Report for OS Project 5

Designing a Thread Pool & Producer-Consumer Problem

陈文迪 519021910071

I. 实验任务

1. 了解 `Pthreads` 中创建和管理线程的相关接口。
2. 了解 `POSIX` 中的互斥锁和信号量以用于线程同步。
3. 实现一个线程池，该线程池具有如下的功能和架构。
    - 任务通过一个队列被提交至线程池
    - 当有空闲线程时，分配一个空闲线程给在队列中等待的任务
    - 当没有空闲线程时，待分配的任务在队列中等待
4. 设计一种解决方案来解决生产者-消费者问题。要求：
    - 使用标准的计数信号量实现 `empty` 和 `full`
    - 使用互斥锁实现 `lock`

II. 实验思路

1. Pthreads

在之前的实验中，我们已经解除了如 `pthread_create` 和 `pthread_join` 这类基本函数。在本次实验中，我们还需要用到一个关键函数 `pthread_cancel` 来进行线程撤销。这类线程撤销属于延迟撤销，因此我们需要在线程中添加一个撤销点。而在线程中我们会使用到的信号量操作 `sem_wait()` 即可作为一个合适的撤销点。

2. POSIX 同步

POSIX中我们可以这样使用互斥锁。

```
pthread_mutex_t mutex;
/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX中我们可以这样使用计数信号量。

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

3. Designing a Thread Pool

设计思路

由于我们的等待队列是有限的，这让我们很快就想到了可以将线程池中的队列转化成一个生产者-消费者问题。然而我们不难发现，该应用场景与标准的生产者-消费者问题是不同的。在此项目中，消费者便是我们最初创建的几个有限的线程，当线程发现等待队列中已经没有需要完成的任务时，线程应该持续等待`sem_wait(&sem)`；而生存者则是用户使用`pool_submit`函数提交新的任务，此时若等待队列已满，我们不应该像标准问题中一样持续等待直到有队列有空间，而是应该直接向用户发出信息表示提交失败。基于这样的思想，我们可以只使用`full`信号量来保证每个线程不会从空队列中抽取任务执行，另一个互斥锁`mutex`用于保证对队列的操作是互斥的，而避免队满的判断则放入临界区，不再使用`empty`信号量。

核心代码如下：

```
pthread_mutex_t mutex;
sem_t full;
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    pthread_mutex_lock(&mutex);
    int flag = 0;
    if(head == (rear+1)%QUEUE_SIZE) flag = 1;
    else{
        queue[rear] = t;
        rear = (rear+1)%QUEUE_SIZE;
    }
    pthread_mutex_unlock(&mutex);
    if(flag==0)sem_post(&full);
    return flag;
}
// remove a task from the queue
task dequeue()
{
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    task work = queue[head];
    head = (head+1)%QUEUE_SIZE;
    pthread_mutex_unlock(&mutex);
    return work;
}
```

项目中的其他细节

为了实现一个线程池库，我们需要为用户提供一系列的操作接口。

在创建线程池时，我们需要使用`pool_init`函数对队列、互斥锁、信号量进行初始化，并创建一定数量的线程以供使用。线程创建的同时，我们需要指定一个工作函数，工作函数的功能非常直观，它不断地从队列中获取任务并执行或等待新任务的到来。

```
// initialize the thread pool
void pool_init(void)
{
    head = 0; rear = 0;
    pthread_mutex_init(&mutex,NULL);
    sem_init(&full, 0, 0);
```

```
    //sem_init(&empty, 0, QUEUE_SIZE);
    for(int i = 0;i<NUMBER_OF_THREADS;++i){
        pthread_create(&(bee[i]),NULL,worker,NULL);
    }
}
// the worker thread in the thread pool
void *worker(void *param)
{
    // execute the task
    do{
        task work = dequeue();
        execute(work.function,work.data);
    }while(TRUE);

    pthread_exit(0);
}
/**
 * Executes the task provided to the thread pool
 */
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}
```

我们提供一个接口`pool_submit`供用户提交新的工作，并告知用户是否提交成功。

```
/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{

    worktodo.function = somefunction;
    worktodo.data = p;
    int state = enqueue(worktodo);
    if(state==0){
        printf("Submission succeeds.\n");
    }
    else{
        printf("Submission fails.\n");
    }
    return state;
}
```

当我们关闭线程池时候，我们提供`pool_shutdown`来撤销线程并归并线程。

```
// shutdown the thread pool
void pool_shutdown(void)
{
    for(int i = 0;i<NUMBER_OF_THREADS;++i){
        pthread_cancel(bee[i]);
        pthread_join(bee[i],NULL);
    }
}
```

4. Producer-Consumer Problem

本问题是标准的生产者-消费者问题，我们可以仿照前一个项目的思路。不同的是，这次我们使用标准的结构，采用两个计数信号量和一个互斥锁。接着，我们按照用户需求创建相应数量的线程，等待指定时间后撤销全部线程并归并线程即可。

为了更好地进行模拟，我们让生产者和消费者随机睡眠一段时间，再进行插入或删除的操作。

核心代码如下：

```
#define true 1
#define MAX 100

pthread_mutex_t mutex;
sem_t full;
sem_t empty;

buffer_item buffer[BUFFER_SIZE];
pthread_t bee[MAX];

int head,rear;

int insert_item(buffer_item item){
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    buffer[rear] = item;
    rear = (rear+1)%BUFFER_SIZE;
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    return 0;
}

int remove_item(buffer_item *item){
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    *item = buffer[head];
    head = (head+1)%BUFFER_SIZE;
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    return 0;
}

void *producer(void *param){
    buffer_item item;
    while(true){
        sleep(rand()%3);
        item = rand()%100;
        if(insert_item(item))
            fprintf(stdin,"report error condition");
        else
            printf("producer produced %d\n",item);
    }
}

void *consumer(void *param){
    buffer_item item;
    while(true){
        sleep(rand()%3);
```

```
        if(remove_item(&item))
            fprintf(stdin,"report error condition");
        else
            printf("consumer consumed %d\n",item);
    }
}

int main(int argc, char *argv[]){
    // parser
    if(argc!=4) return -1;
    int sleep_time = atoi(argv[1]);
    int num_of_producer = atoi(argv[2]);
    int num_of_consumer = atoi(argv[3]);
    if(num_of_consumer+num_of_producer>MAX) return -1;

    //initialize
    head = 0;rear = 0;
    pthread_mutex_init(&mutex,NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    //create threads
    int i = 0;
    for(;i<num_of_producer;i++){
        pthread_create(&(bee[i]),NULL,producer,NULL);
    }
    for(;i<num_of_consumer+num_of_producer;i++){
        pthread_create(&(bee[i]),NULL,consumer,NULL);
    }

    //sleep
    sleep(sleep_time);

    // terminate threads
    for(i=0;i<num_of_consumer+num_of_producer;i++){
        pthread_cancel(bee[i]);
        pthread_join(bee[i],NULL);
    }
    return 0;
}
```

III. 实验过程

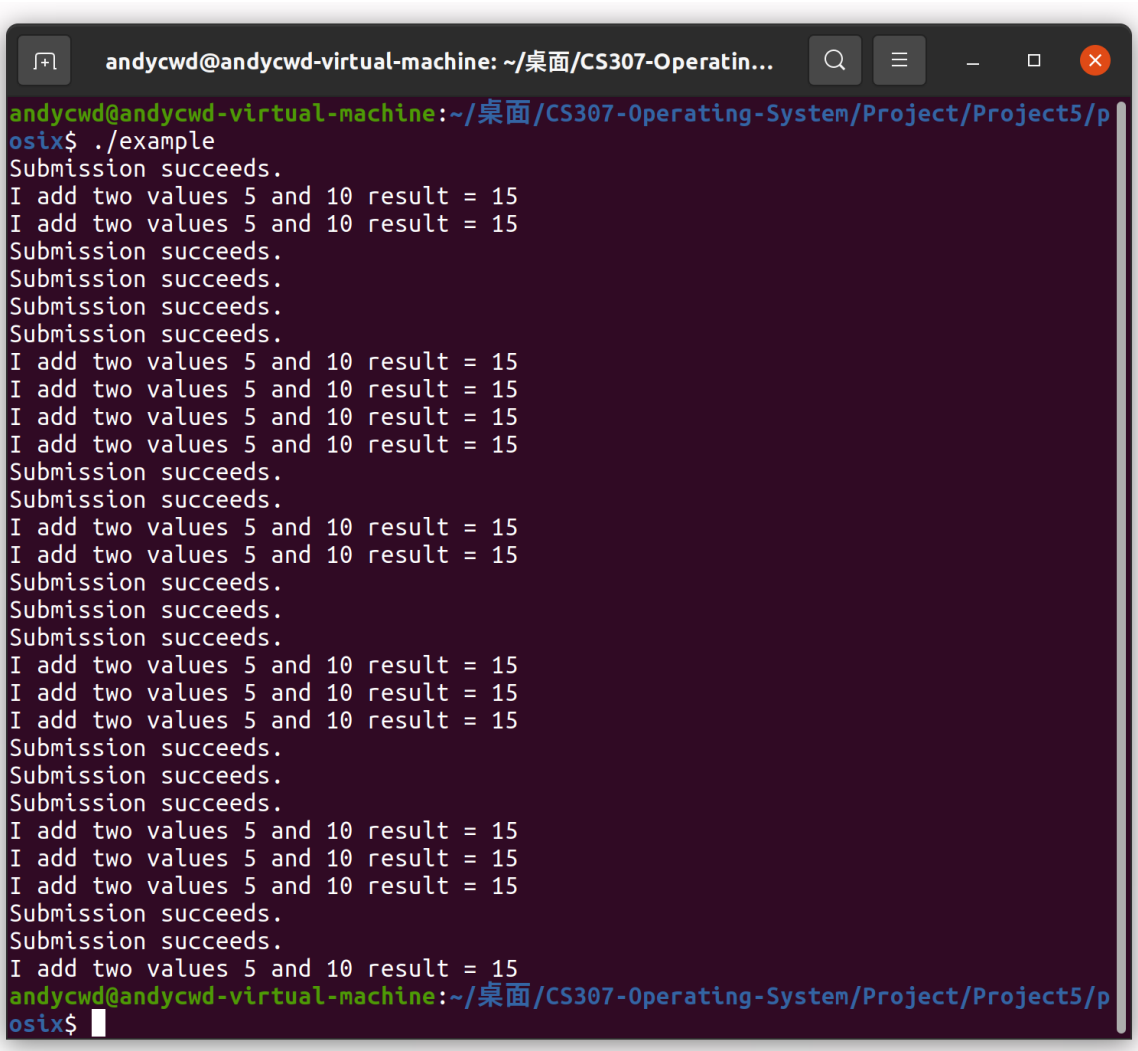
1. Designing a Thread Pool

我们将`client.c`中的`main`函数进行如下修改，共进行15次任务提交。

```
int main(void)
{
    // create some work to do
    struct data work;
    work.a = 5;
    work.b = 10;

    // initialize the thread pool
    pool_init();
```

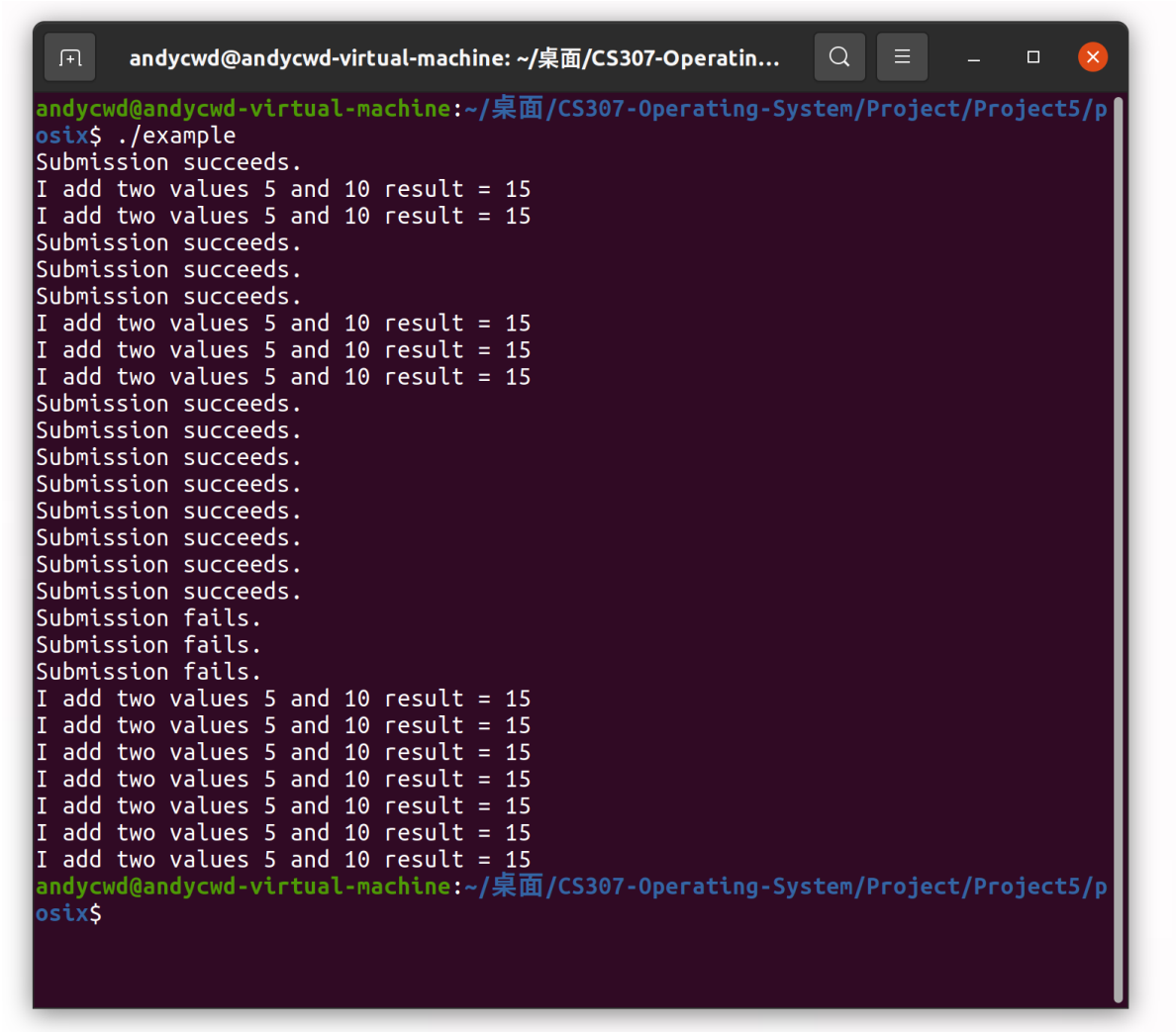
```
    // submit the work to the queue
    for(int i = 0;i<15;++i){
        pool_submit(&add,&work);
    }

    pool_shutdown();
    return 0;
}
```

接着我们先将等待队列的大小设置为10，线程数设置为3，进行实验，结果如下：



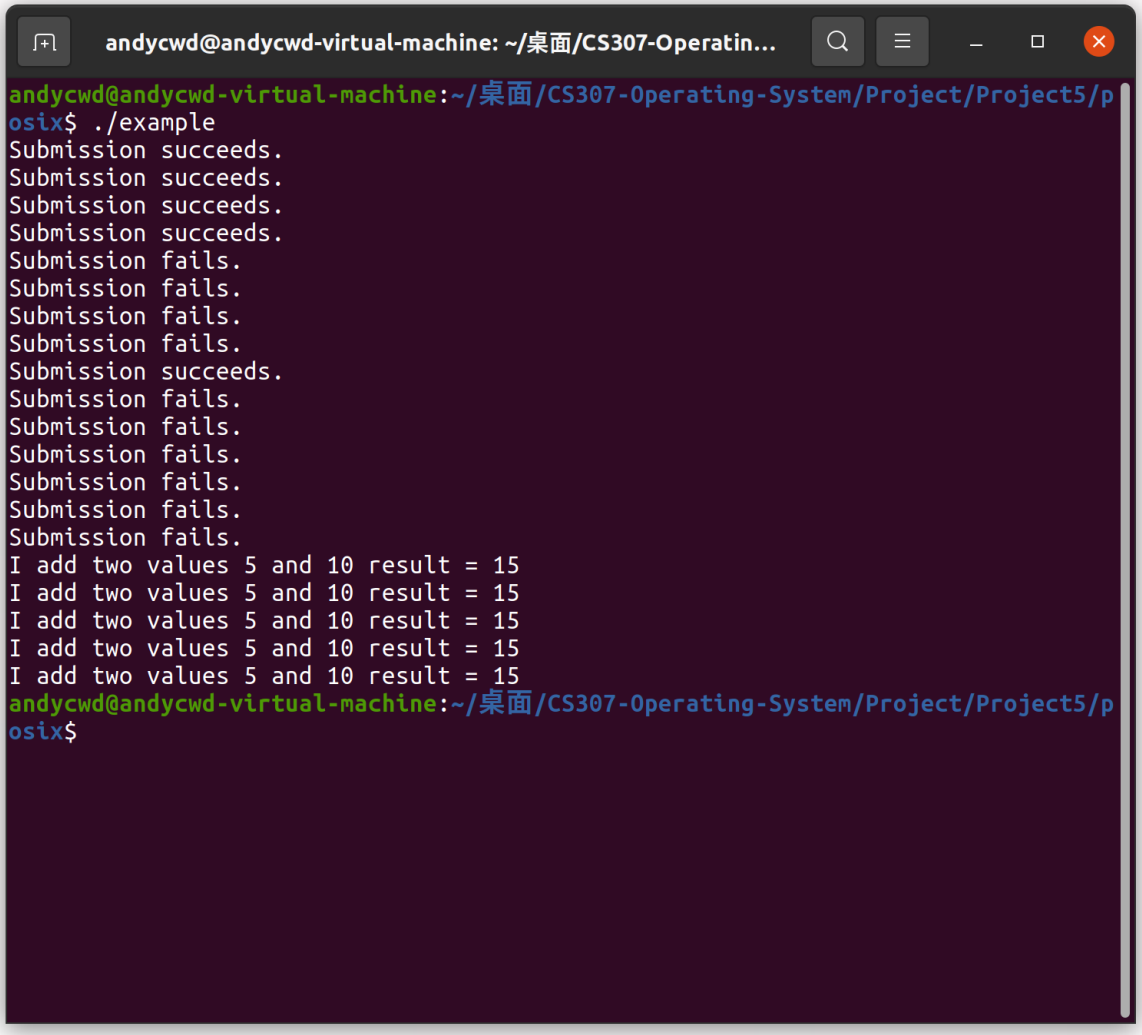
可以看到所有的提交都成功了。

然后，我们将等待队列的大小设置为5，线程数设置为3，再次进行实验，结果如下：



可以看到有部分提交失败了。

然后，我们将等待队列的大小设置为5，线程数设置为1，再次进行实验，结果如下：



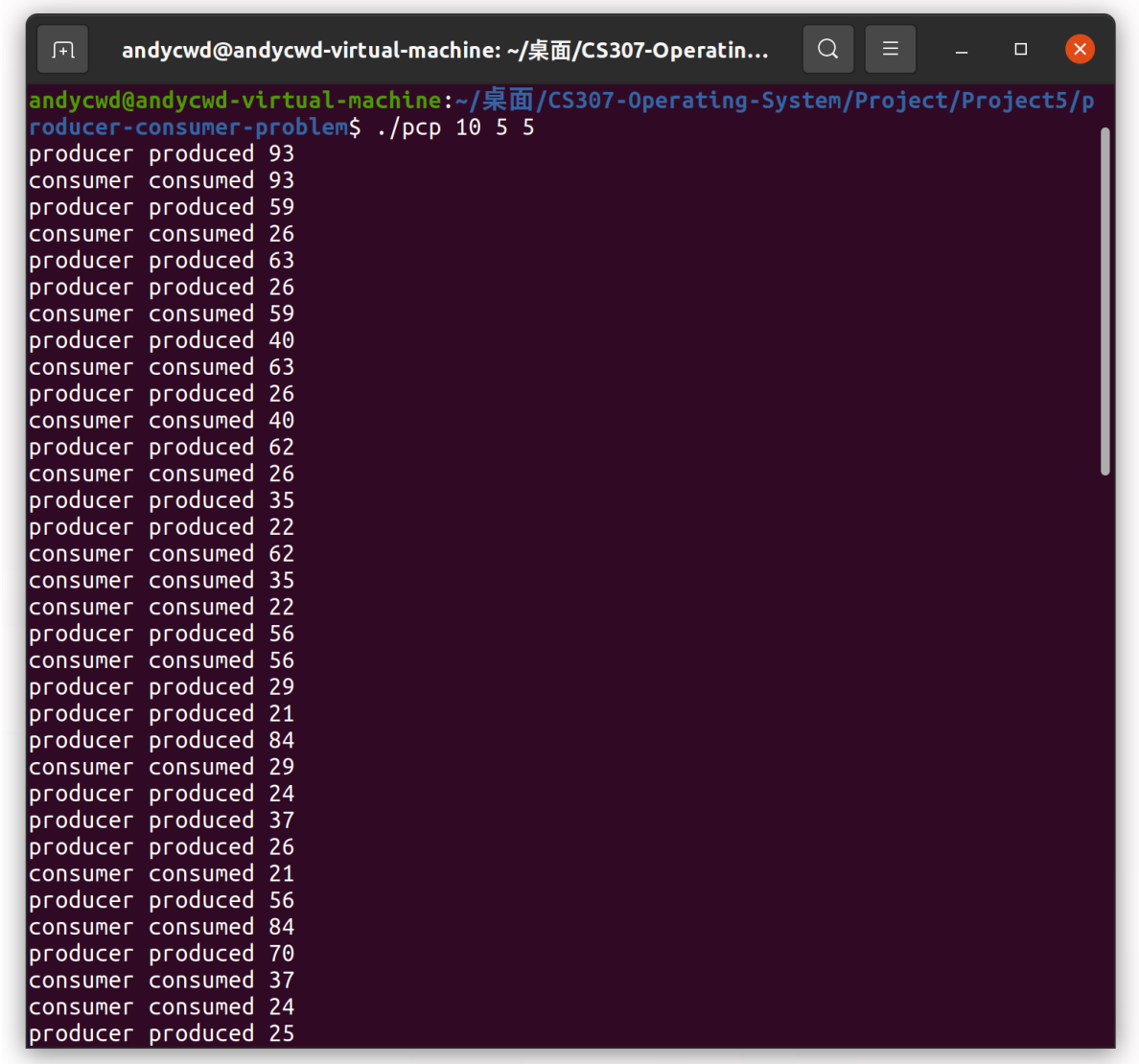
可以发现失败概率进一步增加。

因此，我们得出结论：减小线程池等待队列的大小或是减少可用线程的数量都会增大任务提交的失败概率。

2. Producer-Consumer Problem

生产者-消费者问题的描述保证了我们不会出现项目1中提交失败的情况。

我们采用3s的测试时间，测试5个生产者、5个消费者的情况，测试结果如下（受限于截图大小，结果并不完整）：



符合我们的设计预期。

IV. 遇到的问题

如何合理设计线程池中的队列逻辑?

我们在课本中学习了几类经典的线程同步问题，但是现实中的线程同步问题不会和书本上所描述的完全一致。为了完成线程池的功能，我们应该先进行抽象，再结合实际需求进行分析。例如，本次实验中我们联想到了生产者-消费者问题，但此时的生产者的需求发生了变化，它不应该等待队列中产生空位，而是向用户返回失败的信息。

V. 参考资料

[1] Operating System Concept 10th Edition

[2] The Pthreads Library - Multithreaded Programming Guide (oracle.com)