
实验课题：典型同步问题模拟处理编程设计与实现

计科 1903 班 19281171 王雨潇

1. 实验目的

探索、理解并掌握操作系统同步机制的应用编程方法，针对典型的同步问题，构建基于 Windows（或 Linux）操作系统同步机制的解决方案。

2. 实验环境

运行环境：Windows 10 操作系统

编译器：MinGW-w64 GCC 11.2.0

IDE：VSCode，实验代码见“附录：源程序的完整代码”

3. 实验内容

了解、熟悉和运用 Windows（或 Linux）操作系统同步机制及编程方法，针对典型的同步问题，譬如生产者-消费者问题、读者优先的读者-写者问题、写者优先的读者-写者问题、读者数限定的读者-写者问题、哲学家就餐问题等（任选四个即可），编程模拟实现相应问题的解决方案。

3.1. 生产者-消费者问题

3.1.1. 问题描述

生产者-消费者问题是存在一个多线程共享的临界资源数据区，有一个生产者进程不定时地向数据区放入产品，还有一个消费者进程不定时地要取走数据区中的产品，它们访问临界资源的时间均无法预料，需要保证数据同步，避免生产者和消费者发生数据不一致；

3.1.2. 解决方案

用信号量记录数据区资源的数目，需要两个信号量，分别用于空缓冲区和数据资源，还需要一个互斥锁，生产者、消费者对数据操作前，应当加锁确保一次至多一个进程能进入临界区。

3.1.3. 数据结构和算法流程

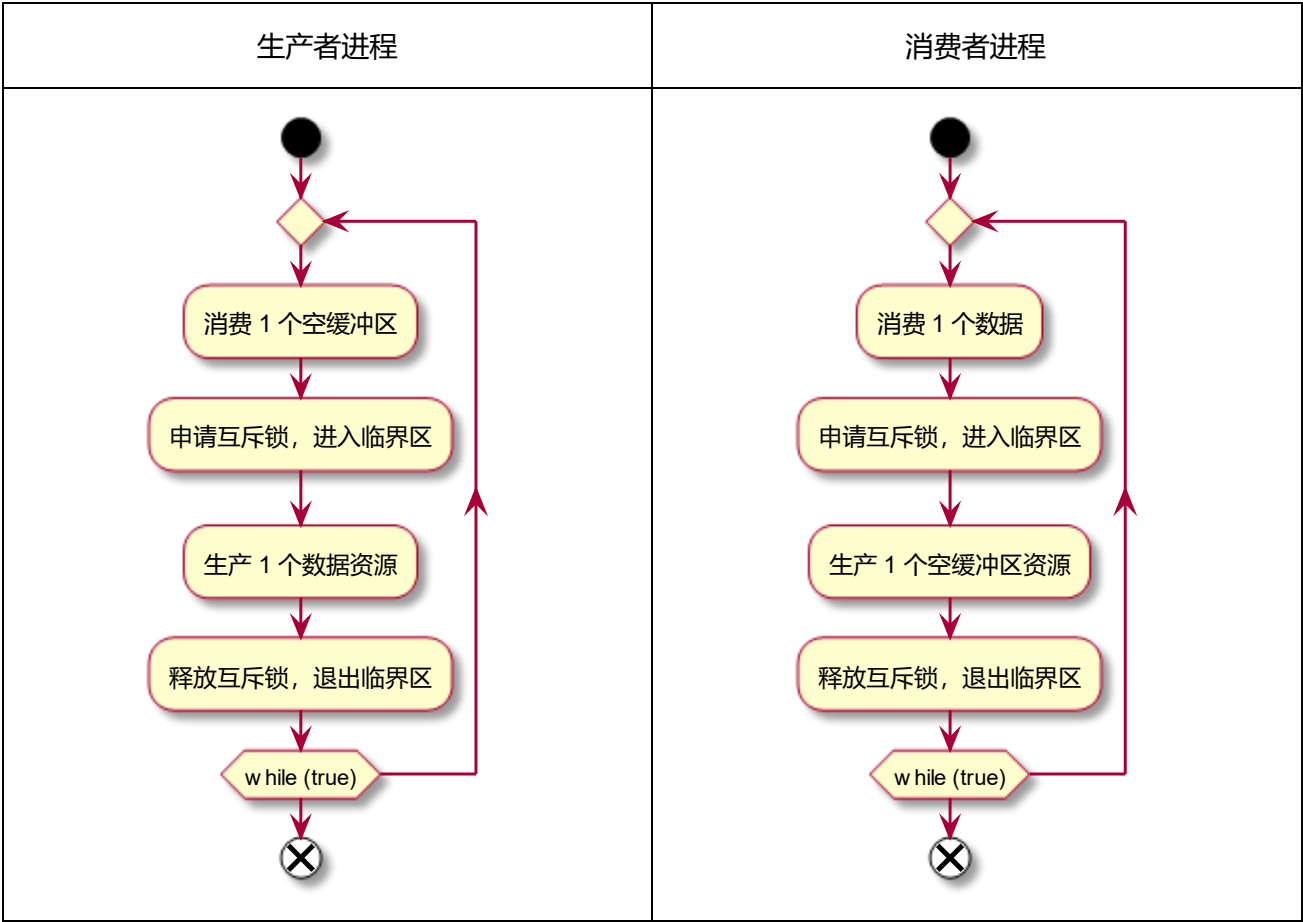
用数组表示缓冲区，信号量、互斥锁分别使用 pthread.h 提供的 sem_t, pthread_mutex_t 定义

```
#define bool      int           // 假装有 bool 类型
#define true      1
#define false     0
#define wait(x)   sem_wait((x)) // 宏定义 wait() 为 pthread 库函数
#define signal(x) sem_post((x)) // 宏定义 signal() 为 pthread 库函数
#define sleeping() Sleep(1000)  // 休眠 1000 ms
#define lock(x)   pthread_mutex_lock((x))
// 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x))
// 宏定义 unlock() 为 pthread 库函数

#define ITEM_TYPE int           // 定义产品类型 为 int
#define BUFFER_SIZE 10          // 缓冲区大小 设为 10

int id = 0;                     // 记录产品主键 id
sem_t empty, full;              // 信号量定义
int in_pos, out_pos;            // 记录缓冲区生产、消费的下标位置
pthread_mutex_t mutex;          // 缓冲区操作互斥锁
ITEM_TYPE buffer[BUFFER_SIZE]; // 缓冲区
```

算法描述流程图如下：



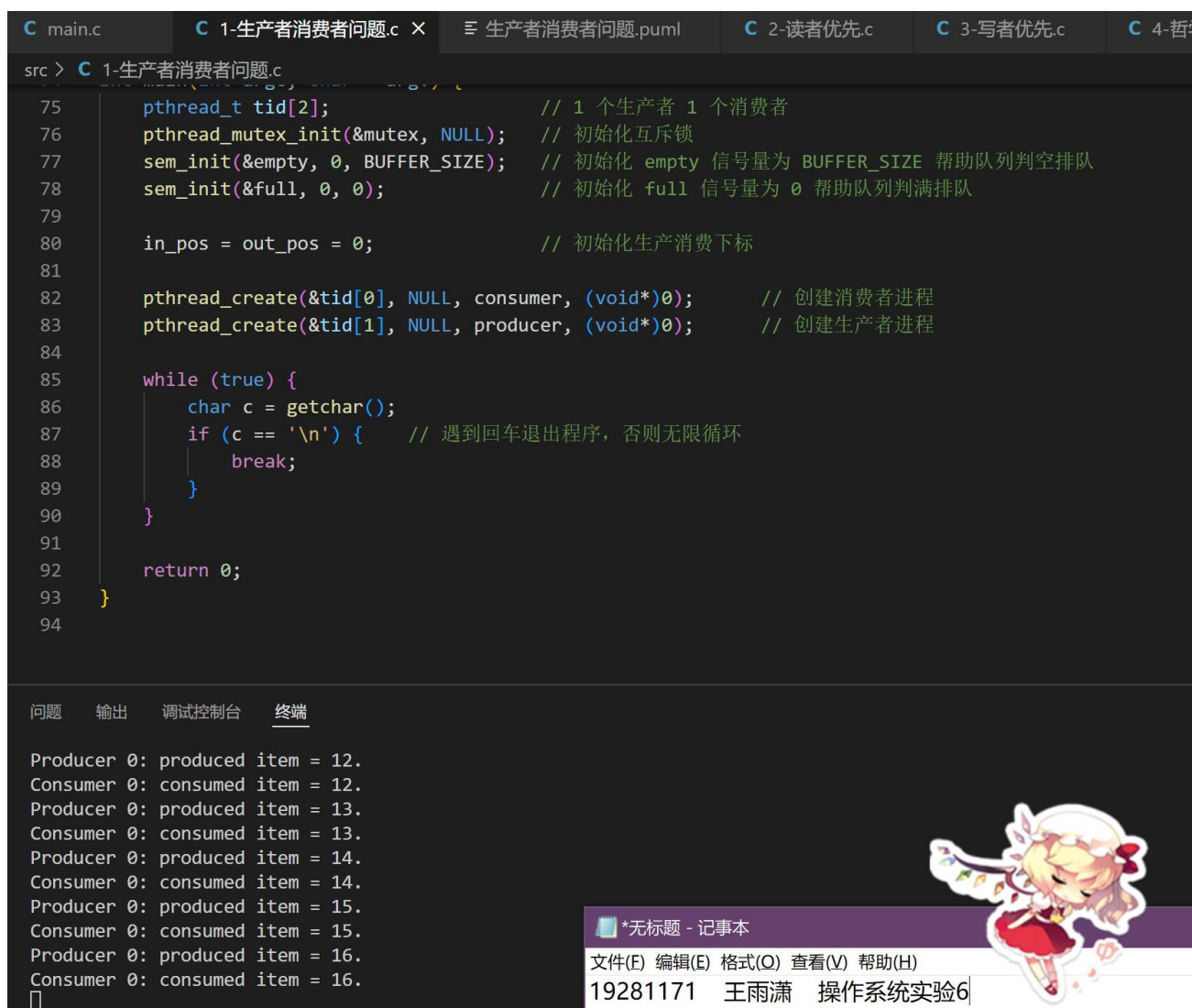
全部代码见“附录：源程序的完整代码”；

3.1.4. 运行截图

```
C main.c  C 1-生产者消费者问题.c x  生产者消费者问题.puml  C 2-读者优先.c  C 3-写者优先.c  C 4-哲学家问题.c
src > C 1-生产者消费者问题.c
75  pthread_t tid[2];                // 1 个生产者 1 个消费者
76  pthread_mutex_init(&mutex, NULL); // 初始化互斥锁
77  sem_init(&empty, 0, BUFFER_SIZE); // 初始化 empty 信号量为 BUFFER_SIZE 帮助队列判空排队
78  sem_init(&full, 0, 0);            // 初始化 full 信号量为 0 帮助队列判满排队
79
80  in_pos = out_pos = 0;              // 初始化生产消费下标
81
82  pthread_create(&tid[0], NULL, consumer, (void*)0); // 创建消费者进程
83  pthread_create(&tid[1], NULL, producer, (void*)0); // 创建生产者进程
84
85  while (true) {
86      char c = getchar();
87      if (c == '\n') { // 遇到回车退出程序, 否则无限循环
88          break;
89      }
90  }
91
92  return 0;
93 }
94

问题  输出  调试控制台  终端
Producer 0: produced item = 12.
Consumer 0: consumed item = 12.
Producer 0: produced item = 13.
Consumer 0: consumed item = 13.
Producer 0: produced item = 14.
Consumer 0: consumed item = 14.
Producer 0: produced item = 15.
Consumer 0: consumed item = 15.
Producer 0: produced item = 16.
Consumer 0: consumed item = 16.

```



3.2. 读者优先的读者-写者问题

3.2.1. 问题描述

读者-写者问题即, 存在一个多线程共享的临界资源数据对象, 有很多个写者进程不定时地对数据进行写入操作, 还有很多个读者进程不定时只要求读取该数据, 它们访问临界资源的时间均无法预料, 需要保证数据同步, 读者之间可以同时访问数据, 读写和写写进程之间不能同时访问。

本问题中需要保证读者进程都优先于写者进程。

3.2.2. 解决方案

首先用互斥锁实现读者-写者问题要求的读读共享，读写互斥，写写互斥机制。

要保证读进程具有优先权，可以按如下思路设计：当有一个及以上读进程正在读时，新来的写进程均被阻塞，新来的读进程可以直接加入；当没有读进程正在读时，写进程才能执行。

3.2.3. 数据结构和算法流程

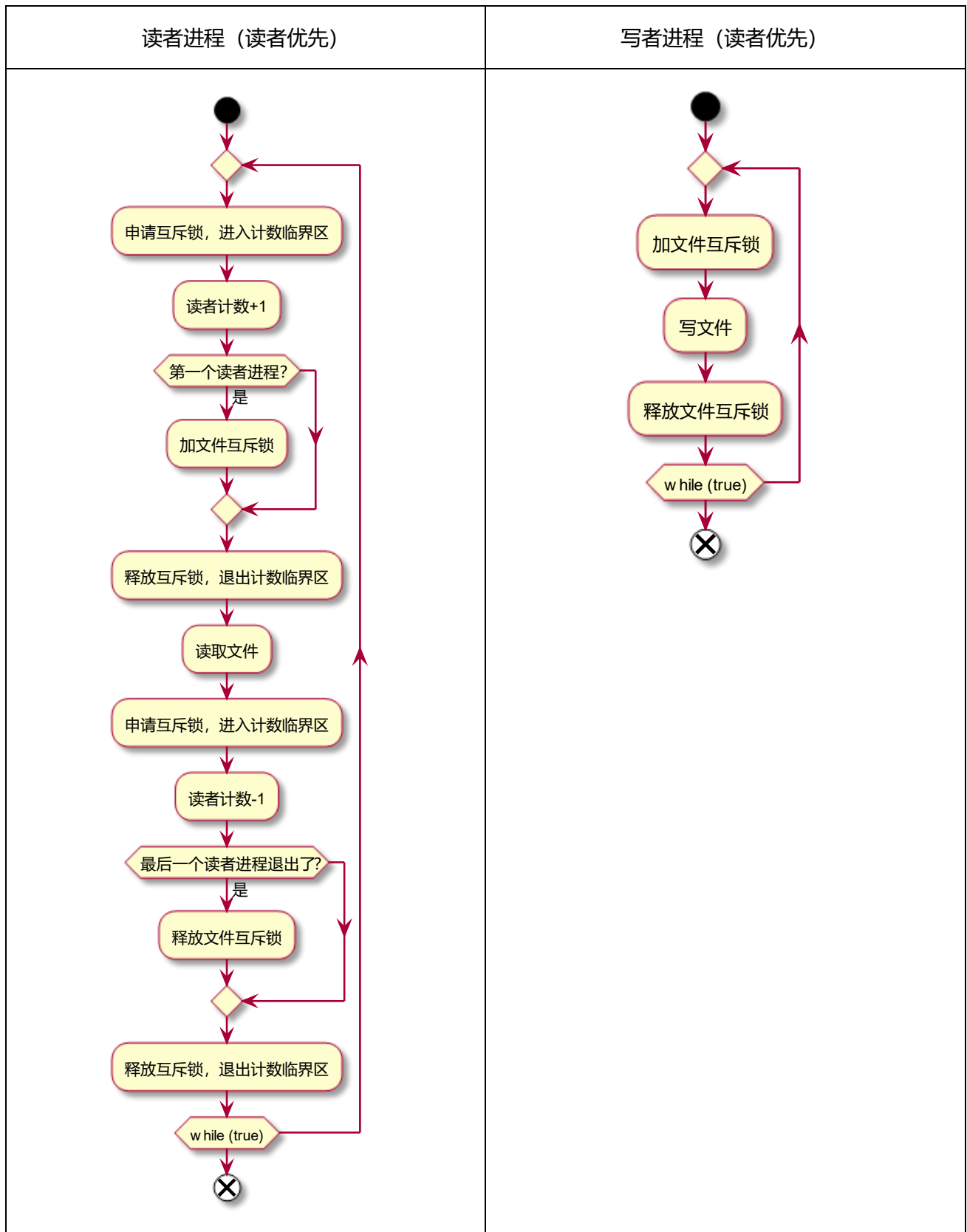
用一个整型变量记录当前同时读的读者数量（因为信号量不能用于判断），两个互斥锁分别用于控制文件读写互斥，读者计数修改互斥。

```
#define bool      int           // 假装有 bool 类型
#define true      1
#define false     0
#define lock(x)   sem_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) sem_post((x)) // 宏定义 unlock() 为 pthread 库函数
#define sleeping() Sleep(3000)  // 休眠 3000 ms
#define lock(x)   pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 unlock() 为 pthread 库函数

#define R_NUM 1 // 定义读者数量
#define W_NUM 4 // 定义写者数量

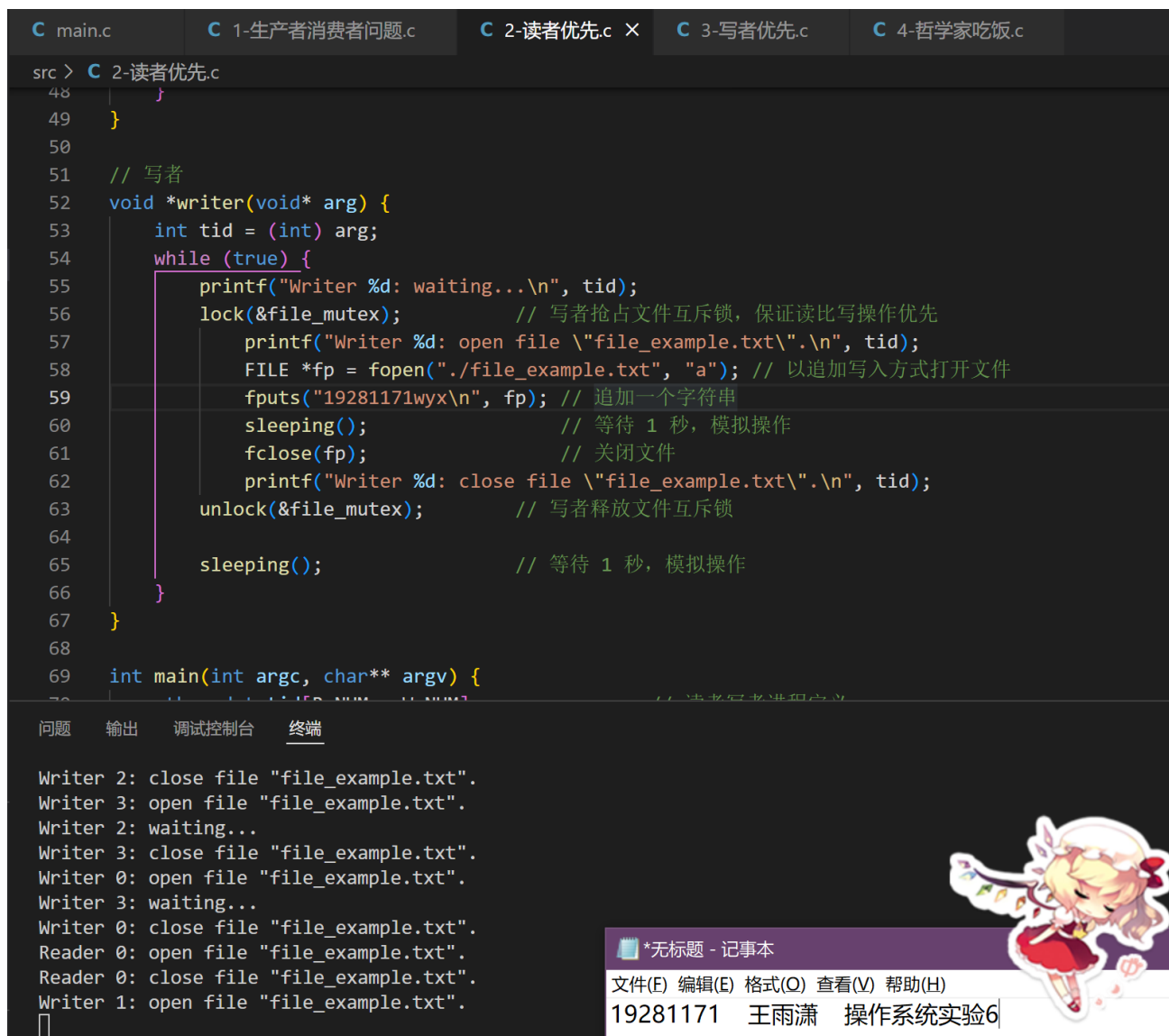
int reader_count = 0; // 当前读者数量
pthread_mutex_t file_mutex, read_cnt_mutex; // 信号量定义
```

算法描述流程图如下：



全部代码见“附录：源程序的完整代码”；

3.2.4. 运行截图



```
src > C 2-读者优先.c
48     }
49 }
50
51 // 写者
52 void *writer(void* arg) {
53     int tid = (int) arg;
54     while (true) {
55         printf("Writer %d: waiting...\n", tid);
56         lock(&file_mutex); // 写者抢占文件互斥锁, 保证读比写操作优先
57         printf("Writer %d: open file \"file_example.txt\".\n", tid);
58         FILE *fp = fopen("./file_example.txt", "a"); // 以追加写入方式打开文件
59         fputs("19281171wyx\n", fp); // 追加一个字符串
60         sleeping(); // 等待 1 秒, 模拟操作
61         fclose(fp); // 关闭文件
62         printf("Writer %d: close file \"file_example.txt\".\n", tid);
63         unlock(&file_mutex); // 写者释放文件互斥锁
64
65         sleeping(); // 等待 1 秒, 模拟操作
66     }
67 }
68
69 int main(int argc, char** argv) {
70     // ... (rest of the code is partially visible) ...
71 }
```

问题 输出 调试控制台 终端

```
Writer 2: close file "file_example.txt".
Writer 3: open file "file_example.txt".
Writer 2: waiting...
Writer 3: close file "file_example.txt".
Writer 0: open file "file_example.txt".
Writer 3: waiting...
Writer 0: close file "file_example.txt".
Reader 0: open file "file_example.txt".
Reader 0: close file "file_example.txt".
Writer 1: open file "file_example.txt".
█
```

*无标题 - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
19281171 王雨潇 操作系统实验6

3.3. 写者优先的读者-写者问题

3.3.1. 问题描述

读者-写者问题即, 存在一个多线程共享的临界资源数据对象, 有很多个写者进程不定时地对数据进行写入操作, 还有很多个读者进程不定时只要求读取该数据, 它们访问临界资源的时间均无法预料, 需要保证

数据同步，读者之间可以同时访问数据，读写和写写进程不能同时访问。

本问题中需要保证写进程都优先于读者进程。

3.3.2. 解决方案

需要记录同时读者、写者数量，用两个整型变量表示；通过互斥锁之间的前驱关系保证写者优先。

3.3.3. 数据结构和算法流程

用 5 把互斥锁控制复杂的互斥和前驱要求：

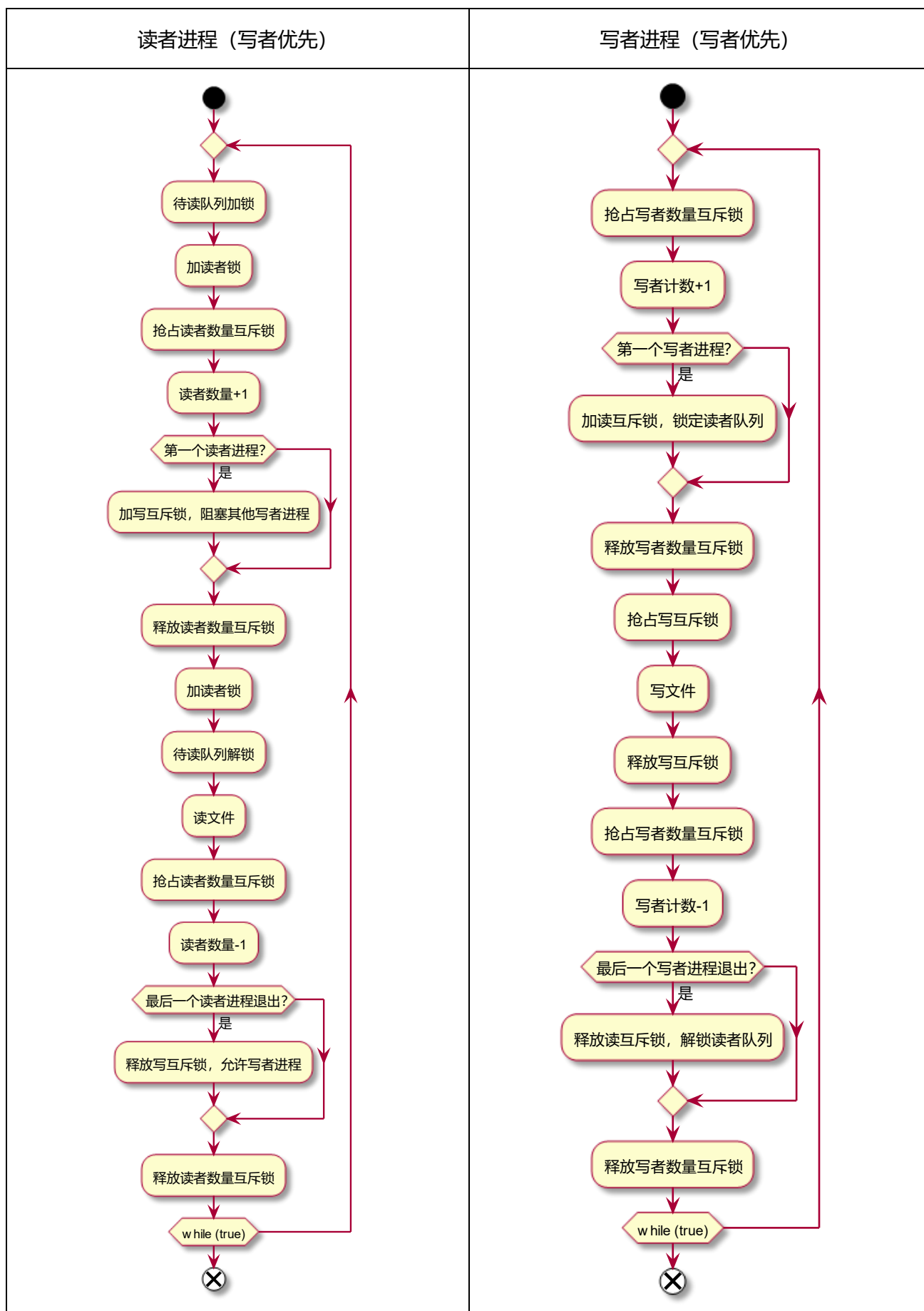
变量名	功能
read_mutex	读者和写者互斥
write_mutex	写者和写者之间互斥
queue_mutex	确保写者优先于读者的阻塞队列锁
read_cnt_mutex	读者计数修改互斥
write_cnt_mutex;	写者计数修改互斥

```
#define bool      int           // 假装有 bool 类型
#define true      1
#define false     0
#define lock(x)   sem_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) sem_post((x)) // 宏定义 unlock() 为 pthread 库函数
#define sleeping() Sleep(3000)   // 休眠 3000 ms
#define lock(x)   pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 unlock() 为 pthread 库函数

#define R_NUM 4 // 定义读者数量
#define W_NUM 1 // 定义写者数量

int reader_count = 0, writer_count = 0; // 当前读者、写者数量
pthread_mutex_t read_mutex, write_mutex, queue_mutex, read_cnt_mutex, write_cnt_mutex;
// 信号量定义
```

算法描述流程图如下：

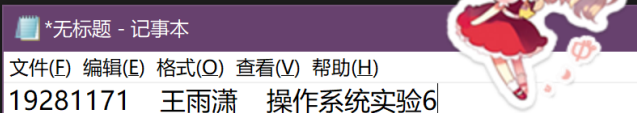


3.3.4. 运行截图

```
C main.c  C 1-生产者消费者问题.c  C 2-读者优先.c  C 3-写者优先.c X  C 4-哲学家吃饭.c
src > C 3-写者优先.c
59     printf("Writer %d: waiting...\n", tid);
60     lock(&write_cnt_mutex);    // 写者抢占写者数量互斥锁
61     if (writer_count == 0) {
62         lock(&read_mutex);    // 第一个写者来，锁定待读队列，保证写比读操作优先
63     }
64     writer_count++;            // 增加当前写者数量
65     unlock(&write_cnt_mutex);  // 写者释放写者数量互斥锁
66
67     lock(&write_mutex);        // 写者抢占写互斥锁
68     FILE *fp = fopen("./file_example.txt", "a"); // 以追加写入方式打开文件
69     printf("Writer %d: open file \"file_example.txt\".\n", tid);
70     fputs("19281171\n", fp);    // 追加一个字符串
71     fclose(fp);                // 关闭文件
72     printf("Writer %d: close file \"file_example.txt\".\n", tid);
73     unlock(&write_mutex);      // 写者释放写互斥锁
74
75     lock(&write_cnt_mutex);    // 写者抢占写者数量互斥锁
76     writer_count--;            // 减少当前写者数量
77     if (writer_count == 0) {
78         unlock(&read_mutex);  // 当最后一个写者离开，解锁读互斥锁，保证写比读操作优先
79     }
80     unlock(&write_cnt_mutex);  // 写者释放写者数量互斥锁

问题  输出  调试控制台  终端

Reader 1: open file "file_example.txt".
Reader 0: waiting...
Reader 0: open file "file_example.txt".
Reader 3: close file "file_example.txt".
Reader 1: close file "file_example.txt".
Reader 2: close file "file_example.txt".
Reader 0: close file "file_example.txt".
Writer 0: waiting...
Writer 0: open file "file_example.txt".
Writer 0: close file "file_example.txt".
Reader 1: waiting...
```



3.4. 哲学家就餐问题

3.4.1. 问题描述

有 N 个哲学家共用 N 支筷子，每支筷子都是一个临界资源，只能一次被一个哲学家持有；哲学家会不定期地使用筷子进餐，每当哲学家饥饿时，他就会依次取用圆桌上他左右两端的两支筷子，只有同时拿到两

支筷子的哲学家可以进餐，进餐完毕则把筷子放回原处继续思考。

3.4.2. 解决方案

本问题采用的解决方案是设置互斥锁，当一个哲学家想吃饭时，先用互斥锁禁止其他哲学家对筷子有任何操作，然后再拿起自己两侧筷子，才释放互斥锁，就餐，放下筷子时故技重施，从而避免死锁发生。

3.4.3. 数据结构和算法流程

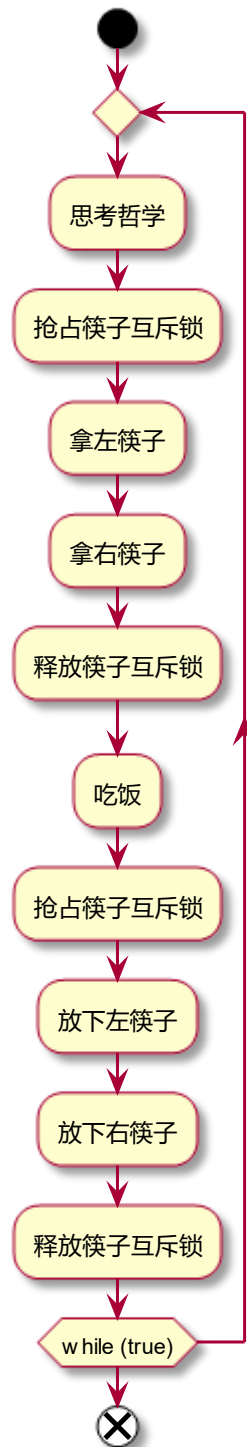
用初始化为 1 的信号量表示临界资源筷子，再设置对筷子操作的互斥锁，对筷子使用情况的操作必须先加锁后解锁。

```
#define bool      int           // 假装有 bool 类型
#define true      1
#define false     0
#define wait(x)   sem_wait((x)) // 宏定义 wait()
#define signal(x) sem_post((x)) // 宏定义 signal()
#define eating()  Sleep(1000)   // 休眠 1000 ms 模拟哲学家吃饭
#define thinking(x) Sleep((x)) // 不定长休眠模拟哲学家思考
#define lock(x)   pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 lock() 为 pthread 库函数

#define P_NUM 5 // 定义哲学家人数

sem_t chopstick[P_NUM]; // 信号量定义
pthread_mutex_t mutex;   // 筷子操作互斥锁
```

算法描述流程图如下：(单个哲学家进程的行为)



3.4.4. 运行截图

C main.cC 1-生产者消费者问题.cC 2-读者优先.cC 3-写者优先.cC 4-哲学家吃饭.c X

src > C 4-哲学家吃饭.c


```
43     }
44
45     int main(int argc, char** argv) {
46         int i;
47         srand(time(NULL));
48         pthread_t tid[P_NUM];           // 定义哲学家
49         pthread_mutex_init(&mutex, NULL); // 初始化互斥锁
50         for(i=0; i<P_NUM; i++) {
51             sem_init(&chopstick[i], 0, 1); // 初始化筷子
52         }
53
54         for(i=0; i<P_NUM; i++) {
55             pthread_create(&tid[i], NULL, philosopher, (void*)i); // 创建哲学家进程
56         }
57
58         while (true) {
59             char c = getchar();
60             if (c == '\n') { // 遇到回车退出程序, 否则无限循环
61                 break;
62             }
63         }
64     }
```

问题 输出 调试控制台 终端

Philosopher 0: I put down a pair of chopsticks.
Philosopher 0: I'm thinking...
Philosopher 1: I'm hungry...
Philosopher 1: I got a pair of chopsticks.
Philosopher 1: I'm eating...
Philosopher 3: I put down a pair of chopsticks.
Philosopher 3: I'm thinking...
Philosopher 4: I'm hungry...
Philosopher 4: I got a pair of chopsticks.
Philosopher 4: I'm eating...
p

*无标题 - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
19281171 王雨潇 操作系统实验6



4. 实验结论和心得体会

通过本次实验, 我了解了 Windows 操作系统下 C 语言 pthread 库提供的同步机制方法, 通过解决这四个经典的同步问题, 我在实践中验证了自己对信号量、互斥锁等概念的理论认知, 也对同步编程有了更深入的理解和认识。本次实验的主要难点在于理清各个事件的前驱条件, 冲突条件, 而且加锁以后就要避免死锁的发生, 才能写出可以正常工作的程序。

5. 附录：源程序的完整代码

5.1. 生产者-消费者问题

```
// 生产者-消费者问题

#include <stdio.h>
#include <windows.h>
#include <pthread.h>
#include <semaphore.h>

#define bool      int           // 假装有 bool 类型
#define true      1
#define false     0
#define wait(x)   sem_wait((x)) // 宏定义 wait() 为 pthread 库函数
#define signal(x) sem_post((x)) // 宏定义 signal() 为 pthread 库函数
#define sleeping() Sleep(1000)  // 休眠 1000 ms
#define lock(x)   pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 unlock() 为 pthread 库函数

#define ITEM_TYPE int           // 定义产品类型 为 int
#define BUFFER_SIZE 10          // 缓冲区大小 设为 10

int id = 0;                      // 记录产品主键 id
sem_t empty, full;               // 信号量定义
int in_pos, out_pos;             // 记录缓冲区生产、消费的下标位置
pthread_mutex_t mutex;           // 缓冲区操作互斥锁
ITEM_TYPE buffer[BUFFER_SIZE];  // 缓冲区

// 生产产品的函数
bool insert_item(ITEM_TYPE item) {
    // 将 item 放进缓冲区
    buffer[out_pos] = item;
    out_pos = (out_pos + 1) % BUFFER_SIZE;
    return true;
}

// 消费产品的函数
bool remove_item(ITEM_TYPE *item) {
    // 将缓冲区 in_pos 下标的产品移除，取到 item 中
    *item = buffer[in_pos];
    in_pos = (in_pos + 1) % BUFFER_SIZE;
    return true;
}

// 生产者进程
void *producer(void* arg) {
    int tid = (int) arg;
```

```

    while (true) {
        wait(&empty);        // 等待队列
        lock(&mutex);        // 申请互斥锁
        insert_item(id);     // 放入生产的产品
        printf("Producer %d: produced item = %d.\n", tid, id);
        id++;
        unlock(&mutex);      // 释放互斥锁
        signal(&full);

        sleeping();          // 休眠
    }
}

// 消费者进程
void *consumer(void* arg) {
    int tid = (int) arg;
    while (true) {
        wait(&full);        // 等待队列
        lock(&mutex);        // 申请互斥锁
        int item;
        remove_item(&item); // 消费生产的产品
        printf("Consumer %d: consumed item = %d.\n", tid, item);
        unlock(&mutex);      // 释放互斥锁
        signal(&empty);

        sleeping();          // 休眠
    }
}

int main(int argc, char** argv) {
    pthread_t tid[2];        // 1 个生产者 1 个消费者
    pthread_mutex_init(&mutex, NULL); // 初始化互斥锁
    sem_init(&empty, 0, BUFFER_SIZE); // 初始化 empty 信号量为 BUFFER_SIZE
    帮助队列判空排队
    sem_init(&full, 0, 0);    // 初始化 full 信号量为 0 帮助队列判满
    排队

    in_pos = out_pos = 0;    // 初始化生产消费下标

    pthread_create(&tid[0], NULL, consumer, (void*)0); // 创建消费者进程
    pthread_create(&tid[1], NULL, producer, (void*)0); // 创建生产者进程

    while (true) {
        char c = getchar();
        if (c == '\n') {    // 遇到回车退出程序，否则无限循环
            break;
        }
    }

    return 0;
}

```

5.2. 读者优先的读者-写者问题

// 读者优先的读者-写者问题

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

#define bool    int           // 假装有 bool 类型
#define true    1
#define false   0
#define lock(x)  sem_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) sem_post((x)) // 宏定义 unlock() 为 pthread 库函数
#define sleeping() Sleep(3000) // 休眠 3000 ms
#define lock(x)  pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 unlock() 为 pthread 库函数

#define R_NUM 1 // 定义读者数量
#define W_NUM 4 // 定义写者数量

int reader_count = 0; // 当前读者数量
pthread_mutex_t file_mutex, read_cnt_mutex; // 信号量定义

// 读者优先的读者进程
void *reader(void* arg) {
    int tid = (int) arg;
    while (true) {
        printf("Reader %d: waiting...\n", tid);
        lock(&read_cnt_mutex); // 读者抢占读者计数互斥锁
        if (reader_count == 0) {
            lock(&file_mutex); // 第一个读者需要抢占文件互斥锁，保证读比写
操作优先
        }
        reader_count++;
        unlock(&read_cnt_mutex); // 读者释放读者计数互斥锁

        printf("Reader %d: open file \"file_example.txt\".\n", tid);
        FILE *fp = fopen("./file_example.txt", "r"); // 以读方式打开文件
        fclose(fp); // 关闭文件
        printf("Reader %d: close file \"file_example.txt\".\n", tid);

        lock(&read_cnt_mutex); // 读者抢占读者计数互斥锁
        reader_count--;
        if (reader_count == 0) {
            unlock(&file_mutex); // 最后一个读者离开需要释放文件互斥锁
        }
        unlock(&read_cnt_mutex); // 读者释放读者计数互斥锁

        sleeping(); // 等待 1 秒，模拟操作
    }
}
```

```

    }
}

// 写者进程
void *writer(void* arg) {
    int tid = (int) arg;
    while (true) {
        printf("Writer %d: waiting...\n", tid);
        lock(&file_mutex);           // 写者抢占文件互斥锁，保证读比写操作优先
        printf("Writer %d: open file \"file_example.txt\".\n", tid);
        FILE *fp = fopen("./file_example.txt", "a"); // 以追加写入方式打开文件
        fputs("19281171wyx\n", fp); // 追加一个字符串
        sleeping();                  // 等待 1 秒，模拟操作
        fclose(fp);                  // 关闭文件
        printf("Writer %d: close file \"file_example.txt\".\n", tid);
        unlock(&file_mutex);         // 写者释放文件互斥锁

        sleeping();                  // 等待 1 秒，模拟操作
    }
}

int main(int argc, char** argv) {
    pthread_t tid[R_NUM + W_NUM];    // 读者写者进程定义
    pthread_mutex_init(&file_mutex, NULL); // 文件互斥锁
    pthread_mutex_init(&read_cnt_mutex, NULL); // 读者数量操作的互斥锁

    int i;
    for(i=0; i<R_NUM; i++) {
        pthread_create(&tid[i], NULL, reader, (void*)i); // 创建读者进程
    }
    for(i=0; i<W_NUM; i++) {
        pthread_create(&tid[R_NUM + i], NULL, writer, (void*)i); // 创建写者进
程
    }

    while (true) {
        char c = getchar();
        if (c == '\n') { // 遇到回车退出程序，否则无限循环
            break;
        }
    }

    return 0;
}

```

5.3. 写者优先的读者-写者问题

// 写者优先的读者-写者问题

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

#define bool    int           // 假装有 bool 类型
#define true    1
#define false   0
#define lock(x)  sem_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) sem_post((x)) // 宏定义 unlock() 为 pthread 库函数
#define sleeping() Sleep(3000) // 休眠 3000 ms
#define lock(x)  pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 unlock() 为 pthread 库函数

#define R_NUM 4 // 定义读者数量
#define W_NUM 1 // 定义写者数量

int reader_count = 0, writer_count = 0; // 当前读者、写者数量
pthread_mutex_t read_mutex, write_mutex, queue_mutex, read_cnt_mutex,
write_cnt_mutex; // 信号量定义

// 读者进程
void *reader(void* arg) {
    int tid = (int) arg;
    while (true) {
        printf("Reader %d: waiting...\n", tid);
        lock(&queue_mutex); // 待读队列加锁
        lock(&read_mutex); // 没等到读机会的读者进程都会阻塞在队列这里
        lock(&read_cnt_mutex); // 读者抢占读数量互斥锁
        if (reader_count == 0) {
            lock(&write_mutex); // 第一个读者
        }
        reader_count++; // 增加当前读者数量
        unlock(&read_cnt_mutex); // 读者释放读数量互斥锁
        unlock(&read_mutex); // 释放读锁
        unlock(&queue_mutex); // 待读队列释放

        printf("Reader %d: open file \"file_example.txt\".\n", tid);
        FILE *fp = fopen("./file_example.txt", "r"); // 以读方式打开文件

        fclose(fp); // 关闭文件
        printf("Reader %d: close file \"file_example.txt\".\n", tid);

        lock(&read_cnt_mutex); // 读者抢占读互斥锁
        reader_count--; // 减少当前读者数量
        if (reader_count == 0) {
```

```

        unlock(&write_mutex);    // 最后一个读者，放开写限制
    }
    unlock(&read_cnt_mutex); // 读者释放读互斥锁

    sleeping();
}
}

// 写者优先的写者进程
void *writer(void* arg) {
    int tid = (int) arg;
    while (true) {
        printf("Writer %d: waiting...\n", tid);
        lock(&write_cnt_mutex);    // 写者抢占写者数量互斥锁
        if (writer_count == 0) {
            lock(&read_mutex);    // 第一个写者来，锁定待读队列，保证写比读操
            作优先
        }
        writer_count++;            // 增加当前写者数量
        unlock(&write_cnt_mutex); // 写者释放写者数量互斥锁

        lock(&write_mutex);        // 写者抢占写互斥锁
        FILE *fp = fopen("./file_example.txt", "a"); // 以追加写入方式打开文件
        printf("Writer %d: open file \"file_example.txt\".\n", tid);
        fputs("19281171\n", fp); // 追加一个字符串
        fclose(fp);              // 关闭文件
        printf("Writer %d: close file \"file_example.txt\".\n", tid);
        unlock(&write_mutex);    // 写者释放写互斥锁

        lock(&write_cnt_mutex);    // 写者抢占写者数量互斥锁
        writer_count--;           // 减少当前写者数量
        if (writer_count == 0) {
            unlock(&read_mutex); // 当最后一个写者离开，解锁读互斥锁，保证写比读
            操作优先
        }
        unlock(&write_cnt_mutex); // 写者释放写者数量互斥锁

        sleeping();
    }
}

int main(int argc, char** argv) {
    pthread_t tid[R_NUM + W_NUM]; // 读者写者进程定义
    pthread_mutex_init(&read_mutex, NULL); // 初始化为 1 作为读写者之间的互
    斥锁使用
    pthread_mutex_init(&write_mutex, NULL); // 初始化为 1 作为写操作之间
    的互斥锁使用
    pthread_mutex_init(&queue_mutex, NULL); // 作为待读者队列互斥锁
    pthread_mutex_init(&read_cnt_mutex, NULL); // 作为读者数量互斥锁
    pthread_mutex_init(&write_cnt_mutex, NULL); // 作为写者数量互斥锁

    int i;
    for(i=0; i<R_NUM; i++) {

```

```

        pthread_create(&tid[i], NULL, reader, (void*)i); // 创建读者进程
    }
    for(i=0; i<W_NUM; i++) {
        pthread_create(&tid[R_NUM + i], NULL, writer, (void*)i); // 创建写者进
程
    }

    while (true) {
        char c = getchar();
        if (c == '\n') { // 遇到回车退出程序，否则无限循环
            break;
        }
    }

    return 0;
}

```

5.4. 哲学家就餐问题

```

// 哲学家就餐问题

#include <stdio.h>
#include <windows.h>
#include <pthread.h>
#include <semaphore.h>

#define bool    int           // 假装有 bool 类型
#define true    1
#define false   0
#define wait(x)  sem_wait((x)) // 宏定义 wait()
#define signal(x) sem_post((x)) // 宏定义 signal()
#define eating() Sleep(1000)    // 休眠 1000 ms 模拟哲学家吃饭
#define thinking(x) Sleep((x)) // 不定长休眠模拟哲学家思考
#define lock(x)  pthread_mutex_lock((x)) // 宏定义 lock() 为 pthread 库
函数
#define unlock(x) pthread_mutex_unlock((x)) // 宏定义 lock() 为 pthread 库
函数

#define P_NUM    5 // 定义哲学家人数

sem_t chopstick[P_NUM]; // 信号量定义
pthread_mutex_t mutex; // 筷子操作互斥锁

// 哲学家进程
void *philosopher(void* arg) {
    int id = (int) arg;
    while (true) {
        printf("Philosopher %d: I'm thinking...\n", id);
        thinking((rand() % 5000)); // 该哲学家思考

        printf("Philosopher %d: I'm hungry...\n", id);
    }
}

```

```

    lock(&mutex);                // 抢占互斥锁以禁止其他哲学家拿筷子
    wait(&chopstick[id]);        // 拿左手筷子
    wait(&chopstick[(id+1) % P_NUM]); // 拿右手筷子
    printf("Philosopher %d: I got a pair of chopsticks.\n", id);
    unlock(&mutex);              // 释放互斥锁，允许其他哲学家拿筷子

    printf("Philosopher %d: I'm eating...\n", id);
    eating();                    // 该哲学家吃饭

    signal(&chopstick[id]);      // 放下左手筷子
    signal(&chopstick[(id+1) % P_NUM]); // 放下右手筷子
    printf("Philosopher %d: I put down a pair of chopsticks.\n", id);
}
}

int main(int argc, char** argv) {
    int i;
    srand(time(NULL));
    pthread_t tid[P_NUM];        // 定义哲学家
    pthread_mutex_init(&mutex, NULL); // 初始化互斥锁
    for(i=0; i<P_NUM; i++) {
        sem_init(&chopstick[i], 0, 1); // 初始化筷子
    }

    for(i=0; i<P_NUM; i++) {
        pthread_create(&tid[i], NULL, philosopher, (void*)i); // 创建哲学家进程
    }

    while (true) {
        char c = getchar();
        if (c == '\n') { // 遇到回车退出程序，否则无限循环
            break;
        }
    }

    return 0;
}

```