Proj5 Designing a Thread Pool & Producer-Consumer Problem

by 郑航 520021911347

1 Project 1—Designing a Thread Pool

1.1 Abstract

实现一个含等待队列的线程池，功能：创建一系列线程，用户可以通过submit向池内提交待完成的task，所有task都会先被置于等待队列上，若有空闲线程则将task从等待队列中取下并执行，否则该task一直等待直到有空闲线程

本次project采用 **Pthread and POSIX** 的实现

主要分五部分进行实现：

- 全局变量的设计
- 线程池的初始化
- 等待队列的实现
- 子线程的执行
- 线程池的关闭

1.2 全局变量的设计

在本project中，我们总共设计了如下全局变量：

- int num_of_threads：线程池中线程数目，可由用户定义
- int* thread_index：线程的人为编号，从0开始递增
- pthread_t *bee：线程数组，存储了各子线程的线程ID
- int shutdown：shutdown的信号，当执行shutdown函数时，会将shutdown置为1，使得所有子线程退出
- task_node *head, *tail：分别表示等待队列的头元素和尾元素
- pthread_mutex_t mutex：互斥锁，用于多线程环境下保证对等待队列的操作彼此间互斥，防止少执行或者多执行任务
- sem_t *sem：named信号量，将等待队列上的task视为共享资源，用sem准确记录其数目并使得线程在无任务可执行的情况下保持等待新任务的状态

1.3 线程池的初始化

初始化由pool_init函数来进行，总共需要完成以下几个部分：

- 依据用户选择的进程数进行动态内存分配：利用malloc函数为bee和thread_index分配空间
- shutdown，head和tail的初始化
- mutex和semaphore的初始化：利用对应函数初始化即可，注意要检查初始化是否成功
- 进程的创建：分别利用pthread_create创建num_of_threads个子线程，并将thread_index作为work的参数传入，注意该部分需要放在init函数最后进行，保证在子线程执行task时其他变量都已得到初始化

代码如下：

```c
// initialize the thread pool
void pool_init(int num)
{
    num_of_threads=num;
    bee=(pthread_t*)malloc(sizeof(pthread_t)*num_of_threads);
    thread_index=(int*)malloc(sizeof(int)*num_of_threads);

    head = tail = NULL;
    shutdown = 0;

    // initialize mutual-exclusion locks and semaphores
    int ret = pthread_mutex_init(&mutex, NULL);
    if (ret)
    {
        printf("mutex initialization failed!\n");
        return;
    }
    sem = sem_open("QUEUE_SEM", O_CREAT, 0666, 0);
    if (sem == SEM_FAILED)
    {
        printf("semaphore initialization failed!\n");
        return;
    }

    // create the threads
    for (int i = 0; i < num_of_threads; i++)
    {
        thread_index[i] = i;
        ret = pthread_create(&bee[i], NULL, worker, &thread_index[i]);
    }
}
```

1.4 等待队列的实现

本次project我们采用动态等待队列的实现方式，即采用linked list作为数据结构，且为了添加删除元素的高效和方便，在head之外还引入了一个tail指针指向链表末尾。为了保证队列的先进先出性质，每次插入元素在tail后进行，而删除则在head处进行，为了使用链表，需要先定义如下的结点结构体：

```c
struct task_node
{
    task tsk;
    struct task_node *next;
};
```

1.4.1 enqueue

首先将传入的参数t（task）存入一个新分配内存的结点中，然后根据目前队列是否为空分两种情况将该结点插入队列中

- 若队列为空，则此时应该head和tail都是NULL，直接将两者都指向新建的结点tmp即可
- 若队列不为空，将结点tmp插入在当前tail所指的结点后

具体代码如下：

```c
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    struct task_node *tmp = (struct task_node *)malloc(sizeof(struct
task_node));
    if (tmp == NULL)
    {
        printf("malloc error and enqueue failed!\n");
        return 1;
    }

    tmp->tsk = t;
    tmp->next = NULL;
    if (tail == NULL) // first task
    {
        head = tmp;
        tail = tmp;
    }
    else // others
    {
        tail->next = tmp;
        tail = tail->next;
    }
    //printf("enqueue seccessful\n");
    return 0;
}
```

1.4.2 dequeue

若队列为空则报错，否则取出head当前所指的结点，并将其task返回，注意若dequeue之后队列已经没有元素了，需要将tail一并更新为NULL，便于下次插入新结点时的空队列条件判断

具体代码如下：

```c
// remove a task from the queue
task dequeue()
{
    if (head == NULL)
    {
        printf("waiting queue empty and dequeue failed!\n");
        return;
    }
    task worktodo = head->tsk;
    struct task_node *tmp = head;
    head = head->next;
    if (head == NULL)
        tail = NULL;
    free(tmp);
    //printf("dequeue seccessful\n");
    return worktodo;
}
```

1.5 子线程的执行

主要分为两部分，一部分是用户可用的pool_submit函数，即用户将其所需执行的task提交到线程池的方法；另一部分是线程池内部的task执行方法

1.5.1 pool_submit

该函数的功能为将用户的task提交到线程池中，主要包含下面几个部分：

- 将用户的目标函数和数据集中打包到一个task中
- 将该task通过enqueue放入等待队列中，注意enqueue的过程需要在mutex的保护下进行，防止多个线程同时进行enqueue或dequeue产生race condition，导致多执行或少执行某个任务
- 判断插入是否成功，若不成功则报错，否则递增信号量，表示等待队列中加入了一个新的可用task

具体代码如下：

```c
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;

    int res;
    pthread_mutex_lock(&mutex);
    res = enqueue(worktodo);
    pthread_mutex_unlock(&mutex);

    if (res == 1)
    {
        printf("task submission to pool failed!\n");
        return 1;
    }
    else
    {
        sem_post(sem); // increase the semaphore
        return 0;
    }
}
```

1.5.2 worker和execute

子线程的执行函数为worker(void *param)，为了使得子线程始终处于工作中，将函数的主体设计为一个条件永远为true的while循环，只有在接收到shutdown信号时才退出循环并终止线程

while循环中主要包括以下部分：

- sem_wait(sem);　等待并递减信号量，如等待队列中有可用task，则递减信号量后进入执行，否则持续等待直到有可用task进入等待队列
- shutdown信号的判断
- 获取当前可用task：通过dequeue从等待队列中获取，注意这个过程需要在mutex的保护下进行，防止多个线程同时进行enqueue或dequeue产生race condition，导致多执行或少执行某个任务
- 对task具体内容进行执行：调用execute函数即可

具体代码如下：

```c
// the worker thread in the thread pool
void *worker(void *param)
{
    while (TRUE)
    {
        sem_wait(sem); // decrease the semaphore
        if (shutdown)
            break;

        pthread_mutex_lock(&mutex);
        task worktodo = dequeue();
        pthread_mutex_unlock(&mutex);

        // execute the task
        printf("In thread %d:    ", *((int *)param));
        execute(worktodo.function, worktodo.data);
    }

    pthread_exit(0);
}

//Executes the task provided to the thread pool
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}
```

1.6 线程池的关闭

通过pool_shutdown函数来完成，其过程与pool_init有类似对称的行为，主要包含以下几个部分：

- 设置shutdown信号：通知各子线程停止等待新任务并终止线程
- 对sem递增num_of_threads次：这一步可能一开始难以理解，考虑到shutdown信号到来之前很可能所有线程都已经完成其可用的所有任务并处于等待新任务的状态中，即此时sem值为0而线程在sem_wait处等待，假如不进行sem的递增，则线程无法接收到shutdown信号然后退出循环，因此必须在此时递增sem将线程从等待中释放出来
- 依次调用pthread_join合并子线程，使得main线程不至于提前于子线程结束
- 摧毁mutex和sem
- 释放申请的动态内存

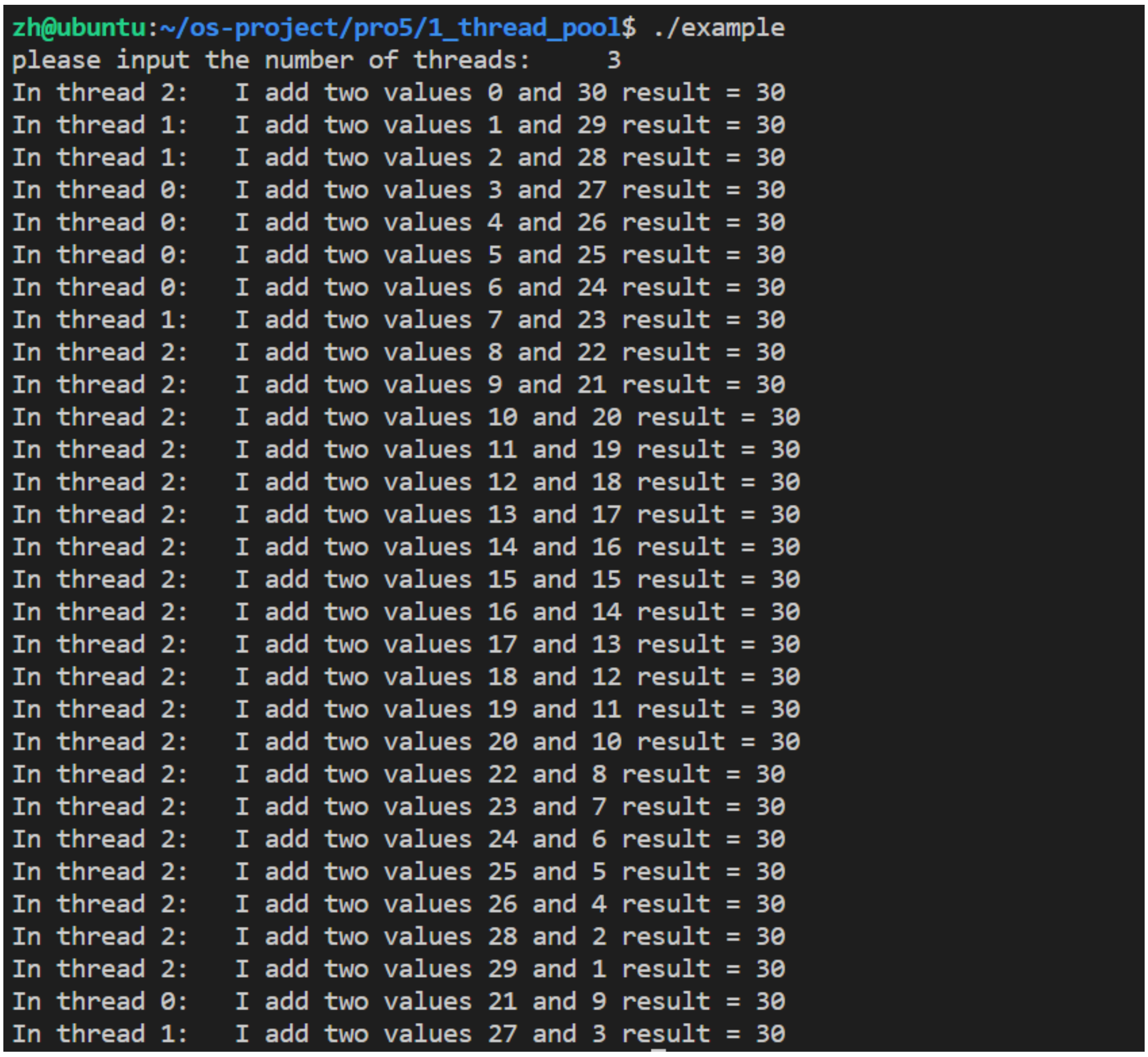
具体代码如下：

```c
// shutdown the thread pool
void pool_shutdown(void)
{
    shutdown = 1;

    // release sem so that threads can exit
    for (int i = 0; i < num_of_threads; i++)
    {
        sem_post(sem);
    }

    //  join the threads
    for (int i = 0; i < num_of_threads; i++)
    {
        pthread_join(bee[i], NULL);
    }

    // destroy mutex and semaphore
    pthread_mutex_destroy(&mutex);
    sem_destroy(sem);

    free(bee);
    free(thread_index);
}
```

1.7 运行结果

首先修改client.c程序，使得主程序可以接收用户输入的线程池线程数，并创建30个add任务，依次提交到线程池中进行执行

测试结果如下：



```
zh@ubuntu:~/os-project/pro5/1_thread_pool$ ./example
please input the number of threads:     3
In thread 2:   I add two values 0 and 30 result = 30
In thread 1:   I add two values 1 and 29 result = 30
In thread 1:   I add two values 2 and 28 result = 30
In thread 0:   I add two values 3 and 27 result = 30
In thread 0:   I add two values 4 and 26 result = 30
In thread 0:   I add two values 5 and 25 result = 30
In thread 0:   I add two values 6 and 24 result = 30
In thread 1:   I add two values 7 and 23 result = 30
In thread 2:   I add two values 8 and 22 result = 30
In thread 2:   I add two values 9 and 21 result = 30
In thread 2:   I add two values 10 and 20 result = 30
In thread 2:   I add two values 11 and 19 result = 30
In thread 2:   I add two values 12 and 18 result = 30
In thread 2:   I add two values 13 and 17 result = 30
In thread 2:   I add two values 14 and 16 result = 30
In thread 2:   I add two values 15 and 15 result = 30
In thread 2:   I add two values 16 and 14 result = 30
In thread 2:   I add two values 17 and 13 result = 30
In thread 2:   I add two values 18 and 12 result = 30
In thread 2:   I add two values 19 and 11 result = 30
In thread 2:   I add two values 20 and 10 result = 30
In thread 2:   I add two values 22 and 8 result = 30
In thread 2:   I add two values 23 and 7 result = 30
In thread 2:   I add two values 24 and 6 result = 30
In thread 2:   I add two values 25 and 5 result = 30
In thread 2:   I add two values 26 and 4 result = 30
In thread 2:   I add two values 28 and 2 result = 30
In thread 2:   I add two values 29 and 1 result = 30
In thread 0:   I add two values 21 and 9 result = 30
In thread 1:   I add two values 27 and 3 result = 30
```

可以看到，线程池实现成功，且可以观察到各个task的执行总体上是按序的，且同一个线程内是严格按序的，这是由于采用的是队列，遵循先进先出的原则；而各线程间可能存在乱序现象。事实上，假如task的耗时更长，线程数更多，我们应该可以看到更加明显的乱序现象

1.8 完整代码

考虑到代码篇幅较长，完整代码详见附录

2 Project 2—Producer-Consumer Problem

2.1 Abstract

- 本次project**基于Pthread**，实现一个producer-consumer问题的模拟程序
- 功能：用户可以指定模拟时长以及生产者消费者的数量，程序内利用一个循环队列模拟有限容量的buffer，生产者和消费者每隔一个随机的时长进行buffer-item的生产或消费，并在模拟时长到了之后退出程序
- 主要分四部分：

    - buffer的实现
    - main的设计和实现
    - producer
    - consumer

3.2 buffer的实现

buffer利用循环队列的数据结构来实现

设置buffer的最大容量为5，根据要求，buffer相关的操作都只能通过insert_item和remove_item来完成，同时还需要提供初始化函数，因此buffer的实现主要包括以下三部分：

- insert_item
- remove_item
- buffer_init

2.2.1 insert_item

我们的循环队列有两个指针front和rear，front指向队列首元素，rear指向尾元素，每次对front和rear操作都可以加入对（BUFFER_SIZE+1）的取模操作，以实现“循环”的特性。注意循环队列的实现中，front指向的是一个空的位置，所以实际buffer数组的大小得设置为（BUFFER_SIZE+1）

insert_item插入函数，首先需要判断队列目前是否已满，若满则返回-1（表示错误信息），否则直接将新的item插入并返回0（表示插入成功）即可，判断队列已满的标志是rear即将赶上front，即(rear + 1) % (BUFFER_SIZE+1) == front

具体代码如下：

```c
// insert an item to the buffer
int insert_item(buffer_item item)
{
    if ((rear + 1) % (BUFFER_SIZE+1) == front)
    {
        return -1;
    }
    rear = (rear + 1) % (BUFFER_SIZE+1);
    buf[rear] = item;
    return 0;
}
```

2.2.2 remove_item

实现与insert_item类似，只需要将判满改为判空即可，判空的标志是rear和front恰好处于同一个位置，即front == rear

具体代码如下：

```c
// remove an item from the buffer
int remove_item(buffer_item *item)
{
    if (front == rear)
    {
        return -1;
    }
    front = (front + 1) % (BUFFER_SIZE + 1);
    *item = buf[front];
    return 0;
}
```

2.2.3 buffer_init

只需要对front和rear进行初始化即可

具体代码如下:

```c
void buffer_init()
{
    front = 0;
    rear = 0;
}
```

2.3 main的设计和实现

main函数是本次project的设计重点，主要包括以下几个部分:

- 参数读取和变量的初始化：读取参数，设置变量存储这些参数，创建锁和信号量
- 子线程的创建：创建给定数目的producer和consumer线程
- sleep 延时一段时间
- 子线程的终止
- 程序退出前的处理：锁和信号量的销毁和动态内存释放

2.3.1 参数读取和变量的初始化

- 首先判断用户输入的参数数目是否正确，并将三个参数经过atoi函数转为int类型后存在对应变量中
- buffer初始化
- 锁和信号量的初始化（与Proj 1 中类似）

具体代码如下:

```c
if (argc != 4)
{
    printf("Error: invalid arguments.\n");
    return 1;
}
int time = atoi(argv[1]);
int num_of_producers = atoi(argv[2]);
int num_of_consumers = atoi(argv[3]);

// buffer initialization
buffer_init();

// mutex and semaphore initialization
pthread_mutex_init(&mutex, NULL);
```

```c
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
```

2.3.2 子线程的创建

利用类似Proj 1中的方式，用Pthread_create函数创建对应数目的子线程即可

- 由于此处需由用户设定线程数，因此需要使用动态内存分配的方式
- 为了更清晰地看到各producer和consumer线程的执行情况，可以将index（从0开始）作为参数传入子线程中

具体代码如下:

```c
pthread_t *producers = (pthread_t *)malloc(sizeof(pthread_t) * num_of_producers);
pthread_t *consumers = (pthread_t *)malloc(sizeof(pthread_t) * num_of_consumers);
for (int i = 0; i < num_of_producers; ++i)
{
    int producer_index = i;
    pthread_create(&producers[i], NULL, &producer, &producer_index);
}
for (int i = 0; i < num_of_consumers; ++i)
{
    int consumer_index = i;
    pthread_create(&consumers[i], NULL, &consumer, &consumer_index);
}
```

2.3.3 sleep 延时一段时间

直接调用sleep函数即可

```c
// sleep before termination
sleep(time);
```

2.3.4 子线程的终止

采用不同于Proj1 中利用pthread_join的退出方式，在这里我们采用pthread_cancel的方式来退出，原因是Proj 1中我们需要等待shutdown函数的调用后才退出，无法预计退出时间，因此main函数必须始终等待子线程的结束。而本project中，我们已经利用sleep函数进行等待，时间结束即可立即终止子线程，因此直接调用pthread_cancel的方式更加方便，也无需考虑信号量的等待等问题

具体代码如下:

```c
// termination
for (int i = 0; i < num_of_producers; ++i)
    pthread_cancel(producers[i]);
for (int i = 0; i < num_of_consumers; ++i)
    pthread_cancel(consumers[i]);
```

2.3.5 程序退出前的处理

进行锁和信号量的销毁以及动态内存的回收，代码如下：

```c
// destroy mutex and semaphore
pthread_mutex_destroy(&mutex);
sem_destroy(&empty);
sem_destroy(&full);

free(producers);
free(consumers);
return 0;
```

2.4 producer

producer的代码框架在课后project的介绍部分已经给出，我们只需要将其完善即可

- 随机数的生成直接调用rand()函数即可，为了保证等待时间不过长，我们sleep的随机时间对3取模
- critical section的部分需要在信号量和锁的保护下执行，具体方式与课内教授的一致，信号量在外层，mutex在内层
- 进入critical section前需要确保此时buffer未满，需要wait一下empty信号量，若empty不为0则表示有空位，可直接进入，并将empty减一，否则等待直到empty不为0；离开critical section前需要post一下full信号量，表示buffer中数据量加一
- 检查写入buffer是否成功，不成功输出错误信息并退出线程，否则输出本次produce的信息（线程index和buffer-item）

具体代码如下：

```c
// producer thread
void *producer(void *param)
{
    buffer_item item;
    int index = *(int *)param;
    while (TRUE)
    {
        /* sleep for a random period of time */
        sleep(rand() % 3);

        /* generate a random number */
        item = rand();

        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        /* critical section */
        if (insert_item(item))
        {
            fprintf(stderr, "buffer is full, producer %d 's insertion failed!\n", index);
            exit(1);
        }
        else
            printf("producer %d produced %d \n", index, item);
        /* critical section */
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_exit(0);
}
```

2.5 consumer

consumer的代码框架在课后project的介绍部分已经给出，我们也只需要将其完善即可

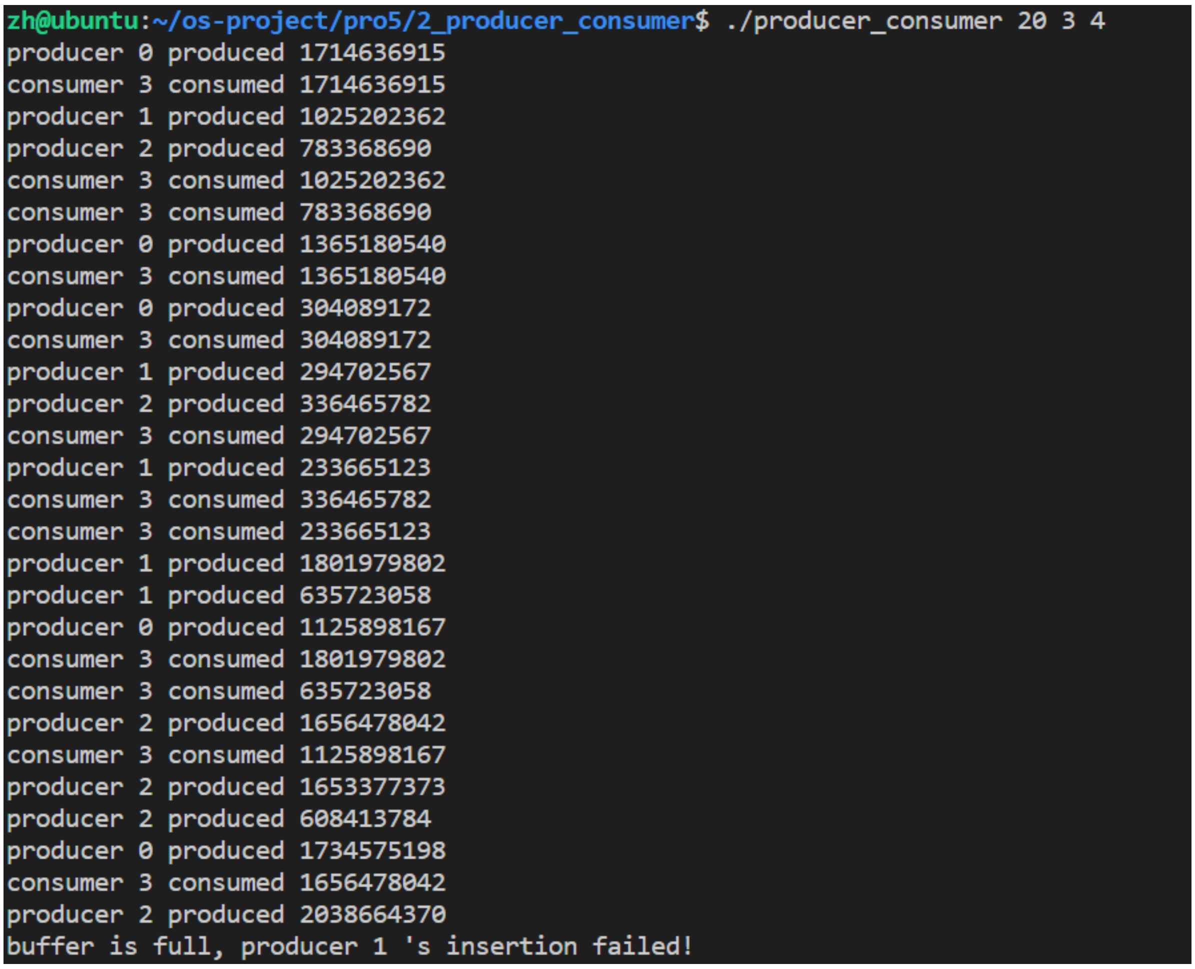
- 随机数的生成直接调用rand()函数即可，为了保证等待时间不过长，我们sleep的随机时间对3取模
- critical section的部分需要在信号量和锁的保护下执行，具体方式与课内教授的一致，信号量在外层，mutex在内层
- 进入critical section前需要确保此时buffer不为空，需要wait一下full信号量，若full不为0则表示buffer内有数据，不为空，可直接取出一个item，并将full减一，否则等待直到full不为0；离开critical section前需要post一下empty信号量，表示buffer中数据量减一
- 检查从buffer中取出数据是否成功，不成功输出错误信息并退出线程，否则输出本次consume的信息（线程index和buffer-item）

具体代码如下：

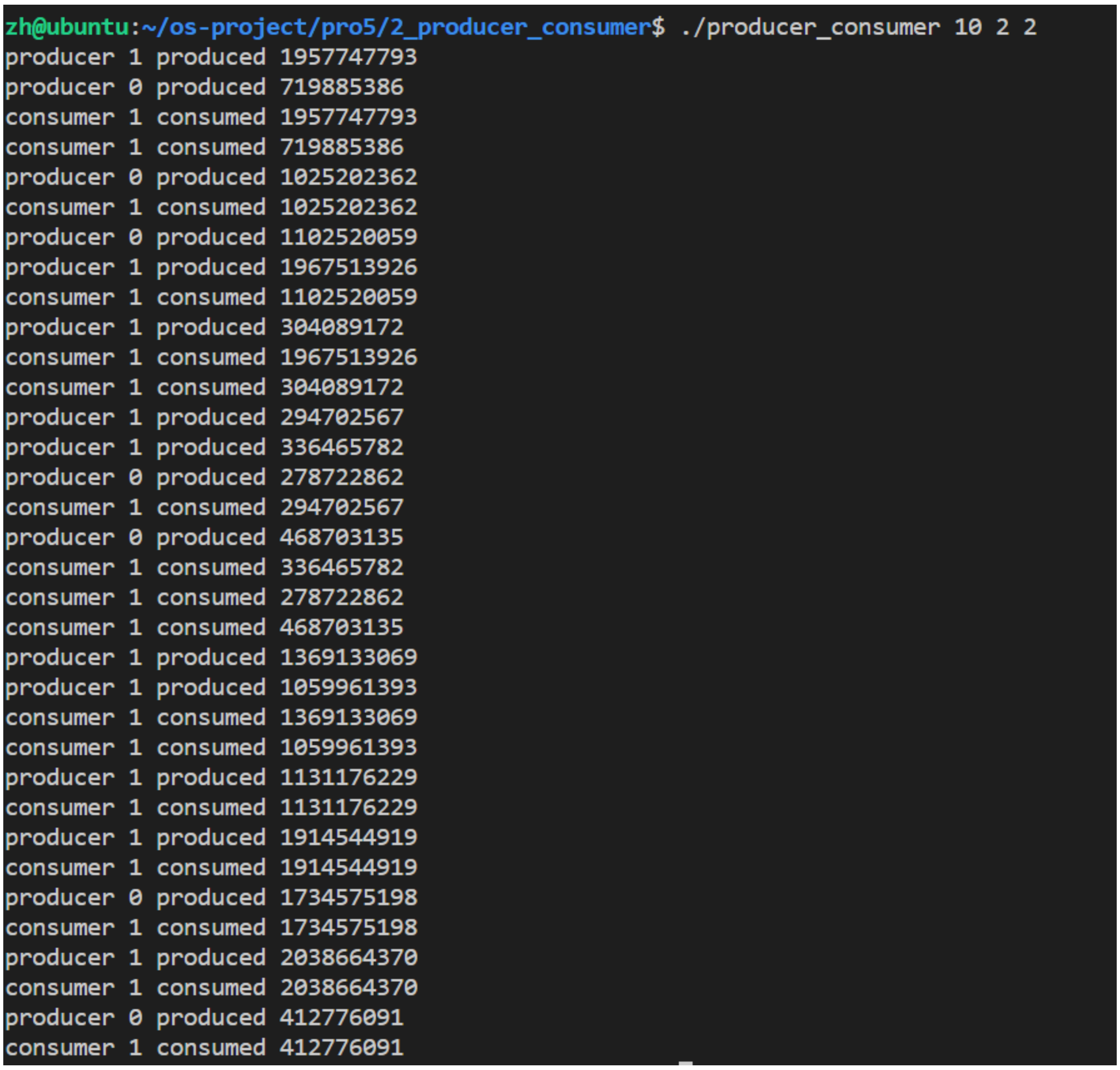
```c
// consumer thread
void *consumer(void *param)
{
    buffer_item item;
    int index = *(int *)param;
    while (TRUE)
    {
        /* sleep for a random period of time */
        sleep(rand() % 3);

        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        /* critical section */
        if (remove_item(&item))
        {
            fprintf(stderr, "buffer is empty, consumer %d 's deletion failed!\n", index);
            exit(1);
        }
        else
            printf("consumer %d consumed %d \n", index, item);
        /* critical section */
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    pthread_exit(0);
}
```

2.6 运行结果

我们类比Proj1 中的方式编写Makefile，对几个文件进行编译，运行后结果如下：



```
zh@ubuntu:~/os-project/pro5/2_producer_consumer$ ./producer_consumer 20 3 4
producer 0 produced 1714636915
consumer 3 consumed 1714636915
producer 1 produced 1025202362
producer 2 produced 783368690
consumer 3 consumed 1025202362
consumer 3 consumed 783368690
producer 0 produced 1365180540
consumer 3 consumed 1365180540
producer 0 produced 304089172
consumer 3 consumed 304089172
producer 1 produced 294702567
producer 2 produced 336465782
consumer 3 consumed 294702567
producer 1 produced 233665123
consumer 3 consumed 336465782
consumer 3 consumed 233665123
producer 1 produced 1801979802
producer 1 produced 635723058
producer 0 produced 1125898167
consumer 3 consumed 1801979802
consumer 3 consumed 635723058
producer 2 produced 1656478042
consumer 3 consumed 1125898167
producer 2 produced 1653377373
producer 2 produced 608413784
producer 0 produced 1734575198
consumer 3 consumed 1656478042
producer 2 produced 2038664370
buffer is full, producer 1 's insertion failed!
```



```
zh@ubuntu:~/os-project/pro5/2_producer_consumer$ ./producer_consumer 10 2 2
producer 1 produced 1957747793
producer 0 produced 719885386
consumer 1 consumed 1957747793
consumer 1 consumed 719885386
producer 0 produced 1025202362
consumer 1 consumed 1025202362
producer 0 produced 1102520059
producer 1 produced 1967513926
consumer 1 consumed 1102520059
producer 1 produced 304089172
consumer 1 consumed 1967513926
consumer 1 consumed 304089172
producer 1 produced 294702567
producer 1 produced 336465782
producer 0 produced 278722862
consumer 1 consumed 294702567
producer 0 produced 468703135
consumer 1 consumed 336465782
consumer 1 consumed 278722862
consumer 1 consumed 468703135
producer 1 produced 1369133069
producer 1 produced 1059961393
consumer 1 consumed 1369133069
consumer 1 consumed 1059961393
producer 1 produced 1131176229
consumer 1 consumed 1131176229
producer 1 produced 1914544919
consumer 1 consumed 1914544919
producer 0 produced 1734575198
consumer 1 consumed 1734575198
producer 1 produced 2038664370
consumer 1 consumed 2038664370
producer 0 produced 412776091
consumer 1 consumed 412776091
```

可以看到，程序成功运行，且由于设计的buffer容量较小（仅为5）和随机时间（0~2s）较短，当进程数目设置的较多时就很可能出现buffer已满无法写入的情况，这也与我们的预期相符合

2.7 完整代码

考虑到代码篇幅较长，完整代码详见附录

3 Difficulty& Summary

3.1 difficulty

依旧不太习惯多线程编程，因为一开始的bug比较多，尤其是Project 1中，pool_init中的线程创建部分，我一开始是放在函数的前半部分，但是后续发现这样始终会导致 Segmentation fault (core dumped) 的错误，这一般是数组溢出或指针无效引用等导致的错误，所以我一开始找不到到底是哪个地方出问题了，定位出错的位置花了较长时间，后来才想明白是因为此时mutex和sem未得到初始化而子线程中就对其进行了引用导致的。

3.2 summary

本次通过模拟线程池和生产者消费者问题，在之前project的基础上，对Pthread线程库有了更深的理解，同时在代码中使用mutex和semaphore也使得我对其理解更加具体深入，是对课内理论知识的良好实践，收获颇丰

4 appendix

4.1 Proj 1 Designing a Thread Pool

4.1.1 client.c

```c
/**
 * Example client program that uses thread pool.
 */

#include <stdio.h>
#include <unistd.h>
#include "threadpool.h"

struct data
{
    int a;
    int b;
};

void add(void *param)
{
    struct data *temp;
    temp = (struct data *)param;

    printf("I add two values %d and %d result = %d\n", temp->a, temp->b, 
temp->a + temp->b);
}

int main(void)
{
    // create some work to do
    struct data work[30];
    for (int i = 0; i < 30; i++)
    {
        work[i].a = i;
        work[i].b = 30 - i;
    }

    int num;
    printf("please input the number of threads:     ");
    scanf("%d",&num);
    // initialize the thread pool
    pool_init(num);

    // submit the work to the queue
    for (int i = 0; i < 30; i++)
    {
        pool_submit(&add, &work[i]);
    }

    // may be helpful
    sleep(1);

    pool_shutdown();

    return 0;
}
```

4.1.2 threadpool.c

```
/**
 * Implementation of thread pool.
 */

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include "threadpool.h"

#define TRUE 1

// this represents work that has to be
// completed by a thread in the pool
typedef struct
{
    void (*function)(void *p);
    void *data;
} task;

// the work queue
struct task_node
{
    task tsk;
    struct task_node *next;
};
struct task_node *head, *tail;

pthread_mutex_t mutex;
sem_t *sem;

int num_of_threads;
int *thread_index;
int shutdown;

// the worker bee
pthread_t *bee;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
```

```c
{
    struct task_node *tmp = (struct task_node *)malloc(sizeof(struct
task_node));
    if (tmp == NULL)
    {
        printf("malloc error and enqueue failed!\n");
        return 1;
    }

    tmp->tsk = t;
    tmp->next = NULL;
    if (tail == NULL) // first task
    {
        head = tmp;
        tail = tmp;
    }
    else // others
    {
        tail->next = tmp;
        tail = tail->next;
    }
    // printf("enqueue seccessful\n");
    return 0;
}

// remove a task from the queue
task dequeue()
{
    if (head == NULL)
    {
        printf("waiting queue empty and dequeue failed!\n");
        return;
    }
    task worktodo = head->tsk;
    struct task_node *tmp = head;
    head = head->next;
    if (head == NULL)
        tail = NULL;
    free(tmp);
    // printf("dequeue seccessful\n");
    return worktodo;
}

// the worker thread in the thread pool
void *worker(void *param)
{
    while (TRUE)
    {
        sem_wait(sem); // decrease the semaphore
        if (shutdown)
            break;

        pthread_mutex_lock(&mutex);
        task worktodo = dequeue();
        pthread_mutex_unlock(&mutex);

        // execute the task
        printf("In thread %d:   ", *((int *)param));
```

```c
            execute(worktodo.function, worktodo.data);
        }

        pthread_exit(0);
}

/**
 * Executes the task provided to the thread pool
 */
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}

/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;

    int res;
    pthread_mutex_lock(&mutex);
    res = enqueue(worktodo);
    pthread_mutex_unlock(&mutex);

    if (res == 1)
    {
        printf("task submission to pool failed!\n");
        return 1;
    }
    else
    {
        sem_post(sem); // increase the semaphore
        return 0;
    }
}

// initialize the thread pool
void pool_init(int num)
{
    num_of_threads = num;
    bee = (pthread_t *)malloc(sizeof(pthread_t) * num_of_threads);
    thread_index = (int *)malloc(sizeof(int) * num_of_threads);

    head = tail = NULL;
    shutdown = 0;

    // initialize mutual-exclusion locks and semaphores
    int ret = pthread_mutex_init(&mutex, NULL);
    if (ret)
    {
        printf("mutex initialization failed!\n");
        return;
    }
    sem = sem_open("QUEUE_SEM", O_CREAT, 0666, 0);
    if (sem == SEM_FAILED)
    {
        printf("semaphore initialization failed!\n");
        return;
    }

    // create the threads
    for (int i = 0; i < num_of_threads; i++)
    {
        thread_index[i] = i;
        ret = pthread_create(&bee[i], NULL, worker, &thread_index[i]);
    }
}

// shutdown the thread pool
void pool_shutdown(void)
{
    shutdown = 1;

    // release sem so that threads can exit
    for (int i = 0; i < num_of_threads; i++)
    {
        sem_post(sem);
    }

    //  join the threads
    for (int i = 0; i < num_of_threads; i++)
    {
        pthread_join(bee[i], NULL);
    }

    // destroy mutex and semaphore
    pthread_mutex_destroy(&mutex);
    sem_destroy(sem);

    free(bee);
    free(thread_index);
}
```

4.1.3 threadpool.h

```c
// function prototypes
void execute(void (*somefunction)(void *p), void *p);
int pool_submit(void (*somefunction)(void *p), void *p);
void *worker(void *param);
void pool_init(int num);
void pool_shutdown(void);
```

4.2 Proj 2 Producer-Consumer Problem

4.2.1 producer_consumer.c

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

#define TRUE 1

sem_t empty, full;
pthread_mutex_t mutex;

// producer thread
void *producer(void *param)
{
    buffer_item item;
    int index = *(int *)param;
    while (TRUE)
    {
        /* sleep for a random period of time */
        sleep(rand() % 3);

        /* generate a random number */
        item = rand();

        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        /* critical section */
        if (insert_item(item))
        {
            fprintf(stderr, "buffer is full, producer %d 's insertion failed!\n", index);
            exit(1);
        }
        else
            printf("producer %d produced %d \n", index, item);
        /* critical section */
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_exit(0);
}

// consumer thread
void *consumer(void *param)
{
    buffer_item item;
    int index = *(int *)param;
    while (TRUE)
    {
        /* sleep for a random period of time */
        sleep(rand() % 3);

        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        /* critical section */
        if (remove_item(&item))
        {
            fprintf(stderr, "buffer is empty, consumer %d 's deletion failed!\n", index);
            exit(1);
        }
        else
            printf("consumer %d consumed %d \n", index, item);
        /* critical section */
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    if (argc != 4)
    {
        printf("Error: invalid arguments.\n");
        return 1;
    }
    int time = atoi(argv[1]);
    int num_of_producers = atoi(argv[2]);
    int num_of_consumers = atoi(argv[3]);

    // buffer initialization
    buffer_init();

    // mutex and semaphore initialization
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_t *producers = (pthread_t *)malloc(sizeof(pthread_t) * num_of_producers);
    pthread_t *consumers = (pthread_t *)malloc(sizeof(pthread_t) * num_of_consumers);
    for (int i = 0; i < num_of_producers; ++i)
    {
        int producer_index = i;
        pthread_create(&producers[i], NULL, &producer, &producer_index);
    }
    for (int i = 0; i < num_of_consumers; ++i)
    {
        int consumer_index = i;
        pthread_create(&consumers[i], NULL, &consumer, &consumer_index);
    }

    //sleep before termination
    sleep(time);

    // termination
    for (int i = 0; i < num_of_producers; ++i)
        pthread_cancel(producers[i]);
    for (int i = 0; i < num_of_consumers; ++i)
        pthread_cancel(consumers[i]);

    // destroy mutex and semaphore
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    free(producers);
    free(consumers);
    return 0;
}
```

4.2.2 buffer.c

```c
//#include <stdlib.h>
#include "buffer.h"

// circular queue implemented with array
buffer_item buf[BUFFER_SIZE + 1];
int front, rear;

// insert an item to the buffer
int insert_item(buffer_item item)
{
    if ((rear + 1) % (BUFFER_SIZE + 1) == front)
    {
        return -1;
    }
    rear = (rear + 1) % (BUFFER_SIZE + 1);
    buf[rear] = item;
    return 0;
}

// remove an item from the buffer
int remove_item(buffer_item *item)
{
    if (front == rear)
    {
        return -1;
    }
    front = (front + 1) % (BUFFER_SIZE + 1);
    *item = buf[front];
    return 0;
}

void buffer_init()
{
    front = 0;
    rear = 0;
}
```

4.2.3 buffer.h

```c
# define BUFFER_SIZE 5

typedef int buffer_item;

int insert_item(buffer_item item);
int remove_item(buffer_item *item);
void buffer_init();
```