solutions for critical section are test and set instruction, disabling interrupt and swapping.

4. Discuss the solutions of critical section using the following methods and also discuss the advantage and shortfalls of each of these methods:

a) Algorithm 1 -
```
do {
    while (turn != i);
        critical section.
    turn = j;
        remainder section
} while (1);
```

Advantage - Mutual exclusion is satisfied.
Shortfall - Does not satisfy progress and bounded wait conditions.

b) Algorithm 2 -
```
boolean flag [2];
do {
    flag [i] = true.
    while (flag [j])
        critical section
    flag [i] = false;
        remainder section;
} while (1);
```

Advantage - In the remainder section, the presence of one process cannot hamper the progress of another process. If another process is ready to enter the critical section, it can enter.

c) Dekker's Algorithm -

```
boolean flag[2];
int turn;
do {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
            flag[i] = true;
        }
    }
    critical section
    turn = j;
    flag[i] = false;
    remainder section
} while (1);
```

Advantage - It does not require special test and set instructions and is therefore highly portable between Languages and machine architecture.

Disadvantage - It is limited to two processes and makes use of busy waiting instead of process suspension.

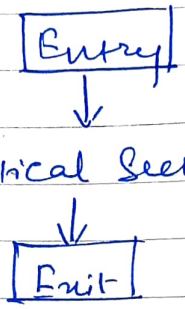
d) Peterson's Algorithm -

```
boolean flag[2];
int turn;
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
    critical section
    flag[i] = false;
    remainder section
} while (1);
```

Advantage - Mutual exclusion is preserved. The progress requirement is satisfied. The bounded-waiting requirement is met.

Disadvantage - This is a busy waiting solution so CPU time is wasted. So that spin lock problem can come.

e) Disabling Interrupt



When a job J_i finishes critical section it will enable the locked interrupts for other processes. So, basically here one process is executing so we disable all other interrupts so there is no completion. So only after current process execution finishes then only other process can start.

Advantages and Shortfall:
In uni-process environment critical section problem can be solved by forbidding interrupts. while a shared variable is being modified by more than one process simultaneously. But in a multiprocessor, it may not work properly.

f) Swapping lock-

```
void swap (boolean &a, boolean &b){
    bodean temp = a;
    a = b;
    b = temp;
    do {
        key = true;
        while ( key == true)
        swap ( lock, key);
```

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

Advantage - Allows you to perform complex transactions automatically.

Disadvantage - This code is embedded into ROM, so making further changes is difficult.

g) Test and Set Lock - ~~bool~~ boolean TestAndSet(boolean &target){

```
    boolean rv = target;
    target = true;
    return rv;
}

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

Advantage - Ensures mutual exclusion

Disadvantage - It does not guarantee bounded waiting and may cause starvation.

5. Discuss Lamport Bakery Algorithm why this is special?

Ans Lamport's Bakery Algorithm is intended to improve the safety in the usage of shared resources among multiple threads by means of mutual exclusion. It is one of many mutual exclusion algorithms designed to prevent concurrent threads entering

critical sections of code concurrently to eliminate the risk of data corruption.

```
boolean choosing [n];
int number [n];
do {
    choosing [i] = true;
    number[i] = max (number [0], number [1],
        number [n-1]) + 1;
for (j=0; j<n; j++) {
    while (choosing [j]);
while ((number[j] != 0) && (number[j,j] <
    number [i, i]));
}
critical section
number[i] = 0;
remainder section
} while (1);
```

When a thread wants to enter the critical section, it has to check whether now is its turn to do so. It should check the number n of every other thread to make sure that it has the smallest one. In case another thread has the same number the thread will the smallest i will enter the critical section first. Once the thread ends its critical job, it gets rid of its number and enters the non-critical section.

6. What is semaphore? What is the different usage of semaphore? what are the different types of semaphore? Explain

Ans Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

Wait - The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, hence no operation is performed.

Signal - The signal operation increments the value of its argument S.

The different usage of semaphore are -

a) Mutual exclusion - The value of S is kept to 1 to maintain mutual exclusion between processes.

b) Synchronization -

P1	P2
S1;	wait(synch);
signal(synch);	S2;

This guarantees that S1 and S2 are working parallelly and S1 executes before S2. By synching a semaphore S with these processes and initializing it to 0, the wait condition is satisfied. P2 will wait till S1 executes. After that S becomes 1 then S2 executes.

c) Multiple copy of same type resource allocation - The value of S is equal to the total number of processes is more than number of resources then one of the process has to wait.

There are 2 types of semaphores - counting semaphores and binary semaphores.

Counting semaphores - There are integer valued semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access,

where the semaphore count is the number of available resources.

- Binary semaphores - The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

7) What is meant by busy waiting or spinlock? Why it is not recommended to adapt this method? How without using spinlock we can implement semaphore?

Ans Busy waiting is a phenomenon in which a process checks repeatedly for a certain condition. It is waiting for the condition to be fulfilled, but it is busy checking for that condition that whether the condition has yet fulfilled or not. The condition is checked everytime if it is true or false hence it is also called spinlock. Busy waiting keeps the CPU all the time in checking the while loops condition continuously although the process is waiting for the critical section to become available. Hence it is not recommended to adapt this method.

To overcome the need for busy waiting we can modify the wait() and signal() semaphore operations. When the process executes the wait() operation then rather than engaging in busy waiting the process can block itself. The process that is blocked waiting on a semaphore should be restarted. When some other process executes a signal() operation, the process is restarted by wakeup() operation.

```
typedef struct{
    int value;
    struct process *L;
} semaphore;
void wait (semaphore s){
    s.value--;
    if (s.value <0){
        add this process to s.L;
        block();
    }
}
void signal (semaphore s){
    s.value ++;
    if (s.value <=0){
        remove a process P from s.L;
        wakeup (P);
    }
}
```

8. With proper example show how using semaphore may result to deadlock

Ans: The implementation of semaphore with a waiting queue may result in starvation where two or more processes are waiting indefinitely for an event that can be caused only by one of the following waiting processes.

```
P0                      P1
wait (S);               wait (Q);
wait (Q);               wait (S);
  :                       :
signal (S);             signal (Q);
signal (Q);             signal (S);
```

P0 executes wait (s) and thus P1 executes wait (Q). When P0 executes wait (Q), it must wait until P1 executes signal (Q) in the same way P1 must wait until P0 executes signal (S). So P0 and P1 are deadlocked.

9. Discuss how using semaphore we can solve the following problems: a) Readers Writers Problem
b) Dining Philosophers Problem
c) Producer Consumer Problem

a) Readers Writers Problems:

```
semaphore mutex, wrt;
int readcount
```

Writers Process:

```
wait (wrt);
---
Writing is performed
---
signal (wrt);
```

Readers Process:

```
wait (mutex);
readcount++;
if (readcount == 1)
    wait (wrt);
signal (mutex);
.....
reading is performed
.....
wait (mutex);
readcount--;
if (readcount == 0)
    signal (wrt);
signal (mutex);
```

Let's say first process P_0 comes, calls wait on mutex and changes readcount = 1 and it calls wait on wrt and then it starts reading. By that time P_1 process comes and gets incremented by 1, ie, readcount = 2.

Now, another process P_2 comes and it waits for P_1 to signal the mutex. When P_1 calls the signal on mutex and P_2 proceeds likewise and also enters the reading part. After sometime P_0 has completed the reading and readcount is decremented by 1, ie, readcount=2.
Now, say P_3 comes comes does the same job and readcount = 3 and enters the reading section.
Now P_1 finishes the reading and makes readcount= 2 and since readcount ≠0, it is calling signal on mutex and going out.
Similarly P_2 finished reading and makes readcount to 1 and again readcount ≠0, so it is going out after calling the signal on mutex.
Now P_3 finishes reading and makes readcount to 0 and it is calling signal on wrt. (ie, it is unlocking wrt).
Now the writers who were waiting, P_3 and P_4 will get the chance to write.

6. Dining philosophers:-

```
semaphores chopstick [5];
do{
    wait (chopstick [i]);
    wait (chopstick [i+1]%5);
    ----------
    eat
    ----------
    signal (chopstick [i]);
    signal (chopstick [i+1]%5);
    ----------
    think
    ----------
} while(1)
```

If all the philosophers started eating parallelly. So P_0 comes in ith wait and it will assign chopstick 0, as it is available
Similarly P_1 acquires chopstick 1; P_2 to chopstick 2, P_3 to chopstick 3, P_4 to chopstick 4.

Next they will be trying to acquire the (i+1)th chopstick, suppose P_0 is trying to get the chopstick 1 and it is waiting for P_1 to release it, so it is a busy waiting. Similarly P_1 requested for chopstick 2 but it is allocated and it is waiting for P_1 to release it, so it is a busy waiting. Similarly P_1 requested for chopstick 2. but it is allocated to P_2 so P_1 is in busy waiting. So in this way everyone is waiting for another process and creating a circular wait thus it is a major situation when deadlock may occur

c) Producer - Consumer

```
Producer:
do{
    Produce item
    :
    wait (empty);
    wait (mutex);
    :
    Put item in buffer
    :
    Signal (mutex);
    Signal (full);
    }while (1);
```

Mutex, empty and full are the semaphores. Initially, mutex = 1, empty = n (n is max size of the buffer) and full = 0.

The mutex semaphore ensures mutual exclusion. The empty and full semaphore count the number of empty and full spaces in buffer.

After the item is produced, wait operation is carried out on empty. Then wait operation is carried out on mutex so that consumer process cannot interfere. After the item is put in the buffer, signal function is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

```
do {
    wait (full);
    wait (mutex);
    :
    Remove item from buffer
    :
    Signal (mutex);
    Signal (empty);
    :
    consume item
    :
} while (1);
```

The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait is carried out on mutex so that producer cannot interfere.

Then the item is removed from buffer. After that signal is carried out on mutex and empty. The former indicates that consumer process can now

act and the latter shows that empty space in the buffer is increased by 1.

10. How can we redesign the solution of Dining philosopher's problem using semaphore to avoid the possibilities of deadlock?

Ans: Solution to avoid deadlock :-

a) Allow at most four philosophers to be eating simultaneously.

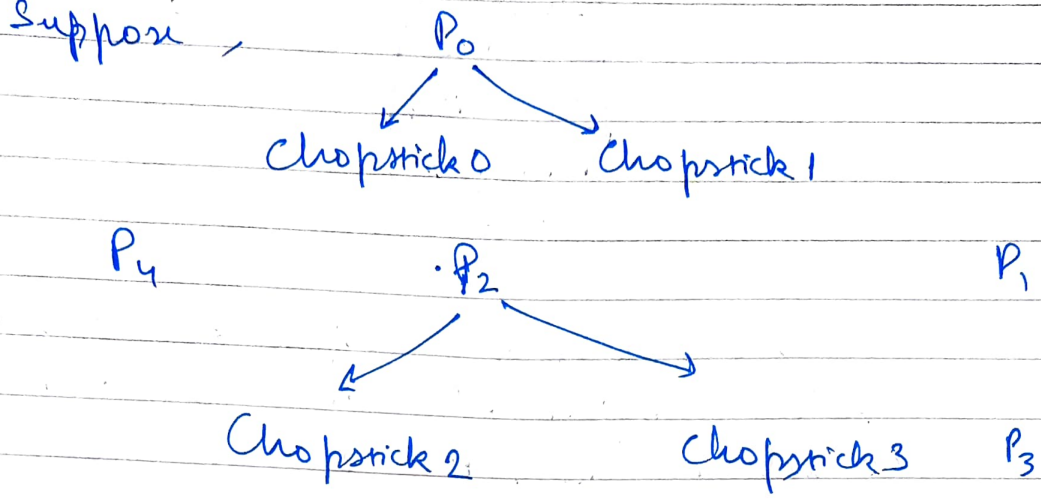
Ex - Suppose active philosophers P_0 get chopstick 0, P_1 got chopstick 1, P_2 got chopstick 2, P_3 got chopstick 3, now P_4 has to wait.

When P_0 trying to get chopstick 1, P_0 has to wait. Similarly P_1 has to wait for chopstick 2, P_2 has to wait for chopstick 3.

But chopstick 4 is free as P_4 is waiting, so P_3 has got chopstick 4, so chopstick 3 is released and thus P_2 gets chopstick 3, simultaneously P_1 get chopstick and P_0 gets chopstick 1.

```
semaphore c=4;
do{ wait(c);
    .
    .
    critical section
    .
    .
    .... think
    signal(c);
}while(1);
```

b) Allow a philosopher to pick the chopsticks if both the chopsticks are freed or available.



So, P_1 and P_4 is waiting, when P_0 and P_2 finishes and if both the chopsticks are freed and all of them are available then only P_1 and P_4 will get the chance.

c) Allow all odd philosophers to pick the left chopstick first and the even philosophers to pick the left chopstick first and the even philosophers to pick the right chopstick first.

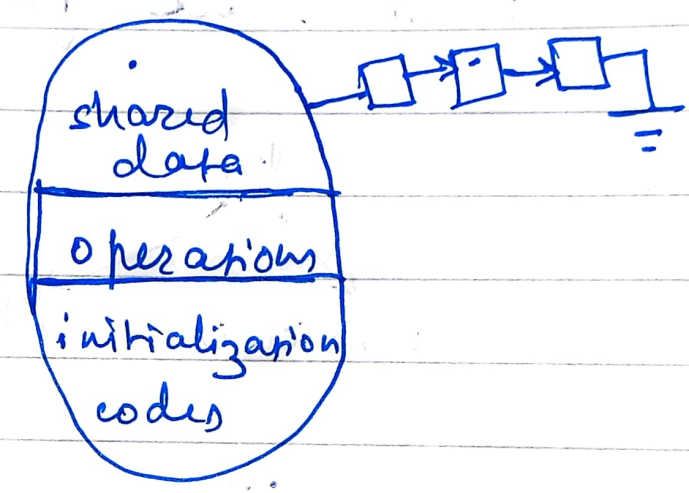
Suppose P_0, P_2, P_4 will get the ith chopstick first as all of them are even processes.

So, $P_0 \rightarrow$ chopstick 1, $P_2 \rightarrow$ chopstick 3, $P_4 \rightarrow$ chopstick 0

Now P_1, P_3 will be trying to get the ith chopstick first but chopstick 1 and chopstick 3 are already allocated, so they are waiting.

Now, P_0, P_2, P_4 will be going for ith chopstick, since chopstick 4 is free, P_4 will get chopstick 4 and chopstick 0 will be free, so P_0 can get chopstick 0 now and chopstick 1 will be free and P_2 will be assigned to chopstick 2 and chopstick 3 will be free. Now the odd processes P_1 and P_3 will get their respective chopsticks as those are free now.

11. What is monitor corresponding to semaphore



Semaphores allows multiple threads to access a shared object. Monitors allow mutually exclusive access to a shared object.