实验 3、同步互斥问题

一、生产者-消费者问题要求

1. 实验要求：

设计一个程序来解决有限缓冲问题，其中的生产者与消费者进程如图 6.10 与图 6.11 所示。
在 6.6.1 小节中，使用了三个信号量: empty (以记录有多少空位)、full (以记录有多少满位)以及 mutex (二进制信号量或互斥信号量，以保护对缓冲插入与删除的操作)。对于本项目， empty 与 full 将采用标准计数信号量，而 mutex 将采用二进制信号量。生产者与消费者作为独立线程，在 empty、full、mutex 的同步前提下，对缓冲进行插入与删除。
本项目，可采用 Pthread 。

2. 实验测试数据格式：

下面是一个测试数据文件的例子：

```
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
```

注意：在创建数据文件时，由于涉及到文件格式问题，最好在记事本中手工逐个键入数据，而不要拷贝粘贴数据，否则，本示例程序运行时可能会出现不可预知的错误。

3. 实验过程

1. 对实验的理解以及代码：

对实验的理解：
消费者和生产者两者的操作是互斥的，不同时进行的，因此在消费或生产的时候要使用 mutex 信号量以保证同一时间只有一个线程对缓冲区进行一种操作；
所有缓冲区都满位时生产者不生产，所有缓冲区都为空时消费者不消费，因此需要信号量 full 和 empty。

设置缓冲区数量以及声明各种全局变量：

```c
typedef int buffer_item;
#define BUFFER_SIZE 3
buffer_item buffer[BUFFER_SIZE]; // 循环队列缓冲区
int buffer_total_num = 0;      // 已满位i的缓冲区数量
int buffer_index = 0;     // 消费的缓冲区位置下标

// 定义信号量
sem_t full, empty;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```c
#define PTHREAD_SIZE 8
pthread_t pthreads[PTHREAD_SIZE];        // 所有线程数组
char pthreads_flag[PTHREAD_SIZE];        // 总的进程
int produce_num[PTHREAD_SIZE];           // 生产的产品号
int delay_times[PTHREAD_SIZE];       // 线程创建后申请资源的延迟时间
int op_times[PTHREAD_SIZE ];         // 进行生产或消费操作的时间
int pthread_index[PTHREAD_SIZE];     // 线程的下标，保证所有线程在运行时能正确获取自己的下标
```

初始化信号量等数据（pthread_index 是为了保证在程序运行时各线程能正确获取自己的序号）

```c
// 对信号量等进行初始化
void init_sems() {
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    pthread_mutex_init(&mutex, NULL);
    int i = 0;
    for (i = 0; i < PTHREAD_SIZE; i++) {
        pthread_index[i] = i;
    }
}
```

根据课本伪代码，写出以下函数：
判断是否合法后决定是否插入或删除缓冲区数据（由于本次实验要求与课本的有所差异，因此这两个函数其实也是可以省略不写的，但这里写上可以检测是否程序运行出错，比如是否在缓冲区满位时插入数据或在缓冲区空时取出数据等错误操作）：

```c
int insert_item(buffer_item item) {
    if (buffer_total_num < BUFFER_SIZE) {
        buffer[(buffer_index+buffer_total_num)%BUFFER_SIZE] = item;
        ++buffer_total_num;
        return 0;
    }
    return -1;
}

int remove_item(buffer_item *item) {
    if (buffer_total_num > 0 && buffer[buffer_index] == *item) {
        buffer_index = (buffer_index + 1) % BUFFER_SIZE;
        --buffer_total_num;
        return 0;
    }
    if (buffer_total_num == 0) {
        printf("buffer_total_num = 0\n");
    }
    if (buffer[buffer_index] != *item) {
        printf("buffer[buffer_index] = %d, *item = %d\n", buffer[buffer_index], *item);
    }
    return -1;
}
```

生产者函数：
在线程开始时打印出”applying”；
如果 empty 为 0，即缓冲区满位时，生产者不生产，生产者生产时利用 mutex 信号量保证同个时间只有该生产者能够进行生产，保证互斥；

```c
void *producer(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d producer is applying\n", i+1);

    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    printf("%d producer starts\n", i+1);
    sleep(op_times[i]);
    printf("%d producer ends\n", i+1);
    if (insert_item(produce_num[i])) {
        printf("report pro error condition\n");
        exit(1);
    }
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}
```

同上，消费者函数：

在线程开始时打印出”applying”；
如果 full 为 0，即缓冲区全为空位时，生产者不生产，生产者生产时利用 mutex 信号量保证同个时间只有该消费者能够进行消费，保证互斥；

```
void *consumer(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d consumer is applying\n", i+1);

    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    printf("%d consumer starts\n", i+1);
    sleep(op_times[i]);
    buffer_item removed_item = buffer[buffer_index];
    if (remove_item(&removed_item)) {
        printf("report con error condition\n");
        exit(1);
    }
    else
        printf("%d consumer ends\n", i+1);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
}
```

main 函数：
1.先判断输入命令行参数；

```
if (argc != 2) {
    printf("Usage: ./a.out [filename]\n");
    exit(0);
}
```

2.读取文件数据

```
// 读取文件数据
FILE *fin;
if ((fin = fopen(argv[1], "r")) == NULL) {
    printf("Open fail\n");
    exit(1);
}

int p_num = 0;
int c_num = 0;
while (!feof(fin)) {
    int index, delay_time, op_time, produce_number;
    char flag;
    fscanf(fin, "%d %c %d %d", &index, &flag, &delay_time, &op_time);
    pthreads_flag[index-1] = flag;
    delay_times[index-1] = delay_time;
    op_times[index-1] = op_time;
    if (flag == 'P') {
        fscanf(fin, "%d\n", &produce_number);
        produce_num[index-1] = produce_number;
        ++p_num;
    }
    else {
        fscanf(fin, "\n");
        ++c_num;
    }
}
fclose(fin);
```

3.初始化信号量

```
// 2.初始化函数
init_sems();
```

4.创建进程

```
int i, all_num = p_num + c_num;
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'P') {
        if (pthread_create(&pthreads[i], NULL, producer, (void*)(&pthread_index[i]))
            printf("producers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d producers pthread is created\n", i+1);
    }
    else {
        if (pthread_create(&pthreads[i], NULL, consumer, (void*)(&pthread_index[i]))
            printf("consumers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d consumers pthread is created\n", i+1);
    }
}
```

5.运行退出

```
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'P') {
        if (pthread_join(pthreads[i], NULL)) {
            printf("producers %d pthread join error\n", i);
            exit(1);
        }
    }
    else {
        if (pthread_join(pthreads[i], NULL)) {
            printf("consumers %d pthread join error\n", i);
            exit(1);
        }
    }
}

pthread_mutex_destroy(&mutex);
return 0;
```

2. 程序运行结果截图及分析：

本次实验测试文件数据主要使用 ppt 给出的示例数据，在进行实验时观察缓冲区长度不同时的运行结果，由于示例数据总共有 3 个消费者和 3 个生产者，运行测试了缓冲区长度从 1 到 5 这几种情况：

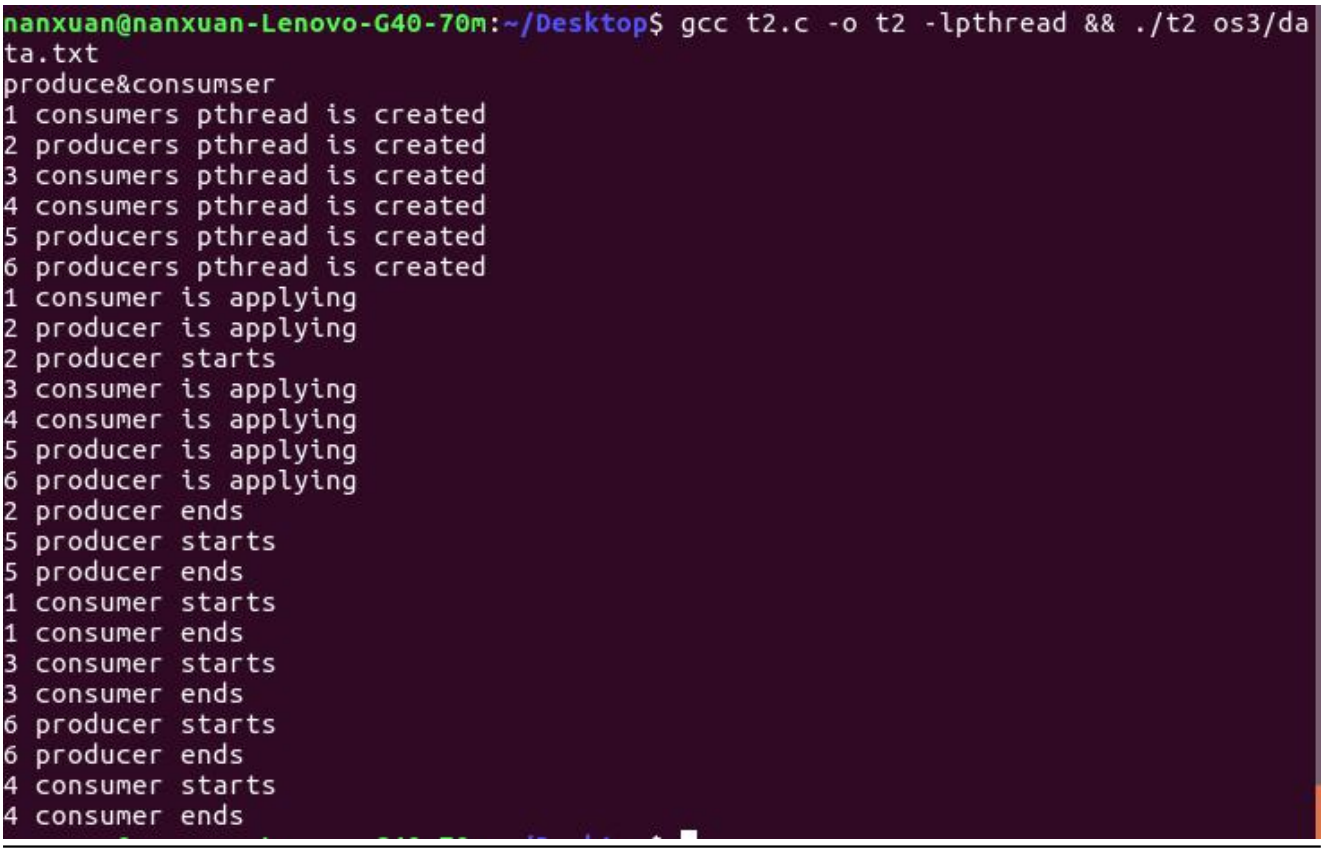
缓冲区长度<生产者/消费者数目

1.当缓冲区长度为 1 时：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop$ gcc t2.c -o t2 -lpthread && ./t2 os3/da
ta.txt
produce&consumser
1 consumers pthread is created
2 producers pthread is created
3 consumers pthread is created
4 consumers pthread is created
5 producers pthread is created
6 producers pthread is created
1 consumer is applying
2 producer is applying
2 producer starts
3 consumer is applying
4 consumer is applying
5 producer is applying
6 producer is applying
2 producer ends
1 consumer starts
1 consumer ends
5 producer starts
5 producer ends
3 consumer starts
3 consumer ends
6 producer starts
6 producer ends
4 consumer starts
4 consumer ends
```

可以看到由于缓冲区长度为 1，因此第一个**生产者 2** 生产结束后，缓冲区就已经满了，于是**消费者 1** 进行消费，消费结束后缓冲区全空，因此需要进行生产；因此实验结果是生产和消费过程交替进行，如上图所示。

2.当缓冲区长度为 2 时：



缓冲区数量为 2 时，最多连续进行两次生产就要进行消费；最多连续进行两次消费就要进行生产

缓冲区长度>=生产者/消费者数目时，就没有像上面的情况那样明显看出缓冲区长度对消费和生产行为的影响，可以 3 个生产者线程全部完成后再进行消费，也可以生产进行一次或两次后进行消费行为（只要缓冲区不为空）

3.当缓冲区长度为 3 时：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop$ gcc t2.c -o t2 -lpthread && ./t2 os3/da
ta.txt
produce&consumser
1 consumers pthread is created
2 producers pthread is created
3 consumers pthread is created
4 consumers pthread is created
5 producers pthread is created
6 producers pthread is created
1 consumer is applying
2 producer is applying
2 producer starts
3 consumer is applying
4 consumer is applying
5 producer is applying
6 producer is applying
2 producer ends
5 producer starts
5 producer ends
6 producer starts
6 producer ends
1 consumer starts
1 consumer ends
3 consumer starts
3 consumer ends
4 consumer starts
4 consumer ends
```

4.当缓冲区长度为 4 时：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop$ gcc t2.c -o t2 -lpthread && ./t2 os3/da
ta.txt
produce&consumser
1 consumers pthread is created
2 producers pthread is created
3 consumers pthread is created
4 consumers pthread is created
5 producers pthread is created
6 producers pthread is created
1 consumer is applying
2 producer is applying
2 producer starts
3 consumer is applying
4 consumer is applying
5 producer is applying
6 producer is applying
2 producer ends
1 consumer starts
1 consumer ends
6 producer starts
6 producer ends
5 producer starts
5 producer ends
3 consumer starts
3 consumer ends
4 consumer starts
4 consumer ends
```

5.当缓冲区长度为 5 时：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop$ gcc t2.c -o t2 -lpthread && ./t2 os3/da
ta.txt
produce&consumser
1 consumers pthread is created
2 producers pthread is created
3 consumers pthread is created
4 consumers pthread is created
5 producers pthread is created
6 producers pthread is created
1 consumer is applying
2 producer is applying
2 producer starts
3 consumer is applying
4 consumer is applying
5 producer is applying
6 producer is applying
2 producer ends
5 producer starts
5 producer ends
6 producer starts
6 producer ends
1 consumer starts
1 consumer ends
3 consumer starts
3 consumer ends
4 consumer starts
4 consumer ends
```

可以看到上面的运行结果都是在一个生产者或消费者在生产或消费结束之后，其他线程才开始进行操作，线程与线程之间的生产或消费过程是互斥的。

二、读者写者问题

1. 实验要求：

在 Linux 环境下，创建一个进程，此进程包含 n 个线程。用这 n 个线程来表示 n 个读者或写者。每个线程按相应测试数据文件(后面有介绍)的要求进行读写操作。用信号量机制分别实现读者优先和写者优先的读者-写者问题。

读者-写者问题的读写操作限制(仅读者优先或写者优先)：

1)写-写互斥，即不能有两个写者同时进行写操作。

2)读-写互斥，即不能同时有一个线程在读，而另一个线程在写。

3)读-读允许，即可以有一个或多个读者在读。

读者优先的附加限制：如果一个读者申请进行读操作时已有另一个读者正在进行读操作，则该读者可直接开始读操作。

写者优先的附加限制：如果一个读者申请进行读操作时已有另一写者在等待访问共享资源，则该读者必须等到没有写者处于等待状态后才能开始读操作。

运行结果显示要求：要求在每个线程创建、发出读写操作申请、开始读写操作和结束读写操作时分别显示一行提示信息，以确定所有处理都遵守相应的读写操作限制。

2. 实验测试数据格式：

下面是一个测试数据文件的例子：

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

注意：在创建数据文件时，由于涉及到文件格式问题，最好在记事本中手工逐个键入数据，而不要拷贝粘贴数据，否则，本示例程序运行时可能会出现不可预知的错误。

3. 实验内容

1）读者优先

读者优先指的是除非有写者在写文件，否则读者不需要等待。所以可以用一个整型变量 read_count 记录当前的读者数目，用于确定是否需要释放正在等待的写者线程(当 read_count=0 时，表明所有的读者读完，需要释放写者等待队列中的一个写者)。每一个读者开始读文件时，必须修改 read_count 变量。因此需要一个互斥对象 mutex 来实现对全局变量 read_count 修改时的互斥。

另外，为了实现写-写互斥，需要增加一个临界区对象 write。当写者发出写请求时，必须申请临界区对象的所有权。通过这种方法，也可以实现读-写互斥，当 read_count=1 时(即第一个读者到来时)，读者线程也必须申请临界区对象的所有权。

当读者拥有临界区的所有权时，写者阻塞在临界区对象 write 上。当写者拥有临界区的所有权时，第一个读者判断完“read_count==1”后阻塞在 write 上，其余的读者由于等待对 read_count 的判断，阻塞在 mutex 上。

2）写者优先

写者优先与读者优先类似。不同之处在于一旦一个写者到来，它应该尽快对文件进行写操作，如果有一个写者在等待，则新到来的读者不允许进行读操作。为此应当添加一个整型变量 write_count，用于记录正在等待的写者的数目，当 write_count=0 时，才可以释放等待的读者线程队列。

为了对全局变量 write_count 实现互斥，必须增加一个互斥对象 mutex3。

为了实现写者优先，应当添加一个临界区对象 read，当有写者在写文件或等待时，读者必须阻塞在 read 上。读者线程除了要对全局变量 read_count 实现操作上的互斥外，还必须有一个互斥对象对阻塞 read 这一过程实现互斥。这两个互斥对象分别命名为 mutex1 和 mutex2。

4. 实验过程

1）读者优先

a.对实验的理解以及代码：

对全局变量如进程和信号量进行声明：

fmutex 保证读写互斥以及写写互斥

```
#define THREAD_NUM 20
pthread_t pthreads[THREAD_NUM];
char pthreads_flag[THREAD_NUM];
int delay_times[THREAD_NUM];
int op_times[THREAD_NUM];
int pthread_index[THREAD_NUM];    // 线程的下标，保证所有线程在运行时能正确获取自己的下标

int reader_count = 0;   // 当前读者数目
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥对象控制对read_count修改
sem_t fmutex;            // 写写互斥和读写互斥的临界对象
sem_t wmutex;           // 阻塞write进程
```

读函数，在进行读操作之前对 read_count 的修改，用 mutex 保证是互斥；reader_count == 0 时，等待某个正在进行写操作的写者完成操作后释放 fmutex

```
void *read_file(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d reader is applying\n", i+1);

    pthread_mutex_lock(&mutex); // 保证对全局read_count修改是互斥的
    if (reader_count == 0) {    // 等待写者写完释放fmutex
        sem_wait(&fmutex);
    }
    ++reader_count;
    pthread_mutex_unlock(&mutex);

    // read
    printf("%d reader starts reading\n", i+1);
    sleep(op_times[i]);
    printf("%d reader ends reading\n", i+1);

    pthread_mutex_lock(&mutex);
    --reader_count;
    if (reader_count == 0) {    // 所有读者读完了，写者可以写了
        sem_post(&fmutex);
    }
    pthread_mutex_unlock(&mutex);
    return (NULL);
}
```

写操作函数，使用两个互斥变量保证读操作优先获取 fmutex：

```c
void *write_file(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d writer is applying\n", i+1);
    // 把许多写线程阻塞在这里，保证读操作会比写操作优先获得释放的fmutex
    sem_wait(&wmutex);
    // 保证每次阻塞在fmutex的写进程只有一个，只有一个写进程等待fmutex的释放
    sem_wait(&fmutex);
    // write
    sleep(op_times[i]);
    printf("%d writer ends writing\n", i+1);

    sem_post(&fmutex);
    sem_post(&wmutex);
    return (NULL);
}
```

信号量的初始化：

```c
void init_sems() {
    pthread_mutex_init(&mutex, NULL);
    sem_init(&fmutex, 0, 1);
    sem_init(&wmutex, 0, 1);
    int i = 0;
    for (i = 0; i < THREAD_NUM; i++) {
        pthread_index[i] = i;
    }
}
```

Main 函数：

命令行参数的获取，文件数据读取，信号量初始化，生产进程和运行结束等步骤都同上（生产者和消费者实验），在这里就不详细阐述，代码如下：

```c
int main(int argc, char *argv[]) {
    printf("reader first\n");
    // 线程序号 线程角色(R/W) 存放或取出操作的开始时间 操作持续时间
    // 判断命令行参数
    if (argc != 2) {
        printf("Usage: ./a.out [filename]\n");
        exit(0);
    }
    // 读取文件数据
    FILE *fin;
    if ((fin = fopen(argv[1], "r")) == NULL) {
        printf("Open fail\n");
        exit(1);
    }
int r_num = 0;
int w_num = 0;
while (!feof(fin)) {
    int index, delay_time, op_time;
    char flag;
    fscanf(fin, "%d %c %d %d\n", &index, &flag, &delay_time, &op_time);
    pthreads_flag[index-1] = flag;
    delay_times[index-1] = delay_time;
    op_times[index-1] = op_time;
    if (flag == 'R') {
        ++r_num;
    }
    else {
        ++w_num;
    }
}
fclose(fin);
```

```
// 初始化信号量
init_sems();

// 生成读者和写者进程
int i = 0, all_num = r_num + w_num;
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'R') {
        if (pthread_create(&pthreads[i], NULL, read_file, (void*)(&i))) {
            printf("readers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d reader pthread is created\n", i+1);
    }
    else {
        if (pthread_create(&pthreads[i], NULL, write_file, (void*)(&i))) {
            printf("writers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d writer pthread is created\n", i+1);
    }
}
```

```
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'R') {
        if (pthread_join(pthreads[i], NULL)) {
            printf("readers %d pthread join error\n", i);
            exit(1);
        }
    }
    else {
        if (pthread_join(pthreads[i], NULL)) {
            printf("writers %d pthread join error\n", i);
            exit(1);
        }
    }
}

pthread_mutex_destroy(&mutex);
return 0;
```

b.程序运行结果截图及分析：

```
nanxuan@nanxuan-Lenovo-G40-70m:/media/nanxuan/LENOVO/ppts/os3/result/2$ gcc t4.c
-o t4 -lpthread && ./t4 data2.txt
reader first
1 reader pthread is created
2 writer pthread is created
3 reader pthread is created
4 reader pthread is created
5 writer pthread is created
1 reader is applying
1 reader starts reading
2 writer is applying
3 reader is applying
3 reader starts reading
4 reader is applying
4 reader starts reading
5 writer is applying
3 reader ends reading
1 reader ends reading
4 reader ends reading
2 writer ends writing
5 writer ends writing
```

根据运行结果可以看出：

读者线程 **3** 和 **4** 比写者线程 **2** 和 **5** 先进行操作并先完成，且写线程之间互斥、读写线程互斥、读进程可以同时进行；实验结果符合读者优先

2）写者优先

a.对实验的理解以及代码：

对全局变量如进程和信号量进行声明：

fmutex 保证读写互斥以及写写互斥

```
#define THREAD_NUM 10
pthread_t pthreads[THREAD_NUM];
char pthreads_flag[THREAD_NUM];
int delay_times[THREAD_NUM];
int op_times[THREAD_NUM];
int pthread_index[THREAD_NUM];    // 线程的下标，保证所有线程在运行时能正确获取自己的下标

int reader_count = 0;   // 当前读者数目
int writer_count = 0;   // 当前写者数目
```

```
// 互斥对象控制对read_count修改
static pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
// 互斥对象控制对write_count修改
static pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;
sem_t fmutex;               // 对文件获取（读或写）的互斥对象
sem_t entermutex;           // 阻塞读者进程，保证只有写者优先拿到quemutex
sem_t quemutex;             // 阻塞读者进程
```

读函数：写线程优先获得 quemutex，因此写线程比其他读者线程优先对文件进行操作

```
void *read_file(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d reader is applying\n", i+1);

    // 很多读者在这里排队
    sem_wait(&entermutex);
    // 只有一个读者进入这里，保证该读者释放quemutex时，写者优先拿到quemutex
    // 临界区
    sem_wait(&quemutex);
    // 临界区
    pthread_mutex_lock(&mutex2);
    if (reader_count == 0) {    // 等待前面的写者写完释放fmutex
        sem_wait(&fmutex);
    }
    ++reader_count;
    pthread_mutex_unlock(&mutex2);
    sem_post(&quemutex);
    sem_post(&entermutex);

    // read
    printf("%d reader starts reading\n", i+1);
    sleep(op_times[i]);
    printf("%d reader ends reading\n", i+1);

    pthread_mutex_lock(&mutex2);
    --reader_count;
    if (reader_count == 0) {    // 读者读完了，写者可以写了
        sem_post(&fmutex);      // 释放时写者优先获取fmutex
    }
    pthread_mutex_unlock(&mutex2);
    return (NULL);
}
```

读函数：等待所有写线程结束后才开始进行读取

```
void *write_file(void *param) {
    int i = *((int*)(param));
    sleep(delay_times[i]);
    printf("%d writer is applying\n", i+1);

    pthread_mutex_lock(&mutex3);
    if (writer_count == 0) {    // 等待某个读者读完释放quemutex
        sem_wait(&quemutex);
    }
    ++writer_count;
    pthread_mutex_unlock(&mutex3);

    // write
    sem_wait(&fmutex);
    printf("%d writer starts writing\n", i+1);
    sleep(op_times[i]);
    printf("%d writer ends writing\n", i+1);
    sem_post(&fmutex);

    pthread_mutex_lock(&mutex3);
    --writer_count;
    if (writer_count == 0) {    // 所有写者写完了，释放quemutex，让读者去读
        sem_post(&quemutex);
    }
    pthread_mutex_unlock(&mutex3);
    return (NULL);
}
```

初始化各种变量：

```
void init_sems() {
    pthread_mutex_init(&mutex2, NULL);
    pthread_mutex_init(&mutex3, NULL);
    sem_init(&fmutex, 0, 1);
    sem_init(&quemutex, 0, 1);
    sem_init(&entermutex, 0, 1);
    int i = 0;
    for (i = 0; i < THREAD_NUM; i++) {
        pthread_index[i] = i;
    }
}
```

Main 函数：

命令行参数的获取，文件数据读取，信号量初始化，生产进程和运行结束等步骤都同上（生产者和消费者实验），在这里就不详细阐述，代码如下：

```
// 判断命令行参数
if (argc != 2) {
    printf("Usage: ./a.out [filename]\n");
    exit(0);
}
// 读取文件数据
FILE *fin;
if ((fin = fopen(argv[1], "r")) == NULL) {
    printf("Open fail\n");
    exit(1);
}
```

```
int r_num = 0;
int w_num = 0;
while (!feof(fin)) {
    int index, delay_time, op_time;
    char flag;
    fscanf(fin, "%d %c %d %d\n", &index, &flag, &delay_time, &op_time);
    pthreads_flag[index-1] = flag;
    delay_times[index-1] = delay_time;
    op_times[index-1] = op_time;
    if (flag == 'R') {
        ++r_num;
    }
    else {
        ++w_num;
    }
}
fclose(fin);

// 初始化信号量和随机数
init_sems();
```

```
// 生成读者和写者进程
int i = 0, all_num = r_num + w_num;
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'R') {
        if (pthread_create(&pthreads[i], NULL, read_file, (void*)(&pthread_index[i]))) {
            printf("readers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d reader pthread is created\n", i+1);
    }
    else {
        if (pthread_create(&pthreads[i], NULL, write_file, (void*)(&pthread_index[i]))) {
            printf("writers %d pthread create error\n", i);
            exit(1);
        }
        printf("%d writer pthread is created\n", i+1);
    }
}
```

```
for (i = 0; i < all_num; i++) {
    if (pthreads_flag[i] == 'R') {
        if (pthread_join(pthreads[i], NULL)) {
            printf("readers %d pthread join error\n", i);
            exit(1);
        }
    }
    else {
        if (pthread_join(pthreads[i], NULL)) {
            printf("writers %d pthread join error\n", i);
            exit(1);
        }
    }
}

pthread_mutex_destroy(&mutex2);
pthread_mutex_destroy(&mutex3);
return 0;
```

b.程序运行结果截图及分析：

```
-o t5 -lpthread && ./t5 data2.txt
writer first
1 reader pthread is created
2 writer pthread is created
3 reader pthread is created
4 reader pthread is created
5 writer pthread is created
1 reader is applying
1 reader starts reading
2 writer is applying
3 reader is applying
4 reader is applying
5 writer is applying
1 reader ends reading
2 writer starts writing
2 writer ends writing
5 writer starts writing
5 writer ends writing
3 reader starts reading
4 reader starts reading
3 reader ends reading
4 reader ends reading
```

根据运行结果可以看出：

写者线程 **2** 和 **5** 比写者线程 **3** 和 **4** 先进行操作并先完成，且写线程之间互斥、读写线程互斥、读进程可以同时进行；实验结果符合写者优先

5. 实验总结

通过这次实验我更加深入了解了同步互斥的相关概念。了解到互斥是在同一时间只运行一个访问者访问某一资源（少数情况允许多个访问者比如读取操作），但互斥无法决定或限制访问者的访问顺序；同步是指实现访问者对资源的有序访问，不同的访问者之间的操作是互斥的，也即是不能同时进行，只能在一个结束后才能进行另外一个。本次实验利用信号量实现多个访问者对同一资源进行有序访问。