姓名：陈伟杰
学号：71066001
操作系统第 4 次作业

1. 读者写者问题（写者优先）: 1）共享读; 2）互斥写、读写互斥; 3）写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。

答：首先定义 共享读是即可以有一个或多个读者在读，互斥写、读写互斥：在写者写的时候，其他读者和写者都不能同时对该资源进行操作。在读者读的时候，其他写者不能对该资源进行写操作，既不存在两个写者同时进行写操作和一个线程在读另一个在写。

代码：

```c
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include <sys/types.h>
# include <pthread.h>
# include <semaphore.h>
# include <string.h>
# include <unistd.h>

//semaphores
sem_t RWMutex, mutex1, mutex2, mutex3, wrt;
//全局变量
int writeCount, readCount;


struct data {
    int id;
    int opTime;
    int lastTime;
};

//读者
void* Reader(void* param) {
    int id = ((struct data*)param)->id;
    int lastTime = ((struct data*)param)->lastTime;
    int opTime = ((struct data*)param)->opTime;

    sleep(opTime);
    printf("Thread %d: waiting to read\n", id);

    sem_wait(&mutex3);
    sem_wait(&RWMutex);
    sem_wait(&mutex2);
    readCount++;
    if(readCount == 1)
        sem_wait(&wrt);
    sem_post(&mutex2);
    sem_post(&RWMutex);
    sem_post(&mutex3);

    printf("Thread %d: start reading\n", id);
    /* reading is performed */
    sleep(lastTime);
    printf("Thread %d: end reading\n", id);

```

```
    sem_wait(&mutex2);
    readCount--;
    if(readCount == 0)
        sem_post(&wrt);
    sem_post(&mutex2);

    pthread_exit(0);
}

//写者
void* Writer(void* param) {
    int id = ((struct data*)param)->id;
    int lastTime = ((struct data*)param)->lastTime;
    int opTime = ((struct data*)param)->opTime;

    sleep(opTime);
    printf("Thread %d: waiting to write\n", id);

    sem_wait(&mutex1);
    writeCount++;
    if(writeCount == 1){//当writeCount等于1的时候，申请信号量RWMutex，其余的写者无需再次申请
        sem_wait(&RWMutex);
    }
    sem_post(&mutex1);

    sem_wait(&wrt);
    printf("Thread %d: start writing\n", id);
    /* writing is performed */
    sleep(lastTime);
    printf("Thread %d: end writing\n", id);
    sem_post(&wrt);

    sem_wait(&mutex1);
    writeCount--;
    if(writeCount == 0) {//当写者的数量writeCount等于0的时候，则证明此时没有没有读者了，释放信号量RWMutex
        sem_post(&RWMutex);
    }
    sem_post(&mutex1);

    pthread_exit(0);
}


int main() {
    //pthread
    pthread_t tid;

    pthread_attr_t attr;


    pthread_attr_init(&attr);


    sem_init(&mutex1, 0, 1);
    sem_init(&mutex2, 0, 1);
    sem_init(&mutex3, 0, 1);
    sem_init(&wrt, 0, 1);
    sem_init(&RWMutex, 0, 1);

    readCount = writeCount = 0;

    int id = 0;
    while(scanf("%d", &id) != EOF) {

        char role;
        int opTime;
        int lastTime;

        scanf("%c%d%d", &role, &opTime, &lastTime);
        struct data* d = (struct data*)malloc(sizeof(struct data));

        d->id = id;
        d->opTime = opTime;
        d->lastTime = lastTime;

        if(role == 'R') {
            printf("Create the %d thread: Reader\n", id);
            pthread_create(&tid, &attr, Reader, d);

        }
        else if(role == 'W') {
            printf("Create the %d thread: Writer\n", id);
            pthread_create(&tid, &attr, Writer, d);
        }
    }

    sem_destroy(&mutex1);
    sem_destroy(&mutex2);
    sem_destroy(&mutex3);
    sem_destroy(&RWMutex);
    sem_destroy(&wrt);

    return 0;
}
```

运行结果解释

```
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3Create the 1 thread: Reader
Create the 2 thread: Writer
Create the 3 thread: Reader
Create the 4 thread: Reader
Thread 1: waiting to read
Thread 1: start reading
Thread 2: waiting to write
Thread 2: start writing
Thread 3: waiting to read
Thread 3: start reading
Thread 4: waiting to read
Thread 4: start reading
Thread 3: end reading
Thread 1: end reading
Thread 2: end writing
Thread 4: end reading
```

共享读：可以看到 thread 3 和 4 在读
互斥写、读写互斥：没有看到一个线程同时 start reading 和 start writing 在同一时间
写者优先：读者必须等到没有写者处于等待状态后才能开始读操作。 Thread2 （写者）start Writing 之后 Thread3 才开始读操作

2. 寿司店问题。假设一个寿司店有 5 个座位，如果你到达的时候有一个空座位，你可以立刻就坐。但是如果你到达的时候 5 个座位都是满的有人已经就坐，这就意味着这些人都是一 起来吃饭的，那么你需要等待所有的人一起离开才能就坐。编写同步原语，实现这个场景的约束。

答：这次代码实现了一个简单的餐厅场景，包含了进入餐厅、用餐、离开餐厅等操作。使用了两个信号量，一个用于控制餐厅座位的数量，另一个用于控制等待座位的人数

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

// Initialize semaphores
sem_t seats;
sem_t waiting;
int num_waiting = 0;
pthread_mutex_t waiting_mutex = PTHREAD_MUTEX_INITIALIZER;

// Function for a person to enter the sushi restaurant
void* enter_restaurant(void* arg) {
    printf("A person enters the restaurant.\n");
    // Try to acquire a seat
    sem_wait(&seats);
    printf("A person sits down.\n");
    // Eat sushi
    printf("A person eats sushi.\n");
    // Leave the restaurant
    sem_post(&seats);
    printf("A person leaves the restaurant.\n");
    // Check if anyone is waiting
    pthread_mutex_lock(&waiting_mutex);
    if (num_waiting > 0) {
        num_waiting--;
        sem_post(&waiting);
        printf("A person from the waiting list sits down.\n");
    }
    pthread_mutex_unlock(&waiting_mutex);
    return NULL;
}

// Function for a group of people to enter the sushi restaurant
void* enter_group(void* arg) {
    int num_people = *(int*)arg;
    free(arg);
    printf("A group of %d people enters the restaurant.\n", num_people);
    // Try to acquire all seats
    for (int i = 0; i < num_people; i++) {
        sem_wait(&seats);
    }
    printf("The group sits down.\n");
    // Eat sushi
    printf("The group eats sushi.\n");

    // Leave the restaurant
    for (int i = 0; i < num_people; i++) {
        sem_post(&seats);
    }
    printf("The group leaves the restaurant.\n");
    // Check if anyone is waiting
    pthread_mutex_lock(&waiting_mutex);
    if (num_waiting > 0) {
        int num_seats = sem_trywait(&seats);
        if (num_seats == 0) {
            // Release extra seat
            sem_post(&seats);
            num_seats--;
        }
        for (int i = 0; i < num_seats && num_waiting > 0; i++) {
            num_waiting--;
            sem_post(&waiting);
            printf("A person from the waiting list sits down.\n");
        }
    }
    pthread_mutex_unlock(&waiting_mutex);
    return NULL;
}

// Function for a person to leave the sushi restaurant
void* leave_restaurant(void* arg) {
    printf("A person leaves the restaurant.\n");
    sem_post(&seats);
    // Check if anyone is waiting
    pthread_mutex_lock(&waiting_mutex);
    if (num_waiting > 0) {
        num_waiting--;
        sem_post(&waiting);
        printf("A person from the waiting list sits down.\n");
    }
    pthread_mutex_unlock(&waiting_mutex);
    return NULL;
}

// Test the synchronization primitives
int main() {
    // Initialize semaphores
    sem_init(&seats, 0, 5);
    sem_init(&waiting, 0, 0);
```

```
    pthread_t t1, t2, t3, t4, t5, t6, t7, t8, t9;
    pthread_create(&t1, NULL, enter_restaurant, NULL);
    pthread_create(&t2, NULL, enter_restaurant, NULL);
    pthread_create(&t3, NULL, enter_restaurant, NULL);
    pthread_create(&t4, NULL, enter_restaurant, NULL);
    pthread_create(&t5, NULL, enter_restaurant, NULL);
    int* arg6 = malloc(sizeof(int));
    *arg6 = 3;
    pthread_create(&t6, NULL, enter_group, arg6);
    pthread_create(&t7, NULL, enter_restaurant, NULL);
    pthread_create(&t8, NULL, leave_restaurant, NULL);
    pthread_create(&t9, NULL, enter_restaurant, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);
    pthread_join(t7, NULL);
    pthread_join(t8, NULL);
    pthread_join(t9, NULL);

    // Destroy semaphores
    sem_destroy(&seats);
    sem_destroy(&waiting);
    return 0;
}
```

```
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A group of 3 people enters the restaurant.
The group sits down.
The group eats sushi.
The group leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person enters the restaurant.
A person sits down.
A person eats sushi.
A person leaves the restaurant.
A person leaves the restaurant.
```

3. 三个进程 P1、P2、P3 互斥使用一个包含 N（N>0）个单元的缓冲区。P1 每次用 produce()生成一个正整数并用 put()送入缓冲区某一个空单元中；P2 每次用 getodd()从该缓冲区中取出一个奇数并用 countodd()统计奇数个数；P3 每次用 geteven()从该缓冲区中取出一个偶数并用 counteven()统计偶数个数。请用信号量机制实现这三个进程的同步与互斥活动，并说明所定义的信号量的含义。要求用伪代码描述。

答：

```
semaphore odd = 0,even = 0,empty = N,mutex = 1;
main()
cobegin{
P1()
{
number = produce (); //生成一个数
P(empty); //判断缓冲区是否有空单元
P (mutex);
Put(); //缓冲区是否被占用
V(mutex); /释放缓冲区
if(number %2 == 0)
V(even); //如果是偶数，向 P3发出信号
else
V(odd); //如果是奇数，向 P2发出信号
}


P2()
{
P (odd); //收到P1发来的信号，已产生一个奇数
P (mutex); // 缓冲区是否被占用
getodd ();
V (mutex); //释放缓冲区
V(empty); //向P1发信号，多出一个空单元
countodd ();



P3()
{
P (even); //收到P1发来的信号，已产生一个偶数
P (mutext); //缓冲区是否被占用
geteven();
V (mutex); //释放缓冲区
V(empty); //向P1发信号，多出一个空单元
counteven();
}
}coend
}
```

4. 搜索-插入-删除问题。三个线程对一个单链表进行并发的访问，分别进行搜索、插入和删除。搜索线程仅仅读取链表，因此多个搜索线程可以并发。插入线程把数据项插入到链表最后的位置；多个插入线程必须互斥防止同时执行插入操作。但是，一个插入线程可以和多个搜索线程并发执行。最后，删除线程可以从链表中任何一个位置删除数据。一次只能有一个删除线程执行；删除线程之间，删除线程和搜索线程，删除线程和插入线程都不能同时执行。请编写三类线程的同步互斥代码，描述这种三路的分类互斥问题。

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

sem_t mutex;
sem_t insert_count;
sem_t search_count;

// 搜索线程
void* search(void* arg) {
    sem_wait(&search_count);   // 获取搜索计数信号量
    read_from_linked_list();
    sem_post(&search_count);   // 释放搜索计数信号量
    return NULL;
}

// 插入线程
void* insert(void* arg) {
    sem_wait(&insert_count);   // 获取插入计数信号量
    sem_wait(&mutex);   // 获取互斥信号量
    insert_into_linked_list();
    sem_post(&mutex);   // 释放互斥信号量
    sem_post(&insert_count);   // 释放插入计数信号量
    return NULL;
}

// 删除线程
void* delete(void* arg) {
    sem_wait(&mutex);   // 获取互斥信号量
    delete_from_linked_list();
    sem_post(&mutex);   // 释放互斥信号量
    return NULL;
}

int main() {
    // 初始化信号量
    sem_init(&mutex, 0, 1);
    sem_init(&insert_count, 0, 1);
    sem_init(&search_count, 0, 10);
```

答：

```c
    // 创建线程
    pthread_t t1, t2, t3, t4, t5, t6, t7, t8, t9;
    pthread_create(&t1, NULL, search, NULL);
    pthread_create(&t2, NULL, search, NULL);
    pthread_create(&t3, NULL, search, NULL);
    pthread_create(&t4, NULL, search, NULL);
    pthread_create(&t5, NULL, search, NULL);
    pthread_create(&t6, NULL, insert, NULL);
    pthread_create(&t7, NULL, search, NULL);
    pthread_create(&t8, NULL, delete, NULL);
    pthread_create(&t9, NULL, search, NULL);

    // 等待线程结束
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);
    pthread_join(t7, NULL);
    pthread_join(t8, NULL);
    pthread_join(t9, NULL);

    // 销毁信号量
    sem_destroy(&mutex);
    sem_destroy(&insert_count);
    sem_destroy(&search_count);
    return 0;
```

这种三路的分类互斥问题就是这三个进程同时访问同一个资源但它们的“工作”是不一样的，也就是说每个进程或线程只能访问特定的分类，而且不同分类之间是互斥的。