OS Assignment-2 Anmol Baranwal--2820208 A3

Ans1. 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 semaphore. It is simply an integer variable that is shared b/w threads. This variable is used to solve the critical section problem using two atomic operations, wait & 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 zero or negative, then no operation is performed.

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

signal => The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

Types of semaphores

i) Counting Semaphores => These are integer value semaphores and have an unrestricted value domain.

2820208

(ii) Binary Semaphore ⇒
The binary semaphores are like counting semaphores but their value is restricted to 0 or 1. The wait operation only works when semaphore is 1 and the signal operation succeeds when semaphore is 0.

Semaphores allow only one process into the critical section and follow mutual exclusion principle strictly.

Ans 2. A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.

Requirements of Critical Section Problem's sol^n

- Mutual Exclusion ⇒ It is a special type of binary semaphore which is used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. Not more than one process can execute in its critical section at one time
- Progress ⇒ This sol^n is used when no one is in the critical section, and someone wants in. Then those processes not in their reminder section should decide who should go in, in a finite time.

Bound waiting ⇒ After a process makes a request for getting into critical section, there is a specific limit about number of processes can get into their critical sections. So, when the limit is reached, the system must allow request to the process to get into its critical section.

Ans 4. a. Need = Max - Allocation

	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

Available

A	B	C	D
1	5	2	0
0	4	2	0
1	1	0	0

Safe sequence is $<P_0, P_2, P_3, P_4, P_1>$

b. As need for $P_0 \leq$ Available.

After completion of P_0.

Available ⇒ Available + allocation = 0 0 1 2 + 1 5 2 0 = 1 5 3 2

Since Need(P_1) > Available, so request of P_1 cannot be satisfied.

As Need(P_2) < Available. After completion of P_2

New Available ⇒ Available + allocation = 1 5 3 2 + 1 3 5 4 = 2 8 8 6

New Available → 2 8 8 6

As Need(P_3) < Available, after completion of P_3.

New Available ⇒ Available + allocation = 2 8 8 6 + 0 6 3 2 = 14 11 8

2820208

As Need (P4) < Available, after completion of P4

New Available => 2 14 11 8
0 0 1 4
2 14 12 12

As Need (P1) < Available, after completion of P1

New Available => 2 14 12 12
1 0 0 0
3 14 12 12

So the system is in safe state.

c. Request (0,4,2,0) ≤ Available (1,5,2,0)

New Snapshot =>

	Allocation				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	0	0	1	1	0	0
P1	1	4	2	0	0	3	3	0				
P2	1	3	5	4	1	0	0	2				
P3	0	6	3	2	0	0	2	0				
P4	0	0	1	4	0	6	4	2				

∴ With this snapshot, we can get a safe sequence
<P0 P2 P3 P4 P1>

Thus, request of P1 for (0,4,2,0) can be granted immediately.

Ans 3. The producer-consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization b/w more than one process. If the result is wrong

ause the consumer should get each integer oduced by the producer exactly once. A problem uch as this is called a race condition.

Each of the processes has some sharable and some non-sharable resources. The sharable resources can be shared among the cooperating processes. The non-cooperating processes don't need to share the resource. Now, the question is what is the race condition. When we synchronize the processes and the synchronization is not proper then the race condition occurs. We can define race cond^{n} as follows —

A race condition is a condition where there are many processes and every process shares the data with each other and accessing the data concurrently, and the output of execution depends on a particular sequence in which they share the data and access.

- counter ++ can be implemented as

```
register 1 = counter
register 1 = register 1 + 1
counter = register 1
```

- counter -- can be implemented as

```
register 2 = counter
register 2 = register 2 - 1
counter = register 2
```

• consider this execution interleaving with "count = 5" initially:

S0: producer execute register 1 = counter {register 1 = 5}
S1: producer execute register 1 = register 1 + 1 {register 1 = 6}
S2: producer execute register 2 = counter {register 2 = 5}
S3: producer execute register 2 = register 2 - 1 {register 2 = 4}
S4: producer execute counter = register 1 {counter = 6}
S5: consumer execute counter = register 2 {counter = 4}

The above depicts the race condition using producer-consumer.

⇒ To prevent the race condition, we need to ensure that only one process can access the shared data at a time.