

# Project 5--Designing a Thread Pool & Producer-Consumer Problem

陈思远 518021910567

## Project 5--Designing a Thread Pool & Producer-Consumer Problem

- 1.实验内容和目标
2. 设计线程池
  - 2.1 要求与实现
    - 2.1.1 线程池初始化
    - 2.1.2 任务提交
    - 2.1.3 关闭线程池
  - 2.2 结果
3. 实现生产者-消费者问题
  - 3.1 要求与实现
    - 3.1.1 Buffer操作
    - 3.1.2 生产者和消费者
    - 3.1.3 main函数
  - 3.2 结果
4. 总结与思考

## 1.实验内容和目标

1. 设计一个线程池，用POSIX或Java实现。
2. 实现生产者-消费者问题。
3. 理解并运用信号量、互斥锁实现线程之间的同步。

## 2. 设计线程池

### 2.1 要求与实现

这部分要求实现一个线程池。当一个任务被提交给线程池时，会被放入一个队列，如果当前有空闲的线程，空闲的线程就会从队列里取出任务并执行；如果没有空闲的线程，则等待。

留给用户的接口有 `pool_init()` , `pool_submit()` , `pool_shutdown()` 分别用于线程池的初始化、任务的提交和线程池的关闭。

#### 2.1.1 线程池初始化

线程初始化时，需要首先初始化互斥锁和信号量，并按照设定的线程数量新建线程。互斥锁 `mutex` 在之后修改队列时可以防止竞态的出现；信号量 `full` 则用于通知每个线程，队列中是否有待执行的任务，`full=0` 表示队列中没有待执行的任务，故初始化为0。

```
// initialize the thread pool
void pool_init(void)
{
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    for(int i=0; i<NUMBER_OF_THREADS; i++)
        pthread_create(&bee[i], NULL, worker, NULL);
}
```

### 2.1.2 任务提交

用户调用 `pool_submit()` 函数将任务提交给线程池。线程池负责将调用 `enqueue()` 函数，将该任务放入一个等待队列中，如果等待队列已满则 `return 1`，如果未滿并成功放入，则信号量 `full` 自增，表示待执行的任务多了一个。

```
/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task submit_task;
    submit_task.function = somefunction;
    submit_task.data = p;
    int error=enqueue(submit_task);
    if(!error)
        sem_post(&full);
    return error;
}
```

为了保障同一时间只有一个线程可以操作等待队列，入队函数 `enqueue()` 需要在进入时申请一个互斥锁，结束时释放调它，保证互斥性。

```
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    pthread_mutex_lock(&mutex);
    if(queue_num==QUEUE_SIZE){
        pthread_mutex_unlock(&mutex);
        return 1;
    }
    workqueue[queue_num]=t;
    queue_num++;
    pthread_mutex_unlock(&mutex);
    return 0;
}
```

每个线程的 `worker()` 函数负责从等待队列中接收任务并执行。如果信号量 `full>0`，说明等待队列中有等待执行的任务，那么将 `full` 自减，并调用 `dequeue()` 函数从等待队列里取出一个任务之后执行它。如果信号量 `full` 等于0，说明队列中没有等待执行的任务，那么 `sem_wait()` 函数将会在内部一直自旋，等待有任务被提交。

```
// the worker thread in the thread pool
void *worker(void *param)
{
    task worktodo;
    while(TRUE){
        sem_wait(&full);
        worktodo=dequeue();
        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);
}

```

类似地，为了保障同一时间只有一个线程可以操作等待队列，出队函数 `dequeue()` 也需要在进入时申请一个互斥锁，结束时释放调它，保证互斥性。

```

// remove a task from the queue
task dequeue()
{
    task worktodo;
    pthread_mutex_lock(&mutex);
    worktodo=workqueue[0];
    queue_num--;
    for(int i=0;i<queue_num;i++){
        workqueue[i]=workqueue[i+1];
    }
    pthread_mutex_unlock(&mutex);
    return worktodo;
}

```

### 2.1.3 关闭线程池

提交的所有任务执行完毕后，线程池可以通过 `pool_shutdown()` 函数关闭。由于每个线程的 `worker()` 函数都是采用 `while(true)` 一直循环的，所以需要先调用 `pthread_cancel()` 函数把这些线程终止，再调用 `pthread_join()`。最后，还需要要销毁信号量和互斥锁。

```

// 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);
    }
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
}

```

## 2.2 结果

为了测试线程池的正确性，我将 `client.c` 修改如下。首先生成一些任务，然后初始化线程池，再将这些任务提交给线程池。若等待队列已满，`pool_submit()` 函数就会返回1，这时，用户程序检测到这个情况，就会 `sleep` 三秒，等待队列里的一些任务完成，等待队列有空位之后再把这个任务提交给线程池。

```

#define WorkNum 15

struct data{
    int a;
    int b;
};

```

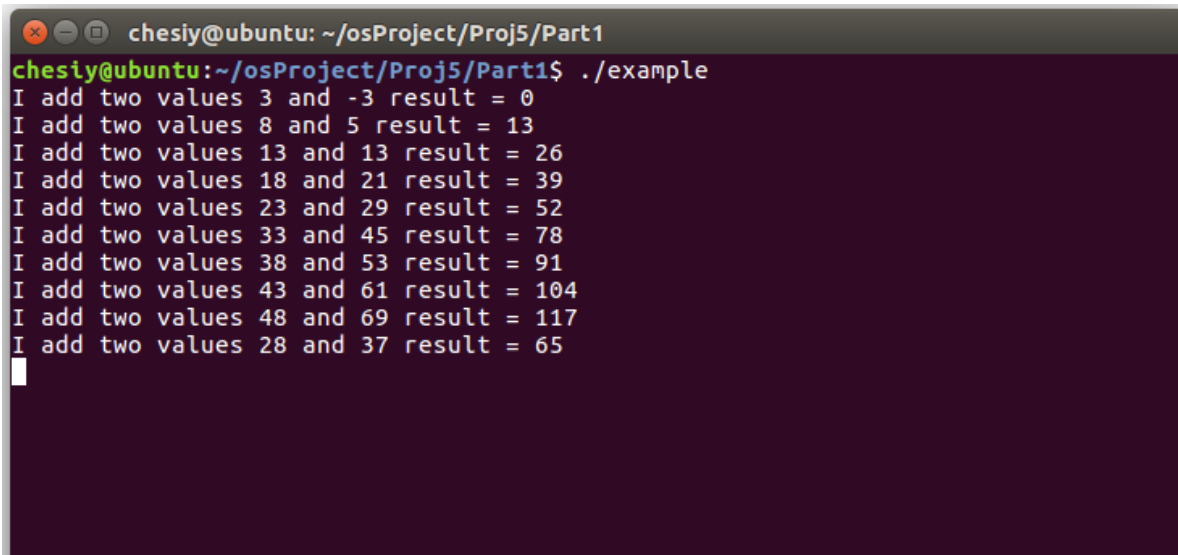
```

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[workNum];
    for(int i=0;i<workNum;i++){
        work[i].a=5*i+3;
        work[i].b=8*i-3;
    }
    // initialize the thread pool
    pool_init();
    // submit the work to the queue
    for(int i=0;i<workNum;i++){
        int err=pool_submit(&add,&work[i]);
        if(err){
            sleep(3);
            i--;
        }
    }
    pool_shutdown();
    return 0;
}

```

我的等待队列最大长度为10，这里设置了15个任务进行测试。可以发现，一开始10个任务很快被执行完，但由于等待队列已满，剩下的5个任务在3秒后被执行。运行截图如下。



```

chesiy@ubuntu: ~/osProject/Proj5/Part1
chesiy@ubuntu:~/osProject/Proj5/Part1$ ./example
I add two values 3 and -3 result = 0
I add two values 8 and 5 result = 13
I add two values 13 and 13 result = 26
I add two values 18 and 21 result = 39
I add two values 23 and 29 result = 52
I add two values 33 and 45 result = 78
I add two values 38 and 53 result = 91
I add two values 43 and 61 result = 104
I add two values 48 and 69 result = 117
I add two values 28 and 37 result = 65

```

```
chesiy@ubuntu: ~/osProject/Proj5/Part1
chesiy@ubuntu:~/osProject/Proj5/Part1$ ./example
I add two values 3 and -3 result = 0
I add two values 8 and 5 result = 13
I add two values 13 and 13 result = 26
I add two values 18 and 21 result = 39
I add two values 23 and 29 result = 52
I add two values 33 and 45 result = 78
I add two values 38 and 53 result = 91
I add two values 43 and 61 result = 104
I add two values 48 and 69 result = 117
I add two values 28 and 37 result = 65
I add two values 53 and 77 result = 130
I add two values 58 and 85 result = 143
I add two values 63 and 93 result = 156
I add two values 68 and 101 result = 169
I add two values 73 and 109 result = 182
chesiy@ubuntu:~/osProject/Proj5/Part1$
```

### 3. 实现生产者-消费者问题

#### 3.1 要求与实现

这部分要求实现生产者-消费者问题，课本中已有用binary semaphore和counting semaphore实现的版本，但实验中要求使用互斥锁和counting semaphore实现。

##### 3.1.1 Buffer操作

每个线程之间共享一个数组buffer，每次只有一个线程（生产者/消费者）可以对数组进行修改，因此可以用一个互斥锁实现。此外，如果buffer为空，那么消费者就不能进行；如果buffer为满，生产者就不能进行。因此需要两个信号量 `full` 和 `empty` 来确保这一点。`full` 表示满的buffer数量，`empty` 表示空的buffer数量，所以分别初始化为0和buffer的大小。

buffer是一个循环队列，所以要把头和尾都初始化为0。

```
void init_buffer(){
    pthread_mutex_init(&mutex,NULL);
    sem_init(&empty,0,BUFFER_SIZE);
    sem_init(&full,0,0);
    buffer_head=0;
    buffer_tail=0;
}
```

生产者会向buffer中插入元素，消费者会从buffer中清除元素，因此需要两个函数来实现这个功能。这两个函数的实现与普通的循环队列几乎相同，但必须用互斥锁保证只有一个线程可以操作buffer，用信号量保证buffer为空时消费者不能进行，buffer为满时，生产者不能进行。

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

int remove_item(buffer_item *item){
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
```

```

    *item=buffer[buffer_head];
    buffer_head=(buffer_head+1)%(BUFFER_SIZE+1);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    return 0;
}

```

运行完成后需要把信号量和互斥锁销毁。

```

void shutdown_buffer(){
    sem_destroy(&full);
    sem_destroy(&empty);
    pthread_mutex_destroy(&mutex);
}

```

### 3.1.2 生产者和消费者

生产者线程和消费者线程都需要生成一个随机数作为sleep的时间。生产者线程还需再生成一个随机数作为item插入buffer。分别调用 `insert_item()` 和 `remove_item()` 即可实现。

```

void *producer(void *param){
    buffer_item item;
    while(1){
        int st=rand()%4+1;
        sleep(st);
        item=rand();
        if(insert_item(item))
            printf("report error condition\n");
        else
            printf("producer produced %d\n",item);
    }
}

void * consumer(void *param){
    buffer_item item;
    while(1){
        int st=rand()%4+1;
        sleep(st);
        if(remove_item(&item)){
            printf("report error condition\n");
        }else{
            printf("consumer consumed %d\n",item);
        }
    }
}

```

### 3.1.3 main函数

在main函数中，首先解析命令行传入的参数，再初始化buffer，之后创建生产者和消费者线程，主函数所在的线程休眠一定时间后终止所有生产者/消费者线程，最后销毁信号量和互斥锁，退出程序。

```

int main(int argc, char *argv[]){
    //get command line arguments
    if(argc!=4){
        printf("error: wrong argument!\n");
    }
}

```

```

        return 1;
    }
    sleep_time=atoi(argv[1]);
    producer_num=atoi(argv[2]);
    consumer_num=atoi(argv[3]);

    //initialize buffer
    init_buffer();

    //Create producer threads
    for(int i=0;i<producer_num;i++){
        pthread_create(&producers[i],NULL,producer,NULL);
    }

    //Create consumer threads
    for(int i=0;i<consumer_num;i++){
        pthread_create(&consumers[i],NULL,consumer,NULL);
    }

    //Sleep
    printf("sleep %d seconds\n",sleep_time);
    sleep(sleep_time);

    //exit
    for(int i=0;i<producer_num;i++){
        pthread_cancel(producers[i]);
        pthread_join(producers[i],NULL);
    }
    for(int i=0;i<consumer_num;i++){
        pthread_cancel(consumers[i]);
        pthread_join(consumers[i],NULL);
    }
    shutdown_buffer();

    return 0;
}

```

### 3.2 结果

我测试了两组数据，结果如下图所示。可以发现，当消费者线程数量等于生产者线程数量时，几乎可以实现生产完立刻被消费；当生产者线程数大于消费者线程数时，会有不少item来不及被消费。

	睡眠时间	生产者线程数	消费者线程数
第一组	5	3	3
第二组	7	4	2

```
chesiy@ubuntu:~/osProject/Proj5/Part2$ gcc ProCon.c -pthread
chesiy@ubuntu:~/osProject/Proj5/Part2$ ./a.out 5 3 3
sleep 5 seconds
producer produced 719885386
consumer consumed 719885386
producer produced 1189641421
consumer consumed 1189641421
producer produced 783368690
consumer consumed 783368690
producer produced 1967513926
consumer consumed 1967513926
chesiy@ubuntu:~/osProject/Proj5/Part2$ ./a.out 7 4 2
sleep 7 seconds
producer produced 719885386
producer produced 596516649
consumer consumed 719885386
producer produced 1350490027
consumer consumed 596516649
producer produced 2044897763
producer produced 1365180540
producer produced 304089172
producer produced 35005211
consumer consumed 1350490027
producer produced 1726956429
```

#### 4. 总结与思考

在本次project中，我实现了线程池和生产者-消费者问题，这让我对进程同步、互斥锁、信号量等的理解更加深入，也让我学习了怎样在实践中使用这些函数，而不仅仅停留在书本上的伪代码。另外，完成这次的project还需要复习之前学过的线程创建、终止、连接和对应的函数，以及线程池的相关知识，是一个很好的温故而知新的过程。