PA2 实验报告

191300073 AI 杨斯凡 191300073@smail.nju.edu.cn

1. 归并排序:

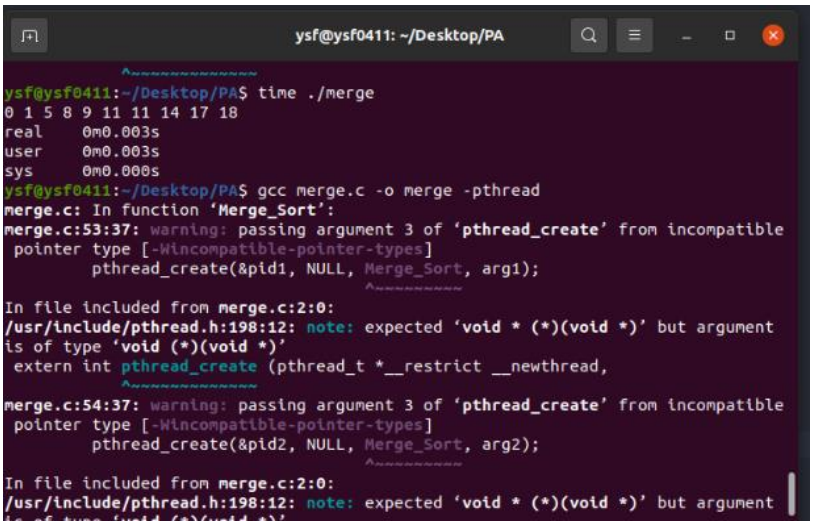
因为上个学期算法课我写过归并排序，我对上个学期的代码进行了微改

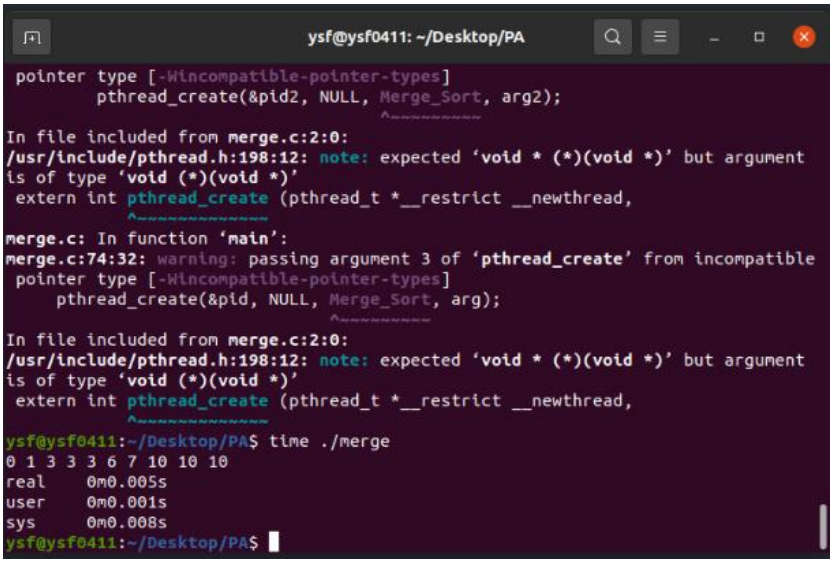
```
void Merge_Sort(void* arg)
{
    int *argu = (int*)arg;
    int left = argu[0];
    int right = argu[1];

    if (left < right)
    {
        pthread_t pid1;
        pthread_t pid2;

        int arg1[2];
        int arg2[2];

        int middle;
        middle = (left + (right - 1)) / 2;
        arg1[0] = left;
        arg1[1] = middle;
        arg2[0] = middle + 1;
        arg2[1] = right;

        pthread_create(&pid1, NULL, Merge_Sort, arg1);
        pthread_create(&pid2, NULL, Merge_Sort, arg2);

        pthread_join(pid1, NULL);
        pthread_join(pid2, NULL);

        merge(left, middle, right);
    }
}
```

将数组起始位置参数设置成数组传入，而进行 mergesort 的时候，不再递归调用，而是创建新的线程去执行，只需等待线程完成即可。

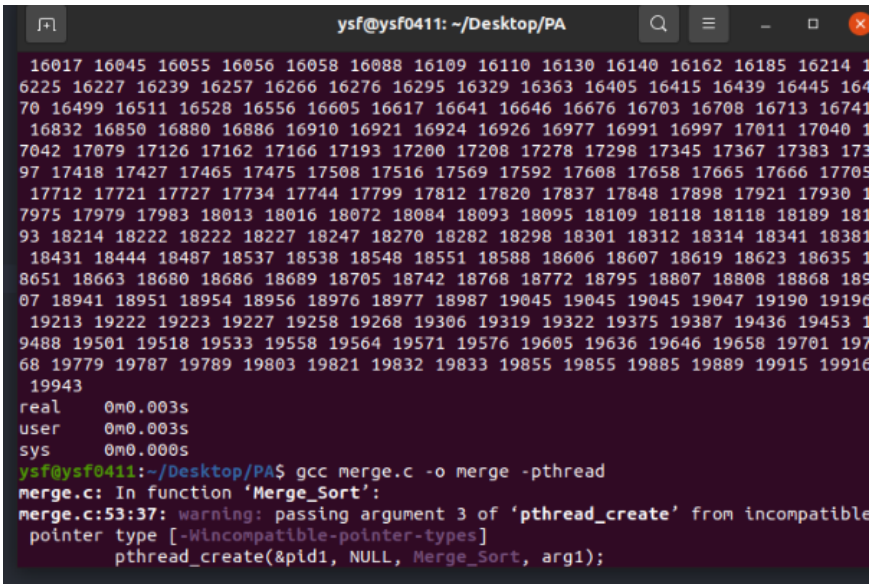
而为了比较时间,我在单线程的时候使用了递归调用.

可以看出，当数组数较少的时候，递归次数少，线程少，两者时间相差无几.

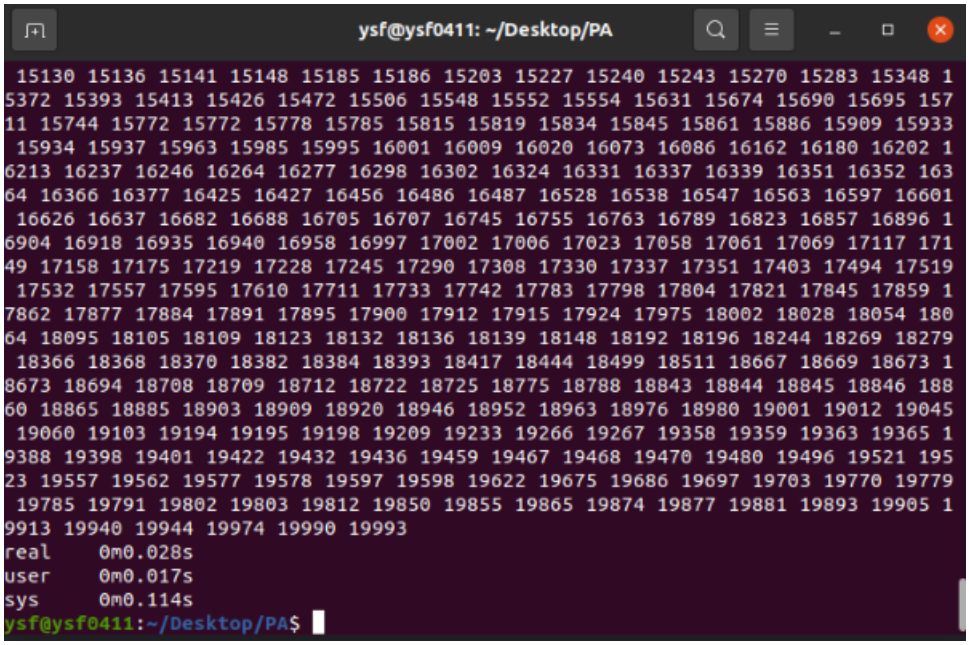




而当数组大小很大的时候，多线程在线程调度上就得花费很多时间



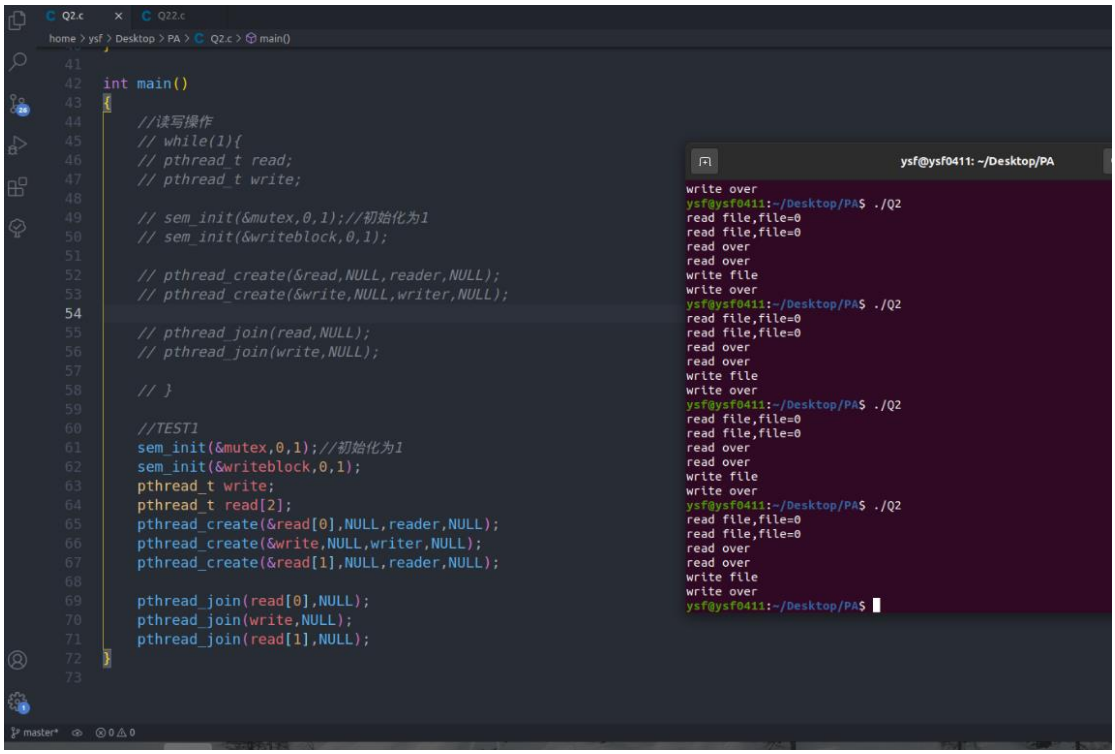
单线程



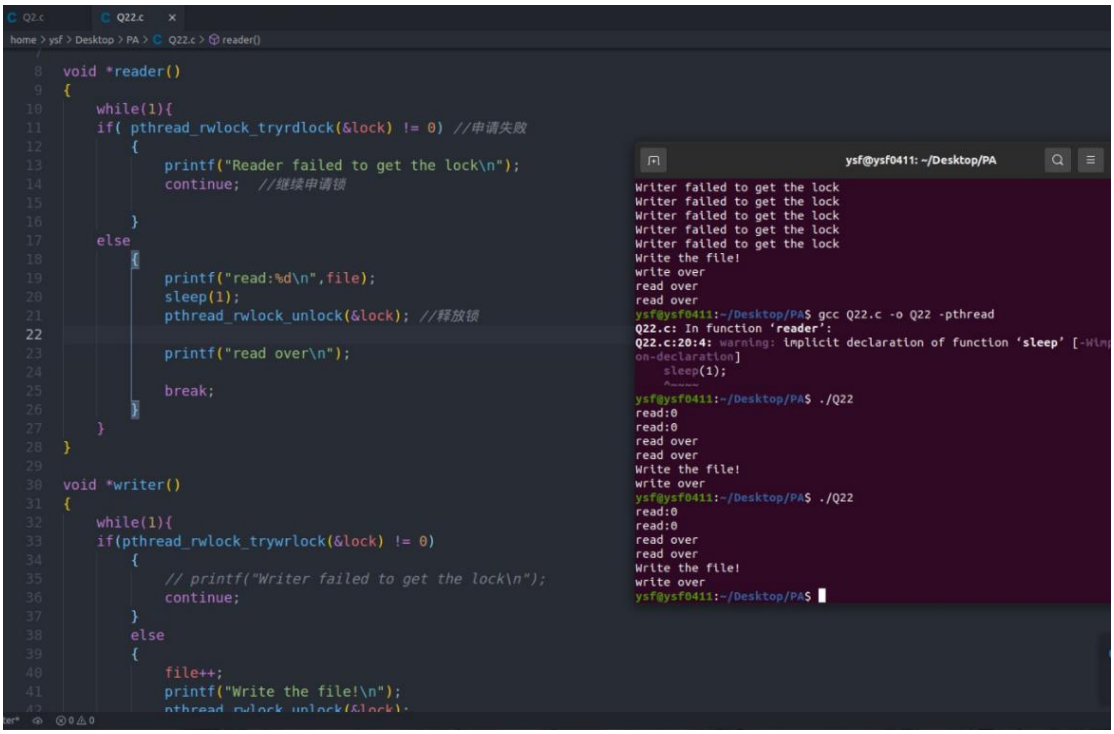
多线程

2. 对于读者写者，我按照课本上的进行 PV 操作，得以正常实现，而使用读写锁，更加容易得实现了读者写者问题。

而两者在同步上也有着区别。为了使得结果更加明显，我使用了 printf 和 sleep 来使得读写请求的间隔更大，最后结果反映两者都是读优先。



（PV 操作）



（读写锁）

3.

我定义了一个类，然后使用构造函数对缓冲区的大小初始化，然后初始化信号量，put, get 函数我对照课本上的读者写者问题实现，使用了 PV 操作来实现。而后面发现在多线程中无法调用类中的函数，我对类进行了修改。

```
class CircleBuffer
{
    public:
        void put(int value);
        int get(void);
        CircleBuffer(int k);//构造函数
        int getsize();
    private:
        int size;
        int in;
        int out;
        int *arr;
        sem_t empty,full,mutex;
};

CircleBuffer::CircleBuffer(int k)
{
    size=k;
    in=0;
    out=0;
    arr=new int[size];
    sem_init(&mutex,0,1);
    sem_init(&empty,0,k);
    sem_init(&full,0,0);
}
int CircleBuffer::getsize(){
    return size;
}
```

```
class CircleBuffer
{
    public:
        void put(int value);
        void get();
        CircleBuffer(int k);//构造函数
        int getsize();
        static void * thread_run1(void* tmp)
        {
            CircleBuffer *p = (CircleBuffer *)tmp;
            p->put(5);
        }
        static void * thread_run2(void* tmp)
        {
            CircleBuffer *p = (CircleBuffer *)tmp;
            p->get();
        }
    private:
        int size;
        int in;
        int out;
        int *arr;
        sem_t empty,full,mutex;
};
```

```
void CircleBuffer::put(int value){
    sem_wait(&empty);
    sem_wait(&mutex);
    arr[in]=value;
    printf("index= %d put %d\n",in,value);
    in=(in+1)%size;

    sem_post(&mutex);
    sem_post(&full);
}
void CircleBuffer::get(){
    int ret;
    sem_wait(&full);
    sem_wait(&mutex);
    ret=arr[out];
    printf("index= %d get %d\n",out,ret);
    out=(out+1)%size;

    sem_post(&mutex);
    sem_post(&empty);
}
```

（修改后的函数）

```
int main(){


    cout<<circlebuffer->getsize()<<"\n";

    for(int i=0;i<10;i++){
        circlebuffer->put(3-i);
        if(i==5) circlebuffer->get();
    }
    pthread_t producer[2];
    pthread_t customer[2];


    pthread_create(&customer[0],NULL,CircleBuffer::thread_r
    pthread_create(&producer[0],NULL,CircleBuffer::thread_r
    pthread_create(&customer[1],NULL,CircleBuffer::thread_r
    pthread_create(&producer[1],NULL,CircleBuffer::thread_r


    pthread_join(customer[0],NULL);
    pthread_join(producer[0],NULL);
    pthread_join(customer[1],NULL);
    pthread_join(producer[1],NULL);


}
```

```
ysf@ysf0411
File  Edit  View  Search  Terminal  Help
index= 1 get 2
haoye
index= 1 put 5
haoye
index= 2 get 1
ysf@ysf0411:~/Desktop$ g++ CircleBuffer.c
ysf@ysf0411:~/Desktop$ ./CircleBuffer
10
index= 0 put 3
index= 1 put 2
index= 2 put 1
index= 3 put 0
index= 4 put -1
index= 5 put -2
index= 0 get 3
index= 6 put -3
index= 7 put -4
index= 8 put -5
index= 9 put -6
index= 0 put 5
index= 1 get 2
index= 1 put 5
index= 2 get 1
ysf@ysf0411:~/Desktop$
```

（实验结果：可以正确同步）

4.死锁，我仿照了书上的哲学家问题，只不过我把人数降到了 2，并且每次对一个信号量进行 P 操作后，都 sleep 挂起当前线程，执行另一个线程，从而提高发生死锁概率，最后成功实现了死锁。

```
void *writer_1()
{
    while(1){
    sem_wait(&writeblock_1);
    sleep(2);
    sem_wait(&writeblock_2);
    printf("write file\n");
    file++;
    sem_post(&writeblock_1);
    sem_post(&writeblock_2);
    }
}

void *writer_2()
{
    while(1){
    sem_wait(&writeblock_2);
    sleep(2);
    sem_wait(&writeblock_1);
    printf("write file\n");
    file++;
    sem_post(&writeblock_2);
    sem_post(&writeblock_1);
    }
}
```