

# Experiment Report

## — Producer-Consumer Problem

Name: Wang Xingyi StudentID: 5140309531

December 22, 2016

## 1 Introduction

In this project, we will design a programming solution to the bounded-buffer problem using producer and consumer process. The solution uses three semaphores: **empty** and **full**, which count the number of empty and full slots in the buffer, and **mutex**, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, standard counting semaphores will be used for **empty** and **full**, and, rather than a binary semaphore, a mutex lock will be used to represent **mutex**. The producer and consumer - running as separate threads - will move items to and from a buffer that is synchronized with these **empty**, **full**, and **mutex** structure. We will solve this problem using Pthread.

## 2 Running environment

→ Ubuntu 16.04

## 3 Experimental procedure

### 3.1 The buffer

The buffer is defined in 'buffer.h' and its size is initially designed as 5.

```
1 /* buffer.h */
2 typedef int buffer_item
3 #define BUFFER_SIZE 5
```

This buffer will be manipulated with two functions, **insert\_item()** and **remove\_item()**.

```

1 #include "buffer.h"
2
3 /* the buffer */
4 buffer_item buffer[BUFFER_SIZE];
5
6 int insert_item(buffer_item item) {
7     /* insert item into buffer
8      * return 0 if successful, otherwise
9      * return -1 indicating an error condition */
10    if (cnt < BUFFER_SIZE) {
11        buffer[cnt++] = item;
12        return 0;
13    }
14    else
15        return -1;
16 }
17
18 int remove_item(buffer_item *item) {
19     /* remove an object from buffer
20      * placing it in item
21      * return 0 if successful, otherwise
22      * return -1 indicating an error condition */
23    if (cnt > 0) {
24        *item = buffer[--cnt];
25        return 0;
26    }
27    else
28        return -1;
29 }

```

### 3.2 Producer and Consumer Threads

We use Pthread to implement **producer** and **consumer** in different threads. Producer first sleeps for a couple of seconds and generate a random number to present that it has produced an item, and insert it into the buffer if the buffer is still available. Similarly, consumer also sleeps for a couple of seconds and consumes the remaining items in the buffer.

```

1 void *producer(void *param) {
2     buffer_item item;
3
4     while (1) {
5         /* sleep for a random period of time */

```

```

6         sleep(rand() % 5);
7         /* generate a random number */
8         item = rand();
9
10        sem_wait(&empty);
11        pthread_mutex_lock(&mutex);
12        if (insert_item(item))
13            fprintf(stderr, "report_error_condition\n"
14                    );
15        else
16            printf("producer_produced_%d\n", item);
17        pthread_mutex_unlock(&mutex);
18        sem_post(&full);
19    }
20}
21
22void *consumer(void *param) {
23    buffer_item item;
24
25    while (1) {
26        /* sleep for a random period of time */
27        sleep(rand() % 5);
28
29        sem_wait(&full);
30        pthread_mutex_lock(&mutex);
31        if (remove_item(&item))
32            fprintf(stderr, "report_error_condition\n"
33                    );
34        else
35            printf("consumer_consumed_%d\n", item);
36        pthread_mutex_unlock(&mutex);
37        sem_post(&empty);
38    }
39}

```

### 3.3 Mutex Locks

Mutex locks are available in the Pthread API and can be used to protect a critical section.

```

1 #include <pthread.h>
2 pthread_mutex_t mutex;
3
4 pthread_mutex_init(&mutex, NULL);

```

```

5 |
6 | pthread_mutex_lock(&mutex);
7 | /* critical section */
8 | pthread_mutex_unlock(&mutex);

```

### 3.4 Semaphores

We use unnamed semaphores provided by Pthread.

```

1 | #include <semaphore.h>
2 | sem_t empty, full;
3 |
4 | sem_init(&empty, 0, BUFFER_SIZE);
5 | sem_init(&full, 0, 0);

```

And the classical **wait()** and **signal()** semaphore operations are named as **sem\_wait()** and **sem\_post()** respectively.

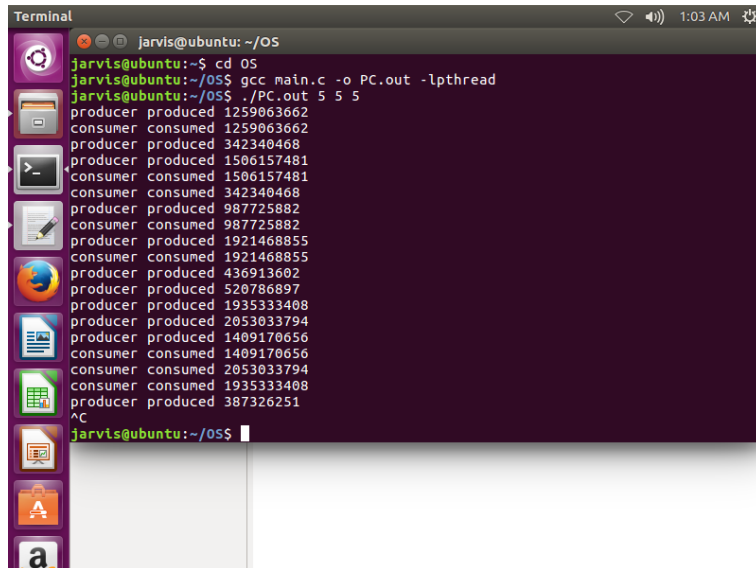
### 3.5 main()

The **main()** function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the **main()** function will sleep for a period of time and, upon awakening, will terminate the application. The **main()** function will be passed three parameters on the command line.

## 4 Conclusion and Discussion

We will demonstrate this producer-consumer system in three different situations.

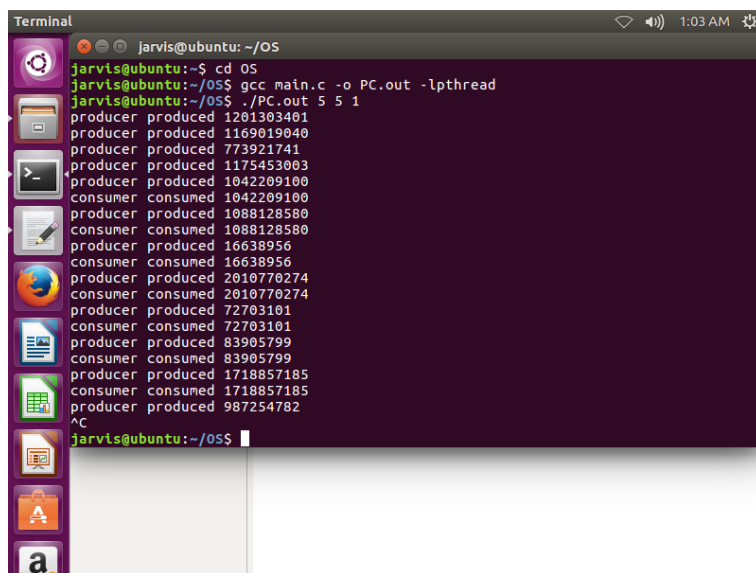
## 4.1 Producers are as many as Consumers

A terminal window titled 'Terminal' showing a command-line interface. The user 'jarvis@ubuntu' is in the directory '~/OS'. They run 'cd OS', then 'gcc main.c -o PC.out -lpthread', and finally './PC.out 5 5 5'. The output shows a sequence of 'producer produced' and 'consumer consumed' messages with memory addresses, indicating that 5 producers and 5 consumers successfully processed items in an alternating fashion.

```
jarvis@ubuntu:~/OS$ cd OS
jarvis@ubuntu:~/OS$ gcc main.c -o PC.out -lpthread
jarvis@ubuntu:~/OS$ ./PC.out 5 5 5
producer produced 1259063662
consumer consumed 1259063662
producer produced 342340468
producer produced 1506157481
consumer consumed 1506157481
consumer consumed 342340468
producer produced 987725882
consumer consumed 987725882
producer produced 1921468855
consumer consumed 1921468855
producer produced 436913602
producer produced 520786897
producer produced 1935333408
producer produced 2053033794
producer produced 1409170656
consumer consumed 1409170656
consumer consumed 2053033794
consumer consumed 1935333408
producer produced 387326251
^C
jarvis@ubuntu:~/OS$
```

In this situation, we allocate 5 producers and 5 consumers. The result shows that producer and consumer randomly produce or consume items. But an item must be consumed after being produced(of course!).

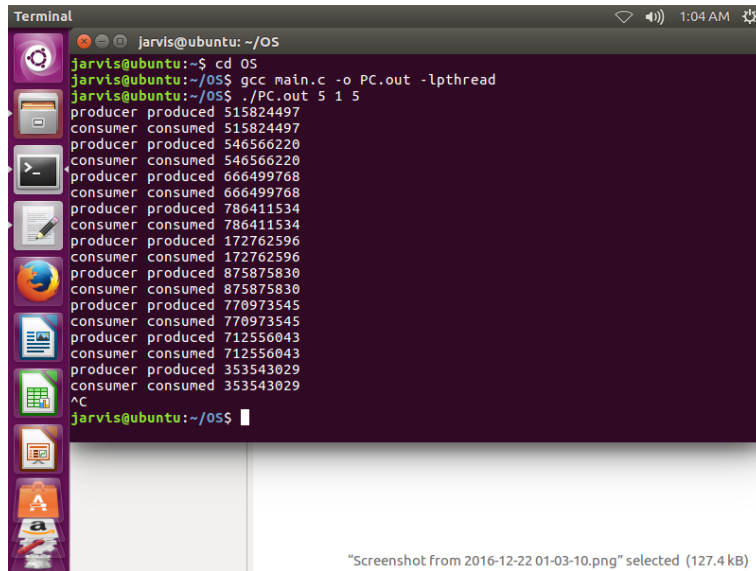
## 4.2 Producers are more than Consumers

A terminal window titled 'Terminal' showing a command-line interface. The user 'jarvis@ubuntu' is in the directory '~/OS'. They run 'cd OS', then 'gcc main.c -o PC.out -lpthread', and finally './PC.out 5 5 1'. The output shows 5 producers producing items first, followed by 1 consumer consuming one item. The sequence of messages indicates that the buffer becomes full after the first 4 items are produced, and no further production occurs until the first item is consumed.

```
jarvis@ubuntu:~/OS$ cd OS
jarvis@ubuntu:~/OS$ gcc main.c -o PC.out -lpthread
jarvis@ubuntu:~/OS$ ./PC.out 5 5 1
producer produced 1201303401
producer produced 1169019040
producer produced 773921741
producer produced 1175453003
producer produced 1042209100
consumer consumed 1042209100
producer produced 1088128580
consumer consumed 1088128580
producer produced 16638956
consumer consumed 16638956
producer produced 2010770274
consumer consumed 2010770274
producer produced 72703101
consumer consumed 72703101
producer produced 83905799
consumer consumed 83905799
producer produced 1718857185
consumer consumed 1718857185
producer produced 987254782
^C
jarvis@ubuntu:~/OS$
```

In this situation, we allocate 5 producers and 1 consumer. The result shows that 5 items are produced first, and the buffer is full. Once an item is consumed, another new item will be produced immediately. So the first 4 items will remain in the bounded buffer.

### 4.3 Producers are less than Consumers

A terminal window titled 'Terminal' with the path 'jarvis@ubuntu: ~/OS'. The user has executed the following commands: 'cd OS', 'gcc main.c -o PC.out -lpthread', and './PC.out 5 1 5'. The output shows a sequence of 'producer produced' and 'consumer consumed' messages with memory addresses. The addresses for producers are 515824497, 546566220, 666499768, 786411534, 172762596, 875875830, 770973545, 712556043, and 353543029. The addresses for consumers are 515824497, 546566220, 666499768, 786411534, 172762596, 875875830, 770973545, 712556043, and 353543029. The output ends with '^C' and the prompt 'jarvis@ubuntu:~/OS\$'.

```
jarvis@ubuntu:~/OS$ cd OS
jarvis@ubuntu:~/OS$ gcc main.c -o PC.out -lpthread
jarvis@ubuntu:~/OS$ ./PC.out 5 1 5
producer produced 515824497
consumer consumed 515824497
producer produced 546566220
consumer consumed 546566220
producer produced 666499768
consumer consumed 666499768
producer produced 786411534
consumer consumed 786411534
producer produced 172762596
consumer consumed 172762596
producer produced 875875830
consumer consumed 875875830
producer produced 770973545
consumer consumed 770973545
producer produced 712556043
consumer consumed 712556043
producer produced 353543029
consumer consumed 353543029
^C
jarvis@ubuntu:~/OS$
```

In this situation, we allocate 1 producers and 5 consumer. An item that is produced will be consumed immediately.