《操作系统综合实验》

实验报告

04 多线程与信号量编程

姓 名　王 栗 政
班 级　2018039
学 号　20179100018
教 师　冯鹏斌

实验日期　2022.5.6

一、实验题目

```
In Section 5.7.1, we presented a semaphore-based solution to the producer– consumer
problem using a bounded buffer. In this project, you will design a programming solution
to the bounded-buffer problem using the producer and consumer processes shown in
Figures 5.9 and 5.10. The solution presented in Section 5.7.1 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, you will use standard
counting semaphores for empty and full and a mutex lock, rather than a binary
semaphore, to represent mutex. The producer and consumer—running as separate threads—
will move items to and from a buffer that is synchronized with the empty, full, and
mutex structures. You can solve this problem using either Pthreads or the Windows API.
```

二、相关原理与知识

1. **Linux信号量和互斥锁**

```
//信号量和互斥锁的区别：
  //作用域：
    //信号量：进程间或线程间(Linux仅线程间的无名信号量pthread semaphore)
    //互斥锁：线程间
  //使用：
    //信号量：只要信号量的value大于0，其他线程就可以sem_wait成功，成功后信号量的value减1。若value值不大于0，则sem_wait使得线程阻塞，直到sem_post释放后value值加一,但是sem_wait返回之前还是会将此value值减一
    //互斥锁：只要被锁住，其他任何线程都不可以访问被保护的资源

int sem_init(sem_t *sem,int pshared,unsigned int value); //初始化
//对由sem指向的信号量进行初始化，设置好它的共享选项，并指定一个整数类型的初始值。pshared参数控制着信号量的类型。如果 pshared的值是 0，就表示它是当前进程的局部信号量；否则，其它进程就能够共享这个信号量。

int sem_destroy(sem_t *sem); //释放信号量
//释放sem指向的信号量，如果destory时有现成正在等待该信号量，则会返回一个错误

int sem_wait(sem_t *sem); //信号量-1
//原子操作，将sem指向的信号量-1。若为正整数才-1，若为0则会等待一直到有线程post 1，才-1。

int sem_trywait(sem_t *sem);// 信号量-1
//原子操作，与sem_wait相似，但是若为0，不会等待，而是直接返回以一个错误，是sem_wait的非阻塞版本

int sem_post(sem_t *sem); //信号量+1
//原子操作，将sem指向的信号量+1

int sem_getvalue(sem_t *sem, int * sval);//取值
//获取sem指向信号量的值，并将其指向sval
```

2. **生产者——消费者模型**

> 生产者消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。
>
> 要解决该问题，就必须让生产者在缓冲区满时休眠（或放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。该问题也能被推广到多个生产者和消费者的情形。

三、实验过程

实验思路：

使用两个信号量`empty`和`full`两个信号量分别记录空缓冲区和满缓冲区，使用三个互斥锁`mutex`、`pro_mutex`和`con_mutex`分别作为缓冲区、生产者计数和消费者计数的互斥锁，通过线程指针实现动态个数的消费者和生产者进程，通过循环进行创建，在生产（消费）的过程中首先获取到自己生产者（消费者）的编号，后续进入死循话，睡眠随机时间后生产（消费），为了使消费和生产不完全一致，使生产者和消费者睡眠不同的时间长度。

1. 获取生产者和消费者数

获取三个参数：程序执行时间、生产者个数和消费者个数，因为获取到的参数为字符串，所以通过`atoi`函数将其转化为`int`。

```
    if(argc != 4){
        puts("\033[31mUsage: ./chapter_4 sleep_time producer_size
consumer_size.\033[m");
        exit(0);
    }

    sleep_time = atoi(argv[1]);
    pro_size = atoi(argv[2]);
    con_size = atoi(argv[3]);

    if (sleep_time <= 0 || pro_size <= 0 || con_size <= 0)
    {
        puts("\033[31mError: the arguments must be more than 0.\033[m");
        return 0;
    }
```

2. 生产和消费

· 首先对信号量和互斥锁进行初始化，将`empty`初始化为缓冲区大小，将`full`初始化为`0`

· 在producer（consumer）函数中，首先获取到该生产者（消费者）的编号，之后进入生产（消费）的死循环，先睡眠随机时间，为了避免随机睡眠时间过长，我将随机取到的数模上一个值，又为了避免两者（大概率）总是消耗相同量的资源，我将生产者模5而消费者模9。

· 在每次生产时，首先减少一个`empty`告诉别的生产者自己现在在生产一个资源，生产成功之后再增加一个`full`告诉消费者自己已经成功生产了一个资源，可以消费了，这样缓冲区永远不可能溢出。在每次消费时同理。

· 在`insert_item`和`remove_item`函数中对于信号量和互斥锁的操作若出现错误，则会返回非`0`值，此时返回`-1`，并输出`errno`中的报错信息。

```
int insert_item(buffer_item item){

    if (sem_wait(&empty))   return -1;
    if (pthread_mutex_lock(&mutex))   return -1;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    if (pthread_mutex_unlock(&mutex))  return -1;
    if(sem_post(&full))  return -1;
    return 0;
}

int remove_item(){
    buffer_item item;

    if (sem_wait(&full))    return -1;
    if (pthread_mutex_lock(&mutex))   return -1;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    if (pthread_mutex_unlock(&mutex))  return -1;
    if(sem_post(&empty))  return -1;
    return item;

}

void *producer(void *param){
    buffer_item item;
    size_t name;
    pthread_mutex_lock(&pro_mutex);
    name = ++produce;
    pthread_mutex_unlock(&pro_mutex);

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

```
        item = rand()%100;
        if(insert_item(item))
            puts("\033[31mError: Something wrong !!!\033[m");
        else
            printf("PRODECER: %zd    PRODUCE: %zd\n",name,item);
        total_produce += item;
    }

}

void *consumer(void *param){
    buffer_item item;
    size_t name;
    pthread_mutex_lock(&con_mutex);
    name = ++consume;
    pthread_mutex_unlock(&con_mutex);
    while (1)
    {
        sleep(rand() % 9);

        if((item = remove_item()) == -1)
            puts("\033[31mError: Something wrong !!!\033[m");
        else
            printf("CONSUMER: %zd    CONSUME: %zd\n",name,item);
        total_consume += item;
    }

}

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&con_mutex, NULL);
    pthread_mutex_init(&pro_mutex, NULL);
    consumer_thread = (pthread_t*) malloc(sizeof(pthread_t) * con_size);
    producer_thread = (pthread_t*) malloc(sizeof(pthread_t) * pro_size);

    srand(time(NULL));

    for (int i = 0; i < pro_size;i++)
        pthread_create(producer_thread + i, NULL, producer, NULL);


    for (int i = 0; i < con_size;i++)
        pthread_create(consumer_thread + i, NULL, consumer, NULL);
```

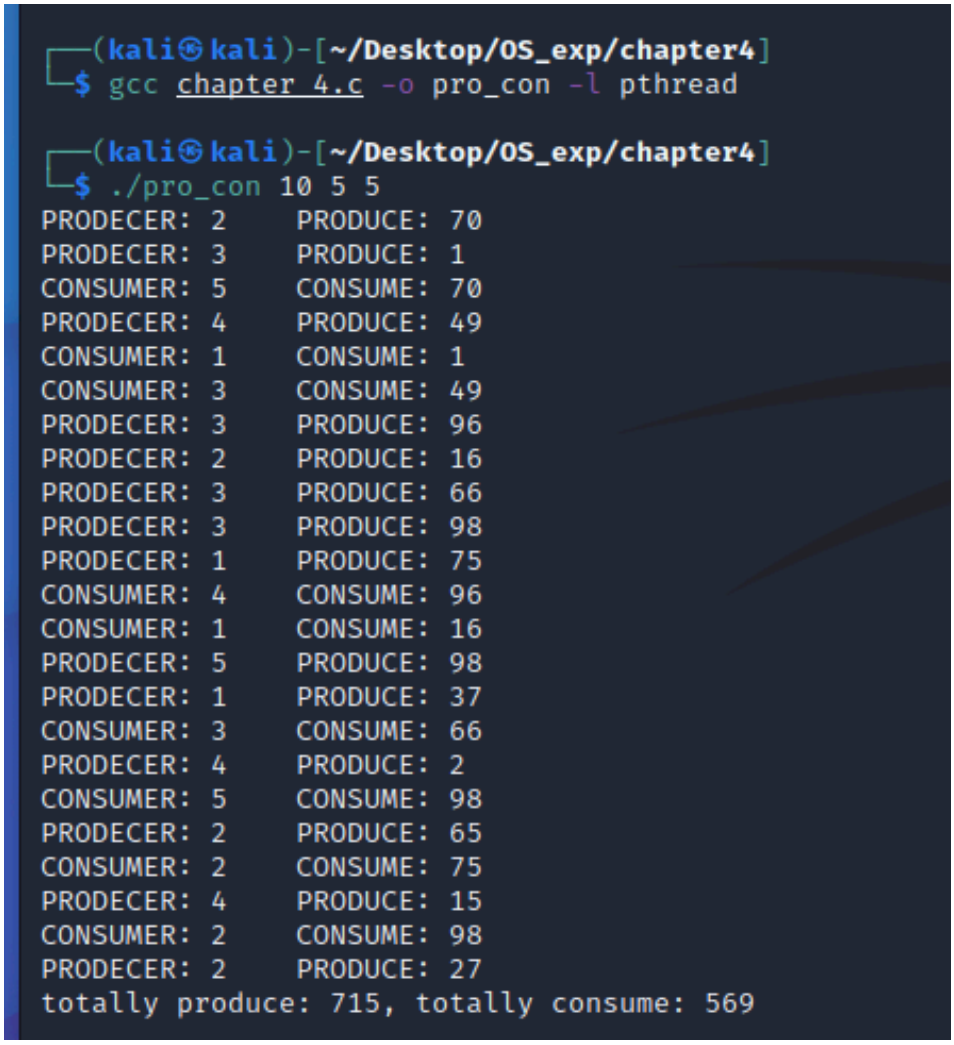
3. 主线程睡眠指定时间之后结束

使主线程睡眠指定时长，此处不需要进行`pthread_join`。只需要按参数的要求等待指定时长即可，之后直接将互斥锁`mutex`加锁，即可停止生产者和消费者进程的生产和消费过程，之后对于生产和消费的总资源数进行输出。

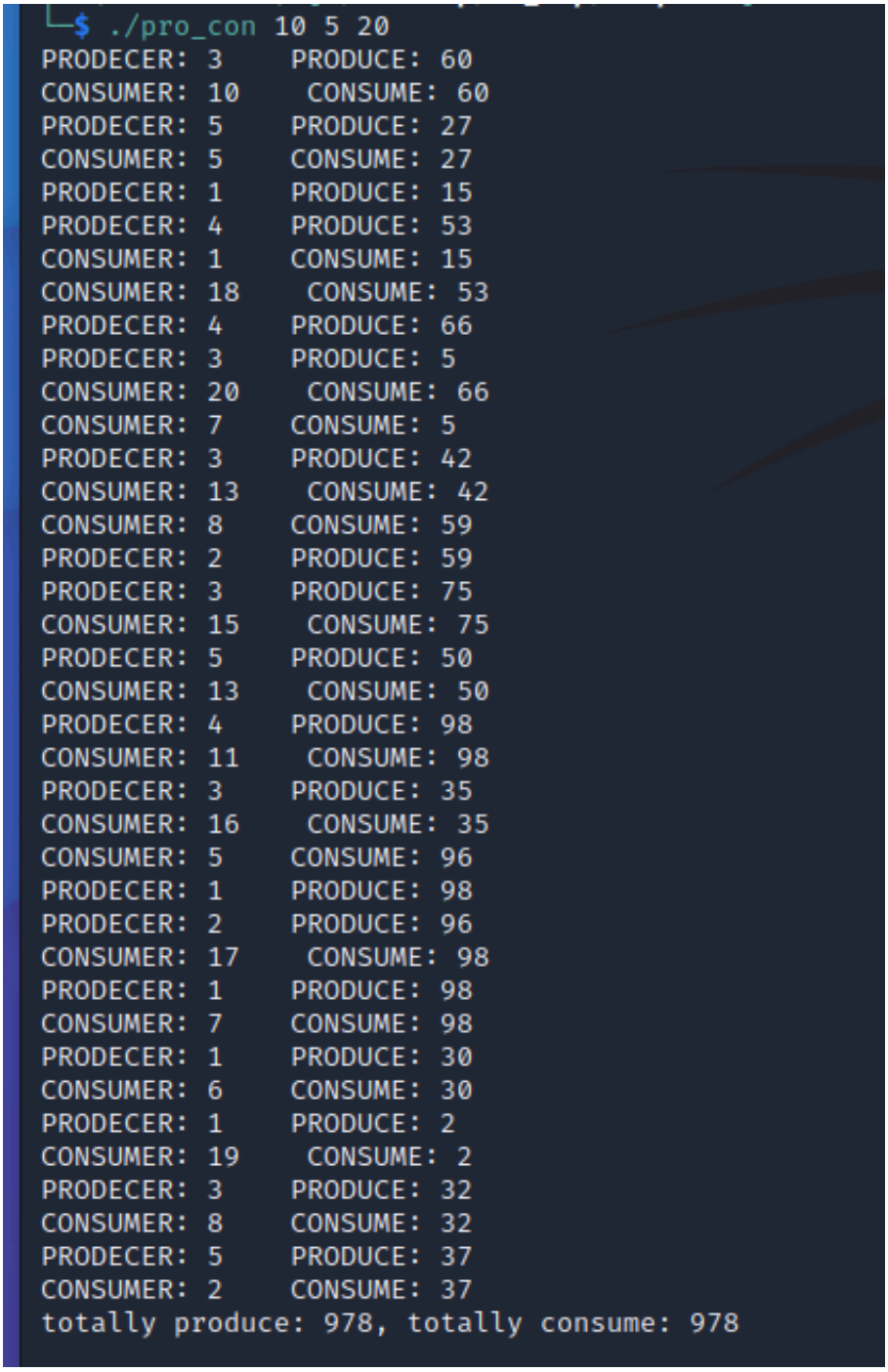
```
    sleep(sleep_time);
    pthread_mutex_lock(&mutex);
    printf("totally produce: %zd, totally consume: %zd\n", total_produce,
total_consume);
    exit(0);
```

四、实验结果与分析

让程序运行10s，创建生产者和消费者均为5个



由于设置的生产者睡眠时间比消费者长，所以资源大概率不被消耗完。可以尝试创建远多于生产者的消费者



此时创建的资源可以被消耗完。

五、问题总结

1. 第一次实验时对于生产者消费者模型理解不清晰

在第一次写的时候，没有理解到`empty`和`full`的妙用，采用了一个用于记录缓冲区现有资源数的值`count`来防止缓冲区溢出，后来发现：生产者每次生产时，先减少一个`empty`告诉别的生产者自己现在在生产一个资源，生产成功之后再增加一个`full`告诉消费者自己已经成功生产了一个资源，可以消费了，这样缓冲区永远不可能溢出。

2. Mac系统和Linux系统对于信号量的使用不同

实验时直接在Mac写的程序进行运行，发现一直报错找不出原因，后来发现Mac不使用`sem_init`函数创建信号量，而是采用`sem_open`函数，这一点需要注意。

六、源代码

1. buffer.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include"buffer.h"
#include<unistd.h>


typedef size_t buffer_item;
#define BUFFER_SIZE 5
buffer_item buffer[BUFFER_SIZE];
size_t sleep_time, pro_size, con_size;
sem_t empty,full;
pthread_t * producer_thread, * consumer_thread;
size_t in=0,out=0,count=0;
size_t produce=0,consume=0;
pthread_mutex_t mutex, pro_mutex, con_mutex;
size_t total_produce=0,total_consume=0;
```

2. buffer.c

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




int insert_item(buffer_item item){
    // while (count == BUFFER_SIZE );//判断是否溢出，但是不完全正确

    if (sem_wait(&empty))   return -1;
    if (pthread_mutex_lock(&mutex))   return -1;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    if (pthread_mutex_unlock(&mutex))  return -1;
```

```
    if(sem_post(&full))  return -1;
    return 0;
}

int remove_item(){
    buffer_item item;

    if (sem_wait(&full))    return -1;
    if (pthread_mutex_lock(&mutex))   return -1;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    if (pthread_mutex_unlock(&mutex))  return -1;
    if(sem_post(&empty))  return -1;
    return item;

}

void *producer(void *param){
    buffer_item item;
    size_t name;
    pthread_mutex_lock(&pro_mutex);
    name = ++produce;
    pthread_mutex_unlock(&pro_mutex);

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

        item = rand()%100;
        if(insert_item(item))
            puts("\033[31mError: Something wrong !!!\033[m");
        else
            printf("PRODECER: %zd    PRODUCE: %zd\n",name,item);
        total_produce += item;
    }

}

void *consumer(void *param){
    buffer_item item;
    size_t name;
    pthread_mutex_lock(&con_mutex);
    name = ++consume;
    pthread_mutex_unlock(&con_mutex);
    while (1)
    {
        sleep(rand() % 9);

        if((item = remove_item()) == -1)
```

```
            puts("\033[31mError: Something wrong !!!\033[m");
        else
            printf("CONSUMER: %zd    CONSUME: %zd\n",name,item);
        total_consume += item;
    }

}
```

3. Chapter_4.c

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



int main(int argc,char *argv[]){
    if(argc != 4){
        puts("\033[31mUsage: ./chapter_4 sleep_time producer_size
consumer_size.\033[m");
        exit(0);
    }

    sleep_time = atoi(argv[1]);
    pro_size = atoi(argv[2]);
    con_size = atoi(argv[3]);

    if (sleep_time <= 0 || pro_size <= 0 || con_size <= 0)
    {
        puts("\033[31mError: the arguments must be more than 0.\033[m");
        return 0;
    }

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&con_mutex, NULL);
    pthread_mutex_init(&pro_mutex, NULL);
    consumer_thread = (pthread_t*) malloc(sizeof(pthread_t) * con_size);
    producer_thread = (pthread_t*) malloc(sizeof(pthread_t) * pro_size);


    srand(time(NULL));

    for (int i = 0; i < pro_size;i++)
        pthread_create(producer_thread + i, NULL, producer, NULL);
```

```
    for (int i = 0; i < con_size;i++)
        pthread_create(consumer_thread + i, NULL, consumer, NULL);


    sleep(sleep_time);
    pthread_mutex_lock(&mutex);
    printf("totally produce: %zd, totally consume: %zd\n", total_produce,
total_consume);
    exit(0);
}
```