Project 5

秦啸涵　**521021910604**

使用Pthread API实现线程池

threadpool.c

相关结构体和信号量的声明

```c
// the work queue
struct worknode
{
    task worktodo;
    struct worknode *next;
} *head, *tail;

// the worker bee
pthread_t bee[NUMBER_OF_THREADS];

pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t sem;
```

enqueue()

```c
int enqueue(task t)
{
    tail->next = (struct worknode*)malloc(sizeof(struct worknode));
    if (tail->next == NULL)
    {
        printf("Queue is full\n");
        return 1;
    }
    tail = tail->next;
    tail->worktodo = t;
    return 0;
}
```

这里并未使用信号量，加锁操作放在函数体之外执行，详见 `pool_submit()`

dequeue()

```c
task dequeue()
{
    if (head == tail)
    {
        printf("Queue is empty\n");
        exit(0);
    }
    struct worknode *old_head = head;
    head = head->next;
    free(old_head);
    return head->worktodo;
}
```

与 `enqueue()` 相同，加锁操作在函数体外执行，需要注意的点是在使用链表模拟队列的过程中，为了方便地判断队列满或空，这里的head实际是虚拟头节点，它指向的数据是无效的，因此 `dequeue()` 操作是先出队再返回 `head->worktodo`

函数指针worker

```c
void *worker(void *param)
{
    // execute the task
    // execute(worktodo.function, worktodo.data);
    static task worktodo;
    // printf("worker created\n");
    while(TRUE)
    {
        sem_wait(&sem);
        if (shutdown)
        {
            pthread_exit(0);
        }
        pthread_mutex_lock(&queue_mutex);
        worktodo = dequeue();
        pthread_mutex_unlock(&queue_mutex);
        execute(worktodo.function, worktodo.data);
    }
    pthread_exit(0);
}
```

线程要执行的操作，从队列中取出数并 `execute()` ，这里需要加锁

- `sem` 信号量进行当前队列是否为空的判断
- `queue_mutex` 是对队列执行操作的互斥锁

pool_submit()

```c
int pool_submit(void (*somefunction)(void *p), void *p)
{
    // worktodo.function = somefunction;
    // worktodo.data = p;
    // printf("submitting work\n");
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;
    // printf("worktodo created\n");
    pthread_mutex_lock(&queue_mutex);
    enqueue(worktodo);
    pthread_mutex_unlock(&queue_mutex);
    sem_post(&sem);
    // printf("work submitted\n");
    return 0;
}
```

向队列中生成一个新的任务，需要加锁

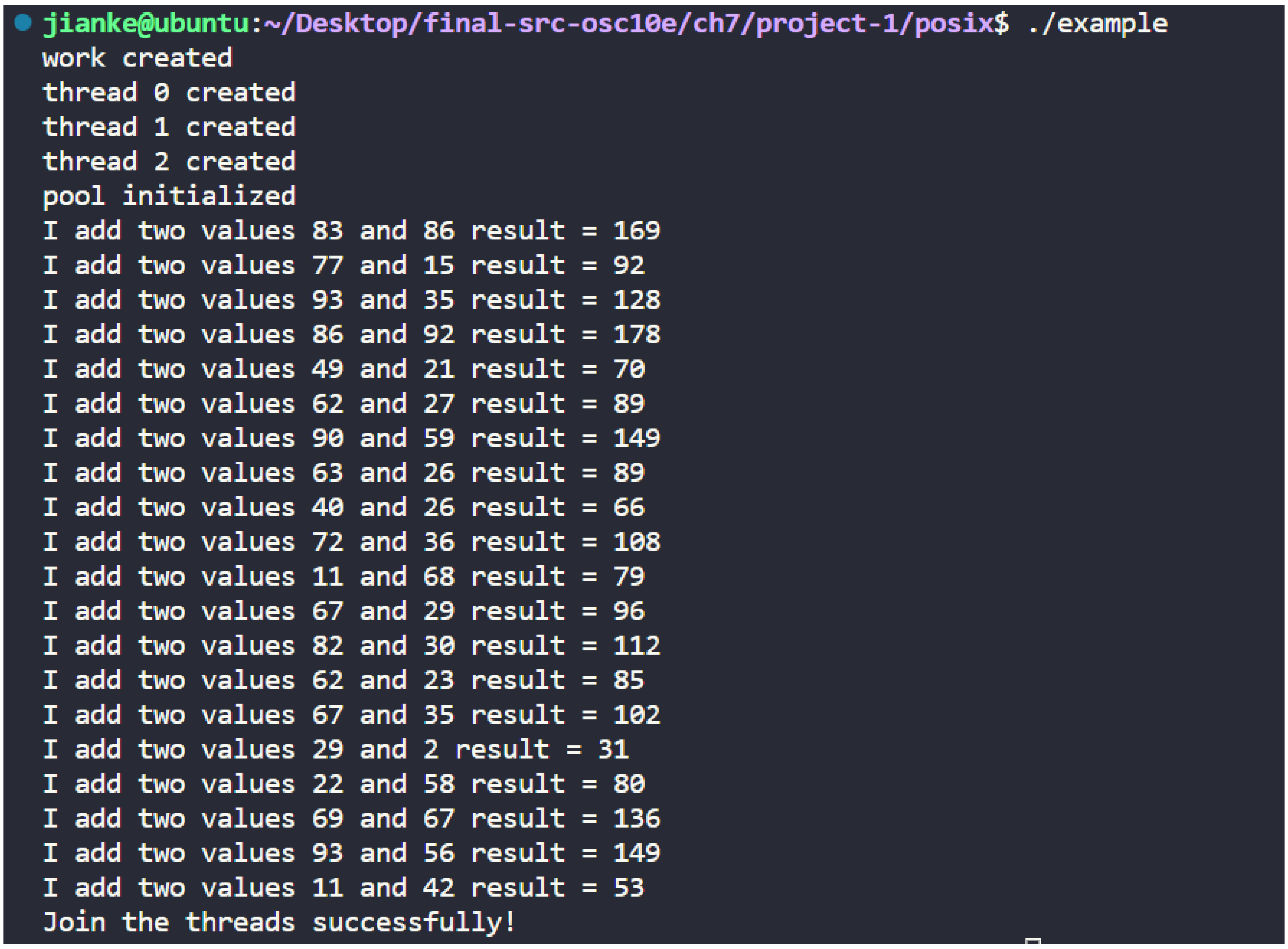
线程池初始化和终止

```c
// initialize the thread pool
void pool_init(void)
{
    head = (struct worknode*)malloc(sizeof(struct worknode));
    tail = head;
    sem_init(&sem, 0, 0);
    for(int i = 0; i < NUMBER_OF_THREADS; i++)
    {
        pthread_create(&bee[i], NULL, worker, NULL);
        printf("thread %d created\n", i);
    }
}

// shutdown the thread pool
void pool_shutdown(void)
{
    shutdown = true;
    for (int i = 0; i < NUMBER_OF_THREADS; ++ i)
        sem_post(&sem);
    for(int i = 0; i < NUMBER_OF_THREADS; i++)
    {
        pthread_join(bee[i], NULL);
    }
    printf("Join the threads successfully!\n");
    sem_destroy(&sem);
    pthread_mutex_destroy(&queue_mutex);

}
```

client.c

```c
int main(void)
{
    // create some work to do
    struct data work[NUMBER_OF_TASKS];
    for(int i=0; i<NUMBER_OF_TASKS; i++)
    {
        work[i].a = rand()%100;
        work[i].b = rand()%100;
    }
    printf("work created\n");
    // initialize the thread pool
    pool_init();
    printf("pool initialized\n");
    // submit the work to the queue
    // pool_submit(&add,&work);
    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        pool_submit(&add,&work[i]);
    }

    // may be helpful
    sleep(3);

    pool_shutdown();

    return 0;
}

```

运行结果:


```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch7/project-1/posix$ ./example
work created
thread 0 created
thread 1 created
thread 2 created
pool initialized
I add two values 83 and 86 result = 169
I add two values 77 and 15 result = 92
I add two values 93 and 35 result = 128
I add two values 86 and 92 result = 178
I add two values 49 and 21 result = 70
I add two values 62 and 27 result = 89
I add two values 90 and 59 result = 149
I add two values 63 and 26 result = 89
I add two values 40 and 26 result = 66
I add two values 72 and 36 result = 108
I add two values 11 and 68 result = 79
I add two values 67 and 29 result = 96
I add two values 82 and 30 result = 112
I add two values 62 and 23 result = 85
I add two values 67 and 35 result = 102
I add two values 29 and 2 result = 31
I add two values 22 and 58 result = 80
I add two values 69 and 67 result = 136
I add two values 93 and 56 result = 149
I add two values 11 and 42 result = 53
Join the threads successfully!
```

生产者-消费者问题

buffer.c

缓冲区 `buffer` 的定义及相关函数实现

```c
int head, tail;
buffer_item buffer[BUFFER_SIZE+1];

int insert_item(buffer_item item) {
    // insert item into buffer
    // return 0 if successful, otherwise
    // return -1 indicating an error condition
    if ((tail+1)%(BUFFER_SIZE+1) != head)
    {
        tail = (tail+1)%(BUFFER_SIZE+1);
        buffer[tail] = item;
        return 0;
    }
    else
    {
        return -1;
    }
}

int remove_item(buffer_item *item) {
    // remove an object from buffer
    // placing it in item
    // return 0 if successful, otherwise
    // return -1 indicating an error condition
    if (head == tail)
    {
        return -1;
    }
    head = (head+1)%(BUFFER_SIZE+1);
    *item = buffer[head];
    return 0;
}

void buffer_init() {
    // initialize buffer
    // return 0 if successful, otherwise
    // return -1 indicating an error condition
    head = 0;
    tail = 0;
}

```

使用队列实现，这里的 `insert_item()` 和 `remove_item()` 的加锁操作依然放在函数体之外实现

producer_consumer.c

相关信号量的定义

```c
sem_t empty, full;
pthread_mutex_t mutex;
int flag = 0;
```

其中 `flag` 用于控制进程的终止

producer和consumer函数指针实现

```c
void *producer(void *param)
{
    buffer_item item;
    while (1)
    {
        /* sleep for a random period of time */
        sleep(rand() % 5);
        /* generate a random number */
        item = rand()%100;
        sem_wait(&empty);
        if (flag)
            break;
        pthread_mutex_lock(&mutex);
        if (insert_item(item))
            fprintf(stderr, "report error condition");
        else
            printf("\033[1;32mproducer produced item 
%d\033[0m\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *param)
{
    buffer_item item;
    while (1)
    {
        /* sleep for a random period of time */
        sleep(rand() % 5);
        sem_wait(&full);
        if (flag)
            break;
        pthread_mutex_lock(&mutex);
        if (remove_item(&item))
            fprintf(stderr, "report error condition");
        else
            printf("\033[1;31mconsumer consumed item %d\033[0m\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}
```

二者都是在循环中随机 `sleep` 一段时间后，若满足信号量条件则进入缓冲区执行，并输出相关信息

main函数实现

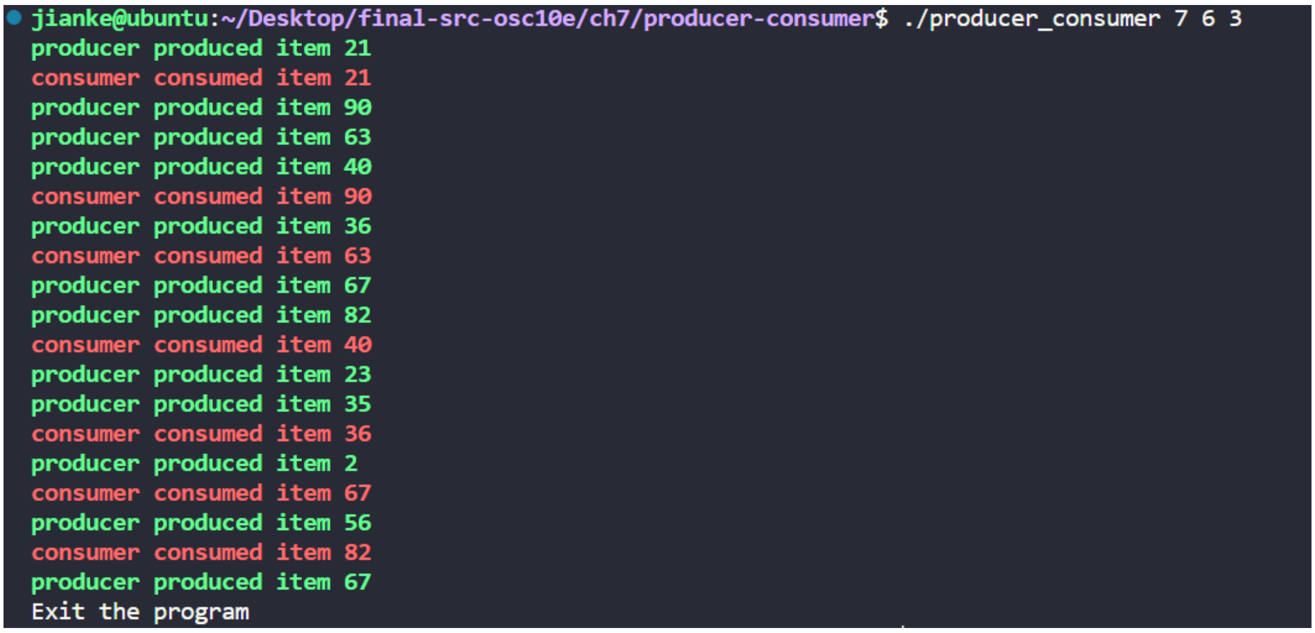
```
int main(int argc, char *argv[])
{
    int sleepTime, producerThreads, consumerThreads;
    int i, j;
    /* 1. Get command line arguments argv[1],argv[2],argv[3] */
    if (argc != 4)
    {
        fprintf(stderr, "Usage: <sleep time> <producer threads> <consumer threads>\n");
        return -1;
    }
    sleepTime = atoi(argv[1]);
    producerThreads = atoi(argv[2]);
    consumerThreads = atoi(argv[3]);
    /* 2. Initialize buffer */
    buffer_init();
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    /* 3. Create producer thread(s) */
    for (i = 0; i < producerThreads; i++)
    {
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, producer, NULL);
```

```c
    }
    /* 4. Create consumer thread(s) */
    for (j = 0; j < consumerThreads; j++)
    {
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, consumer, NULL);
    }
    /* 5. Sleep */
    sleep(sleepTime);
    /* 6. Exit */
    flag = 1;
    for (i = 0; i < producerThreads; i++)
    {
        sem_post(&empty);
    }
    for (j = 0; j < consumerThreads; j++)
    {
        sem_post(&full);
    }
    printf("Exit the program\n");
    exit(0);
}
```

按照教材上的6步执行

1. Get command line arguments argv[1],argv[2],argv[3]
2. Initialize buffer
3. Create producer thread(s)
4. Create consumer thread(s)
5. Sleep
6. Exit

运行结果如下:



```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch7/producer-consumer$ ./producer_consumer 7 6 3
producer produced item 21
consumer consumed item 21
producer produced item 90
producer produced item 63
producer produced item 40
consumer consumed item 90
producer produced item 36
consumer consumed item 63
producer produced item 67
producer produced item 82
consumer consumed item 40
producer produced item 23
producer produced item 35
consumer consumed item 36
producer produced item 2
consumer consumed item 67
producer produced item 56
consumer consumed item 82
producer produced item 67
Exit the program
```

这里的7是运行时间，6代表创建6个生产者线程，3代表创建3个消费者线程

Bonus

线程池的核心线程数量的设置对线程池的性能和资源消耗有重要的影响。过大的核心线程数量会浪费系统资源，而过小的核心线程数量会导致任务等待时间过长和系统负载过高。

一般来说，合理的线程池核心线程数量的设置应该考虑以下几个因素：

1. 任务的类型和特性：任务类型和特性不同，需要的处理时间也不同，处理时间短的任务可以使用更多的线程来提高处理效率，处理时间长的任务则需要适当减少线程数量以避免资源浪费。
2. 系统硬件资源：线程的运行需要占用CPU、内存等系统资源，应该考虑系统的硬件资源情况，避免过多地占用系统资源导致系统负载过高。
3. 线程池的扩展性：在设计线程池的时候应该考虑线程池的扩展性，应该留有一定的余地来扩展线程池的大小，以适应未来的业务增长。

一般来说，可以根据任务的类型和特性以及系统硬件资源情况来合理设置线程池的核心线程数量，一般建议设置在CPU核心数的1-2倍之间。如果系统的硬件资源较为充足，可以适当增加核心线程数量来提高处理效率，如果系统硬件资源有限，则需要适当减少核心线程数量来避免过多地占用系统资源。