LABWORK – 3

TITLE: Producer-Consumer Problem (Using semaphores)

OBJECTIVE:
1.To study use of counting semaphore.
2.To study Producer-Consumer problem of operating system.

What is Producer-consumer Problem?

The producer and consumer share a fixed-size buffer used as a queue. The producer's job is to generate data and put this in the buffer. The consumer's job is to consume the data from this buffer, one at a time.

Problem Statement

How do you make sure that producer doesn't try to put data in buffer when the buffer is full and consumer doesn't try to consumer data when the buffer is empty?
When producer tries to put data into the buffer when it is full, it wastes cpu cycles. The same is true for consumer it tries to consumer from an empty buffer. It's better that they go on sleep in these cases so that the scheduler can schedule another process.

What is a semaphore ?

A semaphore is fundamentally an integer whose value is never allowed to fall below 0. There are two operations on a semaphore: wait and post. The post operation increment the semaphore by 1, and the wait operations does the following: If the semaphore has a value > 0, the semaphore is decremented by 1. If the semaphore has value 0, the caller will be blocked (busy-waiting or more likely on a queue) until the semaphore has a value larger than 0, and then it is decremented by 1.

We declare a semaphore as:

```
sem_t sem;
```

To initialize a semaphore, use sem_init():

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem points to a semaphore object to initialize
- pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes. LinuxThreads does not currently support shared semaphores
- value is an initial value to set the semaphore to

Example of use:

```
sem_init(&sem_name, 0, 10);
```

To wait on a semaphore, use sem_wait:

```
int sem_wait(sem_t *sem);
```

Example of use:

```
sem_wait(&sem_name);
```

- sem_wait is an implementation of the DOWN operation discussed in class. If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem_post.

To increment the value of a semaphore, use sem_post:

```
int sem_post(sem_t *sem);
```

Example of use:

```
sem_post(&sem_name);
```

- sem_post is an implementation of the UP operation discussed in class. It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

To find out the value of a semaphore, use

```
int sem_getvalue(sem_t *sem, int *valp);
```

- gets the current value of sem and places it in the location pointed to by valp

Example of use:

```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

To destroy a semaphore, use

```
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```
sem_destroy(&sem_name);
```

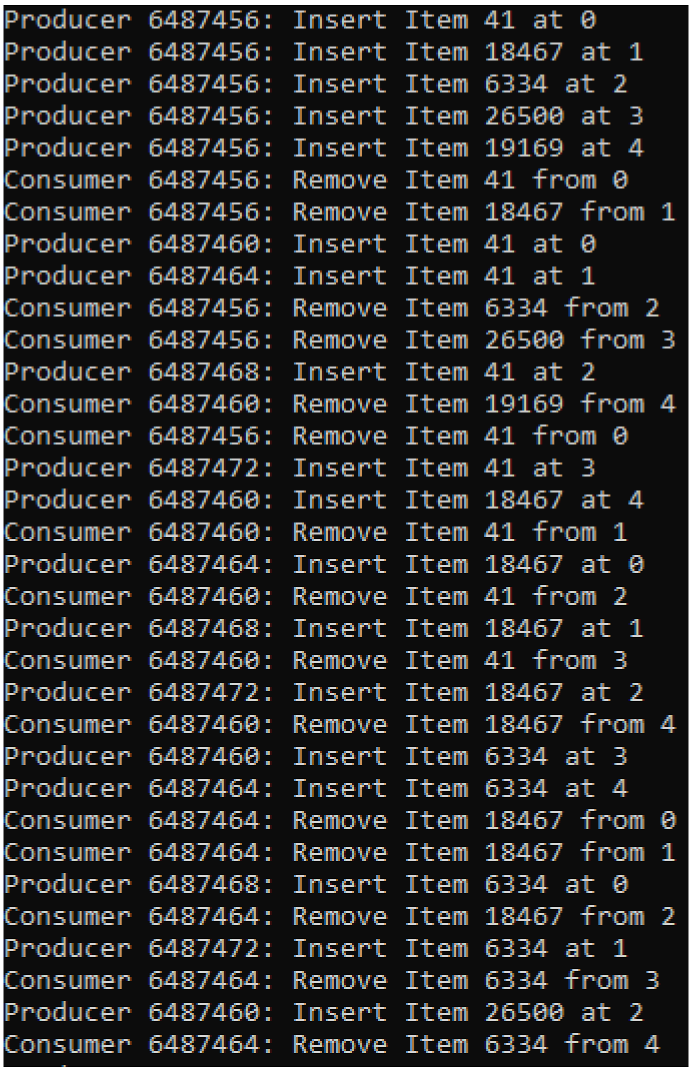
EXAMPLE:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#define MaxItems 5 // Maximum items a producer can produce or a consumer can consume
#define BufferSize 5 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;

void producer(int pno)
{
   int item;
   int i;
   for( i = 0; i < MaxItems; i++) {
      item = rand(); // Produce an random item
      ........
         ........
      buffer[in] = item;
      printf("Producer %d: Insert Item %d at %d\n", pno,buffer[in],in);
      in = (in+1)%BufferSize;
      ........
         ........

   }
}
void consumer(int cno)
{
int i;
   for( i = 0; i < MaxItems; i++) {
      ........
                  ........
```

```
        int item = buffer[out];
        printf("Consumer %d: Remove Item %d from %d\n",cno,item, out);
        out = (out+1)%BufferSize;
        ........
                ........
    }
}

int main()
{

    pthread_t pro[5],con[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty,0,BufferSize);
    sem_init(&full,0,0);

    int a[5] = {1,2,3,4,5}; //Just used for numbering the producer and consumer
        int i;
    for( i = 0; i < 5; i++) {
        pthread_create(&pro[i], NULL, producer, &a[i]);
    }
    for( i = 0; i < 5; i++) {
        pthread_create(&con[i], NULL, consumer, &a[i]);
    }

    for( i = 0; i < 5; i++) {
        pthread_join(pro[i], NULL);
    }
    for( i = 0; i < 5; i++) {
        pthread_join(con[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
```

}

TASK: Complete with necessary functions this example code to implement a solution to the producer consumer problem discussed in class using Posix threads and semaphores. Assume that there is only 10 producer and 10 consumer. The output of your code should be similar to the following and it must run in infinite loop :



```
Producer 6487456: Insert Item 41 at 0
Producer 6487456: Insert Item 18467 at 1
Producer 6487456: Insert Item 6334 at 2
Producer 6487456: Insert Item 26500 at 3
Producer 6487456: Insert Item 19169 at 4
Consumer 6487456: Remove Item 41 from 0
Consumer 6487456: Remove Item 18467 from 1
Producer 6487460: Insert Item 41 at 0
Producer 6487464: Insert Item 41 at 1
Consumer 6487456: Remove Item 6334 from 2
Consumer 6487456: Remove Item 26500 from 3
Producer 6487468: Insert Item 41 at 2
Consumer 6487460: Remove Item 19169 from 4
Consumer 6487456: Remove Item 41 from 0
Producer 6487472: Insert Item 41 at 3
Producer 6487460: Insert Item 18467 at 4
Consumer 6487460: Remove Item 41 from 1
Producer 6487464: Insert Item 18467 at 0
Consumer 6487460: Remove Item 41 from 2
Producer 6487468: Insert Item 18467 at 1
Consumer 6487460: Remove Item 41 from 3
Producer 6487472: Insert Item 18467 at 2
Consumer 6487460: Remove Item 18467 from 4
Producer 6487460: Insert Item 6334 at 3
Producer 6487464: Insert Item 6334 at 4
Consumer 6487464: Remove Item 18467 from 0
Consumer 6487464: Remove Item 18467 from 1
Producer 6487468: Insert Item 6334 at 0
Consumer 6487464: Remove Item 18467 from 2
Producer 6487472: Insert Item 6334 at 1
Consumer 6487464: Remove Item 6334 from 3
Producer 6487460: Insert Item 26500 at 2
Consumer 6487464: Remove Item 6334 from 4
```