操 作 系 统
实 验 报 告

实验名称：实验四 同步互斥问题

姓名：王迎旭

学号：16340226

实验名称：**同步互斥问题**

一、实验目的：

1. 用线程同步机制，实现生产者-消费者问题
2. 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

二、实验要求：

（1） 生产者与消费者问题

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：

```
do {
    ...
  // produce an item in nextp
    ...
  wait(empty);
  wait(mutex);
    ...
  // add nextp to buffer
    ...
  signal(mutex);
  signal(full);
}while (TRUE);
```

图 6.10 生产者进程结构

```
do {
  wait(full);
  wait(mutex);
    ...
  // remove an item from buffer to nextc
    ...
  signal(mutex);
  signal(empty);
    ...
  // consume the item in nextc
    ...
}while (TRUE);
```

图 6.11 消费者进程结构

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。

（2） 读者与写者问题

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。

三、实验过程:

Ⅰ、生产者与消费者问题

（1）明确目标程序的需求:

用线程同步机制，实现生产者-消费者问题

（2）搜索题目所涉及相关资料:

1、Mutex

1）Mutex 的三种类型

① 快速 mutex:一个线程必须在未锁时获得 mutex，否则必须等待。

② 递归 mutex:拥有 mutex 的线程可多次加锁而不必等待，但此时其他线程必须等待。

③ 错误检测 mutex:如果 mutex 已被锁，其他进程使用 pthread_mutex_lock()加锁时，将返回 EDEADLK

2）Mutex 类型选择

通过初始化参数选择类型。POSIX 线程库提供了更快速和直接初始化方法 :

① pthread_mutex_t fastmutex =
PTHREAD_MUTEX_INITIALIZER;//快速 mutex

② pthread_mutex_t recmutex =
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP; //递归 mutex

③ pthread_mutex_t errchkmutex =
PTHREAD_ERRORCECK_MUTEX_INITIALIZER_NP; //错误检测 mutex

3）Mutex 的初始化

线程的互斥量数据类型是 pthread_mutex_t。 使用前要对它进行初始化，有两种方法：

① 静态分配的互斥量的初始化：

```
static pthread_mutex_t mutex =
                    PTHREAD_MUTEX_INITIALIZER
```

② 动态分配的互斥量的初始化：

A、首先申请内存(malloc)，

B、再用 pthread_mutex_init 进行初始化。

C、释放时，必须先调用 pthread_mutex_destroy，而后再释放内存(free)。

4）可对 Mutex 进行操作的函数

① int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);//初始化 mutex.

② int pthread_mutex_lock(pthread_mutex_t *mutex);//对 mutex 的加锁，不成等待

③ int pthread_mutex_trylock(pthread_mutex_t *mutex);//尝试加锁，不成返回

④ int pthread_mutex_unlock(pthread_mutex_t *mutex);//解锁

⑤ int pthread_mutex_destroy(pthread_mutex_t *mutex);//销毁 mutex

5）线程对 Mutex 操作的性质

① 原子性。对 mutex 的加锁和解锁操作是原子的，一个线程进行 mutex 操作的过程中，其他线程不能对同一个 mutex 进行其他操作。

② 单一性。拥有 mutex 的线程除非释放 mutex，否则其他线程不能拥有此 mutex。

③ 非忙等待。等待 mutex 的线程处于等待状态，直到要等待的 mutex 处于未加锁状态，这时操作系统负责唤醒等待此 mutex 的线程。

2、POSIX 信号量

1）POSIX 信号量性质

① POSIX 信号量在多线程编程中可以起到同步或互斥的作用。用 POSIX 信号量可以实现传统操作系统 P、V 操作(即对应课本的 wait、signal)。

② 由于 POSIX 信号量不是内核负责维护，所以当进程退出后，POSIX 信号量自动消亡。

2）POSIX 信号量的含义

① sem_init()原型：int sem_init(sem_t *sem, int pshared, unsigned int value);

作用：创建信号量。sem 为指向信号量结构的一个指针；pshared 不为 0 时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value 给出了信号量的初始值。

② sem_post()

作用：相当于 signal 操作，释放资源。用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程被唤醒。

③ sem_wait()：

作用：相当于 wait 操作，即申请资源。这是一个阻塞的函数。测试所指定信号量的值,它的操作是原子的。若 sem>0，那么它减 1 并立即返回。若 sem==0，则睡眠直到 sem>0，此时立即减 1，然后返回。

④ sem_trywait ()

作用：是函数 sem_wait()的非阻塞版本。若信号量大于 0，它直接将信号量 sem 的值减一并返回 0；否则，它立即返回错误类型 EAGAIN。

⑤ sem_getvalue()

作用：获得信号量当前值

⑥ sem_destroy()

作用：用来释放信号量。

（3）消费者生产者程序设计

思路梳理（截图+文字叙述）

① 创建全局变量用于处理消费者与生产者的编号与其数目统计问题

```
// 生产者总数、消费者总数
int total_producer = 0, total_consumer = 0;
// 生产者存放的产品
int buffer[BUFFER_SIZE];
// 下一个生产者存放的产品、下一个消费者消费的产品
int nextP = 0, nextC = 0;
```

② 创建 POSIX 信号量用于处理互斥问题

```
// 记录有多少空位
sem_t empty;
// 记录有多少满位
sem_t full ;
//保护对缓冲区插入操作
sem_t mutex;
```

③ 创建 command 结构体按顺序存储 PPT 中所给的指令

```
struct command
{
    int pid;// 线程号
    char type;// 线程角色（P：生产者；C：消费者）
    int startTime;// 操作开始的时间
    int lastTime;// 操作的持续时间
    int num;// 生产者存放产品的编号
};
```

④ 创建生产者线程函数

1）将传入参数 param 转化为一个 command 的结构体，包含生产者的 pid(线程号),type(即是 P,线程角色:生产者),startTime（开始时间），lastTime（持续时间），num（生产者所生成的产品编号）。

2） 进入 while 循环：

a. 开始阻塞信号 empty，等待缓冲区是否有空位；

b. 睡眠 sleep 开始时间 startTime，阻塞 mutex 信号，修改缓冲区，把生成的编号为 data->num 的产品放入缓冲区，打印出已完成生产的信息，修改缓冲区指针。

c. 睡眠 sleep 持续时间 lastTime，释放信号量 mutex 和 full，允许消费者对缓冲区进行修改和取出产品。

3）退出循环。

```
36  // 生产者线程
37  void *producer(void *param) {
38      struct command* data = (struct command*)param;
39
40      while (true) {
41          sem_wait(&empty);
42          sleep(data->startTime);
43          sem_wait(&mutex);
44
45          buffer[nextP] = data->num;
46          cout << "Producer No." << data->pid
47              << " produces " << "product No." << data->num << endl;
48          nextP = (nextP + 1) % BUFFER_SIZE;
49
50          sleep(data->lastTime);
51          sem_post(&mutex);
52          sem_post(&full);
53
54          pthread_exit(0);
55      }
56  }
```

⑤ 消费者线程函数

1）将传入参数 param 转化为一个 command 的结构体（注意此时消费者没有 num），包含生产者的 pid（线程号），type（即是 C，线程角色：消费者），startTime（开始时间），lastTime（持续时间）。

2）进入 while 循环：

a. 开始阻塞信号 full，等待缓冲区是否有产品；

b. 睡眠 sleep 开始时间 startTime，阻塞 mutex 信号，修改缓冲区，取出消费编号为 buffer[nextC]的产品，打印出已完成消费的信息，修改缓冲区指针，缓冲区减少一个产品；

c. 睡眠 sleep 持续时间 lastTime，释放 mutex 和 empty，允许生产者对缓冲区进行修改和向其中存放产品。

3）退出循环。

```
void *consumer(void *param) {
    struct command* data = (struct command*)param;

    while (true) {
        sem_wait(&full);
        sleep(data->startTime);
        sem_wait(&mutex);

        cout << "Consumer No." << data->pid
            << " consumes " << "product No." << buffer[nextC] << endl;
        buffer[nextC] = 0;
        nextC = (nextC + 1) % BUFFER_SIZE;

        sleep(data->lastTime);
        sem_post(&mutex);
        sem_post(&empty);

        pthread_exit(0);
    }
}
```

⑥ 创建 ifstream 读写流用于读取 txt 文件中的指令数据，同时完成对指令的解码并按顺序赋值给指令结构体。

这里由于所给的指令中有 4 位的也有 5 位的，所以就选择了用数据流进行操作，来保证指令读取的正确性。

```
    ifstream in("/home/dell/Desktop/data.txt");
    if (!in.is_open()) {
        exit(1);
    }

    char buffer[256];
    int j = 0;
    char ch;
    while (!in.eof()) {
        in.read(&ch, 1);
        if (ch != ' ' && ch != '\n' && ch != '\r')
            buffer[j++] = ch;
    }
    j = 0;

    for (int i = 0; i < total; i++) {
        information[i].pid = buffer[j++] - '0';
        information[i].type = buffer[j++];
        information[i].startTime = buffer[j++] - '0';
        information[i].lastTime = buffer[j++] - '0';
        if (information[i].type == 'P')
            information[i].num = buffer[j++] - '0';
    }
```

⑦ 根据指令集的指示，完成线程的创建并为每个线程分配对应的行为。

```
119     for (int i = 0; i < total; i++) {
120         if (information[i].type == 'P') {
121             total_producer++;
122             cout << "Create Producer No." << information[i].pid << endl;
123             pthread_create(&Pid[i], NULL, producer, &information[i]);
124         }
125 
126         if (information[i].type == 'C') {
127             total_consumer++;
128             cout << "Create Consumer No." << information[i].pid << endl;
129             pthread_create(&Pid[i], NULL, consumer, &information[i]);
130         }
131     }
```

⑧ 非阻塞式调用线程

```
133     for (int i = 0; i < total; i++) {
134         pthread_join(Pid[i], NULL);
135     }
```

⑨ 删除信号量，终止程序

```
137     sem_destroy(&full);
138     sem_destroy(&empty);
139     sem_destroy(&mutex);
```

（4）程序测试与结果分析

程序测试：

```
[dell@localhost Desktop]$ g++ -g question.cpp -o question -lpthread
[dell@localhost Desktop]$ ./question 6
Create Consumer No.1
Create Producer No.2
Create Consumer No.3
Create Consumer No.4
Create Producer No.5
Create Producer No.6
Producer No.2 produces product No.1
Producer No.5 produces product No.2
Producer No.6 produces product No.3
Consumer No.1 consumes product No.1
Consumer No.3 consumes product No.2
Consumer No.4 consumes product No.3
```

结果分析：

从运行结果上我们可以看到，三个生产者进程完成自己的任务之后，三个消费者才开始自己的消费过程，但是为什么是如此？我们不妨从指令集上

一条一条来分析。

指令集的指令如下：

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

① 第一条指令，消费者线程请求访问，但是这时候由于 **full == 0**，所以这个线程就被阻塞在 wait（full）这里，无法继续往下运行。

② 第二条指令，生产者线程请求访问，这时候由于程序开始时候 empty 被初始化为 5，所以可以越过 wait（empty）这一步，empty - 1 后往下走，随后等待 4 毫秒，在进行到 wait（mutex）这一步时候，由于 mutex 是 1，所以可以越过 wait（mutex）这一步，mutex - 1 后继续往下走，这时候由于信号量 mutex = 0 ，所以其他线程也就无权访问共享变量，也就保证了互斥性。往下走的过程中，生产了编号为 1 的产品，**第一个输出提示信息**。随后等待 5 毫秒，发出信号 post（mutex），此时 mutex -> 1，接着 post（full），full ++ ，这一步后也就保证了消费者可以进行消费。

③ 第三条指令第四条指令，仍然是和第一条一样卡在 wait（full）这里；这两条指令卡在这里是因为，第二条指令尚未完成时候，第三条第四条指令已经在 wait（full）了，full 仍然是 0，所以就被阻塞。

④ 第五条指令，随着第二条指令的 post（mutex）完成，第五条指令开始进入像第二条指令一样的运行过程，所以**第二个输出提示信息**。

⑤ 第六条指令，由于线程调用的顺序是先 No.5 后 No.6 所以，生产者 5 的生产过程早于生产者 6。随后第五条指令完成任务后，紧接着就把访问权交给了第六条指令，进而完成创建任务，所以**第三个输出提示信息**。

⑥ 第六条指令完成后，当 mutex 重新被设置为 1 后，第一条指令率先夺取到访问权，于是就完成消费过程，所以**第四个输出提示信息**；剩下的第三条第四条指令过程与此类似，都是按照被创建的先后顺序完成信息输出。

II、读者与写者问题

Part 1 读者优先

（1）明确目标程序的需求

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）读者优先程序设计

思路梳理（截图+文字叙述）

① 创建全局变量用于记录读者的数目以及读取的内容

```
13  //记录数据的读取情况
14  int data = 0;
15  // 记录读者的数量
16  int read_count = 0;
```

② 创建 POSIX 信号量用于处理互斥问题

```
18  // 临界区对象writer用于改变读者数量
19  sem_t writer,
20  // 临界区对象mutex用于阻塞读写操作
21  sem_t mutex;
```

③ 创建 command 结构体按顺序存储 PPT 中所给的指令

```
23  struct command
24  {
25      // 线程号
26      int pid;
27      // 线程角色 (R: 读者; W: 写者)
28      char type;
29      // 操作开始的时间
30      int startTime;
31      // 操作的持续时间
32      int lastTime;
33  };
```

④ 创建读写函数用于显示此过程中处理的数据

```
35  // 写操作函数
36  void write() {
37      int rd = rand() % MAX_RAND;
38      cout << "Write data " << rd << endl;
39      data = rd;
40  }
41  // 读操作函数
42  void read() {
43      cout << "Read data " << data << endl;
44  }
```

⑤ 创建写者线程函数：

1）将传入参数 param 转化为一个 command 的结构体，包含写者的 pid（线程号），type（即是 W，线程角色：写者），startTime（开始时间），lastTime（持续时间）。

2）进入 while 循环

a. 睡眠 sleep 开始时间 startTime；

b. 打印出线程号为 pid 的线程正在申请资源信息，并开始阻塞 writer；

c. 打印出开始写的信息，并执行写操作函数 write()，显示写入的信息；

d. 睡眠 sleep 持续时间 lastTime； 打印出结束写的信息并释放资源，解除阻塞信号 writer；

3）退出循环。

```
46  // 写者线程
47  void *Writer(void *param) {
48      struct command* c = (struct command*)param;
49      while (true) {
50          sleep(c->startTime);
51          cout << "Writer(the " << c->pid << " pthread) requests to write." << endl;
52          sem_wait(&writer);
53
54          cout << "Writer(the " << c->pid << " pthread) begins to write." << endl;
55          write();
56
57          sleep(c->lastTime);
58          cout << "Writer(the " << c->pid << " pthread) stops writing." << endl;
59          sem_post(&writer);
60
61          pthread_exit(0);
62      }
63  }
```

⑥ 创建读者线程函数：

1）将传入参数 param 转化为一个 command 的结构体，包含读者的 pid（线程号），type（即是 R，线程角色：读者），startTime（开始时间），lastTime（持续时间）。

2）进入 while 循环

a. 睡眠 sleep 开始时间 startTime;

b. 打印出线程号为 pid 的线程正在申请资源信息，并开始阻塞互斥信号 mutex； 读者数量 read_count 加 1，检查 read_count，如果为 1，那么阻塞 writer。判断后释放互斥信号 mutex。

c. 打印出开始读取的信息并执行读函数 read()；睡眠 sleep 持续时间 lastTime;

d. 打印出结束写的信息并释放资源，并阻塞互斥信号 mutex； 读者数量 read_count 减 1，检查 read_count，如果为 0，那么释放信号 writer。判断后解除互斥信号 mutex。

3）退出循环

```
65  // 读者线程
66  void *reader(void *param) {
67      struct command* c = (struct command*)param;
68      while (true) {
69          sleep(c->startTime);
70          cout << "Reader(the " << c->pid << " pthread) requests to read." << endl;
71          sem_wait(&mutex);
72
73          read_count++;
74          if (read_count == 1)
75              sem_wait(&writer);
76          sem_post(&mutex);
77
78          cout << "Reader(the " << c->pid << " pthread) begins to read." << endl;
79          read();
80
81          sleep(c->lastTime);
82          cout << "Reader(the " << c->pid << " pthread) stops reading." << endl;
83          sem_wait(&mutex);
84
85          read_count--;
86          if (read_count == 0)
87              sem_post(&writer);
88          sem_post(&mutex);
89
90          pthread_exit(0);
91      }
92  }
```

⑦ 进入 main 函数：

a. 创建信号量并完成初始化
b. 创建线程结构体
c. 完成指令的输入

```
96      int number_person = atoi(argv[1]);
97      sem_init(&writer, 0, 1);
98      sem_init(&mutex, 0, 1);
99      struct command information[number_person];
100     pthread_t pid[number_person];
101
102     for (int i = 0; i < number_person; i++) {
103         cin >> information[i].pid >> information[i].type
104                 >> information[i].startTime >> information[i].lastTime;
105     }
```

⑧ 顺序遍历指令集完成线程的创建：

```
107     for (int i = 0; i < number_person; i++) {
108         if (information[i].type == 'R') {
109             cout << "Create a reader pthread, it's the " << information[i].pid << " pthread."
110             pthread_create(&pid[i], NULL, reader, &information[i]);
111         }
112
113         if (information[i].type == 'W') {
114             pthread_create(&pid[i], NULL, Writer, &information[i]);
115             cout << "Create a writer pthread, it's the " << information[i].pid << " pthread."
116         }
117     }
```

⑨ 非阻塞调用线程随后销毁信号量结束程序：

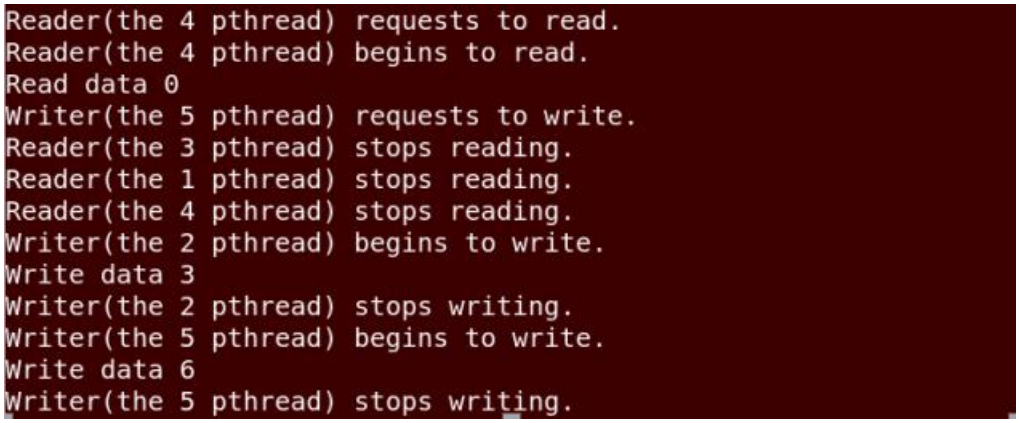
```
119     for (int i = 0; i < number_person; i++) {
120         pthread_join(pid[i], NULL);
121     }
122
123     sem_destroy(&writer);
124     sem_destroy(&mutex);
```

（3）程序测试与结果分析

程序测试：+

```
[dell@localhost Desktop]$ ./question 5
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3

Create a reader pthread, it's the 1 pthread.
Create a writer pthread, it's the 2 pthread.
Create a reader pthread, it's the 3 pthread.
Create a reader pthread, it's the 4 pthread.
Create a writer pthread, it's the 5 pthread.
Reader(the 1 pthread) requests to read.
Reader(the 1 pthread) begins to read.
Read data 0
Writer(the 2 pthread) requests to write.
Reader(the 3 pthread) requests to read.
Reader(the 3 pthread) begins to read.
Read data 0
```



结果分析：

读者优先就意味着当有读者写者同时请求进行线程操作时候，读者要先进行操作，所以程序的运行结果中

这一段：读者线程 2 与写者线程 3 同时请求操作，但是读者线程 2 先进行操作，同时这个过程中也有个读者线程 4 请求操作，当这两个线程读操作完成之后，发出了 stop 指令之后，写者线程 2 才开始进行写操作，这也就体现了读者优先的操作，也就说明了程序的正确性。

Part 2 写者优先

（1）明确目标程序的需求

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 mutex2。

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。同样，有读者读时，写者必须等待。于是，必须有一个互斥对象 RW_mutex 来实现这个互斥。

有写者在写时，写者必须等待。

读者线程要对全局变量 read_count 实现操作上的互斥，必须有一个互斥对象命名为 mutex1。

（2）写者优先程序设计

① 创建全局变量用于记录读者写者的数目以及读取的内容

```
13  //记录数据的读取情况
14  int data = 0;
15  // 记录读者的数量
16  int read_count = 0;
17  // 记录写者的数量
18  int write_count = 0;
```

② 创建 POSIX 信号量用于处理互斥问题

```
20  // writeAccess:对全局变量write_count实现互斥
21  sem_t writeAccess ;
22  // readAccess: 对全局变量read_count实现互斥
23  sem_t readAccess ;
24  // mutexR: 对阻塞read这一过程实现互斥
25  sem_t mutexR ;
26  // mutexW: 当有写者在写文件或者等待时，读者阻塞在mutexW上
27  sem_t mutexW;
```

③ 创建 command 结构体按顺序存储 PPT 中所给的指令

```
29  struct command
30  {
31      int pid;// 线程号
32      char type;// 线程角色（R：读者；W：写者）
33      int startTime;// 操作开始的时间
34      int lastTime;// 操作的持续时间
35  };
```

④ 创建读写函数用于显示此过程中处理的数据

```
37  // 写操作函数
38  void write() {
39      int rd = rand() % MAX_RAND;
40      cout << "Write data " << rd << "." << endl;
41      data = rd;
42  }
43
44  // 读操作函数
45  void read() {
46      cout << "Read data " << data << "." << endl;
47  }
```

⑤ 创建写者线程函数：

1）将传入参数 param 转化为一个 command 的结构体，包含写者的 pid（线程号），type（即是 W，线程角色：写者），startTime（开始时间），lastTime（持续时间）。

2）进入 while 循环：

a. 睡眠 sleep 开始时间 startTime； 打印出线程号为 pid 的线程正在申请资源信息，并开始阻塞 writerAccess；

b. 写者数量 write_count 加 1，检查 write_count，如果为 1，那么阻塞 mutexR。判断后释放互斥信号 writeAccess； 阻塞 mutexW 信号，打印出开始写的信息并执行写函数 write()；

c. 睡眠 sleep 持续时间 lastTime；

d. 打印出结束写的信息并释放资源，阻塞互斥信号 mutexW；

e. 阻塞互斥信号 writeAccess，读者数量 write_count 减 1，检查 write_count，如果为 0，那么释放信号 mutexR。判断后解除互斥信号 writeAccess。

3）退出循环。

```
// 写者线程
void *writer(void *param) {
    struct command* c = (struct command*)param;
    while (true) {
        sleep(c->startTime);
        cout << "Writer(the " << c->pid << " pthread) requests to write." << endl;
        sem_wait(&writeAccess);

        write_count++;
        if (write_count == 1)
            sem_wait(&mutexR);
        sem_post(&writeAccess);

        sem_wait(&mutexW);
        cout << "Writer(the " << c->pid << " pthread) begins to write." << endl;
        write();

        sleep(c->lastTime);
        cout << "Writer(the " << c->pid << " pthread) stops writing." << endl;
        sem_post(&mutexW);

        sem_wait(&writeAccess);
        write_count--;
        if (write_count == 0)
            sem_post(&mutexR);
        sem_post(&writeAccess);

        pthread_exit(0);
    }
}
```

⑥ 创建读者线程函数：

1）将传入参数 param 转化为一个 command 的结构体，包含读者的 pid（线程号），type（即是 W，线程角色：读者），startTime（开始时间），lastTime（持续时间）。

2）进入 while 循环：

a. 睡眠 sleep 开始时间 startTime；

b. 打印出线程号为 pid 的线程正在申请资源信息，并开始阻塞 mutexR 检查读者是否允许读取，同时也阻塞信号 readAccess；

c. 读者数量 read_count 加 1，检查 read_count，如果为 1，那么阻塞 mutexW。判断后释放互斥信号 readAccess 和 mutexR；打印出开始读取的信息并执行读函数 read()；

d. 睡眠 sleep 持续时间 lastTime；

e. 打印出结束写的信息；

f. 阻塞互斥信号 readAccess，读者数量 read_count 减 1，检查 read_count，如果为 0，那么释放信号 mutexW。判断后解

除互斥信号 readAccess。

3）退出循环。

```
// 读者线程
void *reader(void *param) {
    struct command* c = (struct command*)param;
    while (true) {
        sleep(c->startTime);
        cout << "Reader(the " << c->pid << " pthread) requests to read." << endl;
        sem_wait(&mutexR);
        sem_wait(&readAccess);

        read_count++;
        if (read_count == 1)
            sem_wait(&mutexW);
        sem_post(&readAccess);
        sem_post(&mutexR);

        cout << "Reader(the " << c->pid << " pthread) begins to read." << endl;
        read();

        sleep(c->lastTime);
        cout << "Reader(the " << c->pid << " pthread) stops reading." << endl;

        sem_wait(&readAccess);
        read_count--;
        if (read_count == 0)
            sem_post(&mutexW);
        sem_post(&readAccess);

        pthread_exit(0);
    }
}
```

⑦ 进入 main 函数：

d. 创建信号量并完成初始化
e. 创建线程结构体
f. 完成指令的输入

```
    int number_person = atoi(argv[1]);
    sem_init(&writeAccess, 0, 1);
    sem_init(&readAccess, 0, 1);
    sem_init(&mutexR, 0, 1);
    sem_init(&mutexW, 0, 1);

    struct command information[number_person];
    pthread_t pid[number_person];

    for (int i = 0; i < number_person; i++) {
        cin >> information[i].pid >> information[i].type
            >> information[i].startTime >> information[i].lastTime;
    }
```

⑧ 顺序遍历指令集完成线程的创建：

```
127     for (int i = 0; i < number_person; i++) {
128         if (information[i].type == 'R') {
129             cout << "Create a reader pthread-No." << information[i].pid << " pthread." << endl;
130             pthread_create(&pid[i], NULL, reader, &information[i]);
131         }
132 
133         if (information[i].type == 'W') {
134             pthread_create(&pid[i], NULL, writer, &information[i]);
135             cout << "Create a writer pthread-No." << information[i].pid << " pthread." << endl;
136         }
137     }
```

⑨ 非阻塞调用线程随后销毁信号量结束程序：

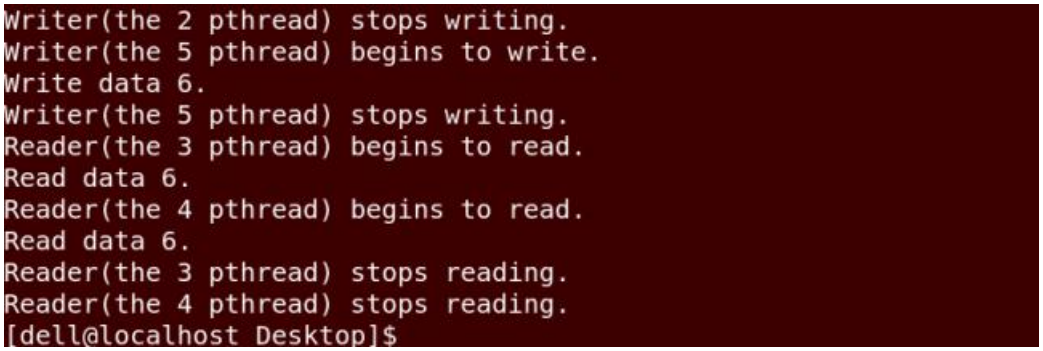
```
139     for (int i = 0; i < number_person; i++) {
140         pthread_join(pid[i], NULL);
141     }
142 
143     sem_destroy(&writeAccess);
144     sem_destroy(&readAccess);
145     sem_destroy(&mutexW);
146     sem_destroy(&mutexR);
```

（3）程序测试与结果分析

程序测试：

```
[dell@localhost Desktop]$ g++ question3.cpp -o question -lpthread
[dell@localhost Desktop]$ ./question 5
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3

Create a reader pthread-No.1 pthread.
Create a writer pthread-No.2 pthread.
Create a reader pthread-No.3 pthread.
Create a reader pthread-No.4 pthread.
Create a writer pthread-No.5 pthread.
Reader(the 1 pthread) requests to read.
Reader(the 1 pthread) begins to read.
Read data 0.
Writer(the 2 pthread) requests to write.
Reader(the 3 pthread) requests to read.
Reader(the 4 pthread) requests to read.
Writer(the 5 pthread) requests to write.
Reader(the 1 pthread) stops reading.
Writer(the 2 pthread) begins to write.
Write data 3.
```



结果分析：

读者优先就意味着当有读者写者同时请求进行线程操作时候，先者要先进行操作，所以程序的运行结果中

这一段：写者线程 2、读者线程 3、读者线程 4、写者线程 5 同时请求操作，读者线程 1 由于是程序开始进行了读操作，所以率先终止，紧接着写者线程 2 进行写操作，结束写操作，写者进程 5 进行写操作，结束写操作，当这两个写线程操作完成之后，发出了 stop 指令之后，读者线程 3 才开始进行毒操作，这也就体现了写者优先的操作，也就说明了程序的正确性。

四、实验心得：

1. 实验之前要理清楚老师给的资料中的内容，不要盲目的下手先做，否则只会是浪费了时间还没有什么收获。

2. 关于线程之间的互斥关系，通过这一次实验之后，也有了一些比较清楚的理解，同时也是对共享数据区的操作有了更熟练的掌握。

3. 同时实验中也处理了多线程之间的互斥和同步问题，再次遇到这类问题我们可以通过使用线程同步和互斥信号量来解决。

4. 这次实验也是自己第一次使用互斥信号量，相信在以后的学习中，会越发熟练掌握互斥信号量的使用。