

## Assignment 6

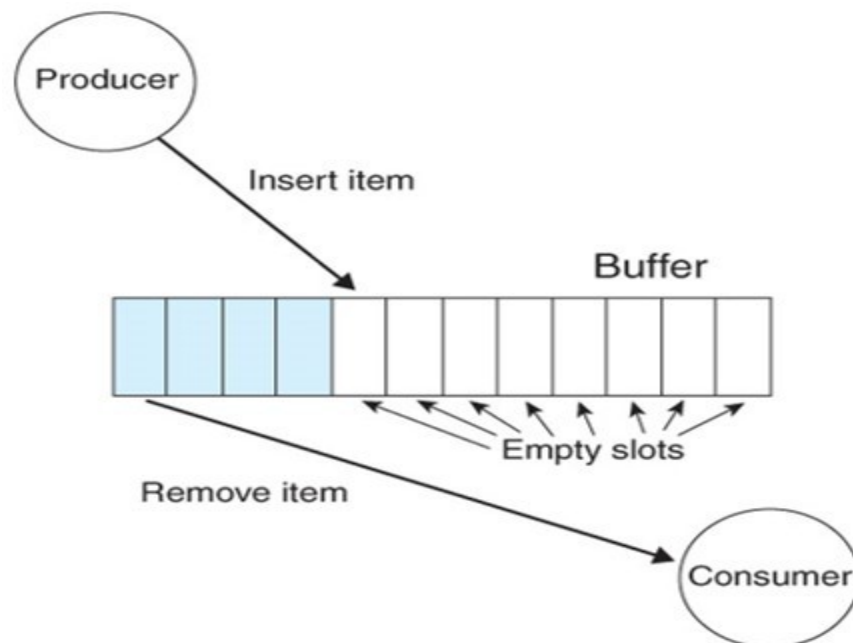
221071

1) **AIM:** Producer Consumer using semaphore & mutex.

2) **THEORY:**

### 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.



### What's the problem here?

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.

### Producer Process

The code that defines the producer process is given below:

```
do {
    . PRODUCE ITEM

    wait(empty);
    wait(mutex);

    . PUT ITEM IN BUFFER

    signal(mutex);

    signal(full);

} while(1);
```

- In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.
- The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty and full spaces in the buffer.
- After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.
- After the item is put in the buffer, signal operation 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

The code that defines the consumer process is given below:

```

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 operation is carried out on mutex so that producer process cannot interfere.
- Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

## Mutex

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.

This is shown with the help of the following example:

```

wait (mutex);

.....

Critical Section

.....

signal (mutex);

```

A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signalling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

## Semaphore

A semaphore is a signalling 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 called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization.

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

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

    S--;
}
```

The signal operation increments the value of its argument S.

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

There are mainly two types of semaphores i.e. counting semaphores and binary semaphores.

Counting Semaphores are integer value 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.

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

### 3) code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include </usr/include/semaphore.h>
#include <unistd.h>
#define BUFF_SIZE 5      // total number of slots
#define NITERS 4        // number of items produced/consumed
int NP,NC;
typedef struct
{
    int buf[BUFF_SIZE]; // shared var
    int in;              // buf[in%BUFF_SIZE] is the first empty slot
    int out;             // buf[out%BUFF_SIZE] is the first full slot
    sem_t full;          // keep track of the number of full spots
    sem_t empty;         // keep track of the number of empty spots

    // use correct type here
    pthread_mutex_t mutex; // enforce mutual exclusion to shared data
} sbuf_t;

sbuf_t shared;

void *Producer(void *arg)
{
    int i, item, index;

    index = (intptr_t)arg;

    for (i=0; i < NITERS; i++)
    {
        /* Produce item */
        item = i;

        // Prepare to write item to buf

        // If there are no empty slots, wait
        sem_wait(&shared.empty);
        // If another thread uses the buffer, wait
        pthread_mutex_lock(&shared.mutex);
        shared.buf[shared.in] = item;
        shared.in = (shared.in+1)%BUFF_SIZE;
        printf("[P%d] Producing= %d \n", index, item);
        fflush(stdout);
```

```

    // Release the buffer
    pthread_mutex_unlock(&shared.mutex);
    // Increment the number of full slots
    sem_post(&shared.full);

    // Interleave producer and consumer execution
    if (i % 2 == 1)
    {
        sleep(1);
    }
}
return NULL;
}

void *Consumer(void *arg)
{
    int i, item, index;

    index = (intptr_t)arg;
    for (i=NITERS; i > 0; i--) {
        sem_wait(&shared.full);
        pthread_mutex_lock(&shared.mutex);
        item=i;
        item=shared.buf[shared.out];
        shared.out = (shared.out+1)%BUFF_SIZE;
        printf("[C%d] Consuming= %d \n", index, item);
        fflush(stdout);
        // Release the buffer
        pthread_mutex_unlock(&shared.mutex);
        // Increment the number of full slots
        sem_post(&shared.empty);

        // Interleave producer and consumer execution
        if (i % 2 == 1)
        {
            sleep(1);
            //printf("\nError");
        }
    }
    return NULL;
}

int main()
{
    pthread_t idP, idC;
    int index;

    printf("Please enter the no of producers= \n");
    scanf("%d",&NP);

```

```

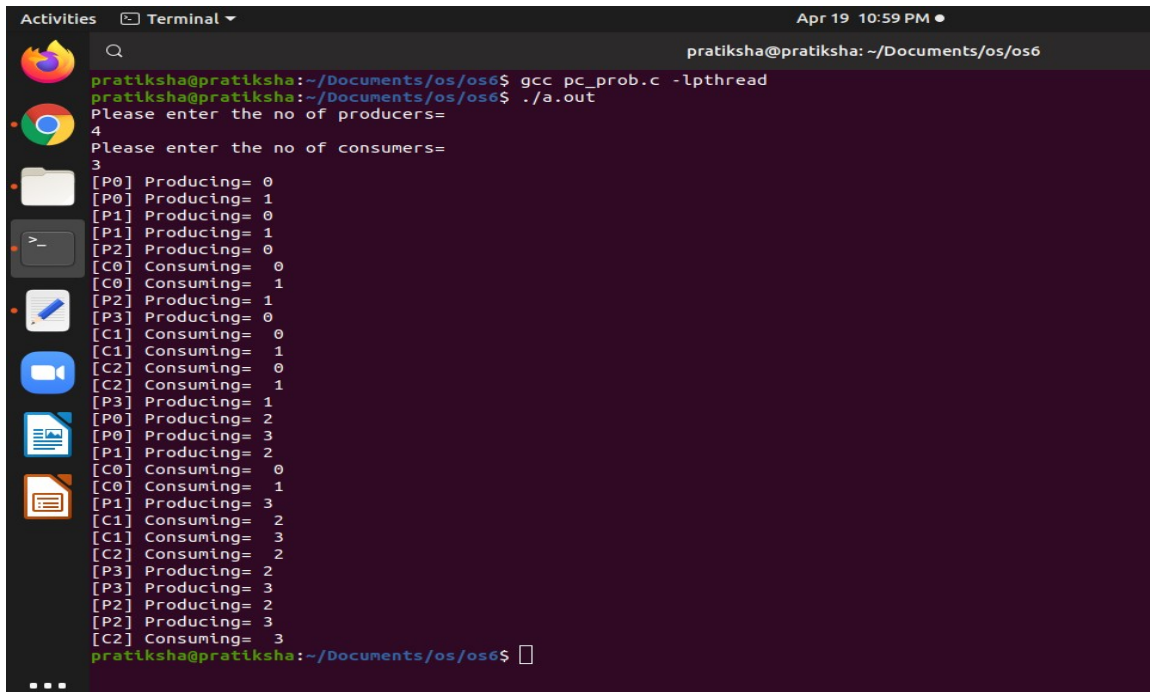
printf("Please enter the no of consumers= \n");
scanf("%d",&NC);

sem_init(&shared.full,0,0);
sem_init(&shared.empty,0,BUFF_SIZE);
pthread_mutex_init(&shared.mutex, NULL);
for (index = 0; index < NP; index++)
{
    /* Create a new producer */
    pthread_create(&idP, NULL, Producer, (void*)(intptr_t)index);
}
/*create a new Consumer*/
for(index=0; index<NC; index++)
{
    pthread_create(&idC, NULL, Consumer, (void*)(intptr_t)index);
}

pthread_exit(NULL);
printf("\nThank You!");
}

```

#### 4) output



```

pratiksha@pratiksha:~/Documents/os/os6$ gcc pc_prob.c -lpthread
pratiksha@pratiksha:~/Documents/os/os6$ ./a.out
Please enter the no of producers=
4
Please enter the no of consumers=
3
[P0] Producing= 0
[P0] Producing= 1
[P1] Producing= 0
[P1] Producing= 1
[P2] Producing= 0
[P2] Producing= 1
[P3] Producing= 0
[P3] Producing= 1
[C0] Consuming= 0
[C0] Consuming= 1
[C1] Consuming= 0
[C1] Consuming= 1
[C2] Consuming= 0
[C2] Consuming= 1
[P0] Producing= 2
[P0] Producing= 3
[P1] Producing= 2
[P1] Producing= 3
[C0] Consuming= 0
[C0] Consuming= 1
[P1] Producing= 3
[C1] Consuming= 2
[C1] Consuming= 3
[C2] Consuming= 2
[P3] Producing= 2
[P3] Producing= 3
[P2] Producing= 2
[P2] Producing= 3
[C2] Consuming= 3
pratiksha@pratiksha:~/Documents/os/os6$

```

