PA2 实验报告

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

1. 归并排序:

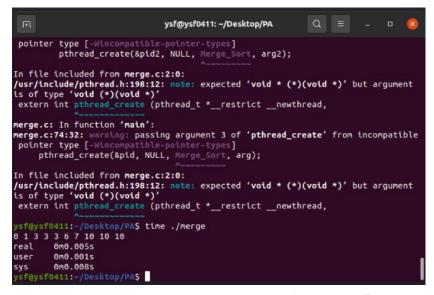
因为上个学期算法课我写过归并排序,我对上个学期的代码进行了微改

```
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];
        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。

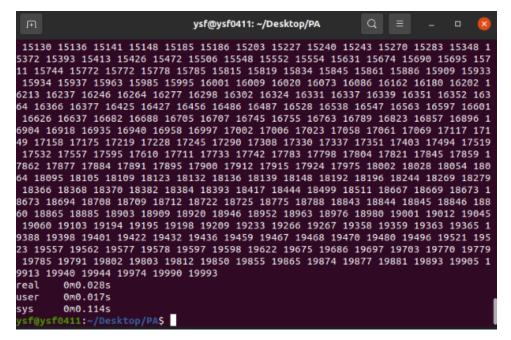
而为了比较时间,我在单线程的时候使用了递归调用.

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

```
ysf@ysf0411: ~/Desktop/PA
 16017 16045 16055 16056 16058 16088 16109 16110 16130 16140 16162 16185 16214
6225 16227 16239 16257 16266 16276 16295 16329 16363 16405 16415 16439 16445 16470 16499 16511 16528 16556 16605 16617 16641 16646 16676 16703 16708 16713 16743
16832 16850 16886 16986 16921 16924 16926 16977 16991 16997 17011 17040 7042 17079 17126 17162 17166 17193 17200 17208 17278 17298 17345 17367 17383 17
97 17418 17427 17465 17475 17508 17516 17569 17592 17608 17658 17665 17666 1770
17712 17721 17727 17734 17744 17799 17812 17820 17837 17848 17898 17921 17930 17975 17979 17983 18013 18016 18072 18084 18093 18095 18109 18118 18118 18189 18193 18214 18222 18222 18227 18247 18270 18282 18298 18301 18312 18314 18341 18381
 18431 18444 18487 18537 18538 18548 18551 18588 18606 18607 18619 18623 18635
8651 18663 18680 18686 18689 18705 18742 18768 18772 18795 18807 18808 18868 189
07 18941 18951 18954 18956 18976 18977 18987 19045 19045 19045 19047 19190 19190
19213 19222 19223 19227 19258 19268 19306 19319 19322 19375 19387 19436 19453 1
9488 19501 19518 19533 19558 19564 19571 19576 19605 19636 19646 19658 19701 197
68 19779 19787 19789 19803 19821 19832 19833 19855 19855 19885 19889 19915 19916
19943
real
            0m0.003s
            0m0.003s
user
            0m0.000s
sys
ysf@ysf0411:-/Desktop/PA$ gcc merge.c -o merge -pthread merge.c: In function 'Merge_Sort':
merge.c:53:37: warning: passing argument 3 of 'pthread_create' from incompatible
             type [-Wincompatible-pointer-types]
pthread_create(&pid1, NULL, Merge_Sort, arg1);
 pointer type [-Wincom
```



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(&mpty,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)

```
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_rivsfeysfo411:-/Desktop$ g++ CircleBuffer cysfeysfo411:-/Desktop$ ./CircleBuffer cysfeysfo411:-/
```

(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);
    }
}
```