操作系统原理

理论课实验3 - 同步互斥问题

学院	专业	班级	学号	姓名
数据科学与计算机学院	软件工程	教务2班	17343131	许滨楠

实验目的及具体要求

- 利用线程同步机制，实现生产者-消费者问题；
- 利用信号量机制，分别实现读者写者问题中的两个原则：
  - 以读者优先的原则进行线程的竞争；
  - 以写者优先的原则进行线程的竞争。

实验环境及工具准备

实验环境

- 主系统为MacOS X 10.14 Mojave
- 虚拟机系统为Linux Ubuntu 14.04 LTS

实验工具

- 在Parallels Desktop软件上运行虚拟机
- 代码文件编辑器：Visual Studio Code、Gedit Text Editor
- 终端：Mac下iTerm、Ubuntu下Terminal
- 终端Shell：zsh、其他环境包括gcc编译器、链接器以及一些小工具
- 实验报告用Markdown语言在Typora软件上编写并导出pdf

实验流程及结果验证

> 实验报告中的说明仅展示必要的关键代码，具体的实现代码请见附录中 **/code/对应题目文件夹** 。

生产者-消费者问题

> 利用线程同步机制和 Pthread 库实现课本中的第六章 Project 。

具体流程

根据课本中 Project 的要求，缓冲区大小设置为5。同时，程序利用三个信号量：

- empty - 记录当前缓冲区空位数量；
- full - 记录当前缓冲区满位数量；
- mutex - 二进制信号量，保护对缓冲区的操作，实现互斥。

来实现生产者-消费者问题中的操作。

主函数中，实现了读取测试文件的处理、线程的创建和控制，关键代码大致如下：

```
// 主函数对所创建的线程的传参通过结构体完成
typedef struct {
    int id, begin_time, last_time, item;
} args;

int main() {
    // 打开文件，存在 test_file 指针

    // 信号量等的初始化

    while (fscanf(test_file,"%d %c %d %d",&id,&pc,&begin,&last) != EOF) {
        // 参数结构体设置
        args tmp = {id, begin, last, -1};
        t[ptcnt] = tmp;
        if ('P' == pc) {
            fscanf(test_file,"%d",&item);
            t[ptcnt].item = item;
            // 线程创建
            pthread_create(&tid[ptcnt], &attr, producer, &t[ptcnt]);
            printf("[  Main  ] create producer thread %d.\n", id);
        }
        else {
            // 线程创建
            pthread_create(&tid[ptcnt], &attr, consumer, &t[ptcnt]);
            printf("[  Main  ] create consumer thread %d.\n", id);
        }
        // 为了防止主函数中的更改导致线程的参数传递出错
        // 用数组对参数进行存储 此处需要累加下标
        ++ptcnt;
    }
    // 主函数等待所有线程执行完毕
    for (int i = 0; i < ptcnt; ++i) {
        pthread_join(tid[i], NULL);
    }

    // 信号量等的“析构”
    exit(0);
}
```

生产者和消费者需要分别编写函数，实现基本的判定和操作：

```
void * producer(void * param) {
    // 存储参数列表
    args tmp = *((args *) param);
    buffer_item item;
    // 挂起定义的相应时间 模拟开始的时延
    sleep(tmp.begin_time);

    int flag = 1;
    while (flag) {
        // 即将进入临界区 设置信号量
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        // 临界操作
        // 模拟进行生产操作 打印相关信息
        // 挂起模拟操作耗时
        // 出临界区
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}

void * consumer(void * param) {
    // 存储参数列表
    args tmp = *((args *) param);
    buffer_item item;
    // 挂起定义的相应时间 模拟开始的时延
    sleep(tmp.begin_time);

    int flag = 1;
    while (flag) {
        // 即将进入临界区 设置信号量
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        // 临界操作
        // 模拟进行消费操作 打印相关信息
        // 挂起模拟操作耗时
        // 出临界区
        sleep(tmp.last_time);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}
```

测试文件 (test.txt)：

```
1 C 3 5
2 P 4 5 1
3 C 5 2
4 C 6 5
5 P 7 3 2
6 P 8 4 3
```

结果验证

代码及Makefile编写完毕后，在同级目录下完成测试文件的输入，在Ubuntu终端cd进入代码目录，执行 **make** 语句完成代码编译，接着键入命令： **./producer_consumer** 执行程序，观察输出情况。

```
xubn@ubuntu code/producer_and_consumer make clean
rm producer_consumer
xubn@ubuntu code/producer_and_consumer make
gcc --std=c99 producer_consumer.c -o producer_consumer -pthread
xubn@ubuntu code/producer_and_consumer ./producer_consumer
[  Main  ] create consumer thread 1.
[  Main  ] create producer thread 2.
[  Main  ] create consumer thread 3.
[  Main  ] create consumer thread 4.
[  Main  ] create producer thread 5.
[  Main  ] create producer thread 6.
[Producer] item  1 inserted successfully [Thread 2].
[ Buffer ]    1   0   0   0   0
[Consumer] item  1  removed successfully [Thread 1].
[ Buffer ]    0   0   0   0   0
[Producer] item  3 inserted successfully [Thread 6].
[ Buffer ]    0   3   0   0   0
[Consumer] item  3  removed successfully [Thread 3].
[ Buffer ]    0   0   0   0   0
[Producer] item  2 inserted successfully [Thread 5].
[ Buffer ]    0   0   2   0   0
[Consumer] item  2  removed successfully [Thread 4].
[ Buffer ]    0   0   0   0   0
xubn@ubuntu code/producer_and_consumer _
```

可以看到消费者线程1创建后，由于此时临界区缓存为空，无产品可供消费，挂起等待；生产者线程2创建后，缓存有空位，进行生产，打印相关信息；此时缓存中已经有产品，可以进行消费，消费者线程1进行消费，打印信息……后序操作及缓存区情况已经打印出来，基本符合生产者-消费者问题情况。

结果满足实验要求。

读者-写者问题

> 利用信号量机制，分别以读者优先和写者优先的原则实现读者-写者问题。

具体流程

根据测试要求，利用互斥信号量 mutex 和读/写信号量 read_signal, write_signal，辅以读者写者等待队列的计数器 reader_cnt, writer_cnt 对线程进行控制，来实现读者-写者问题的基本操作。

在读者优先或写者优先的视线中，主函数的功能都是一样的。主函数中，实现了测试文件的读取，线程的创建和控制等操作，关键代码大致如下：

```c
// 用枚举类型标识当前的读/写/等待操作状态 reading/writing/waiting flag
enum {
    f_waiting, f_reading, f_writing
} state = f_waiting;

// 线程创建所需参数传递结构体
typedef struct {
    int id, begin_time, last_time;
} character;

int main() {
    // 读取测试文件

    // 相关信号量等的初始化
    sem_init(&read_signal, 0, 0);
    sem_init(&write_signal, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    while (fscanf(test_file, "%d %c %d %d",&id,&ch,&begin,&last) != EOF) {
        // 初始化线程创建相关参数
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        character tmp = {id, begin, last};
        chrt[ptcnt] = tmp;
        // 创建读者进程 打印信息
        if ('R' == ch) {
            pthread_create(&tid[ptcnt], &attr, reader, &chrt[ptcnt]);
            printf("[ Main ] create reader thread %d.\n", id);
        }
        // 创建写者进程 打印信息
        else {
            pthread_create(&tid[ptcnt], &attr, writer, &chrt[ptcnt]);
            printf("[ Main ] create writer thread %d.\n", id);
        }
        ++ptcnt;
    }

    // 主进程等待所有线程执行完毕
    for (int i = 0; i < ptcnt; ++i) {
        pthread_join(tid[i], NULL);
    }

    // 释放信号量资源
    pthread_mutex_destroy(&mutex);
    sem_destroy(&read_signal);
    sem_destroy(&write_signal);

    exit(0);
}
```

读者写者各自的线程函数的基本实现：

```
void * reader(void * args) {
    // 获取参数列表
    // 挂起相应时间模拟开始的时延
    // 打印发出请求的信息
    character chrt = *((character *)args);
    sleep(chrt.begin_time);
    printf("[Reader] thread %d is waiting to read.\n", chrt.id);

    // 临界区操作
    pthread_mutex_lock(&mutex);

    // 读者排队等待/可读判定
    // ……

    // 出临界区 打印相关信息与睡眠模拟操作过程
    pthread_mutex_unlock(&mutex);
    sem_wait(&read_signal);
    printf("[Reader] thread %d is reading.\n", chrt.id);
    sleep(chrt.last_time);
    printf("[Reader] thread %d finished reading.\n", chrt.id);

    pthread_mutex_lock(&mutex);

    // 完成操作 更新队列情况
    // ……

    pthread_mutex_unlock(&mutex);
}


void * writer(void * args) {
    // 获取参数列表
    // 挂起相应时间模拟开始的时延
    // 打印发出请求的信息
    character chrt = *((character *)args);
    sleep(chrt.begin_time);
    printf("[Writer] thread %d is waiting to write.\n", chrt.id);


    // 临界区操作
    pthread_mutex_lock(&mutex);

    // 写者排队等待/可写判定
    // ……

    // 出临界区 打印相关信息与睡眠模拟操作过程
```

```
    pthread_mutex_unlock(&mutex);
    sem_wait(&write_signal);
    printf("[Writer] thread %d is writing.\n", chrt.id);
    sleep(chrt.last_time);
    printf("[Writer] thread %d finished writing.\n", chrt.id);

    pthread_mutex_lock(&mutex);

    // 完成操作 更新队列情况
    // ……

    pthread_mutex_unlock(&mutex);
}
```

测试文件 (test.txt)：

```
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3
```

读者优先

在读者优先的原则中，若某读者申请进行读操作的时候正有其他读者正在进行读操作，则该读者可以直接开始进行读操作，无需阻塞等待。这样的操作在读者、写者各自的进程中上述代码 …… 的部分实现即可，具体实现如下：

```
void * reader(void * args) {
    // ……

    // 读者排队等待/可读判定
    ++reader_cnt;
    if (f_waiting == state || f_reading == state) {
        sem_post(&read_signal);
        state = f_reading;
    }
    // 读者优先过程中，只要当前读/写状态是有读者正在读
    // 或者没有读者写者正在访问，则新的读者可以直接开始读操作

    // ……

    // 完成操作 更新队列情况
    --reader_cnt;
    if (0 == reader_cnt) {
        if (0 != writer_cnt) {
            sem_post(&write_signal);
            state = f_writing;
        }
```

```
        else {
            state = f_waiting;
        }
    }
    // 读者读操作完成后，如果没有其他读者了才考虑写者情况
    // 如果没有读者但有写者，开始考虑写者，读/写状态为写
    // 如果没有读者也没有写者，那么读/写情况为等待；

    // ……
}

void * writer(void * args) {
    // ……

    // 写者排队等待/可写判定
    ++writer_cnt;
    if (f_waiting == state) {
        sem_post(&write_signal);
        state = f_writing;
    }
    // 读者优先过程中，只有当没有读者或其他写者的时候
    // 新的写者才能得到写的机会

    // ……

    // 完成操作 更新队列情况
    --writer_cnt;
    if (0 != reader_cnt) {
        sem_post(&read_signal);
        state = f_reading;
    }
    else if (0 != writer_cnt) {
        sem_post(&write_signal);
        state = f_writing;
    }
    else {
        state = f_waiting;
    }
    // 写者写操作完成后，先判断有无读者
    // 若有读者正在等待，则让读者进行读操作
    // 若没有读者正在等待且有写者正在等待，则继续进行其他写者的写操作
    // 否则，为等待状态

    // ……
}
```

写者优先

在写者优先的原则中，若某读者申请进行读操作的时候正有一写者在等待访问共享资源，则该读者必须等到没有写者正在等待的状态后才能开始进行读操作。这样的操作在读者、写者各自的进程中上述代码 ..... 的部分实现即可，具体实现如下：

```
void * reader(void * args) {
    // ……

    // 读者排队等待/可读判定
    ++reader_cnt;
    if ((0 == writer_cnt && f_reading == state) || f_waiting == state) {
        sem_post(&read_signal);
        state = f_reading;
    }
    // 写者优先过程中，只有没有写者正在等待且已经进入读状态
    // 或者队列正处于等待状态，先前没有读者写者等待的情况下才可以进行读操作

    // ……

    // 完成操作 更新队列情况
    --reader_cnt;
    if (0 != writer_cnt) {
        sem_post(&write_signal);
        state = f_writing;
    }
    else if (0 == reader_cnt) {
        state = f_waiting;
    }
    // 读操作完成后，若有写者在等待，则优先执行写操作
    // 读操作完成后，若无其他读者，则队列进入等待状态

    // ……
}

void * writer(void * args) {
    // ……

    // 写者排队等待/可写判定
    ++writer_cnt;
    if (f_waiting == state) {
        sem_post(&write_signal);
        state = f_writing;
    }
    // 写者优先过程中，只要队列处于等待态，写者就可以进行写操作
    // 否则写者将进行等待，（其优先操作已经在读者进程体现）

    // ……

    // 完成操作 更新队列情况
    --writer_cnt;
```

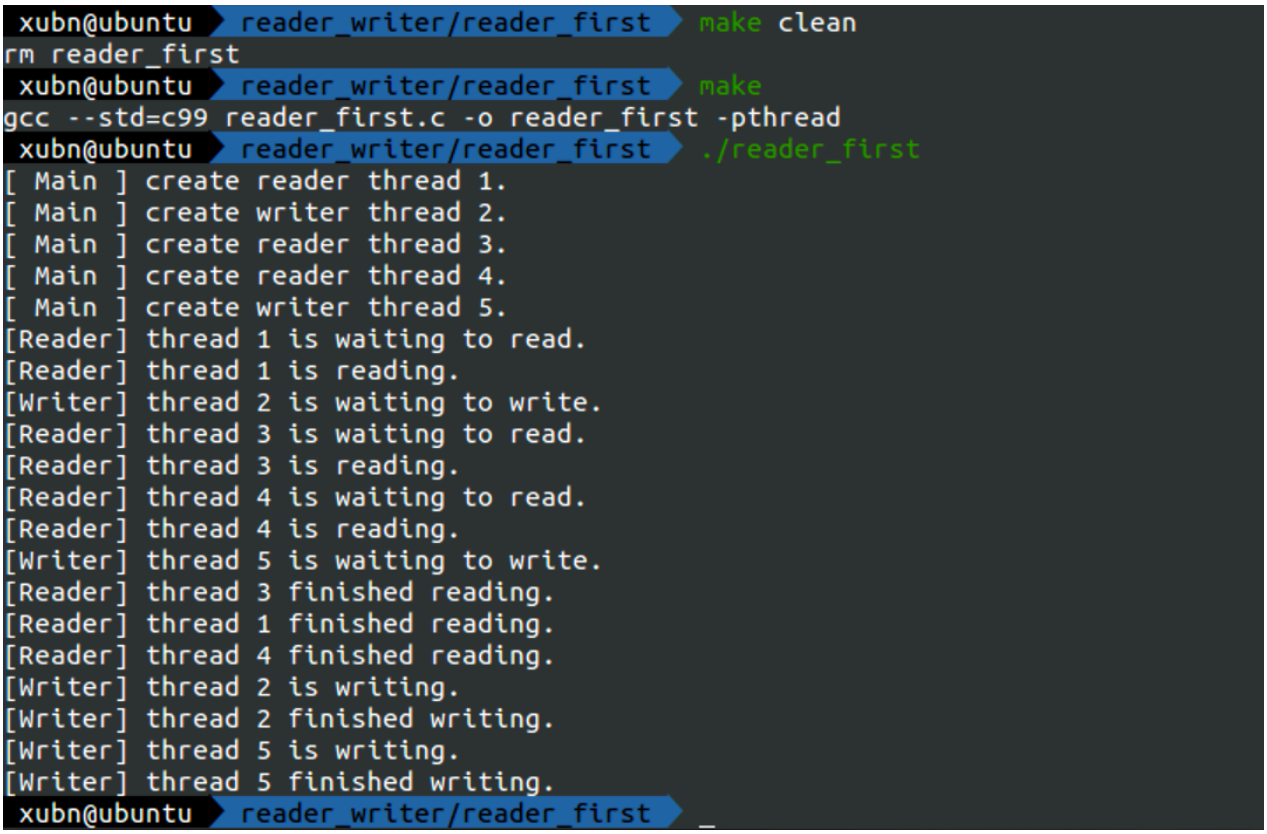
```
    if (0 != writer_cnt) {
        sem_post(&write_signal);
        state = f_writing;
    }
    else if (0 != reader_cnt) {
        for (int i = 0; i < reader_cnt; ++i) {
            sem_post(&read_signal);
        }
        state = f_reading;
    }
    else {
        state = f_waiting;
    }
    // 写操作完成后，若有其他写者正在等待，则优先执行写操作
    // 否则若有读者，则进行读操作，若无人等待，则队列等待

    // ……
}
```

结果验证

读者优先

代码及Makefile编写完毕后，在同级目录下完成测试文件的输入，在Ubuntu终端cd进入代码目录，执行 **make** 语句完成代码编译，接着键入命令： **./reader_first** 执行程序，观察输出情况。



观察到：

- 读者线程1创建后，队列为等待状态，直接开始读操作；
- 写者线程2创建后，等待读者完成操作；
- 读者线程3、4创建后，直接开始读操作，加入读者行列；
- 写者线程5创建后，等待读者完成操作；
- 读者线程1、3、4完成读操作，资源被释放，先前排队的写者2、5分别开始写。

基本符合实验要求。

写者优先

代码及Makefile编写完毕后，在同级目录下完成测试文件的输入，在Ubuntu终端cd进入代码目录，执行 **make** 语句完成代码编译，接着键入命令： **./writer_first** 执行程序，观察输出情况。

```
xubn@ubuntu  reader_writer/writer_first  make clean
rm writer_first
xubn@ubuntu  reader_writer/writer_first  make
gcc --std=c99 writer_first.c -o writer_first -pthread
xubn@ubuntu  reader_writer/writer_first  ./writer_first
[ Main ] create reader thread 1.
[ Main ] create writer thread 2.
[ Main ] create reader thread 3.
[ Main ] create reader thread 4.
[ Main ] create writer thread 5.
[Reader] thread 1 is waiting to read.
[Reader] thread 1 is reading.
[Writer] thread 2 is waiting to write.
[Reader] thread 3 is waiting to read.
[Reader] thread 4 is waiting to read.
[Writer] thread 5 is waiting to write.
[Reader] thread 1 finished reading.
[Writer] thread 2 is writing.
[Writer] thread 2 finished writing.
[Writer] thread 5 is writing.
[Writer] thread 5 finished writing.
[Reader] thread 4 is reading.
[Reader] thread 3 is reading.
[Reader] thread 3 finished reading.
[Reader] thread 4 finished reading.
xubn@ubuntu  reader_writer/writer_first  _
```

观察到：

- 读者线程1创建后，队列为等待状态，直接开始读操作；
- 写者线程2创建后，等待读者完成操作；
- 读者线程3、4创建后，因为此时已经有读者正在等待，所以后续读者线程等待；
- 写者线程5创建后，加入写者等待队列，等待读者完成操作；
- 读者线程1完成读操作，资源被释放，先前排队的写者2、5分别开始写，排队的读者3、4继续等待；
- 写者线程2、5都完成写操作后，先前排队的读者3、4分别开始读。

基本符合实验要求。

思考与总结

这是操作系统理论课的最后一个实验，两块内容都是第六章课上讲过的重点。因为有了课上的理论支持和指导课件中的提示，在熟悉了相关库和信号量的使用之后，还是可以基本完成的。实验中对理论课上讲过的生产者消费者问题、读者写者中的读者优先和写者优先原则进行了实现，实践起来和知识理论学习还是非常不同的，通过自己的代码实现，不仅熟悉了相关库函数和变量功能及使用，也对理论课上的内容有更深的理解。

生产者消费者问题、还有读者写者问题中的两个原则，其实考察和锻炼的都是对同步互斥问题的理解、对信号量的控制和使用、对多线程的运用。通过这次实验，各方面的认识、知识、技能等都得到了提升。同时，还进一步锻炼了代码方面的一些规范和能力，还是颇有收获的。