ASSIGNMENT NO: 4-A

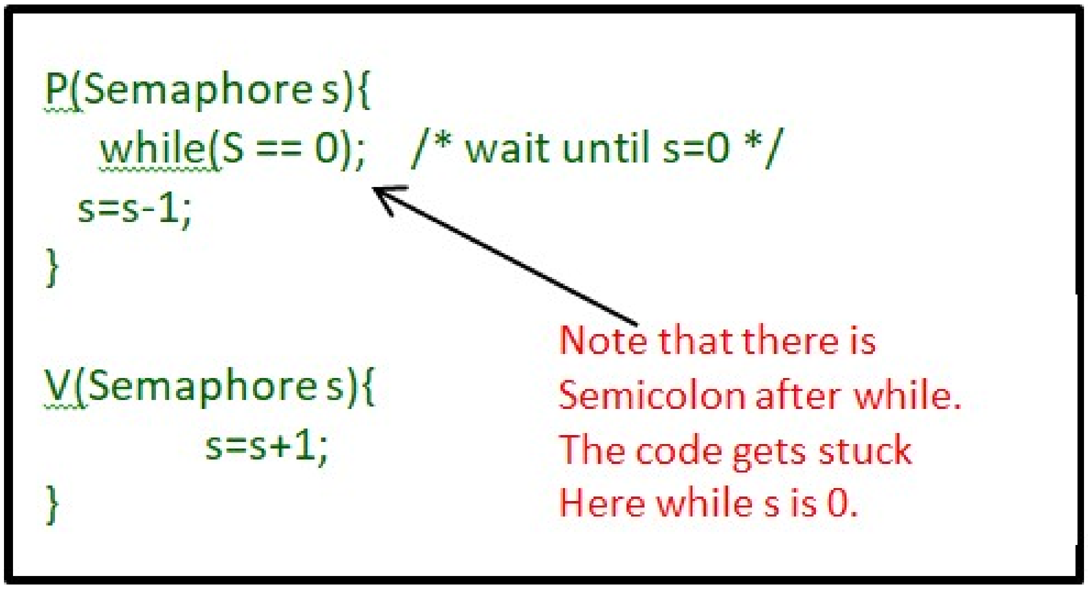
TITLE: Thread synchronization using counting semaphores.

AIM: Application to demonstrate producer-consumer problem with counting semaphores and mutex.

THEORY:

Semaphores:

- An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
- The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.
- The process of using Semaphores provides two operations: wait (P) and signal (V).
- The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore.
- When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.
- **Semaphores** are the **OS tools** for **synchronization**. Two types:
    1. **Binary Semaphore**.
    2. **Counting Semaphore**.



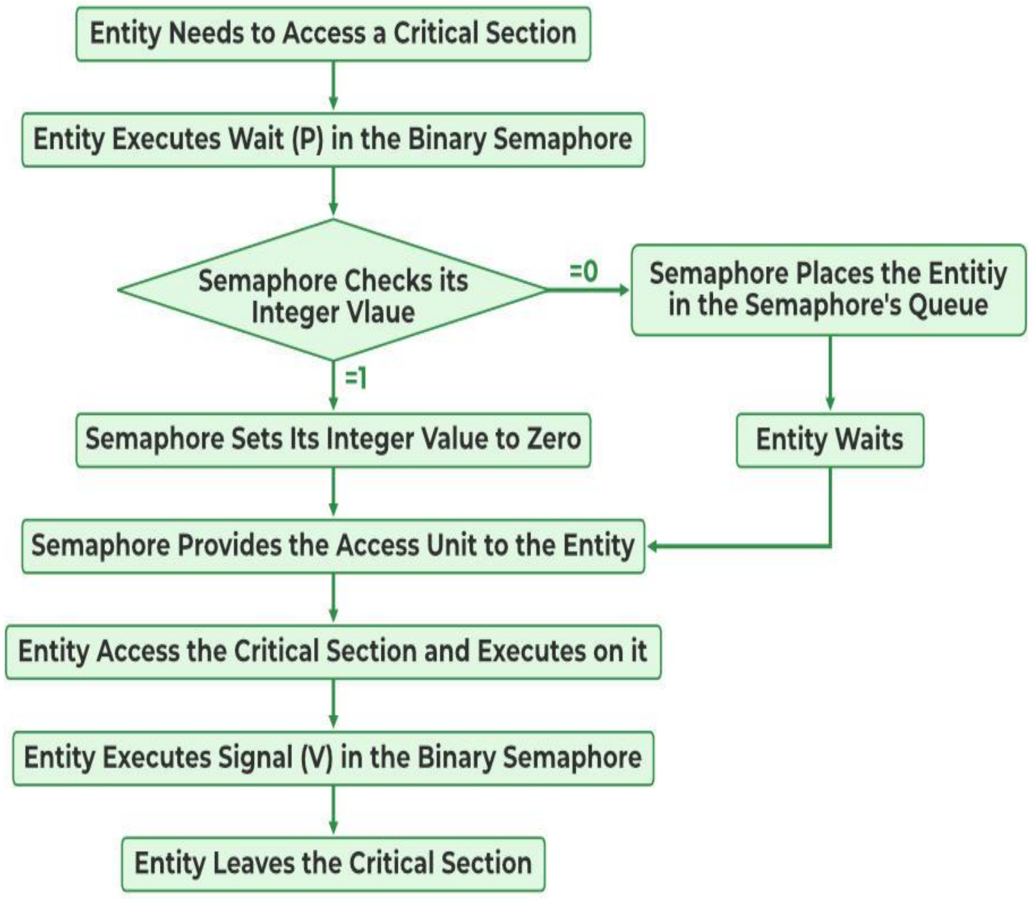
```
P(Semaphore s){
    while(S == 0);    /* wait until s=0 */
   s=s-1;
}

V(Semaphore s){
            s=s+1;
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

Semaphores are of two types:

1. **Binary Semaphore –**
    - The idea behind using a binary semaphore is that, it allows only one process at a time to enter the critical section(thus allowing it to access the shared resource).
    - Here, 0 represents that a process or a thread is in the critical section(i.e. it is accessing the shared resource), while the other process or thread should wait for it to complete. On the other hand, 1 means that no process is accessing the shared resource, and the critical section is free.
    - It guarantees mutual exclusion since no two processes can be in the critical section at any point in time.
    - This is also known as a mutex lock. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.


Entity Needs to Access a Critical Section
Entity Executes Wait (P) in the Binary Semaphore
Semaphore Checks its Integer Vlaue
=0
Semaphore Places the Entitiy in the Semaphore's Queue
=1
Semaphore Sets Its Integer Value to Zero
Entity Waits
Semaphore Provides the Access Unit to the Entity
Entity Access the Critical Section and Executes on it
Entity Executes Signal (V) in the Binary Semaphore
Entity Leaves the Critical Section

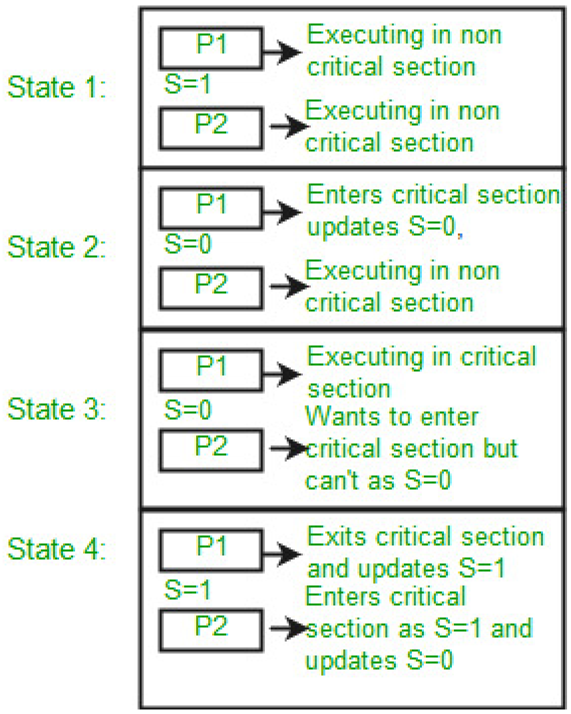
Binary Semaphores are different from Counting semaphores in terms of values because binary semaphores can take only 0 or 1. At the same time, counting semaphores can take from $+\infty$ to $-\infty$ **.** The binary semaphores are different from counting semaphores because counting semaphore allows more than one process to enter into critical sections simultaneously.

2. **Counting Semaphore –**
    - A counting semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.

- It is a structure, which comprises a variable, known as a semaphore variable that can take more than two values and a list of task or entity, which is nothing but the process or the thread.
- The value of the semaphore variable is the number of process or thread that is to be allowed inside the critical section.
- The value of the counting semaphore can range between 0 to N, where N is the number of the number of process which is free to enter and exit the critical section.
- As mentioned, a counting semaphore can allow multiple processes or threads to access the critical section, hence mutual exclusion is not guaranteed.
- Since multiple instances of process can access the shared resource at any time, counting semaphore guarantees bounded wait. Using such a semaphore, a process which enters the critical section has to wait for the other process to get in-side the critical section, implying that no process will starve.
- Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Let us see how it implements mutual exclusion.

- Let there be two processes P1 and P2 and a semaphore s is initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0.
- Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
- This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.


State 1:
P1
Executing in non critical section
S=1
P2
Executing in non critical section
State 2:
P1
Enters critical section updates S=0,
S=0
P2
Executing in non critical section
State 3:
P1
Executing in critical section
S=0
Wants to enter
P2
critical section but can't as S=0
State 4:
P1
Exits critical section and updates S=1
S=1
Enters critical
P2
section as S=1 and updates S=0

Some points regarding P and V operation:

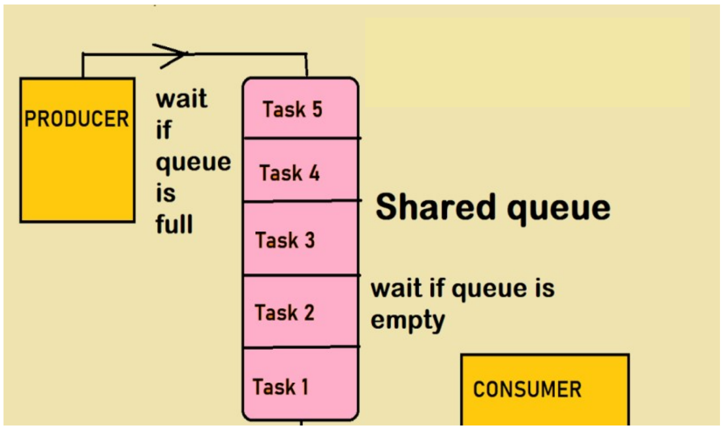
P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.

A solution to the mutual exclusion problem must satisfy three conditions:

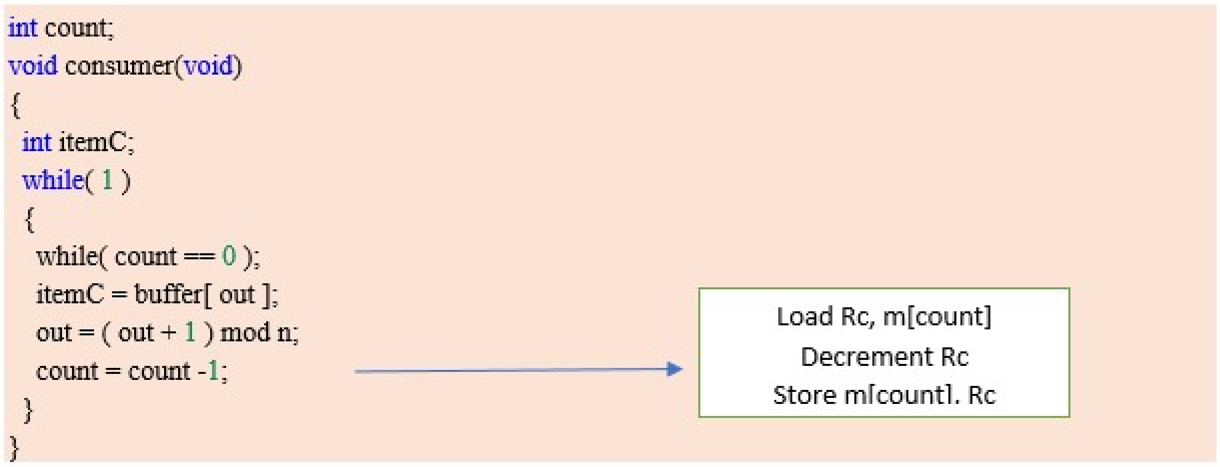
1. Mutual Exclusion: Only one process can be in the critical section at a time -- otherwise what critical section?.
2. Progress: No process is forced to wait for an available resource -- otherwise very wasteful.
3. Bounded Waiting: No process can wait forever for a resource -- otherwise an easy solution: no one gets in.

The Producer/Consumer Problem

- The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem.
- The general statement is this: there are one or more producers generating some type of data (records, characters)and placing these in a buffer.
- There is a single consumer that is taking items out of the buffer one at a time.
- The system is to be constrained to prevent the overlap of buffer operations.
- That is, only one agent (producer or consumer) may access the buffer at any one time.
- The problem is to make sure that the producer won't try to add data in to the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
- **Below are a few points that considered as the problems occur in Producer-Consumer:**
    - The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
    - Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.

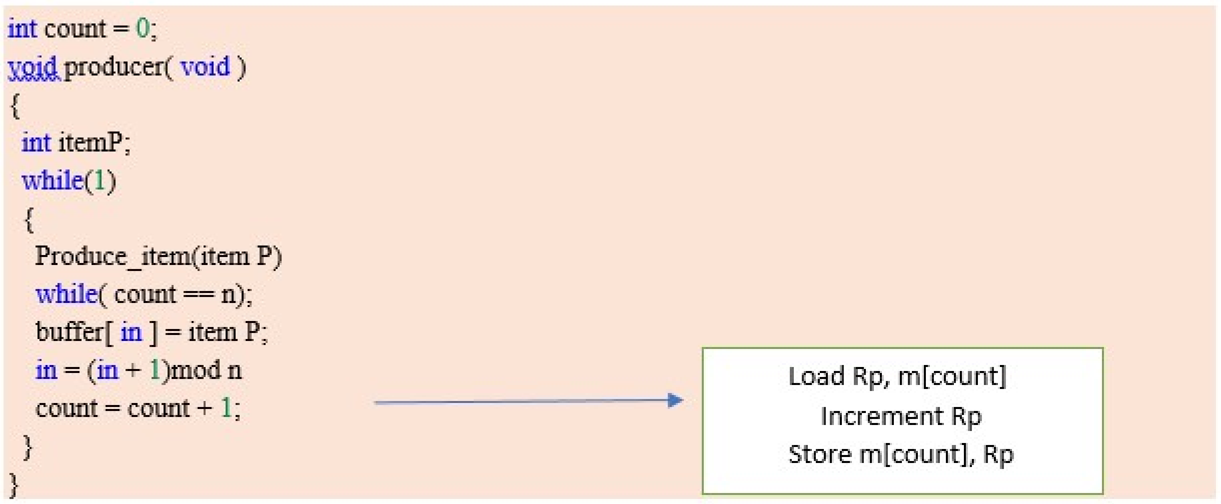


- Accessing memory buffer should not be allowed to producer and consumer at the same time.



```
int count;
void consumer(void)
{
 int itemC;
 while( 1 )
 {
  while( count == 0 );
  itemC = buffer[ out ];
  out = ( out + 1 ) mod n;
  count = count -1;
 }
}
```

```
Load Rc, m[count]
Decrement Rc
Store m[count], Rc
```



```
int count = 0;
void producer( void )
{
 int itemP;
 while(1)
 {
  Produce_item(item P)
  while( count == n);
  buffer[ in ] = item P;
  in = (in + 1)mod n
  count = count + 1;
 }
}
```

```
Load Rp, m[count]
Increment Rp
Store m[count], Rp
```

Let's understand above Producer and Consumer code:
Before Starting an explanation of code, first, understand the few terms used in the above code:

1. "**in**" used in a producer code represent the next **empty buffer**
2. "**out**" used in consumer code represent first **filled buffer**
3. count keeps the count number of elements in the buffer
4. count is further divided into 3 lines code represented in the block in both the producer and consumer code.

If we talk about Producer code first:
--Rp is a register which keeps the value of m[count]
--Rp is incremented (As element has been added to buffer)
--an Incremented value of Rp is stored back to m[count]
Similarly, if we talk about Consumer code next:
--Rc is a register which keeps the value of m[count]
--Rc is decremented (As element has been removed out of buffer)
--the decremented value of Rc is stored back to m[count].
The solution of Producer-Consumer Problem using Semaphore
The above problems of Producer and Consumer which occurred due to context switch and producing inconsistent result can be solved with the help of semaphores.

To solve the problem occurred above of race condition, we are going to use **Binary Semaphore and Counting Semaphore**

ALGORITHM:

Step 1: Start the program.

Step 2: Declare and initialize the necessary variables.

Step 3: Create a Producer.

Step 4: Producer (Child Process) performs a down operation and writes a message.

Step 5: Producer performs an up operation for the consumer to consume.

Step 6: Consumer (Parent Process) performs a down operation and reads or consumes the data (message).

Step 7: Consumer then performs an up operation.

Step 8: Stop the program.

<u>Conclusion:</u>

ASSIGNMENT NO: 4-B

TITLE: Thread synchronization and mutual exclusion using mutex.

AIM: Application to demonstrate Reader-Writer problem with reader priority.

THEORY:

Reader-Writer problem with readers priority

- The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.
- Consider a situation where we have a file shared between many people.
    - If one of the person 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 the **readers-writer**, **problem**
  Problem parameters: that
    - 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

- The solution of readers and writers can be implemented using binary semaphores.
- We use two binary semaphores "write" and "mutex", where binary semaphore can be defined as:
- Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Solution using semaphore:

```
int readcount;
```

```
semaphore x = 1,wsem = 1;

void reader()
{
    while (true)
    {
        semWait (x);
        readcount++;
        if(readcount == 1)
            semWait (wsem);
        semSignal (x);
        READUNIT();

        semWait (x);

        readcount;

        if(readcount == 0)

            semSignal (wsem);

        semSignal (x);
    }
}

void writer()
{
    while (true)
    {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

ALGORITHM:

Step 1: Start the program.
Step 2: Declare and initialize the necessary variables.
Step 3: Create a Reader.
Step 4: Reader performs a down operation and read a message in the database.
Step 5: Reader performs an up operation for the Writer to access a message from the database.
Step 6: Write performs a down operation and writes or access the data (message) from the database.
Step 7: Writer then performs an up operation.
Step 8: Stop the program.
PROGRAM:

Conclusion: