17级并行与分布式计算

HW3 17341137 宋震鹏

$1 实验要求

Implement a multi-access threaded queue with multiple threads inserting and multiple threads extracting from the queue. Use mutex-locks to synchronize access to the queue. Document the time for 1000 insertions and 1000 extractions each by 64 insertions threads (Producers) and 64 extraction threads (Consumer).

- 语言限制：C/C++/Java
- PS：不能直接使用STL或者JDK中现有的并发访问队列，请基于普通的queue或自行实现

$2 实验分析

本次实验要求实现一个并发访问队列，实现一个“生产者-消费者”问题。

根据分析，该问题大致可以表达为：

- 一个单向队列，存储元素(商品)。
- 新加入（生产）元素从队尾加入队列，取出（消费）元素从队首取出。
- 当队列满时，不再生产。
- 当队列空时，不再消费。
- 每个子线程代表一个生产/消费者。

$3 实验代码

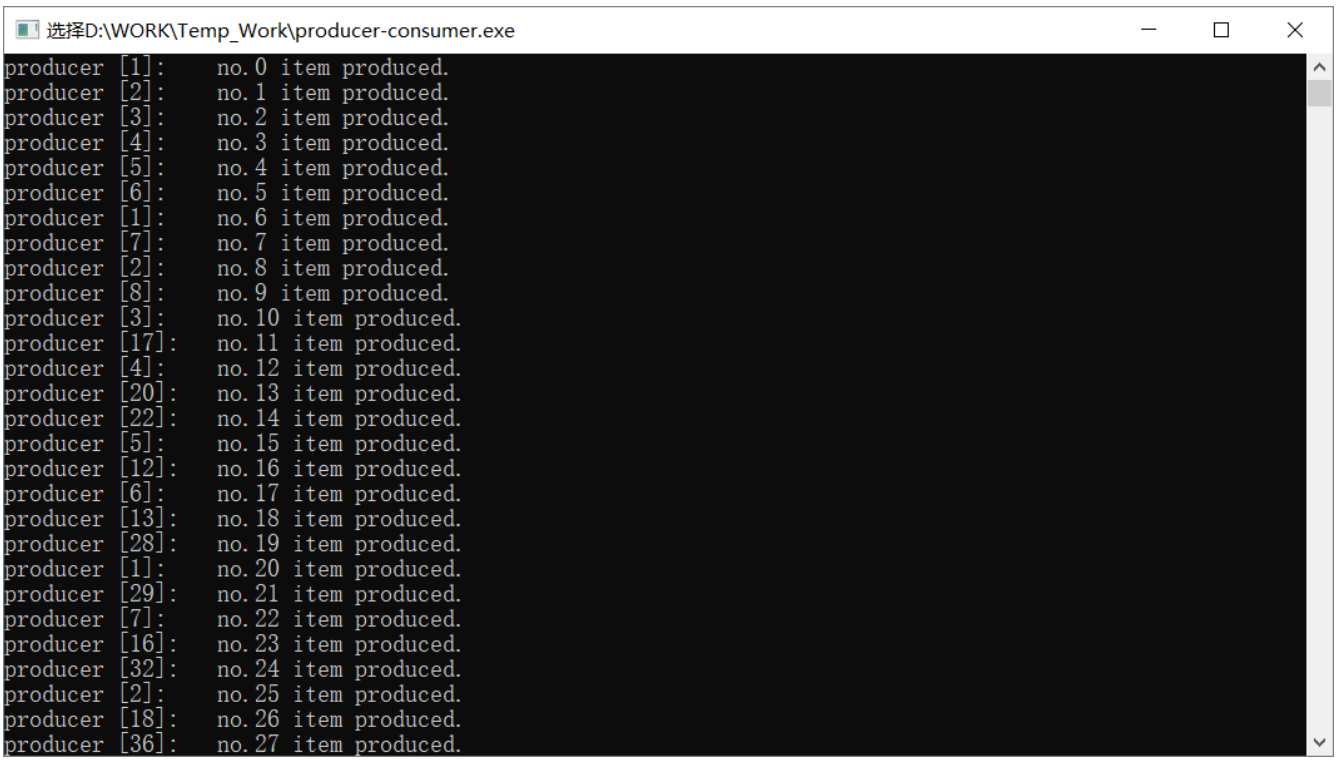
```
#include <stdio.h>
#include <pthread.h>
#define queue_size 1000

int in = 0, out = 0;    // 队首、队尾；从队首取，从队尾存入
void *producer(void *); // 生产者函数
void *consumer(void *); // 消费者函数

pthread_mutex_t read_mutex      = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t write_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t Queue_Not_Full   = PTHREAD_COND_INITIALIZER;
pthread_cond_t Queue_Not_Empty  = PTHREAD_COND_INITIALIZER;

/* 判断队空 */
int queue_is_empty() {
    return (in == out);
}
int queue_is_full() {
    return (out == ((in + 1) % queue_size));
}

/* 主线程 */
```

```c
int main() {
    pthread_t tid[128];
    int p_array[64];
    int c_array[64];
    int icount;

    /* 各线程编号 */
    for (int i = 0; i < 64; i++) {
        p_array[i] = i + 1;
        c_array[i] = i + 1;
    }

    /* 创建线程 */
    for (int i = 0; i < 64; i++)
        pthread_create( & tid[i], NULL, consumer, (void * )c_array[i]);
    for (int i = 64; i < 128; i++)
        pthread_create( & tid[i], NULL, producer, (void * )p_array[i]);

    /* 挂起等待结束 */
    for (icount = 0; icount < 128; icount++) {
        pthread_join(tid[icount], NULL);
    }
    return 0;
}

void *producer(void *arg) {
    int *pno;
    pno = (int *)arg;
    while(1) {
        pthread_mutex_lock( &write_mutex);
        if (queue_is_full()) {
            pthread_cond_wait( &Queue_Not_Full,  &write_mutex);
        }
        printf("producer [%d]:\t no.%d item produced. \n", pno, in);
        in = (in + 1) % queue_size;
        pthread_mutex_unlock( &write_mutex);
        pthread_cond_signal( &Queue_Not_Empty);
    }
}

void *consumer(void *arg) {
    int *cno;
    cno = (int *)arg;
    while(1) {
        pthread_mutex_lock( &read_mutex);
        if (queue_is_empty) {
            pthread_cond_wait( &Queue_Not_Empty,  &read_mutex);
        }
        printf("consumer [%d]:\t no.%d item consumed. \n", cno, out);
        out = (out + 1) % queue_size;
        pthread_mutex_unlock( &read_mutex);
        pthread_cond_signal( &Queue_Not_Full);
    }
```

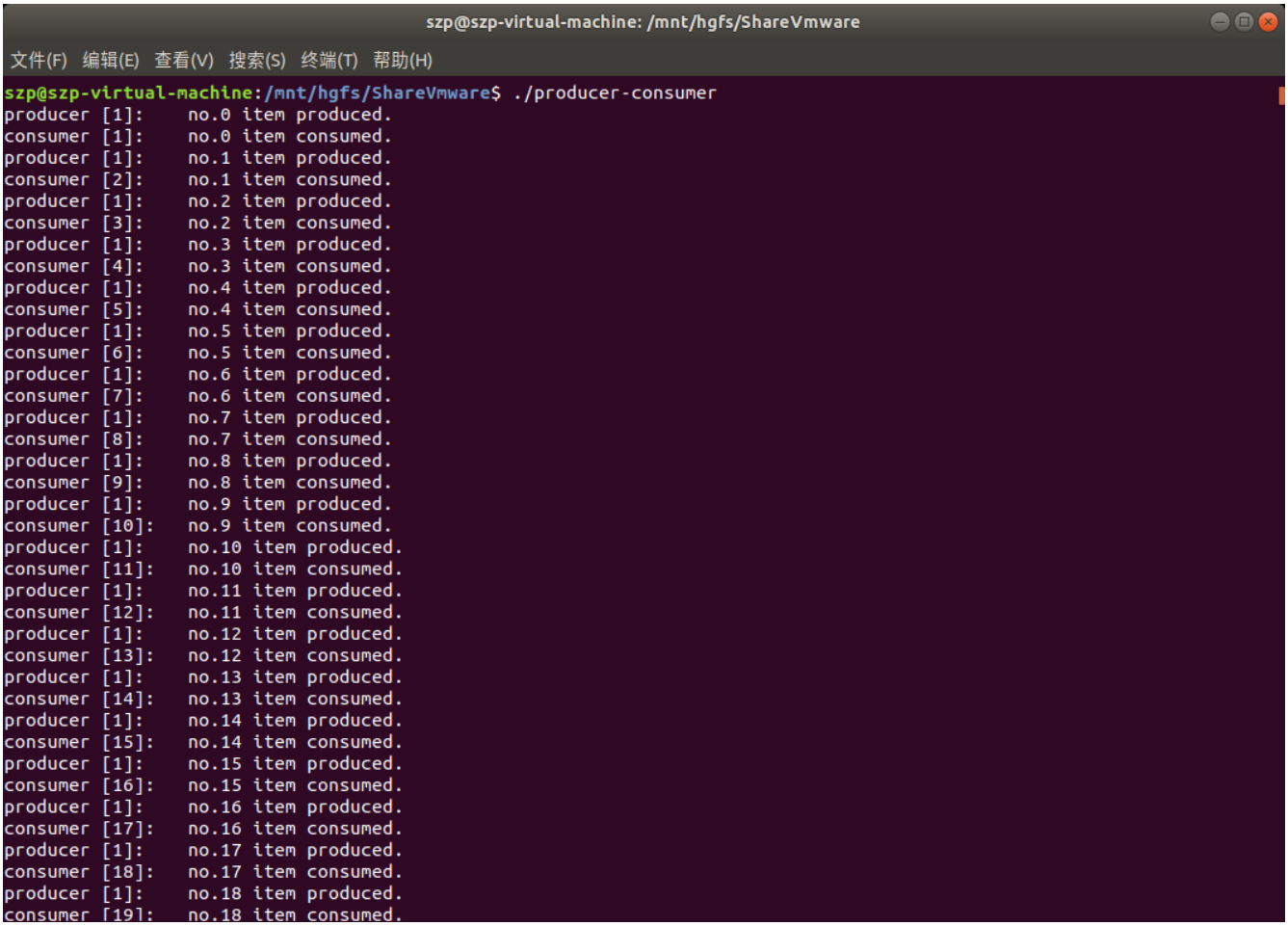
```
}
```

$4 实验截图

Windows下运行：



Linux下运行：



$5 实验总结

在本次实验中，我加深了对<pthread.h>库的理解，了解了多线程并发访问队列的实现。

在开始实验时，不太清楚本次实验具体需要实现的内容，以至花了不少时间在理解题意上。后来整理出“生产者-消费者”问题的本质后，在设计代码时就更有条理了。

主要的问题还是在<pthread.h>库多线程函数的用法上有比较多的不理解之处，对以下函数进行了研究和学习：

```
/* 1 pthread_create() */
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void * (*start_routine)(void *), void *arg);
// func: 创建一个由调用线程控制的新的线程并发运行。
// args: thread: 指向线程标识符的指针; attr: 设置线程属性;
//       (void *): 线程运行函数的起始地址; arg: 运行函数的参数。

/* 2 pthread_join() */
int pthread_join(pthread_t th, void **thread_return);
// func: 挂载一个在执行的线程th直到该线程通过调用pthread_exit或者cancelled结束。
// args: th: 挂载的线程; thread_return: 所指的区域保存线程th的返回值。

/* 3 pthread_mutex_lock() & pthread_mutex_unlock()*/
int pthread_mutex_lock(pthread_mutex_t *mutex);
// func: 互斥锁加锁/解锁
// args: 互斥锁
```

```
/* 4 pthread_cond_wait() */
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
// func: 自动解锁mutex(pthread_unlock_mutex)等待条件变量cond发送。
// args: cond: 条件变量; mutex: 互斥锁

/* 5 pthread_cond_signal();*/
int pthread_cond_signal(pthread_cond_t *cond);
// func: 激活一个正在等待条件变量cond的线程。如果没有线程在等待则什么也不会发生，如果有多个 线程在等待，则只能激活一个线程。
// args: cond: 条件变量。
```

至此本实验学习就告一段落，在并行多线程的学习上又进一步。

17341137 宋震鹏 19/4/17