JUNAID GIRKAR
60004190057
SE COMPS A-3

OPERATING SYSTEMS
EXPERIMENT - 6
THEORY

AIM: Case Study based on Producer Consumer problem using Semaphores

THEORY:

About Producer-Consumer problem:

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

About the problem:

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

Solution:

The above three problems can be solved with the help of semaphores

A semaphore S is an integer variable that can be accessed only through two standard operations : wait() and signal().

The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

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

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

Semaphores are of two types:

1. **Binary Semaphore** – This is similar to mutex lock but not the same thing. 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.

In the producer-consumer problem, we use three semaphore variables:

1. **Semaphore S:** This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.

2. **Semaphore E:** This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.
3. **Semaphore F:** This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the *wait()* and *signal()* function, we can solve our problem(the *wait()* function decreases the semaphore variable by 1 and the *signal()* function increases the semaphore variable by 1). So. let's see how.

The following is the pseudo-code for the producer:

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

The above code can be summarized as:

- ***while()*** is used to produce data, again and again, if it wishes to produce, again and again.
- ***produce()*** function is called to produce data by the producer.
- ***wait(E)*** will reduce the value of the semaphore variable "E" by one i.e. when the producer produces something then there is a decrease in the value of the empty space in the buffer. If the buffer is full i.e. the vale of the semaphore variable "E" is "0", then the program will stop its execution and no production will be done.
- ***wait(S)*** is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- ***append()*** function is used to append the newly produced data in the buffer.
- ***signal(s)*** is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the production is done and the append operation is also done.
- ***signal(F)*** is used to increase the semaphore variable "F" by one because after adding the data into the buffer, one space is filled in the buffer and the variable "F" must be updated.

This is how we solve the produce part of the producer-consumer problem. Now, let's see the consumer solution. The following is the code for the consumer:

```
void consumer() {
    while(T) {
        wait(F)
        wait(S)
        take()
        signal(S)
        signal(E)
        use()
    }
}
```

The above code can be summarized as:

- ***while()*** is used to consume data, again and again, if it wishes to consume, again and again.
- ***wait(F)*** is used to decrease the semaphore variable "F" by one because if some data is consumed by the consumer then the variable "F" must be decreased by one.
- ***wait(S)*** is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- ***take()*** function is used to take the data from the buffer by the consumer.
- ***signal(S)*** is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the consumption is done and the take operation is also done.

- ***signal(E)*** is used to increase the semaphore variable "E" by one because after taking the data from the buffer, one space is freed from the buffer and the variable "E" must be increased.
- **use()** is a function that is used to use the data taken from the buffer by the process to do some operation.

CONCLUSION:

We learnt about Semaphores and how they can be implemented to solve the Producer-Consumer problem which is used for multi-process synchronization.