北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **典型同步问题验证性实验**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年10月12日**

目录

1. 开发运行环境和工具

操作系统	Linux内核版本	处理器	GCC版本
Deepin 20.9	5.18.17-amd64-desktop-hwe	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (8核，非大小核)	gcc (Uos 8.3.0.3-3+rebuild) 8.3.0

- **其他工具**:
  - **编辑**：VSCode (version 1.83.1)
  - **编译和运行**：Terminal

2. 实验过程、分析和结论

2.1. 阅读算法

算法实现的思路写在以下的代码注释中。

2.1.1. producer-consumer

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define n 5

// ===临界资源===
int in = 0, out = 0;
// in和out变量分别表示下一个要写入数据的位置和下一个要读取数据的位置。
int buffer[n];  // 缓冲
// buffer数组表示缓冲池。生产者将生产的产品放入缓冲区的一个位置，消费者从缓冲区中取出产品进行消费
// ===临界资源===

// empty: 资源信号量，表示缓冲池中空缓冲区的数量，其初始值为n
// full: 资源信号量，表示缓冲池中有数据的缓冲区的数量，其初始值为0
// mutex: 互斥信号量，实现进程对缓冲池的互斥使用
sem_t empty, full, mutex;
// mutex信号量用于实现对缓冲区的互斥访问。在生产者和消费者访问缓冲区时，mutex信号量保证了同一时刻只有一个线程可以进入临界区（访问缓冲区）
// empty信号量用于表示空缓冲区的数量，初始值为缓冲区的大小。当一个生产者想要将产品放入缓冲区时，它会尝试获取empty信号量。如果empty大于0，说明有空缓冲区，生产者可以放入产品，然后empty减1
// full信号量用于表示非空缓冲区的数量，初始值为0。当一个消费者想要从缓冲区中取出产品时，它会尝试获取full信号量。如果full大于0，说明有产品可以被消费，消费者可以取出产品，然后full减1

// ===临界资源===
int id = 0; // 记录产品的id
// ===临界资源===
```

```
25
26  // 生产者
27  void* producer(void* param) {
28      long ThreadId = (long)param;    // 线程id
29      while (1) {
30          sem_wait(&empty);   // wait(empty)
31          sem_wait(&mutex);   // wait(mutex)
32          // 生产者线程首先获取empty信号量，如果empty大于0，说明有空缓冲区，可以放入产品；然后获取mutex信号量，与消费者互斥访问临界区
33          // 生产产品
34          buffer[in] = id;    // 生产者在缓冲区的下一个位置放入产品
35          in = (in + 1) % n;  // 缓冲区输入下标变量向后移动一位，以便下次放入下一个位置的缓冲区
36          sleep(2);   // 生产产品花费时间
37          printf("Thread-%ld : Producer produce product %d \n", ThreadId, id++);
38          sem_post(&mutex);   // signal(mutex)
39          sem_post(&full);    // signal(full)
40          // 生产者放入产品后，释放mutex信号量，表示生产者退出临界区；并增加full信号量，表示缓冲区中有产品了
41      }
42  }
43
44  // 消费者
45  void* consumer(void* param) {
46      long ThreadId = (long)param;
47      while (1) {
48          sem_wait(&full);    // wait(full)
49          sem_wait(&mutex);   // wait(mutex)
50          // 消费者线程首先尝试获取full信号量，如果full大于0，说明有产品可以被消费；然后获取mutex信号量，与生产者互斥访问临界区
51          // 消费产品
52          int item = buffer[out]; //消费者从缓冲区的下一个位置取出产品
53          out = (out + 1) % n;    //缓冲区输出下标变量向后移动一位，以便下次取出下一个位置的缓冲区的产品
54          sleep(1);   // 消费产品花费时间
55          printf("Thread-%ld : Consumer consume product %d \n", ThreadId, item);
56          sem_post(&mutex);   // signal(mutex)
57          sem_post(&empty);   // signal(empty)
58          // 消费者取出产品后，释放mutex信号量，表示消费者退出临界区；并增加empty信号量，表示缓冲区中有空缓冲区了
59      }
60  }
61
62  int main() {
63      // 线程id
64      pthread_t tid[4];
65
66      // 初始化信号量
67      // 第二个参数用于区分进程/线程间共享(0为当前进程各线程间共享)
68      // 第三个参数为信号量初始值
69      sem_init(&mutex, 0, 1);
70      sem_init(&empty, 0, n);
71      sem_init(&full, 0, 0);
72
```

```
    // 2个生产者，2个消费者
    pthread_create(&tid[0], NULL, consumer, (void*)0);
    pthread_create(&tid[1], NULL, producer, (void*)1);
    pthread_create(&tid[2], NULL, consumer, (void*)2);
    pthread_create(&tid[3], NULL, producer, (void*)3);

    // 输入q或Q结束进程
    while (1) {
        char c = getchar();
        if (c == 'q' || c == 'Q') {
            for (int i = 0; i < 4; i++) {
                pthread_cancel(tid[i]);
            }
            break;
        }
    }

    // 释放信号量
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

2.1.2. PhD

```
# include <stdio.h>
# include <pthread.h>
# include <semaphore.h>
# include <unistd.h>

// mutex互斥地取筷子
sem_t chopstick[5];

// ===添加的代码===
sem_t philosopher;
// 设置额外的信号量，用于限制最多只有四位哲学家同时拿起筷子，避免死锁
// ===添加的代码===

void* eat_think(void* arg) {

    // ===修改的代码===
    // int phi = (int)arg;
    // **************
    int phi = (long)arg;
    // ===修改的代码===

    while (1) {

        // ===添加的代码===
        sem_wait(&philosopher);
        // 哲学家线程首先尝试获取philosopher信号量，如果信号量的值大于0，表示有资源可以使用，可以继续尝试获取筷子。
        // ===添加的代码===

```

```
        // ===修改的代码===
        // sem_wait(&chopstick[phi]);   // 拿左
        // sem_wait(&chopstick[(phi + 1) % 5]);    // 拿右
        // printf("Philosopher %d fetches chopstick %d\n", phi, phi);
        // printf("Philosopher %d fetches chopstick %d\n", phi, phi + 1);
        // **************
        sem_wait(&chopstick[phi]);  // 拿左
        printf("Philosopher %d fetches chopstick %d\n", phi, phi);  // 先输出提示信息
        sem_wait(&chopstick[(phi + 1) % 5]);    // 拿右
        printf("Philosopher %d fetches chopstick %d\n", phi, phi + 1);
        // 哲学家在尝试获取左边和右边的筷子时，分别对应chopstick[phi]和chopstick[(phi + 1) % 5]。
        // 如果两把筷子都能获取到（chopstick[phi]和chopstick[(phi + 1) % 5]的值为1），则表示可以进餐。
        // ===修改的代码===

        // ===修改的代码===
        // sleep(5);   // 吃饭
        // printf("Philosopher %d is eating...\n", phi);
        // **************
        printf("Philosopher %d is eating...\n", phi);   // 先输出状态提示
        sleep(5);   // 吃饭
        // ===修改的代码===

        sem_post(&chopstick[phi]);
        sem_post(&chopstick[(phi + 1) % 5]);
        // 吃完后，哲学家释放左右两边的筷子（分别对应chopstick[phi]和chopstick[(phi + 1) % 5]的值为1）

        printf("Philosopher %d release chopstick %d\n", phi, phi);
        printf("Philosopher %d release chopstick %d\n", phi, phi + 1);

        // ===添加的代码===
        sem_post(&philosopher);
        // 释放philosopher信号量，表示自己不再占用资源
        // ===添加的代码===

        // 思考
        sleep(3);
    }
}

int main() {
    // 线程id
    pthread_t tid[5];
    for (int i = 0; i < 5; ++i) {
        sem_init(&chopstick[i], 0, 1);
    }
    // 数组的每个元素表示一把筷子，初始值为1。哲学家在进餐时需要先获取左边的筷子，再获取右边的筷子，避免死锁。

    // ===添加的代码===
    sem_init(&philosopher, 0, 4);
    // 信号量的初始值为4，表示最多有4位哲学家可以同时尝试获取筷子。
    // ===添加的代码===
```

```
80
81      for (int i = 0; i < 5; ++i) {
82          pthread_create(&tid[i], NULL, eat_think, (void*)i);
83      }
84      while (1) {
85          char c = getchar();
86          if (c == 'q' || c == 'Q') {
87              for (int i = 0; i < 4; ++i) {
88                  pthread_cancel(tid[i]);
89              }
90              break;
91          }
92
93          // ===添加的代码===
94          else if (c == 'l' || c == 'L') {
95              int chop_vals[5];
96              for (int i = 0; i < 5; ++i) {
97                  sem_getvalue(&chopstick[i], &chop_vals[i]);
98              }
99              printf("chops sem_t value: [%d, %d, %d, %d, %d]\n",
100                 chop_vals[0], chop_vals[1], chop_vals[2], chop_vals[3],
    chop_vals[4]);
101         }
102         // 设置一个接口，按"l"或"L"获取筷子状态
103         // ===添加的代码===
104
105     }
106     for (int i = 0; i < 5; ++i) {
107         sem_destroy(&chopstick[i]);
108     }
109
110     // ===添加的代码===
111     sem_destroy(&philosopher);
112     // ===添加的代码===
113
114     return 0;
115 }
```

2.1.3. ReaderWriter

```
1  # include <stdio.h>
2  # include <pthread.h>
3  # include <semaphore.h>
4  # include <unistd.h>
5
6  // 互斥信号量 wmutex 用于实现对文件的互斥访问
7  // rmutex 用于对readCount变量的互斥访问
8  sem_t wmutex, rmutex;
9  // wmutex信号量用于对文件的读写互斥、写写互斥访问
10 // rmutex信号量用于对readCount变量的互斥访问，确保多个读者不会同时修改readCount
11
12 // ===临界资源===
13 int readCount = 0;
14 // 记录当前有几个读进程在访问文件
15 // ===临界资源===
```

```
16
17  // 读者
18  void* Reader(void* param) {
19      long threadid = (long)param;     //进程id
20      while (1) {
21          sem_wait(&rmutex);
22          // 读者线程在读取文件前，首先获取rmutex信号量，确保在读取readCount变量时没有其
    他读者在修改它
23          if (readCount == 0) {    //第一个读者到来
24              sem_wait(&wmutex);
25              // 如果是第一个读者到来，读者会获取wmutex信号量，确保没有写者在写文件
26              // 这防止了在有写者时有新的读者加入，保证了读写互斥
27          }
28          readCount++;
29          sem_post(&rmutex);
30          // 访问临界资源结束，释放rmutex信号量
31          printf("Thread-%ld: is reading...\n", threadid);
32          sleep(1);
33          printf("Thread-%ld: ends reading.\n", threadid);
34
35          sem_wait(&rmutex);
36          // 需要访问临界资源readCount，再次获取rmutex信号量
37          readCount--;
38          if (readCount == 0) {    //所有读者读完，释放写者
39              sem_post(&wmutex);
40              // 如果是最后一个读者离开，读者会释放wmutex信号量，允许写者访问文件
41          }
42          sem_post(&rmutex);
43          // 访问临界资源结束，释放rmutex信号量
44      }
45  }
46
47  // 写者
48  void* Writer(void* param) {
49      long threadid = (long)param;
50      while (1) {
51          sem_wait(&wmutex);
52          // 写者线程在写文件前，首先获取wmutex信号量，确保没有其他读者或写者在访问文件。
53          printf("Thread-%ld: is writing...\n", threadid);
54          sleep(2);
55          printf("Thread-%ld: ends writing.\n", threadid);
56          sem_post(&wmutex);
57          // 写者写完文件后，释放wmutex信号量，允许其他读者或写者访问文件。
58      }
59  }
60
61  int main() {
62      // 信号量初始化
63      sem_init(&rmutex, 0, 1);
64      sem_init(&wmutex, 0, 1);
65      pthread_t tid[4];
66      // 两个读者、两个写者
67      pthread_create(&tid[0], NULL, Reader, (void*)0);
68      pthread_create(&tid[1], NULL, Writer, (void*)1);
69      pthread_create(&tid[2], NULL, Reader, (void*)2);
70      pthread_create(&tid[3], NULL, Writer, (void*)3);
```

```
    //输入q或Q退出
    while (1) {
        char c = getchar();
        if (c == 'q' || c == 'Q') {
            for (int i = 0; i < 4; ++i) {
                pthread_cancel(tid[i]);
            }
            break;
        }
    }
    // 释放信号量
    sem_destroy(&rmutex);
    sem_destroy(&wmutex);
    return 0;
}

```

2.2. 测试生产者-消费者代码

首先由于Deepin中`gcc (Uos 8.3.0.3-3+rebuild) 8.3.0`的特性，使用`# include <pthread.h>`需要在编译时添加参数`-pthread`（这在上一次实验报告中提到）。因此为了方便，我直接将`-pthread`参数与`gcc`绑定，即运行以下命令：

```
alias gcc="gcc -pthread"
```

接下来对代码进行测试。

2.2.1. 一个生产者一个消费者

在`main`函数中将生产者和消费者分别设置1个线程。

线程1为生产者，线程0为消费者。

```
// 1个生产者，1个消费者
pthread_create(&tid[0], NULL, consumer, (void*)0);
pthread_create(&tid[1], NULL, producer, (void*)1);
```

编译运行。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./producer-consumer.c -o producer-consumer
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
Thread-1 : Producer produce product 0
Thread-1 : Producer produce product 1
Thread-1 : Producer produce product 2
Thread-1 : Producer produce product 3
Thread-1 : Producer produce product 4
Thread-0 : Consumer consume product 0
Thread-0 : Consumer consume product 1
Thread-0 : Consumer consume product 2
Thread-0 : Consumer consume product 3
Thread-0 : Consumer consume product 4
Thread-1 : Producer produce product 5
Thread-1 : Producer produce product 6
```

```
q
```

可以看到，生产者线程和消费者线程的行为交替执行，能正确同步。

1. 生产者线程1生产了5个数据`[0, 1, 2, 3, 4]`，消费者线程0等待生产者线程生产完数据；
2. 消费者进程0消费了5个数据`[0, 1, 2, 3, 4]`，生产者进程1等待消费者线程消费完数据；
3. 生产者线程1又生产了2个数据`[5, 6]`，消费者线程0等待生产者线程生产完数据；
4. 按q销毁线程退出

2.2.2. 多个生产者多个消费者

在`main`函数中将生产者和消费者分别设置2个线程。

线程1、3为生产者，线程0、2为消费者。

```
// 2个生产者，2个消费者
pthread_create(&tid[0], NULL, consumer, (void*)0);
pthread_create(&tid[1], NULL, producer, (void*)1);
pthread_create(&tid[2], NULL, consumer, (void*)2);
pthread_create(&tid[3], NULL, producer, (void*)3);
```

编译运行。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./producer-consumer.c -o producer-consumer
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
Thread-1 : Producer produce product 0
Thread-1 : Producer produce product 1
Thread-1 : Producer produce product 2
Thread-1 : Producer produce product 3
Thread-0 : Consumer consume product 0
Thread-0 : Consumer consume product 1
Thread-0 : Consumer consume product 2
Thread-1 : Producer produce product 4
Thread-1 : Producer produce product 5
Thread-1 : Producer produce product 6
Thread-0 : Consumer consume product 3
Thread-0 : Consumer consume product 4
Thread-0 : Consumer consume product 5
Thread-2 : Consumer consume product 6
Thread-1 : Producer produce product 7
q
```

输出结果表明，程序中多个生产者多个消费者都能正确同步。

1. 生产者线程1生产了4个数据`[0, 1, 2, 3]`，生产者线程3互斥等待，消费者线程0、2等待生产者线程生产完数据；
2. 消费者进程0消费了3个数据`[0, 1, 2]`，消费者进程2互斥等待，生产者进程1、3等待消费者线程消费完数据；
3. 生产者线程1又生产了3个数据`[4, 5, 6]`，生产者线程3互斥等待，消费者线程0、2等待生产者线程生产完数据；

4. 消费者进程0消费了3个数据`[3, 4, 5]`，消费者进程2互斥等待，生产者进程1、3等待消费者线程消费完数据；
5. 消费者进程2消费了1个数据`[6]`，消费者进程0互斥等待，生产者进程1、3等待消费者线程消费完数据；
6. 生产者线程1生产1个数据`[7]`，生产者线程3互斥等待，消费者线程0、2等待生产者线程生产完数据；
7. 按q销毁线程退出

2.2.3. 测试去掉同步操作命令

2.2.3.1. 测试去掉互斥访问同步信号量mutex

在源代码中注释掉信号量`mutex`，只保留信号量`empty`和`full`。

编译运行。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./producer-consumer.c -o producer-consumer
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
Thread-3 : Producer produce product 0
Thread-1 : Producer produce product 0
Thread-0 : Consumer consume product 0
Thread-2 : Consumer consume product 0
q
```

发现临界资源`id`没有被保护互斥访问。

生产者线程1和3都对同一个缓冲区生产了`id`为0的数据；消费者线程0和2都从同一个缓冲区消费了`id`为0的数据。

2.2.3.2. 测试去掉资源同步信号量empty、full

在源代码中注释掉信号量`empty`和`full`，只保留信号量`mutex`。

修改`buffer`的初始化代码为：

```
int buffer[n] = { 4, 3, 2, 1, 0 };
```

以观察消费者访问缓冲区的情况。

编译运行。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./producer-consumer.c -o producer-consumer
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
Thread-0 : Consumer consume product 4
Thread-0 : Consumer consume product 3
Thread-0 : Consumer consume product 2
Thread-0 : Consumer consume product 1
Thread-0 : Consumer consume product 0
Thread-0 : Consumer consume product 4
Thread-0 : Consumer consume product 3
q
```

发现临界资源 `buffer` 即缓冲区没有被保护互斥访问。

没有 `full` 信号量的保护，在生产者生产数据前，消费者线程2就已经开始访问缓冲区获取数据，即消费数据量超出了生产的数据量。

同理，没有 `empty` 信号量的保护，生产者生产的数据也会超出缓冲区的大小，覆盖掉之前生产且没有被消费者消费掉的数据。

2.2.4. 测试将进入临界区的两个wait操作位置互换

在源代码中将进入临界区的两个wait操作位置互换。

为了更好观察信号量情况，在 `main` 函数 `if (c == 'q' || c == 'Q')` 后加一个 `else if (c == 'l' || c == 'L')` 用于创建获取信号量状态的接口。

```
else if (c == 'l' || c == 'L') {
    int mutex_value, full_value, empty_value;
    sem_getvalue(&mutex, &mutex_value);
    sem_getvalue(&full, &full_value);
    sem_getvalue(&empty, &empty_value);
    printf("mutex=%d, full=%d, empty=%d\n",
        mutex_value, full_value, empty_value);
}
```

编译后多次测试，出现如下两种情况：

1. 消费者无法进行消费

   ```
   (base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
   Thread-3 : Producer produce product 0
   Thread-3 : Producer produce product 1
   Thread-3 : Producer produce product 2
   Thread-3 : Producer produce product 3
   Thread-3 : Producer produce product 4
   l
   mutex=0, full=5, empty=0
   q
   ```

   观察到生产者将缓冲区填满后，消费者无法消费。

   此时信号量 `empty` 为0，生产者无法再进行生产，但此时信号量 `mutex` 也为0，生产者和消费者都不能进入临界区。

   所以发生了死锁。

2. 生产者无法进行生产

   ```
   (base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./producer-consumer
   l
   mutex=0, full=0, empty=5
   q
   ```

   观察到程序一开始运行，生产者就没有进行生产。

   此时信号量 `mutex` 为0，生产者和消费者都不能进入临界区。

   在程序运行过程中：

1. `mutex`信号量首先被消费者修改，然后消费者访问`full`信号量，此时`full`为0，没有可消费的资源，无法进入临界区。
2. 但此时`mutex`信号量为0，生产者也无法进入临界区。

发生了死锁。

2.3. 测试哲学家进餐问题代码

仅在原代码中作如下修改：

- 添加输出信号量的功能代码，输入“l”或“L”打印筷子信号量状态；
- 没拿起一支筷子就输出提示信息，而不是拿起两只筷子后才输出提示信息。

不添加进餐哲学家信号量。

编译运行。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./PhD.c -o ./PhD
./PhD.c: In function ‘main’:
./PhD.c:88:50: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
         pthread_create(&tid[i], NULL, eat_think, (void*)i);
                                                  ^
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./PhD
Philosopher 1 fetches chopstick 1
Philosopher 4 fetches chopstick 4
Philosopher 0 fetches chopstick 0
Philosopher 3 fetches chopstick 3
Philosopher 2 fetches chopstick 2
l
chops sem_t value: [0, 0, 0, 0, 0]
q
```

发生死锁。

原因很明显，哲学家`[1, 4, 0, 3, 2]`各自拿起了其左手边的筷子，而右手边的筷子被其他哲学家占有了，所以五位哲学家都无法进餐。

2.3.1. 修改方案1

添加哲学家信号量，限定持有筷子的哲学家数小于5。

修改代码，最终代码如**2.1.2. PhD**代码块中代码所示。该代码能避免死锁，使程序正常运行。

2.3.2. 修改方案2

使用mutex互斥信号量，一次取够一双筷子。

此时将筷子信号量作为临界资源，一次只允许一个哲学家线程修改。

修改代码，删去哲学家信号量，添加`mutex`信号量，将拿取筷子的行为代码包在`wait(mutex)`和`signal(mutex)`内。

```
sem_t mutex;
// 设置额外的信号量，用于互斥拿起两只筷子的行为

```

```c
void* eat_think(void* arg) {
    int phi = (long)arg;
    while (1) {
        sem_wait(&mutex);
        sem_wait(&chopstick[phi]);  // 拿左
        printf("Philosopher %d fetches chopstick %d\n", phi, phi);
        sem_wait(&chopstick[(phi + 1) % 5]);    // 拿右
        printf("Philosopher %d fetches chopstick %d\n", phi, phi + 1);
        sem_post(&mutex);

        printf("Philosopher %d is eating...\n", phi);
        sleep(5);   // 进餐
        sem_post(&chopstick[phi]);
        sem_post(&chopstick[(phi + 1) % 5]);
        printf("Philosopher %d release chopstick %d\n", phi, phi);
        printf("Philosopher %d release chopstick %d\n", phi, phi + 1);
        sleep(3);   // 思考
    }
}
```

编译运行：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./PhD_1.c -o
./PhD_1
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./PhD_1
Philosopher 2 fetches chopstick 2
Philosopher 2 fetches chopstick 3
Philosopher 2 is eating...
Philosopher 1 fetches chopstick 1
l
chops sem_t value: [1, 0, 0, 0, 1]
Philosopher 2 release chopstick 2
Philosopher 2 release chopstick 3
Philosopher 1 fetches chopstick 2
Philosopher 1 is eating...
Philosopher 3 fetches chopstick 3
Philosopher 3 fetches chopstick 4
Philosopher 3 is eating...
l
chops sem_t value: [1, 0, 0, 0, 0]
Philosopher 1 release chopstick 1
Philosopher 1 release chopstick 2
Philosopher 3 release chopstick 3
Philosopher 3 release chopstick 4
Philosopher 4 fetches chopstick 4
Philosopher 4 fetches chopstick 5
Philosopher 4 is eating...
l
chops sem_t value: [0, 1, 1, 1, 0]
q
```

没有发生死锁，程序正常运行。

2.3.3. 修改方案3

编号为奇数的哲学家先尝试拿左手边的筷子再尝试拿右手边的筷子；编号为偶数的哲学家先尝试拿右手边的筷子再尝试拿左手边的筷子。

这样就不需要添加额外的信号量了。

修改代码，删去`mutex`信号量，定义先后拿筷子的编号：

- `phi`为奇时，`(phi + 1) % 2`为0，`first`为`phi`；`phi % 2`为1，`second`为`(phi + 1) % 5`；
- `phi`为偶时，`(phi + 1) % 2`为1，`first`为`(phi + 1) % 5`；`phi % 2`为0，`second`为`phi`；

```
void* eat_think(void* arg) {
    int phi = (long)arg;
    while (1) {
        int first = (phi + ((phi + 1) % 2)) % 5, second = (phi + (phi % 2)) % 5;
        sem_wait(&chopstick[first]);
        printf("Philosopher %d fetches chopstick %d\n", phi, first);
        sem_wait(&chopstick[second]);
        printf("Philosopher %d fetches chopstick %d\n", phi, second);

        printf("Philosopher %d is eating...\n", phi);
        sleep(5);   // 吃饭
        sem_post(&chopstick[phi]);
        sem_post(&chopstick[(phi + 1) % 5]);
        printf("Philosopher %d release chopstick %d\n", phi, phi);
        printf("Philosopher %d release chopstick %d\n", phi, phi + 1);
        sleep(3);   // 思考
    }
}
```

编译运行：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./PhD_2.c -o ./PhD_2
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./PhD_2
Philosopher 2 fetches chopstick 3
Philosopher 2 fetches chopstick 2
Philosopher 2 is eating...
Philosopher 4 fetches chopstick 0
Philosopher 4 fetches chopstick 4
Philosopher 4 is eating...
Philosopher 0 fetches chopstick 1
1
chops sem_t value: [0, 0, 0, 0, 0]
Philosopher 2 release chopstick 2
Philosopher 2 release chopstick 3
Philosopher 3 fetches chopstick 3
Philosopher 3 fetches chopstick 4
Philosopher 3 is eating...
Philosopher 4 release chopstick 4
Philosopher 4 release chopstick 5
Philosopher 0 fetches chopstick 0
Philosopher 0 is eating...
```

```
l
chops sem_t value: [0, 0, 1, 0, 0]
q
```

没有发生死锁，程序正常运行。

2.4. 测试读者写者问题

2.4.1. 测试ReaderWriter.c原代码

按照原代码（仅更改：在读写行为结束后输出提示信息），编译运行，出现两种结果。

1. 读者线程一直挤占互斥信号量，使得写者线程无法进入临界区。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter.c -o ./ReaderWriter
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter
Thread-0: is reading...
Thread-2: is reading...
Thread-0: ends reading.
Thread-0: is reading...
Thread-2: ends reading.
Thread-2: is reading...
Thread-0: ends reading.
Thread-0: is reading...
Thread-2: ends reading.
Thread-2: is reading...
q
```

2. 写者线程一直挤占互斥信号量，使得读者线程无法进入临界区。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter
Thread-1: is writing...
Thread-1: ends writing.
Thread-1: is writing...
Thread-1: ends writing.
Thread-1: is writing...
Thread-1: ends writing.
Thread-1: is writing...
q
```

出现上述问题的原因是读写行为过于频繁。修改原代码，在每次读写行为结束后`sleep(2)`以减少读写行为频率。

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter.c -o ./ReaderWriter
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter
Thread-0: is reading...
Thread-2: is reading...
Thread-0: ends reading.
Thread-2: ends reading.
Thread-1: is writing...
Thread-1: ends writing.
Thread-3: is writing...
q
```

从以上输出可见，读写顺序正常了。

2.4.2. 构造读写者执行序列测试

修改头文件等引用，以精确获取当前时间，如下：

```
# include <sys/time.h>
typedef struct timeval tv;
```

修改读者函数，去掉`while()`循环，每个线程仅读操作一次，如下：

```
// 读者
void* Reader(void* param) {
    long threadid = (long)param;
    tv reach_t;
    gettimeofday(&reach_t, NULL);
    printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
    sem_wait(&rmutex);
    if (readCount == 0) {
        sem_wait(&wmutex);
    }
    tv exec_t;
    gettimeofday(&exec_t, NULL);
    printf("Thread-%ld starts executing at time %ld, waited %ldus\n", threadid, exec_t.tv_usec, exec_t.tv_usec - reach_t.tv_usec);
    readCount++;
    sem_post(&rmutex);

    printf("Thread-%ld: is reading...\n", threadid);
    sleep(1);
    printf("Thread-%ld: ends reading.\n", threadid);

    sem_wait(&rmutex);
    readCount--;
    if (readCount == 0) {
        sem_post(&wmutex);
    }
    sem_post(&rmutex);
}
```

修改写者函数，去掉`while()`循环，每个线程仅写操作一次，如下：

```
// 写者
void* Writer(void* param) {
    long threadid = (long)param;
    tv reach_t;
    gettimeofday(&reach_t, NULL);
    printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
    sem_wait(&wmutex);
    tv exec_t;
    gettimeofday(&exec_t, NULL);
    printf("Thread-%ld starts executing at time %ld, waited %ldus\n",
threadid, exec_t.tv_usec, exec_t.tv_usec - reach_t.tv_usec);
    printf("Thread-%ld: is writing...\n", threadid);
    sleep(2);
    printf("Thread-%ld: ends writing.\n", threadid);
    sem_post(&wmutex);
}
```

2.4.2.1. 序列1测试

首先执行1个写者线程，然后执行9个读者线程。

`main`函数设计如下：

```
int main() {
    sem_init(&rmutex, 0, 1);
    sem_init(&wmutex, 0, 1);
    pthread_t tid[10];
    int num_pthread = 10;
    pthread_create(&tid[0], NULL, Writer, (void*)0);
    for (int i = 1; i < num_pthread; ++i) {
        pthread_create(&tid[i], NULL, Reader, (void*)i);
    }
    while (1) {
        char c = getchar();
        if (c == 'q' || c == 'Q') {
            for (int i = 0; i < num_pthread; ++i) {
                pthread_cancel(tid[i]);
            }
            break;
        }
    }
    sem_destroy(&rmutex);
    sem_destroy(&wmutex);
    return 0;
}
```

编译运行如下：

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_1.c
-o ./ReaderWriter_1
./ReaderWriter_1.c: In function ‘main’:
./ReaderWriter_1.c:64:41: warning: cast to pointer from integer of different
size [-Wint-to-pointer-cast]
   pthread_create(&tid[i], NULL, Reader, (void*)i);
```

```
                                        ^
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ # first execution
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-1 reached at time 381669
Thread-1 starts executing at time 381726, waited 57us
Thread-5 reached at time 381729
Thread-5 starts executing at time 381736, waited 7us
Thread-5: is reading...
Thread-6 reached at time 381740
Thread-6 starts executing at time 381747, waited 7us
Thread-6: is reading...
Thread-1: is reading...
Thread-4 reached at time 381713
Thread-0 reached at time 381672
Thread-9 reached at time 381780
Thread-3 reached at time 381681
Thread-7 reached at time 381757
Thread-8 reached at time 381765
Thread-4 starts executing at time 381782, waited 69us
Thread-4: is reading...
Thread-9 starts executing at time 381808, waited 28us
Thread-9: is reading...
Thread-3 starts executing at time 381811, waited 130us
Thread-8 starts executing at time 381813, waited 48us
Thread-8: is reading...
Thread-3: is reading...
Thread-7 starts executing at time 381817, waited 60us
Thread-7: is reading...
Thread-2 reached at time 381751
Thread-2 starts executing at time 381845, waited 94us
Thread-2: is reading...
Thread-5: ends reading.
Thread-1: ends reading.
Thread-7: ends reading.
Thread-3: ends reading.
Thread-8: ends reading.
Thread-9: ends reading.
Thread-6: ends reading.
Thread-4: ends reading.
Thread-2: ends reading.
Thread-0 starts executing at time 381937, waited 265us
Thread-0: is writing...
Thread-0: ends writing.
q
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ # second execution
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-0 reached at time 742894
Thread-0 starts executing at time 742941, waited 47us
Thread-0: is writing...
Thread-1 reached at time 742901
Thread-2 reached at time 742930
Thread-3 reached at time 742961
Thread-4 reached at time 742982
Thread-5 reached at time 743000
Thread-6 reached at time 743017
Thread-7 reached at time 743031
```

```
Thread-8 reached at time 743046
Thread-9 reached at time 743061
Thread-0: ends writing.
Thread-1 starts executing at time 743130, waited 229us
Thread-1: is reading...
Thread-2 starts executing at time 743144, waited 214us
Thread-2: is reading...
Thread-3 starts executing at time 743153, waited 192us
Thread-3: is reading...
Thread-4 starts executing at time 743161, waited 179us
Thread-4: is reading...
Thread-5 starts executing at time 743171, waited 171us
Thread-5: is reading...
Thread-6 starts executing at time 743182, waited 165us
Thread-6: is reading...
Thread-7 starts executing at time 743194, waited 163us
Thread-7: is reading...
Thread-8 starts executing at time 743202, waited 156us
Thread-8: is reading...
Thread-9 starts executing at time 743210, waited 149us
Thread-9: is reading...
Thread-3: ends reading.
Thread-7: ends reading.
Thread-1: ends reading.
Thread-5: ends reading.
Thread-2: ends reading.
Thread-8: ends reading.
Thread-4: ends reading.
Thread-9: ends reading.
Thread-6: ends reading.
q
```

可见：

- 当读者线程比写者线程先占用了`mutex`互斥信号量，那么写者就会在所有读者线程结束后最后执行临界区，如第1次执行结果所示；
- 如果写者线程比读者线程先占用了`mutex`互斥信号量，那么读者进程会在该写者进程结束后执行，如第2次执行结果所示。

即：读者优先。

在读者线程之间，执行顺序也不是按照线程创建函数`pthread_create()`的执行顺序执行的。线程的创建、过程执行顺序都是随机的。例证如下：

在第1次执行过程中，第21行`Thread-7 reached at time 381757`和第22行`Thread-8 reached at time 381765`显示第7个线程（读者）比第8个线程（读者）到达时间早，但是第28行`Thread-8 starts executing at time 381813, waited 48us`和第31行`Thread-7 starts executing at time 381817, waited 60us`显示第7个线程（读者）比第8个线程（读者）执行时间晚。猜测是因为执行顺序如下：

1. [Thread-7] printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
2. [Thread-8] printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
3. [Thread-8] sem_wait(&rmutex);
4. [Thread-7] sem_wait(&rmutex);

2.4.2.2. 序列2测试

首先执行1个读者线程，然后执行9个写者线程。

`main`函数中`pthread_create()`部分设计如下：

```
pthread_create(&tid[0], NULL, Reader, (void*)0);
for (int i = 1; i < num_pthread; ++i) {
    pthread_create(&tid[i], NULL, Writer, (void*)i);
}
```

编译运行如下：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_1.c -o ./ReaderWriter_1
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-1 reached at time 591009
Thread-0 reached at time 591009
Thread-2 reached at time 591026
Thread-3 reached at time 591041
Thread-4 reached at time 591051
Thread-1 starts executing at time 591055, waited 46us
Thread-7 reached at time 591084
Thread-6 reached at time 591072
Thread-1: is writing...
Thread-5 reached at time 591061
Thread-8 reached at time 591095
Thread-9 reached at time 591107
Thread-1: ends writing.
Thread-0 starts executing at time 591371, waited 362us
Thread-0: is reading...
Thread-0: ends reading.
Thread-2 starts executing at time 591553, waited 527us
Thread-2: is writing...
Thread-2: ends writing.
Thread-3 starts executing at time 591650, waited 609us
Thread-3: is writing...
Thread-3: ends writing.
Thread-4 starts executing at time 591789, waited 738us
Thread-4: is writing...
Thread-4: ends writing.
Thread-7 starts executing at time 591876, waited 792us
Thread-7: is writing...
Thread-7: ends writing.
Thread-6 starts executing at time 592060, waited 988us
Thread-6: is writing...
Thread-6: ends writing.
Thread-5 starts executing at time 592292, waited 1231us
Thread-5: is writing...
Thread-5: ends writing.
Thread-8 starts executing at time 592461, waited 1366us
Thread-8: is writing...
Thread-8: ends writing.
Thread-9 starts executing at time 592694, waited 1587us
Thread-9: is writing...
```

```
Thread-9: ends writing.
q
```

可见，只要有读者线程尝试执行，就一定会在当前写者线程进行完之后立即执行，即读者优先。

2.4.2.3. 序列3测试

交替执行5个写者线程和5个读者线程。

`main`函数中`pthread_create()`部分设计如下：

```
for (int i = 0; i < num_pthread / 2; ++i) {
    pthread_create(&tid[2 * i], NULL, Writer, (void*)(2 * i));
    pthread_create(&tid[2 * i + 1], NULL, Reader, (void*)(2 * i + 1));
}
```

编译运行如下：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_1.c -o ./ReaderWriter_1
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-1 reached at time 338262
Thread-5 reached at time 338314
Thread-6 reached at time 338327
Thread-3 reached at time 338262
Thread-4 reached at time 338280
Thread-0 reached at time 338307
Thread-2 reached at time 338268
Thread-1 starts executing at time 338314, waited 52us
Thread-1: is reading...
Thread-8 reached at time 338353
Thread-7 reached at time 338340
Thread-5 starts executing at time 338360, waited 46us
Thread-5: is reading...
Thread-9 reached at time 338365
Thread-3 starts executing at time 338381, waited 119us
Thread-3: is reading...
Thread-7 starts executing at time 338397, waited 57us
Thread-7: is reading...
Thread-9 starts executing at time 338410, waited 45us
Thread-9: is reading...
Thread-1: ends reading.
Thread-9: ends reading.
Thread-5: ends reading.
Thread-7: ends reading.
Thread-3: ends reading.
Thread-6 starts executing at time 338589, waited 262us
Thread-6: is writing...
Thread-6: ends writing.
Thread-4 starts executing at time 338797, waited 517us
Thread-4: is writing...
Thread-4: ends writing.
Thread-0 starts executing at time 339102, waited 795us
Thread-0: is writing...
Thread-0: ends writing.
```

```
Thread-2 starts executing at time 339415, waited 1147us
Thread-2: is writing...
Thread-2: ends writing.
Thread-8 starts executing at time 339989, waited 1636us
Thread-8: is writing...
Thread-8: ends writing.
q
```

可见执行顺序还是：只要有读者线程尝试执行，就一定会在当前写者线程进行完之后立即执行，即读者优先。

2.4.2.4. 序列4测试

执行10个写者线程。

`main`函数中`pthread_create()`部分设计如下：

```
for (int i = 0; i < num_pthread; ++i) {
    pthread_create(&tid[i], NULL, Writer, (void*)i);
}
```

编译运行如下：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_1.c -o ./ReaderWriter_1
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-4 reached at time 493308
Thread-4 starts executing at time 493362, waited 54us
Thread-4: is writing...
Thread-3 reached at time 493302
Thread-8 reached at time 493379
Thread-1 reached at time 493390
Thread-5 reached at time 493334
Thread-6 reached at time 493353
Thread-0 reached at time 493307
Thread-7 reached at time 493367
Thread-9 reached at time 493391
Thread-2 reached at time 493331
Thread-4: ends writing.
Thread-3 starts executing at time 493608, waited 306us
Thread-3: is writing...
Thread-3: ends writing.
Thread-8 starts executing at time 493828, waited 449us
Thread-8: is writing...
Thread-8: ends writing.
Thread-1 starts executing at time 494108, waited 718us
Thread-1: is writing...
Thread-1: ends writing.
Thread-5 starts executing at time 494346, waited 1012us
Thread-5: is writing...
Thread-5: ends writing.
Thread-6 starts executing at time 494664, waited 1311us
Thread-6: is writing...
Thread-6: ends writing.
Thread-7 starts executing at time 495280, waited 1913us
```

```
Thread-7: is writing...
Thread-7: ends writing.
Thread-9 starts executing at time 495533, waited 2142us
Thread-9: is writing...
Thread-9: ends writing.
Thread-0 starts executing at time 495772, waited 2465us
Thread-0: is writing...
Thread-0: ends writing.
Thread-2 starts executing at time 496053, waited 2722us
Thread-2: is writing...
Thread-2: ends writing.
q
```

写写互斥，同步机制没有出现问题。

2.4.2.5. 序列5测试

执行10个读者线程。

`main`函数中`pthread_create()`部分设计如下：

```
for (int i = 0; i < num_pthread; ++i) {
    pthread_create(&tid[i], NULL, Reader, (void*)i);
}
```

编译运行如下：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_1.c -o ./ReaderWriter_1
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_1
Thread-1 reached at time 875992
Thread-1 starts executing at time 876030, waited 38us
Thread-1: is reading...
Thread-4 reached at time 876047
Thread-4 starts executing at time 876054, waited 7us
Thread-4: is reading...
Thread-2 reached at time 876011
Thread-2 starts executing at time 876068, waited 57us
Thread-2: is reading...
Thread-5 reached at time 876059
Thread-5 starts executing at time 876076, waited 17us
Thread-5: is reading...
Thread-3 reached at time 876030
Thread-3 starts executing at time 876088, waited 58us
Thread-3: is reading...
Thread-0 reached at time 875991
Thread-9 reached at time 876102
Thread-6 reached at time 876069
Thread-0 starts executing at time 876102, waited 111us
Thread-7 reached at time 876080
Thread-8 reached at time 876091
Thread-0: is reading...
Thread-9 starts executing at time 876118, waited 16us
Thread-9: is reading...
Thread-6 starts executing at time 876142, waited 73us
```

```
Thread-6: is reading...
Thread-7 starts executing at time 876149, waited 69us
Thread-7: is reading...
Thread-8 starts executing at time 876158, waited 67us
Thread-8: is reading...
Thread-2: ends reading.
Thread-1: ends reading.
Thread-5: ends reading.
Thread-3: ends reading.
Thread-4: ends reading.
Thread-0: ends reading.
Thread-9: ends reading.
Thread-6: ends reading.
Thread-7: ends reading.
Thread-8: ends reading.
q
```

读读不互斥，读者到达即读，程序正常运行。

2.5. 实现并测试写者优先的读者写者算法

2.5.1. 算法实现

算法逻辑写在注释中：

```c
# include <stdio.h>
# include <pthread.h>
# include <semaphore.h>
# include <unistd.h>

sem_t wmutex, rmutex, wmutex_1, rmutex_1;
// wmutex信号量用于对文件的读写互斥、写写互斥访问
// rmutex信号量用于对readCount变量的互斥访问，确保多个读者不会同时修改readCount
// wmutex_1信号量用于对writeCount变量的互斥访问，确保多个写者不会同时修改writeCount
// rmutex_1信号量用于对文件的写者优先的读写互斥

int readCount = 0;
// 记录当前有几个读进程在访问文件
int writeCount = 0;
// 记录当前有几个写进程在访问文件

// 读者
void* Reader(void* param) {
    long threadid = (long)param;    //进程id
    while (1) {
        sem_wait(&rmutex_1);
        // 首先获取rmutex_1信号量，确保没有写者在访问文件。
        sem_wait(&rmutex);
        if (readCount == 0) {
            sem_wait(&wmutex);
        }
        readCount++;
        sem_post(&rmutex);

        printf("Thread-%ld: is reading...\n", threadid);
```

```
        sleep(1);
        printf("Thread-%ld: ends reading.\n", threadid);

        sem_wait(&rmutex);
        readCount--;
        if (readCount == 0) {
            sem_post(&wmutex);
        }
        sem_post(&rmutex);
        sem_post(&rmutex_1);    // 释放rmutex_1信号量
        sleep(2);
    }
}

// 写者
void* Writer(void* param) {
    long threadid = (long)param;
    while (1) {
        sem_wait(&wmutex_1);
        // 首先获取wmutex_1信号量，确保在读取writeCount变量时没有其他写者在修改它
        if (writeCount == 0) {
            sem_wait(&rmutex_1);
            // 如果是第一个写者到来，写者会获取rmutex_1信号量，确保没有读者在写文件
            // 这防止了在有读者时有新的写者加入，保证了读写互斥
        }
        writeCount++;
        sem_post(&wmutex_1);
        // 释放wmutex_1信号量
        sem_wait(&wmutex);

        printf("Thread-%ld: is writing...\n", threadid);
        sleep(2);
        printf("Thread-%ld: ends writing.\n", threadid);

        sem_post(&wmutex);
        sem_wait(&wmutex_1);
        // 获取wmutex_1信号量，确保在读取writeCount变量时没有其他写者在修改它
        writeCount--;
        if (writeCount == 0) {  //所有写者写完，释放读者
            sem_post(&rmutex_1);
            // 如果是最后一个写者离开，写者会释放rmutex_1信号量，允许读者访问文件
        }
        sem_post(&wmutex_1);
        // 释放wmutex_1信号量
        sleep(2);
    }
}

int main() {
    // 信号量初始化
    sem_init(&rmutex, 0, 1);
    sem_init(&wmutex, 0, 1);
    sem_init(&rmutex_1, 0, 1);
    sem_init(&wmutex_1, 0, 1);
    pthread_t tid[4];
    // 两个读者、两个写者
```

```
    pthread_create(&tid[0], NULL, Reader, (void*)0);
    pthread_create(&tid[1], NULL, Writer, (void*)1);
    pthread_create(&tid[2], NULL, Reader, (void*)2);
    pthread_create(&tid[3], NULL, Writer, (void*)3);
    //输入q或Q退出
    while (1) {
        char c = getchar();
        if (c == 'q' || c == 'Q') {
            for (int i = 0; i < 4; ++i) {
                pthread_cancel(tid[i]);
            }
            break;
        }
    }
    // 释放信号量
    sem_destroy(&rmutex);
    sem_destroy(&wmutex);
    sem_destroy(&rmutex_1);
    sem_destroy(&wmutex_1);
    return 0;
}
```

编译运行：

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_2.c -o ./ReaderWriter_2
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_2
Thread-3: is writing...
Thread-3: ends writing.
Thread-1: is writing...
Thread-1: ends writing.
Thread-3: is writing...
Thread-3: ends writing.
Thread-0: is reading...
Thread-0: ends reading.
Thread-2: is reading...
Thread-2: ends reading.
Thread-1: is writing...
Thread-1: ends writing.
Thread-3: is writing...
Thread-3: ends writing.
Thread-1: is writing...
q
```

看得出来，程序主要由写者掌控。

2.5.2. 构造执行序列测试

我们把这个算法移植到2.4.2.3. 序列3（交替执行5个写者线程和5个读者线程）中，用到达时间和执行时间评价运行效果。

代码如下：

```
# include <stdio.h>
# include <pthread.h>
```

```c
# include <semaphore.h>
# include <unistd.h>
# include <sys/time.h>

typedef struct timeval tv;

sem_t wmutex, rmutex, wmutex_1, rmutex_1;

int readCount = 0, writeCount = 0;

// 读者
void* Reader(void* param) {
    long threadid = (long)param;
    tv reach_t;
    gettimeofday(&reach_t, NULL);
    printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
    sem_wait(&rmutex_1);
    sem_wait(&rmutex);
    if (readCount == 0) {
        sem_wait(&wmutex);
    }
    tv exec_t;
    gettimeofday(&exec_t, NULL);
    printf("Thread-%ld starts executing at time %ld, waited %ldus\n",
threadid, exec_t.tv_usec, exec_t.tv_usec - reach_t.tv_usec);
    readCount++;
    sem_post(&rmutex);

    printf("Thread-%ld: is reading...\n", threadid);
    sleep(1);
    printf("Thread-%ld: ends reading.\n", threadid);

    sem_wait(&rmutex);
    readCount--;
    if (readCount == 0) {
        sem_post(&wmutex);
    }
    sem_post(&rmutex);
    sem_post(&rmutex_1);
}

// 写者
void* Writer(void* param) {
    long threadid = (long)param;
    tv reach_t;
    gettimeofday(&reach_t, NULL);
    printf("Thread-%ld reached at time %ld\n", threadid, reach_t.tv_usec);
    sem_wait(&wmutex_1);
    if (writeCount == 0) {
        sem_wait(&rmutex_1);
    }
    writeCount++;
    sem_post(&wmutex_1);
    sem_wait(&wmutex);
    tv exec_t;
    gettimeofday(&exec_t, NULL);
```

```
    printf("Thread-%ld starts executing at time %ld, waited %ldus\n",
threadid, exec_t.tv_usec, exec_t.tv_usec - reach_t.tv_usec);
    printf("Thread-%ld: is writing...\n", threadid);
    sleep(2);
    printf("Thread-%ld: ends writing.\n", threadid);
    sem_post(&wmutex);
    sem_wait(&wmutex_1);
    writeCount--;
    if (writeCount == 0) {
        sem_post(&rmutex_1);
    }
    sem_post(&wmutex_1);
}

int main() {
    sem_init(&rmutex, 0, 1);
    sem_init(&wmutex, 0, 1);
    sem_init(&rmutex_1, 0, 1);
    sem_init(&wmutex_1, 0, 1);
    pthread_t tid[10];
    int num_pthread = 10;
    for (int i = 0; i < num_pthread / 2; ++i) {
        pthread_create(&tid[2 * i], NULL, Reader, (void*)(2 * i));
        pthread_create(&tid[2 * i + 1], NULL, Writer, (void*)(2 * i + 1));
    }
    while (1) {
        char c = getchar();
        if (c == 'q' || c == 'Q') {
            for (int i = 0; i < num_pthread; ++i) {
                pthread_cancel(tid[i]);
            }
            break;
        }
    }
    sem_destroy(&rmutex);
    sem_destroy(&wmutex);
    sem_destroy(&rmutex_1);
    sem_destroy(&wmutex_1);
    return 0;
}
```

编译运行如下：（省略编译警告信息）

```
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ gcc ./ReaderWriter_3.c
-o ./ReaderWriter_3
(base) kejingfan@KJF-Huawei-PC:~/CodeDir/OS_Lab/Lab2$ ./ReaderWriter_3
Thread-0 reached at time 325598
Thread-0 starts executing at time 325635, waited 37us
Thread-0: is reading...
Thread-2 reached at time 325603
Thread-3 reached at time 325623
Thread-1 reached at time 325598
Thread-4 reached at time 325649
Thread-5 reached at time 325683
Thread-6 reached at time 325696
```

```
Thread-7 reached at time 325708
Thread-8 reached at time 325720
Thread-9 reached at time 325734
Thread-0: ends reading.
Thread-2 starts executing at time 325720, waited 117us
Thread-2: is reading...
Thread-2: ends reading.
Thread-3 starts executing at time 325844, waited 221us
Thread-3: is writing...
Thread-3: ends writing.
Thread-1 starts executing at time 325923, waited 325us
Thread-1: is writing...
Thread-1: ends writing.
Thread-5 starts executing at time 326187, waited 504us
Thread-5: is writing...
Thread-5: ends writing.
Thread-7 starts executing at time 326272, waited 564us
Thread-7: is writing...
Thread-7: ends writing.
Thread-9 starts executing at time 326365, waited 631us
Thread-9: is writing...
Thread-9: ends writing.
Thread-4 starts executing at time 326564, waited 915us
Thread-4: is reading...
Thread-4: ends reading.
Thread-6 starts executing at time 326878, waited 1182us
Thread-6: is reading...
Thread-6: ends reading.
Thread-8 starts executing at time 327463, waited 1743us
Thread-8: is reading...
Thread-8: ends reading.
q
```

执行顺序为：

1. 读者线程0、2到达
2. 读者线程0执行
3. 写者线程3、1到达
4. 读者线程4到达
5. ……
6. 读者线程2执行
7. **写者线程所有线程执行**
8. **读者线程4执行**
9. ……

总结即为，当有写者线程到达，则将当前读者线程队列执行完毕后立即执行接下来到达的所有写者线程，当所有写者线程结束后，才执行到达的读者线程。

写者优先实现了。

3. 问题与讨论

本实验我没有遇到特别大的问题。

4. 实验总结

- 信号量可以用于对共享资源的互斥访问和线程间的同步通信；
- 在多线程编程中，需要仔细设计同步和互斥的策略，才能写出健壮的多线程程序，否则容易导致死锁等问题。