Jigar Makwana

Assignment 2

Due Date 10/23/17

Purpose of the project:

Main purpose of this file is to initialize producer and buyer threads. Then producer produce product and adds it on the queue and buyer thread takes away the product. The synchronization is achieved using one mutex and two semaphores.

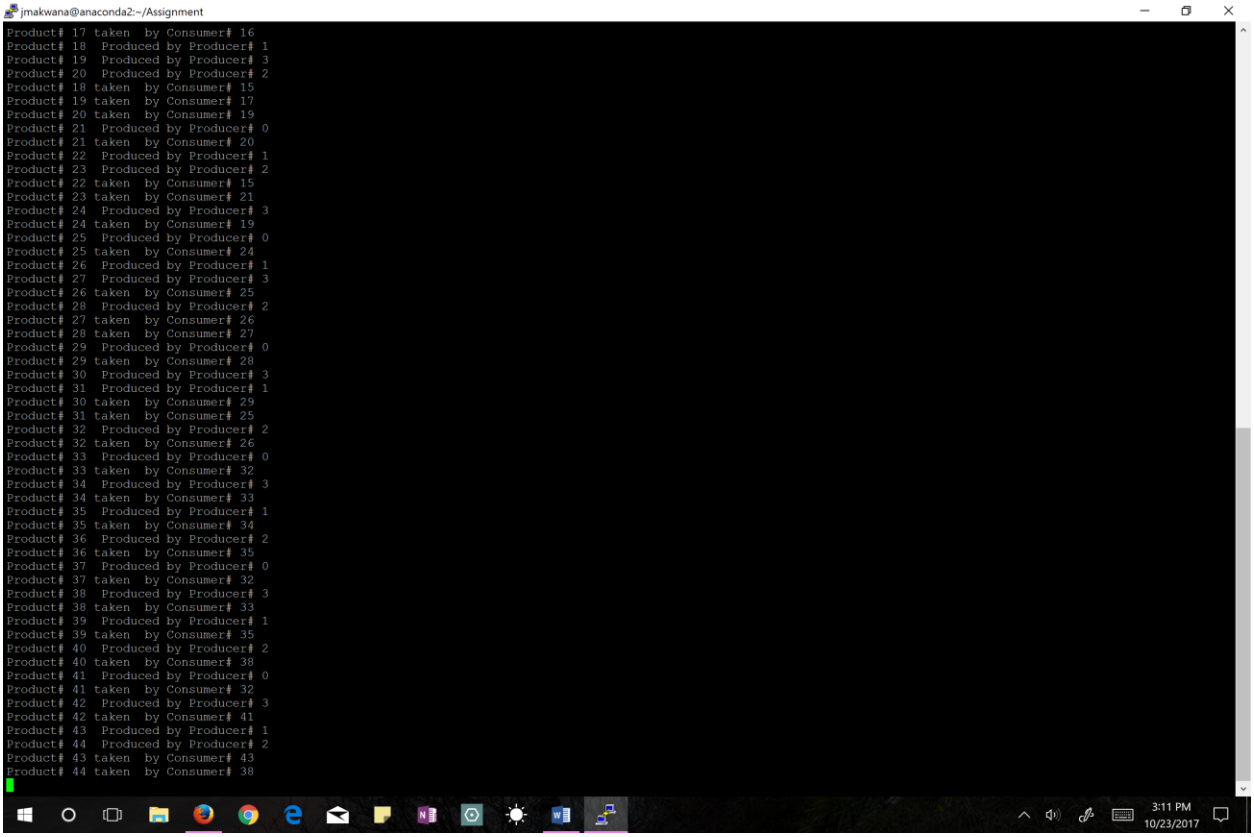
Background of the assignment:

Mutex are used for synchronization of the threads. It uses 1 or 0(lock or unlock) for giving time to access critical section of program. It gives access to only one thread at a time. On other hand Semaphore allows multiple threads to access the critical section. If it is more than 0 it allows thread to use critical section, but thread must wait if it is zero, if it is >0 thread can take the Semaphore to access the critical section. Also, thread can post the value to the semaphore after it done with the critical section.

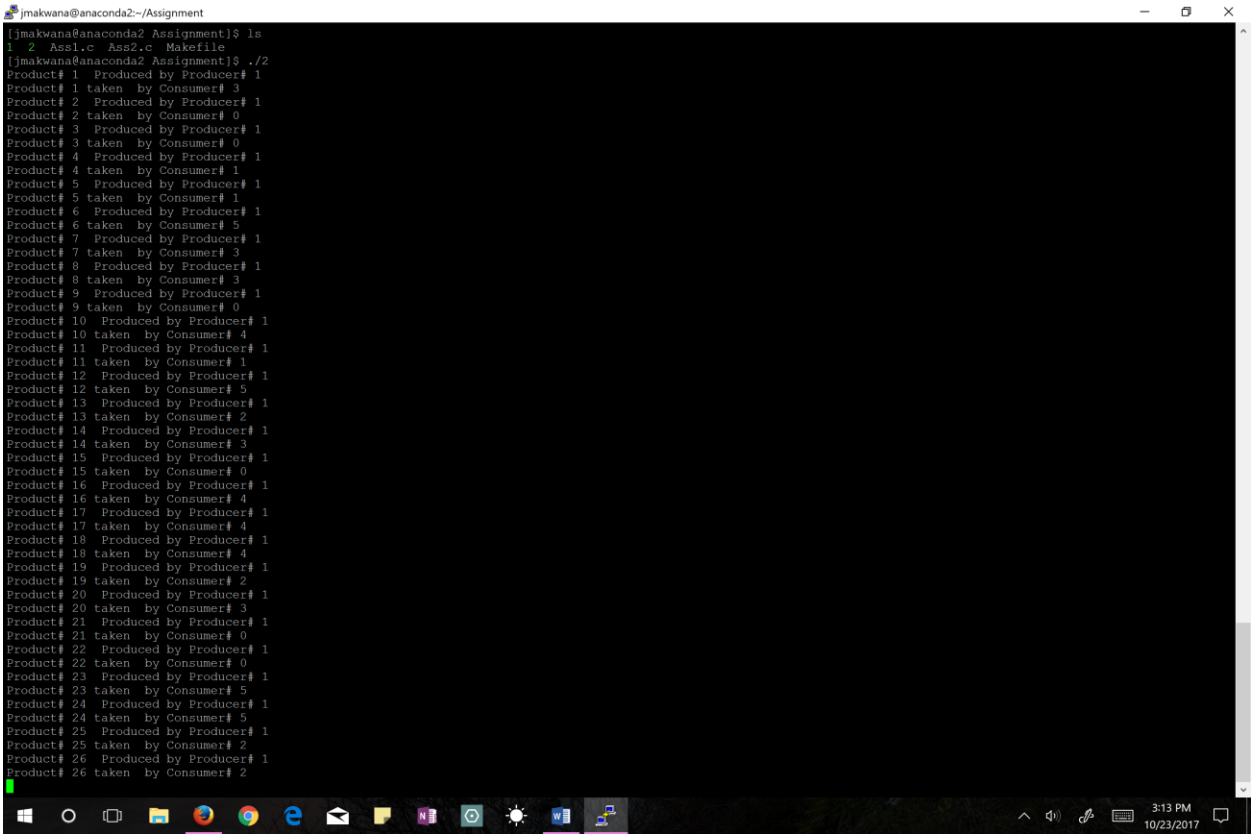
Algorithms/Functions used in assignment:

In this algorithm we used many important function like pthread_mutex_init() - to initialize mutex, sem_init() – to initialize semaphore, malloc() – to allocate memory, pthread_create() – to create threads, pthread_join() – to wait for threads to complete and join, pthread_mutex_destroy() – destroy mutex, sem_destroy() – destroy semaphore, sem_wait() – wait on semaphore, sem_post () – post the semaphore, pthread_mutex_lock() – mutex lock, pthread_mutex_unlock() -mutex unlock.
we also used two user define functions Gen and Take. Gen function generates the product on other hand Take function takes the product away. In both program we used buffer of 10. Hence there where 10 places for keeping product, 10 threads can go in and either add or take the product. In program one we implemented 4 threads called provider and 260 threads for buyer. Program handles synchronization of those 264 threads using Mutex and Semaphores. In second program we only have single buyer. So we can reduce semaphore unlike in program one. Hence, we used only one mutex and one semaphore in second program.

Result:



Result of 1.


jmakwana@anaconda2:~/Assignment
[jmakwana@anaconda2 Assignment]$ ls
1 2 Ass1.c Ass2.c Makefile
[jmakwana@anaconda2 Assignment]$ ./2
Product# 1 Produced by Producer# 1
Product# 1 taken by Consumer# 3
Product# 2 Produced by Producer# 1
Product# 2 taken by Consumer# 0
Product# 3 Produced by Producer# 1
Product# 3 taken by Consumer# 0
Product# 4 Produced by Producer# 1
Product# 4 taken by Consumer# 1
Product# 5 Produced by Producer# 1
Product# 5 taken by Consumer# 1
Product# 6 Produced by Producer# 1
Product# 6 taken by Consumer# 5
Product# 7 Produced by Producer# 1
Product# 7 taken by Consumer# 3
Product# 8 Produced by Producer# 1
Product# 8 taken by Consumer# 3
Product# 9 Produced by Producer# 1
Product# 9 taken by Consumer# 0
Product# 10 Produced by Producer# 1
Product# 10 taken by Consumer# 4
Product# 11 Produced by Producer# 1
Product# 11 taken by Consumer# 1
Product# 12 Produced by Producer# 1
Product# 12 taken by Consumer# 5
Product# 13 Produced by Producer# 1
Product# 13 taken by Consumer# 2
Product# 14 Produced by Producer# 1
Product# 14 taken by Consumer# 3
Product# 15 Produced by Producer# 1
Product# 15 taken by Consumer# 0
Product# 16 Produced by Producer# 1
Product# 16 taken by Consumer# 4
Product# 17 Produced by Producer# 1
Product# 17 taken by Consumer# 4
Product# 18 Produced by Producer# 1
Product# 18 taken by Consumer# 4
Product# 19 Produced by Producer# 1
Product# 19 taken by Consumer# 2
Product# 20 Produced by Producer# 1
Product# 20 taken by Consumer# 3
Product# 21 Produced by Producer# 1
Product# 21 taken by Consumer# 0
Product# 22 Produced by Producer# 1
Product# 22 taken by Consumer# 0
Product# 23 Produced by Producer# 1
Product# 23 taken by Consumer# 5
Product# 24 Produced by Producer# 1
Product# 24 taken by Consumer# 5
Product# 25 Produced by Producer# 1
Product# 25 taken by Consumer# 2
Product# 26 Produced by Producer# 1
Product# 26 taken by Consumer# 2
3:13 PM
10/23/2017

Result of 2.

Observations:

In program 2 we only have one buyer, so we can reduce semaphore unlike in program one. Hence, we used only one mutex and one semaphore in second program.

Conclusion:

Successfully created the producer and buyer threads according to the problem designed. Producer produce at least one item and buyer consumes at least one item.

Source Code:

Ass2_1.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>


#define BufferSize 10

typedef struct
        {
  int add;
  int take;
  int buffer[BufferSize];
        }product;


product proBuff = {0,0,{0}};
unsigned int proId = 1;

pthread_mutex_t mutex;
sem_t full;
sem_t empty;
```

```
void *Gen(void *id)
{
  while(1)
    {
      //wait till there is an empty place in buffer for product
      sem_wait(&empty);
      //wait for the critical area access
      pthread_mutex_lock(&mutex);
      //check if queue is full
      if(!(((proBuff.add + 1) % BufferSize ) == proBuff.take))
        {
          //Add the product in buffer
          proBuff.buffer[proBuff.add] = proId++;
          printf("Product# %d  Produced by Producer# %d
\n",proBuff.buffer[proBuff.add],(long*)id);
          //Increment the index for add
          proBuff.add = (proBuff.add + 1) % BufferSize;
        }
      else
        {
          printf("no place to enter \n",(long*)id);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex);
      //Post the produced item
      sem_post(&full);
      sleep(1);
    }
}


void *Take(void *id)
{
  while(1)
    {

      sem_wait(&full);
      //wait for the critical area access
      pthread_mutex_lock(&mutex);
      if(!(proBuff.add == proBuff.take))
        {
          printf("Product# %d taken  by Consumer# %d
\n",proBuff.buffer[proBuff.take],(long*)id);
          //take the product
          proBuff.buffer[proBuff.take] = 0;
          //Increment the index for take
          proBuff.take = (proBuff.take + 1) % BufferSize ;
        }
      else
        {
          printf("Nothing is avaiable when accessed by Consumer# %d
\n",(long*)id);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex);
      //Post the empty place
```

```
      sem_post(&empty);
      sleep(1);
    }
}


int main(int argc, char *argv[])
{
  unsigned int buyers = 260;
  unsigned int providers = 4;
  pthread_t pro[providers];
  pthread_t* bu;

  //Init Threads
  int t1 = pthread_mutex_init(&mutex, NULL);
  int t2 = sem_init(&full, 0, 0);
  int t3 = sem_init(&empty, 0, BufferSize);

  //Notify if failed
  if(t1!=0||t2!=0||t3!=0)
    printf("Intialization Error");

  bu = malloc(buyers*sizeof(pthread_t));

  long i;
  //Create threads for Producer
  for(i=0;i<providers;i++)
    pthread_create(&pro[i], NULL, Gen, (void*) i);

  //Create threads for buyer
  for(i=0;i<buyers;i++)
    pthread_create(&bu[i], NULL, Take, (void*) i);

  //Wait for Gen and Take threads to finish
  for(i=0;i<providers;i++)
    pthread_join(pro[i], NULL);
  for(i=0;i<buyers;i++)
    pthread_join(bu[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex);
  sem_destroy(&full);
  sem_destroy(&empty);
  return 0;
}
```

Ass2_2

```
#include<stdio.h>
#include<stdlib.h>
```

```
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>


#define BufferSize 10

typedef struct
        {
  int add;
  int take;
  int buffer[BufferSize];
        }product;


product proBuff = {0,0,{0}};
unsigned int proId = 1;

pthread_mutex_t mutex;
sem_t full;
sem_t empty;


void *Gen(void *id)
{
  while(1)
    {
      //wait till there is an empty place in buffer for product
      //sem_wait(&empty);
      //wait for the critical area access
      pthread_mutex_lock(&mutex);
      //check if queue is full
      if(!(((proBuff.add + 1) % BufferSize ) == proBuff.take))
        {
          //Add the product in buffer
          proBuff.buffer[proBuff.add] = proId++;
          printf("Product# %d  Produced by Producer# 1
\n",proBuff.buffer[proBuff.add],(long*)id);
          //Increment the index for add
          proBuff.add = (proBuff.add + 1) % BufferSize;
        }
      else
        {
          printf("no place to enter \n",(long*)id);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex);
      //Post the produced item
      sem_post(&full);
      sleep(1);
    }
}


void *Take(void *id)
{
```

```
  while(1)
    {

      sem_wait(&full);
      //wait for the critical area access
      pthread_mutex_lock(&mutex);
      if(!(proBuff.add == proBuff.take))
        {
          printf("Product# %d taken  by Consumer# %d
\n",proBuff.buffer[proBuff.take],(long*)id);
          //take the product
          proBuff.buffer[proBuff.take] = 0;
          //Increment the index for take
          proBuff.take = (proBuff.take + 1) % BufferSize ;
        }
      else
        {
          printf("Nothing is avaiable when accessed by Consumer# %d
\n",(long*)id);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex);
      //Post the empty place
      sem_post(&empty);
      sleep(1);
    }
}


int main(int argc, char *argv[])
{
  unsigned int buyers = 6;
  unsigned int providers = 1;
  pthread_t pro[providers];
  pthread_t* bu;

  //Init Threads
  int t1 = pthread_mutex_init(&mutex, NULL);
  int t2 = sem_init(&full, 0, 0);
  //int t3 = sem_init(&empty, 0, BufferSize);

  //Notify if failed
  if(t1!=0||t2!=0)
    printf("Intialization Error");

  bu = malloc(buyers*sizeof(pthread_t));

  long i;
  //Create threads for Producer
  for(i=0;i<providers;i++)
    pthread_create(&pro[i], NULL, Gen, (void*) i);

  //Create threads for buyer
  for(i=0;i<buyers;i++)
    pthread_create(&bu[i], NULL, Take, (void*) i);

  //Wait for Gen and Take threads to finish
```

```
  for(i=0;i<providers;i++)
    pthread_join(pro[i], NULL);
  for(i=0;i<buyers;i++)
    pthread_join(bu[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex);
  sem_destroy(&full);
  //sem_destroy(&empty);
  return 0;
}
```