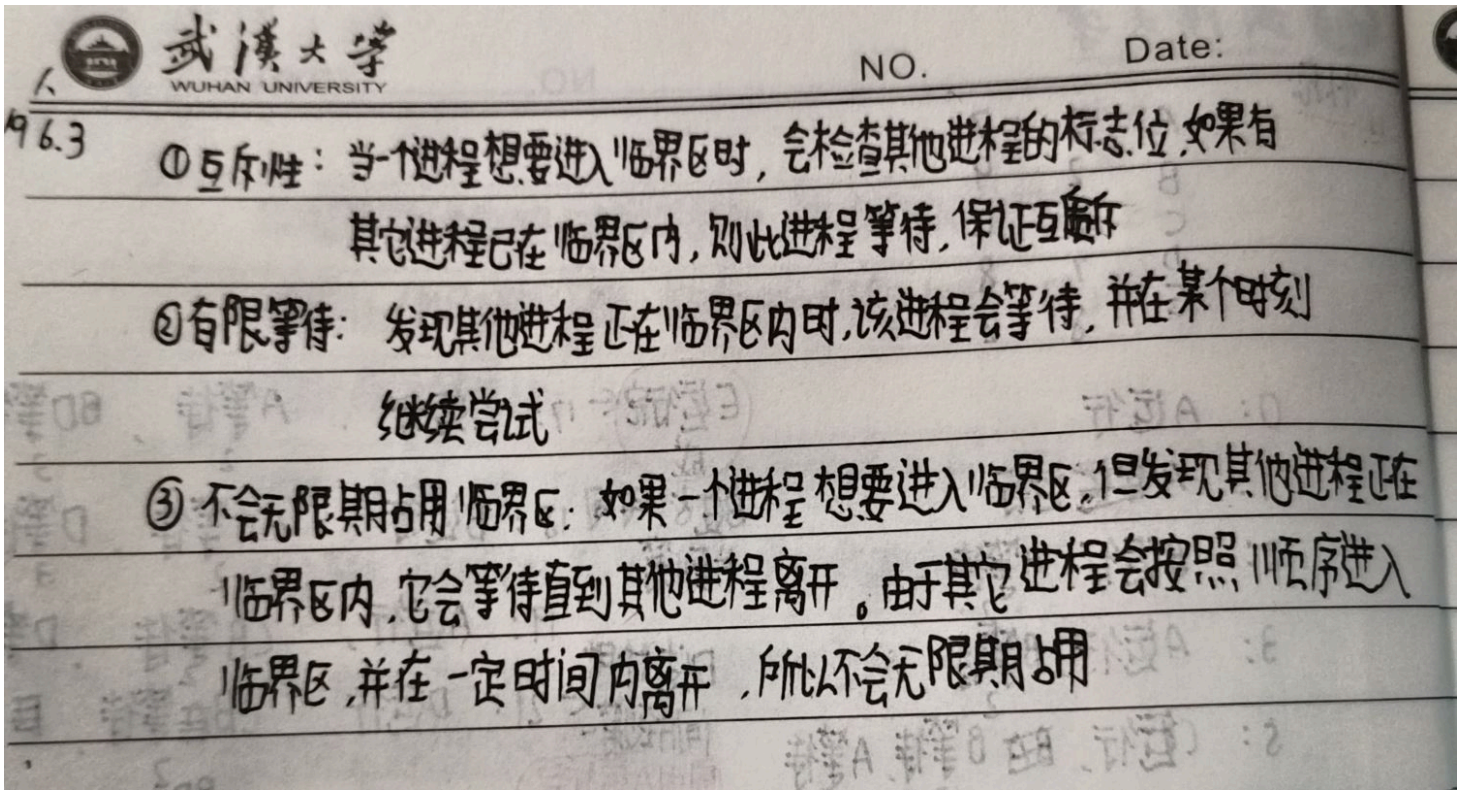
V9 6.3



第二题

- 注意这个要求进程可同时读同一个Fi（0≤i ≤ N）；但有进程写时，其他进程不能读和写。因此不需要读信号量来让读互斥，但是仍然需要写信号量
- P操作就是sem_wait，V操作就是sem_post
- 算法分析
  对于P1进程，它只对表格F进行读取操作.
  对于P2进程，它只对表格F进行写入操作，因此在写入时需要获取写信号量，以确保在有其他进程读取或写入时不会进行写入操作，保证了共享数据的一致性。
  对于P3进程，它先进行读取操作，再进行写入操作，因此需要获取写信号量，以保证一致性。

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

#define N 10 // 表格F的大小
#define K 5  // 每类进程的数量

sem_t write_sem,count;
int read_count = 0;
sem_init(&write_sem, 0, 1);
sem_init(&count, 0, 1); //保证修改readcount的互斥

void *P1(void *arg) {
    int *index = (int *)arg;
    printf("P1[%d] starts\n", *index);

    // 读取表格F
    sem_wait(&count);
    read_count++;
    sem_post(&count);
    if (read_count == 1) {
        sem_wait(&write_sem);
    }
    printf("P1[%d] reads from F\n", *index);

    // 读取完成
    sem_wait(&count);
    read_count--;
    sem_post(&count);
    if (read_count == 0) {
        sem_post(&write_sem);
    }
    printf("P1[%d] finishes\n", *index);
    pthread_exit(NULL);
}

void *P2(void *arg) {
    int *index = (int *)arg;
    printf("P2[%d] starts\n", *index);

    // 写入表格F
    sem_wait(&write_sem);
    printf("P2[%d] writes to F\n", *index);
```

```
    sem_post(&write_sem);

    printf("P2[%d] finishes\n", *index);
    pthread_exit(NULL);
}

void *P3(void *arg) {
    int *index = (int *)arg;
    printf("P3[%d] starts\n", *index);

    // 读取表格F
    sem_wait(&count);
    read_count++;
    sem_post(&count);
    if (read_count == 1) {
        sem_wait(&write_sem);
    }
    printf("P3[%d] reads from F\n", *index);

    sem_wait(&count);
    read_count--;
    sem_post(&count);
    if (read_count == 0) {
        sem_post(&write_sem);
    }

    // 写入表格F
    sem_wait(&write_sem);
    printf("P3[%d] writes to F\n", *index);
    sem_post(&write_sem);

    printf("P3[%d] finishes\n", *index);
    pthread_exit(NULL);
}
```

第三题

- 这个问题第一遍看感觉进程太多想不清楚，但是慢慢看下来可以用生产者-消费者的相关知识解决此问题

- P1与P2、P3，P1就是生产者，对于他们之间的缓冲区，需要用mutex信号量进行互斥访问控制，同时注意这个题缓冲区还有大小，因此引入了full和empty信号量，full信号量初值为0，empty信号量初值为N。对于full和empty两个信号量，自己认为直接用数字来理解更容易，full的s为多少就代表有多少P2,P3可以读，empty的s为多少就代表缓冲区有多少P1可以写。那么对于P1，显然是要先P(&empty)再V(&full);对于P2,P3，显然是要先P(&full)再V(&empty)，同时注意P2,P3还有自己的私有缓冲区，因此还要V(&count_odd)/V(&count_even)
- 对于P2和P2-1，P3和P3-1，此时P2,P3就相当于生产者，分析方法和上述类似

```
/*
mutex：用于对缓冲区的访问进行互斥控制。
full：用于表示缓冲区是否已满，当缓冲区中的单元数达到最大值时，生产者进程需要等待。
empty：用于表示缓冲区是否为空，当缓冲区中的单元数为0时，消费者进程需要等待。
count_odd：用于同步 P2 和 P2-1 进程之间的统计操作。
count_even：用于同步 P3 和 P3-1 进程之间的统计操作。
*/
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define N 10 // 缓冲区buf1的大小
#define M 10 // 私有缓冲区buf2和buf3的大小

sem_t mutex, full, empty, count_odd, count_even;
int buffer[N];
int odd_count = 0, even_count = 0;

void *P1(void *arg) {
    while (1) {

        sem_wait(&empty);

        sem_wait(&mutex);
        // 生成一个正整数并放入缓冲区buf1中
        produce();
        put();
        sem_post(&mutex);

        sem_post(&full);
    }
}

void *P2(void *arg) {
    while (1) {
        sem_wait(&full);

        sem_wait(&mutex);
        // 从缓冲区buf1中取出一个奇数，并放入私有缓冲区buf2中
        getodd();
        sem_post(&mutex);

        sem_post(&empty);
```

```
        sem_post(&count_odd);
    }
}

void *P3(void *arg) {
    while (1) {
        sem_wait(&full);

        sem_wait(&mutex);
        // 从缓冲区buf1中取出一个偶数，并放入私有缓冲区buf3中
        geteven();
        sem_post(&mutex);

        sem_post(&empty);
        sem_post(&count_even);
    }
}

void *P2_1(void *arg) {
    while (1) {
        sem_wait(&count_odd);
        // 读取P2的私有缓冲区buf2，并统计奇数个数
        countodd();
    }
}

void *P3_1(void *arg) {
    while (1) {
        sem_wait(&count_even);
        // 读取P3的私有缓冲区buf3，并统计偶数个数
        counteven();
    }
}

void *P4(void *arg) {
    while (1) {
        // 输出输出一个包含统计时间的结果
        // ...
    }
}

int main() {
    // 初始化信号量
```

```
    sem_init(&mutex, 0, 1);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, N);
    sem_init(&count_odd, 0, 0);
    sem_init(&count_even, 0, 0);

    // 创建线程
    pthread_t threads[5];
    pthread_create(&threads[0], NULL, P1, NULL);
    pthread_create(&threads[1], NULL, P2, NULL);
    pthread_create(&threads[2], NULL, P3, NULL);
    pthread_create(&threads[3], NULL, P2_1, NULL);
    pthread_create(&threads[4], NULL, P3_1, NULL);
    pthread_create(&threads[5], NULL, P4, NULL);

    // 等待线程结束
    for (int i = 0; i < 6; i++) {
        pthread_join(threads[i], NULL);
    }

    // 销毁信号量
    sem_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);
    sem_destroy(&count_odd);
    sem_destroy(&count_even);

    return 0;
}
```

第四题

- 第一问

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

sem_t mutex;

void *car(void *arg) {
    int car_id = *(int *)arg;

    sem_wait(&mutex);
    printf("Car %d is crossing the bridge.\n", car_id);
    sem_post(&mutex);
    pthread_exit(NULL);
}

// 初始化信号量
sem_init(&mutex, 0, 1); // 初始时桥上无车
```

- 第二问
  相比于第一问，第二问主要考虑的问题在于可以最多有个N个车同向通过，因此引入bridge来表示对桥的控制权，当一方有控制权另一方将一直等待直到对方全部过桥
  mutex信号量保证count的修改完整执行不必多说，最多有个N个车同向通过因此引入mutex_east和mutex_west，他们的初始值为N，也就是表示最多有个N个车同向通过

```
semaphore bridge = 1;   // 互斥信号量，表示独木桥的数量
int count1 = 0;         // 东侧车辆在独木桥上的数量
semaphore mutex1 = 1; // 东侧车辆的互斥信号量，保证count1操作的完整执行
semaphore mutex_east=N; //东边最多有个N个车同向通过
int count2 = 0;         // 西侧车辆在独木桥上的数量
semaphore mutex2 = 1; // 西侧车辆的互斥信号量，保证count2操作的完整执行
semaphore mutex_west=N; //西边最多有个N个车同向通过

void east() {
        P(mutex1);
        count1++;
        if(count1 == 1)  // 东侧第一个准备上桥的车去抢夺独木桥
                P(bridge);
        V(mutex1);

    P(mutex_east);
        {过独木桥};
    V(mutex_east);

        P(mutex1);
        count1--;
        if(count1 == 0)  // 东侧最后一个已经下桥的车去释放独木桥
                V(bridge);
        V(mutex1);
}

void west() {
        P(mutex2);
        count2++;
        if(count2 == 1)  // 西侧第一个准备上桥的车去抢夺独木桥
                P(bridge);
        V(mutex2);

    P(mutex_west);
        {过独木桥};
    V(mutex_west);

        P(mutex2);
        count2--;
        if(count2 == 0)  // 西侧最后一个已经下桥的车去释放独木桥
                V(bridge);
```

```
        V(mutex2);
}
```

第五题

参考博客

- 自己一开始是没有理清楚这里面的逻辑的，实际上可以发船的条件就三种：1.四红 2.四黑 3.两红两黑
- 注意S_red和S_black初值都为0，因为在没有发船之前都需要等待，等待条件满足
- 注意boats信号量，不是人员满足条件就可以发船，还得有船

```
semapore  boats=1;    //河上船只的数量
semapore  full=0;        //船的满员状态
semapore  S_red=0;    //控制红客上船
semapore  S_black=0; //控制黑客上船
semapore  mutex=1;   //由于互斥
int reds=0;          //等待上船的红客数
int blacks=0;         //等待上船的黑客数

Red()
{
  P(mutex);//进入临界区
  reds++;  //等待上船的红客数加1

  if(reds >=2 && blacks >=2)
    {//2个红客和黑客的组合
       P(boats);   //准备上船，无船则等待
       take_boat();//该红客上船
       reds=reds-2;//等待上船的红客数减2
       V(S_red);   //通知另一个红客上船

       blacks=blacks-2;//等待上船的黑客数减2
       //通知其他两黑客上船
       V(S_black);
       V(S_black);

       V(full);//通知船满员
       V(mutex);//退出临界区
    }
  else if(reds==4)
    {//4个红客的组合
       P(boats);    //准备上船，无船则等待
       take_boat(); //该红客上船

       //递减等待上船的红客数，通知其他3个红客上船
       while(--reds)
         V(S_red);

       V(full);//通知船满员
       V(mutex);//退出临界区
    }
  else
   {
       V(mutex); //退出临界区，此步必在P(S_red)之前，不然会产生死锁
```

```
        //该红客等待直至条件满足时上船
        P(S_red);
        take_boat();
    }
}

Black()
{
  P(mutex);
  blacks++;

  if(blacks >=2 && reds >=2)
    {
        P(boats);
        take_boat();
        blacks=blacks-2;
        V(S_black);

        reds=reds-2;
        V(S_red);
        V(S_red);

        V(full);
        V(mutex);
    }
  else if(blacks==4)
    {
        P(boats);
        take_boat();
        while(--blacks)
          V(S_black);

        V(full);
        V(mutex);

    }

  else
   {
        V(mutex);
        P(S_black);
        take_boat();
    }
```

```
}


Boat()
{
  while(TRUE){
        P(full);            //等待满员
        shove_off();   //开船过河
        boat_return();//空船返回
        V(boats);        //通知可以上船了
        }
}
```