Project 4: Producer-Consumer Problem

In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes. In Section 6.6.1 in the book, we have already discussed the solution. We should introduce three semaphores:

- empty - count the number of blanks in the buffer.
- full - count the number of items in the buffer.
- mutex - a binary semaphore that protects the actual insertion or removal of items in the buffer.

We also generate several producers and consumers - running as separate threads - will move items to and from a buffer that is synchronized with these empty, full, and mutex structures.

Getting Started

Since I have constituted the linux environment in previous project, this part is skimmed. In this project, *Ubuntu-16.04.1* for linux is used, and its kernel version is *4.4.0*.

The Buffer

The buffer will consist of a fix-size array of type buffer_item (which will be defined using a typedef). The array of buffer_item objects will be manipulated as a circular queue. We use the variables `in` and `out` to represent the tail and head of the queue.

```
typedef int buffer_item;
#define BUFFER_SIZE 5
buffer_item buffer[BUFFER_SIZE]; //the shared buffer

//in represents the tail while out represents the head of the circular queue
int in = 0, out = 0;
```

Code 1: Initialize the buffer

Notice that buffer have two operations: `insert_item()` and `remove_item()`. They are called by the producer and consumer threads, respectively. Considering that these two functions are easy to realize, I simply put them in the producer function and consumer function for convenience. Code 2 and 3 will only show the insert and remove part.

```
buffer[in] = ran;
printf("Producer %d has produced %d\tto buffer[%d] \n", id, ran, in);
in = (in + 1) % BUFFER_SIZE;
```

Code 2: Insert item for producers

```
ran = buffer[out];
printf("Consumer %d has consumed %d\tin buffer[%d] \n", id, ran, out);
out = (out + 1) % BUFFER_SIZE;
```

Code 3: Remove item for consumers

However, these functions should be treated carefully. The buffer should require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores, which will be discussed in next section.

Producer and Consumer Threads

The producer thread will create a random integer and insert it to the buffer. In fact, the producer will consume some time to produce the product. As a result, we can alternate between sleeping for a random period of time as the time spent. In order to protect the critical section, we must use mutex locks and semaphores. According to framework in the book, the producer thread is displayed in Code 4.

```
void *producer(void *param)
{
    buffer_item ran;
    int id = *(int *) param;

    while(1)
    {
        sleep(rand() % 5);
        ran = rand() % 1000;

        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = ran;
        printf("Producer %d has produced %d\tto buffer[%d] \n", id, ran, in);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);
    }
    return NULL;
}
```

Code 4: Producer thread

The consumer thread is almost the same. It will sleep for a random period of time, as the time spent consuming the product. Then it attempt to remove an item form the buffer.

```
void *consumer(void *param)
{
    buffer_item ran;
    int id = *(int *) param;

    while(1)
    {
        sleep(rand() % 5);

        sem_wait(&full);
        sem_wait(&mutex);

        ran = buffer[out];
        printf("Consumer %d has consumed %d\tin buffer[%d] \n", id, ran, out);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
    }
    return NULL;
}
```

Code 5: Consumer thread

Pthreads Thread Creation

We must create create producers and consumers using Pthreads, according to the input. Since last project it is fully discussed on this topic, the detailed is omitted here. The code is presented in Code 6.

```
//creation of produce threads
for(i = 0; i < pNum; ++i)
{
    pOrder[i] = 1 + i;
    ret = pthread_create(&p_thread[i], NULL, producer, &pOrder[i]);
    if(ret != 0) callError();
}

//creation of consumer threads
for(i = 0; i < cNum; ++i)
{
    cOrder[i] = 1 + i;
    ret = pthread_create(&c_thread[i], NULL, consumer, &cOrder[i]);
    if(ret != 0) callError();
}
```

Code 6: Pthreads thread creation

Producers and Consumers will keep running until the sleeping time is over.

Running Result

Figure 2 shows the running result.

```
Please input the sleep time, the number of producers and consumers:
5 5 6
Producer 4 produced 492 to buffer[0]
Producer 2 produced 421 to buffer[1]
Consumer 4 consumed 492 in buffer[0]
Consumer 1 consumed 421 in buffer[1]
Producer 5 produced 426 to buffer[2]
Producer 2 produced 736 to buffer[3]
Consumer 2 consumed 426 in buffer[2]
Consumer 3 consumed 736 in buffer[3]
Producer 1 produced 429 to buffer[4]
Producer 3 produced 530 to buffer[0]
Producer 2 produced 123 to buffer[1]
Consumer 1 consumed 429 in buffer[4]
Consumer 4 consumed 530 in buffer[0]
Consumer 6 consumed 123 in buffer[1]
Producer 4 produced 22  to buffer[2]
Producer 5 produced 69  to buffer[3]
Consumer 1 consumed 22  in buffer[2]
Consumer 5 consumed 69  in buffer[3]
```

Figure 1: Running result of Producer-Consumer

As can be depicted in Figure 2, the producers produce the items in the buffer while the consumers consume them later. The whole process works in good order and no deadlocks occurred. The full code is available in the APPENDIX.

I also write a program under Win32 environment. Though etails concerning thread creation using the Win32 API may differ, the structure resembles our previous one on the whole. The running result is shown as follows:

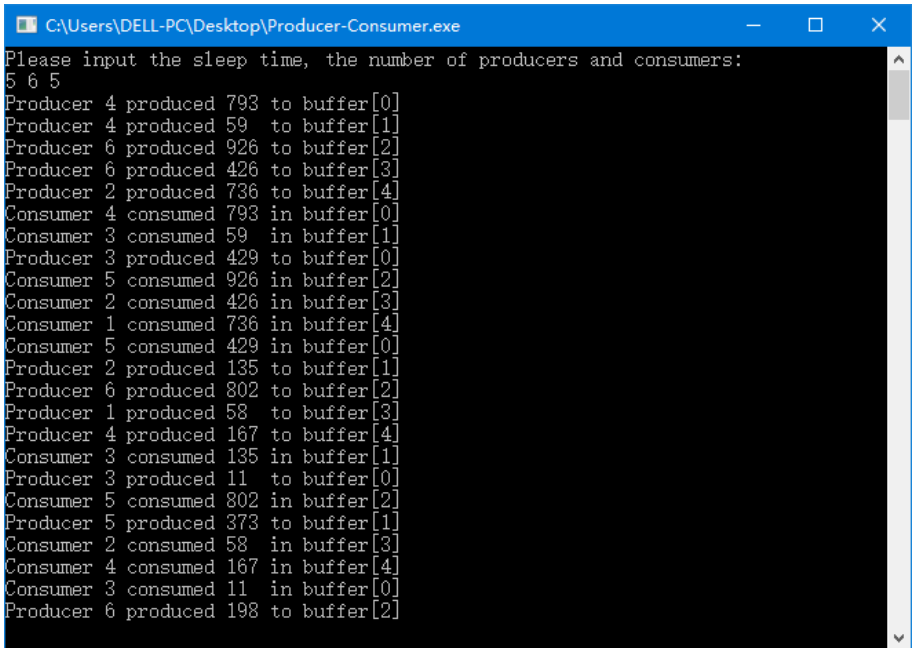

Figure 2: Running result of Producer-Consumer in Win32

Summary

This project is relatively easy since the structure of code has given in the book. But as far as I am concerned, it is quite meaningful. In this project, we reviewed the application of multithread when create producer and consumer threads. What's more, we can learn a lot through this bounded-buffer problem, which is one of the classic problems of synchronization. Now we have a deeper understanding of how to tackle the critical-section problem and some classic solutions to them.

Student name: Han Kunyan
Student No.:5140309534

Appendix

Full code of Producer-Consumer.c

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

typedef int buffer_item;
#define BUFFER_SIZE 5
#define NUM 100 //maximum num of producer_threads or consumer_threads
buffer_item buffer[BUFFER_SIZE]; //the shared buffer

sem_t empty, full;
sem_t mutex;
int in = 0, out = 0; //in = first empty buffer, out = first full buffer
int pOrder[NUM], cOrder[NUM];

void *producer(void *param)
{
    buffer_item ran;
    int id = *(int *) param;

    while(1)
    {
        sleep(rand() % 5);
        ran = rand() % 1000;

        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = ran;
        printf("Producer %d has produced %d\tto buffer[%d] \n", id, ran, in);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);

    }
    return NULL;
}

void *consumer(void *param)
{
    buffer_item ran;
    int id = *(int *) param;

    while(1)
    {
        sleep(rand() % 5);

        sem_wait(&full);
        sem_wait(&mutex);

        ran = buffer[out];
        printf("Consumer %d has consumed %d\tin buffer[%d] \n", id, ran, out);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
```

```
    }
    return NULL;
}

void callError()
{
    perror("create thread error!\n");
    exit(0);
}

int main(int argc, char *argv[])
{
    int pNum, cNum, sleepTime, i, ret;
    pthread_t p_thread[NUM];
    pthread_t c_thread[NUM];

    scanf("%d",&sleepTime);
    scanf("%d",&pNum);
    scanf("%d",&cNum);


    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);



    for(i = 0; i < pNum; ++i)
    {
        pOrder[i] = 1 + i;
        ret = pthread_create(&p_thread[i], NULL, producer, &pOrder[i]);
        if(ret != 0) callError();
    }


    for(i = 0; i < cNum; ++i)
    {
        cOrder[i] = 1 + i;
        ret = pthread_create(&c_thread[i], NULL, consumer, &cOrder[i]);
        if(ret != 0) callError();
    }

    sleep(sleepTime);
    return 0;

}
```