PA2 实验报告

1.merge-sort的非多线程和多线程版本

(1)非多线程下的merge-sort仅需利用算法课所学内容，在`merge_sort`函数中递归调用即可，利用`sys/time.h`以及`time.h`的函数可以求得计算时间，编译运行时使用指令

```
gcc mergesorttest.c -o merge
./merge
```

(2)根据提示写出的多线程merge-sort中，需要在`main`函数与`merge_sort`两个函数中做出调整

```
int main() {

    // make a parent thread
    float time_use = 0;
    struct timeval start_;
    struct timeval end_;
    gettimeofday(&start_,NULL);
    pthread_t start_thread;
    b = (int*)malloc(sizeof(int)*SIZE);

    // start index
    index start;
    start.p = 0;
    start.r = SIZE-1;

    // print before the sort
    printf("Before Sort: ");
    for(int i = 0; i < SIZE; i++){
        printf("%i ",a[i]);
    }
    printf("\n");

    // create thread to merge sort
    pthread_create(&start_thread, NULL, merge_sort, &start);
    pthread_join(start_thread,NULL);

    // print after the sort
    printf("After Sort: ");
    for(int i = 0; i < SIZE; i++){
        printf("%i ",a[i]);
    }
    printf("\n");
    gettimeofday(&end_,NULL);
    time_use=(end_.tv_sec-start_.tv_sec)*1000000+(end_.tv_usec-start_.tv_usec);
    printf("%f\n",time_use);
    return 0;
}
```

```
    pthread_t thd1,thd2;
    // find the mid point of the array
    int mid = (pr->p + pr->r)/2;
```

```
    // indexs for each side of the array
    index side1,side2;
    // spliting the array indexs in 2
    side1.p = pr->p;
    side1.r = mid;
    side2.p = mid+1;
    side2.r = pr->r;

    // create thread to sort half of the array
    sort1 = pthread_create(&thd1, NULL, merge_sort,(void*)&side1);
    if(sort1 >0){
        printf("Failed to create new thread1\n");
    }

    // create thread to sort other half
    sort2 = pthread_create(&thd2, NULL, merge_sort,(void*)&side2);
    if(sort2 >0){
        printf("Failed to create new thread2\n");
    }

    // join the threads
    pthread_join(thd1, NULL);
    pthread_join(thd2, NULL);
```

在将两种实现方式的运行时间进行对比后发现，在要求排序的数字个数不多时，非多线程的算法运行速度更快，原因在于多线程算法需要消耗时间用于线程调度，在要求排序的数字个数较多时，多线程算法具有较为明显的优势

2.信号量与PV操作

本次采用的是写者优先的思路，也就是说当一个写者到来的时候，需要尽快对该文件进行写操作，通过设置五个信号量，`RWmutex`、`mutex1`、`mutex2`、`mutex3`、`wrt`,保证了每次只读或者只写、写的时候只有一个人可以写、防止多个读者同时修改一个`readCount`、在`readCount=1`的时候阻止写者进行写操作以及防止读者和写者竞争，其中信号量`RWmutex`比`mutex3`先释放，从而一旦有写者就可以获取资源。在主函数中，需要输入进程序列号、所执行的操作以及开始和结束时间

```
    while(scanf("%d", &id) != EOF) {

        char role;      //producer or consumer
        int opTime;     //operating time
        int lastTime;   //run time

        scanf("%c%d%d", &role, &opTime, &lastTime);
        struct data* d = (struct data*)malloc(sizeof(struct data));

        d->id = id;
        d->opTime = opTime;
        d->lastTime = lastTime;

        if(role == 'R') {
            printf("Create the %d thread: Reader\n", id);
            pthread_create(&tid, &attr, Reader, d);

        }
        else if(role == 'W') {
```

```
            printf("Create the %d thread: Writer\n", id);
            pthread_create(&tid, &attr, Writer, d);
        }
    }
```

读写锁实际是一种自旋锁，本身保证了在读的时候可以有多个进程执行，但是在写的时候只能有一个进程执行，并且保障了读者和写者之间的互斥

3.管程实现与应用

首先是利用一般信号量进行实现,其中capacity用于表示`circlebuffer`的容量，其中的`consumer_sem`和`producer_sem`分别为当前队列中可以取出的资源的计数器和空闲位置的计数器，在往`circlebuffer`放入资源的过程中，首先要自动判断是否有空闲空间，没有则阻塞，如果取出资源的过程中资源数量小于0则陷入等待，`sem_wait`计数-1

```
class BlockQueue{
     public:
     BlockQueue():safe_queue(CAPACITY)
     {
         capacity = CAPACITY;
         sem_init(&lock, 0, 1);
         sem_init(&producer_sem, 0, capacity);
         sem_init(&consumer_sem, 0, 0);
         pos_read = pos_write = 0;
     }
     ~BlockQueue()
     {
         sem_destroy(&lock);
         sem_destroy(&consumer_sem);
         sem_destroy(&producer_sem);
     }

     void push(T data)
     {
         sem_wait(&producer_sem);
         sem_wait(&lock);

         safe_queue[pos_write] = data;
         pos_write = (pos_write + 1) % capacity;

         sem_post(&lock);
         sem_post(&consumer_sem);
     }
     T pop()
     {
         sem_wait(&consumer_sem);
         sem_wait(&lock);

         T data = safe_queue[pos_read];
         pos_read = (pos_read + 1) % capacity;

         sem_post(&lock);
         sem_post(&producer_sem);

         return data;
     }
```

```
    private:
    vector<T> safe_queue;
    size_t capacity;
    sem_t lock, producer_sem, consumer_sem;

    size_t pos_write;
    size_t pos_read;
};
```

下面是利用互斥锁和条件变量进行管程的实现，其中capacity表示`circlebuffer`的容量，互斥通过`pthread`库中的`pthread_mutex_t`实现，而进程同步通过`pthread_cond_t`实现，`push`和`pop`分别用于向队列中插入数据和取出数据，是对外提供的接口

```
class Queue{
private:
  queue<int> _store;
  int _capacity;
  pthread_mutex_t mutexs;//互斥锁
  pthread_cond_t  cond_Preducer;
  pthread_cond_t  cond_Consumer;

public:
  Queue(int capacity = MAX_CAPACITY)//构造函数 将队列的总容量设为5
    :_capacity(capacity)
  {
    pthread_mutex_init(&mutexs, NULL);//初始化互斥锁
    pthread_cond_init(&cond_Consumer, NULL);
    pthread_cond_init(&cond_Preducer, NULL);
  }
  ~Queue()//析构函数
  {
    pthread_mutex_destroy(&mutexs);//释放互斥锁所占用的资源
    pthread_cond_destroy(&cond_Consumer);
    pthread_cond_destroy(&cond_Preducer);
  }

  bool push(const int& data)
  {
      pthread_mutex_lock(&mutexs);//上锁
      while(_capacity ==_store.size())//判断当前队列是否已满
      {//如果队列满，则
        pthread_cond_signal(&cond_Consumer);
        pthread_cond_wait(&cond_Preducer,&mutexs);
      }
      //如果队列不满，则
      _store.push(data);//插入数据
      pthread_cond_signal(&cond_Consumer);
      pthread_mutex_unlock(&mutexs);//解锁
    return true;
  }

  bool pop(int& data)
  {
    pthread_mutex_lock(&mutexs);//上锁
    while(_store.empty())//判断队列是否为空
    {//如果队列为空，则
```

```
        pthread_cond_signal(&cond_Preducer);
        pthread_cond_wait(&cond_Consumer,&mutexs);
    }
    //如果队列不为空，则
    data = _store.front();//从队列中取出一个元素，这里取出的是当前队列的第一个元素
    _store.pop();//pop操作
    pthread_cond_signal(&cond_Preducer);
    pthread_mutex_unlock(&mutexs);//解锁
    return true;
  }
};
```

由于在生产者消费者模型中，临界资源其实就是一个环形缓冲区，所以要想测试上述`circlebuffer`类的正确性，只需要用生产者消费者的问题进行测试即可

4.死锁问题

本次死锁问题选择的样例是哲学家吃面问题，由于在哲学家吃面问题中，只有五只筷子，所以如果每一个哲学家都拿一个筷子，又因为在这个程序中不存在抢占调度，所以资源平均分配之后，没有资源会返回，那么这个程序产生死锁，无法继续运行

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#define N 5
sem_t sfork[N];

void* philosopher(void*);

int main()
{
    for(int i = 0; i < N; i++)
        sem_init(&sfork[i], 0, 1);
    pthread_t tid[N];
    for(int i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, philosopher, (void*)i);
    for(int i = 0; i < N; i++)
        pthread_join(tid[i], NULL);

    return 0;
}

void* philosopher(void* p)
{
    int id = (int)p;
    while(1)
    {
        printf("(%d)The philosopher is thinking......\n", id);
        sem_wait(&sfork[id]);
        printf("(%d)The philosopher has got left (%d)fork\n", id, id);
        sleep(5);
        sem_wait(&sfork[(id+1)%N]);
        printf("(%d)The philosopher has got right (%d)fork\n", id+1, id+1);
        printf("(%d)The philosopher is eating......\n", id);
```

```
        sem_post(&sfork[id]);
        printf("(%d)The philosopher has put down the left (%d)fork\n", id, id);
        sem_post(&sfork[(id+1)%N]);
         printf("(%d)The philosopher has put down the right (%d)fork\n", id+1,
id+1);
    }
    return NULL;
}
```