浙江大学实验报告

课程名称：操作系统 实验类型：综合型

实验项目名称：同步互斥和 Linux 内核模块

学生姓名：沈子衿 学号：3160104734

电子邮件地址：zijinshen@zju.edu.cn

实验日期：2018 年 11 月 30 日

一、实验环境

实验一：

- 代码编辑环境：Windows 10 家庭中文版
- 代码编译与运行环境：Ubuntu shell for Windows（Linux LAPTOP-I21OTIEM 4.4.0-17763-Microsoft）

实验二：

- 代码编辑环境：Windows 10 家庭中文版
- 代码编译与运行环境：Ubuntu 发行版（Linux ubuntu 4.15.0-39-generic）

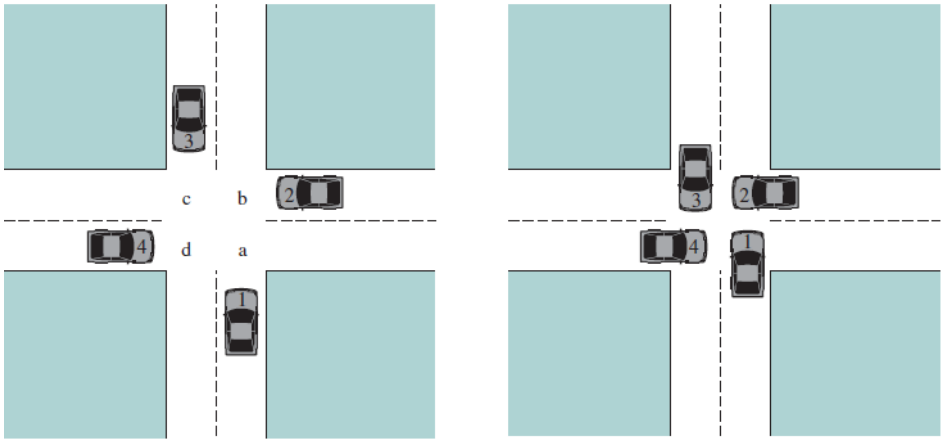
二、实验内容和结果及分析

实验一

2.1.1 题干

有两条道路双向两个车道，即每条路每个方向只有一个车道，两条道路十字交叉。假设车辆只能向前直行，而不允许转弯和后退。如果有 4 辆车几乎同时到达这个十字路口，如图（a）所示；相互交叉地停下来，如图（b），此时 4 辆车都将不能继续向前，这是一个典型的死锁问题。从操作系统原理的资源分配观点，如果 4 辆车都想驶过十字路口，那么对资源的要求如下：

- 向北行驶的车 1 需要象限 a 和 b；
- 向西行驶的车 2 需要象限 b 和 c；
- 向南行驶的车 3 需要象限 c 和 d；
- 向东行驶的车 4 需要象限 d 和 a。



(a) Deadlock possible　　(b) Deadlock

我们要实现十字路口交通的车辆同步问题，防止汽车在经过十字路口时产生死锁和饥饿。在我们的系统中，东西南北各个方向不断地有车辆经过十字路口（注意：不只有4辆），同一个方向的车辆依次排队通过十字路口。按照交通规则是右边车辆优先通行，如图(a)中，若只有 car1、car2、car3，那么车辆通过十字路口的顺序是 car3->car2->car1。车辆通行总的规则：

1) 来自同一个方向多个车辆到达十字路口时，车辆靠右行驶，依次顺序通过；

2) 有多个方向的车辆同时到达十字路口时，按照右边车辆优先通行规则，除非该车在十字路口等待时收到一个立即通行的信号；

3) 避免产生死锁；

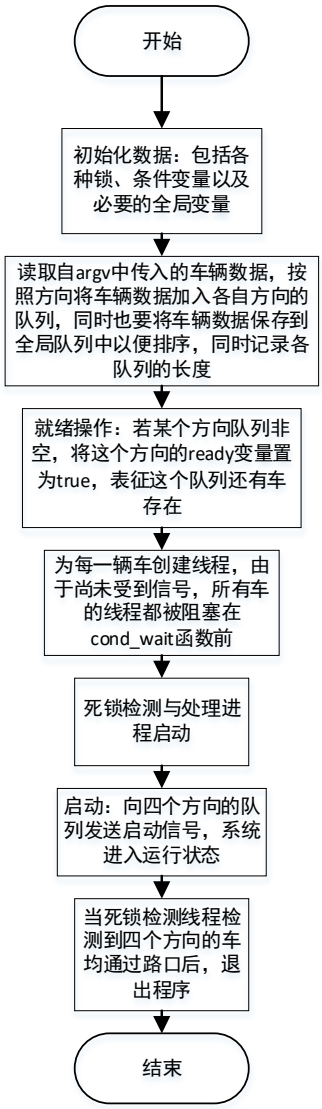
4) 避免产生饥饿；

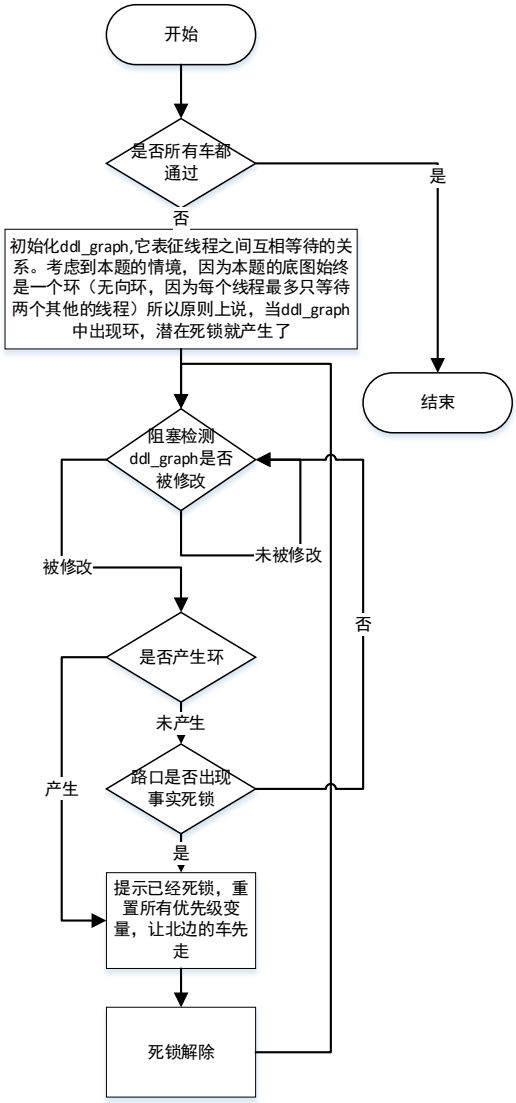
5) 任何一个线程（车辆）不得采用单点调度策略；

6) 由于使用 AND 型信号量机制会使线程（车辆）并发度降低且引起不公平（部分线程饥饿），本题不得使用 AND 型信号量机制，即在上图中车辆不能要求同时满足两个象限才能顺利通过，如南方车辆不能同时判断 a 和 b 是否有空。

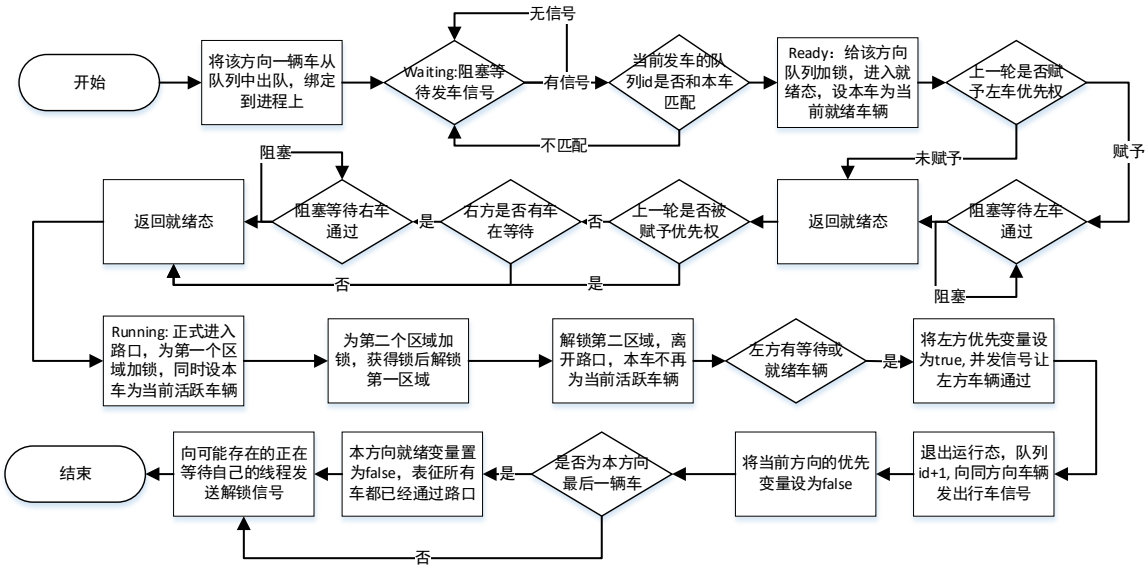
2.1.2 设计文档

2.1.2.1 程序整体设计流程图与死锁检测线程流程图





2.1.2.2 各方向行车逻辑流程图



2.1.2.3 数据结构声明与数据定义

dir_t

```
enum dir_t {NORTH, WEST, SOUTH, EAST};
```

dir_t 类型枚举变量，用于表征行车方向。

car_t

```
struct car_t
{
    int id;
    dir_t dir;
    int order;
};
```

car_t 类型结构变量，用于表征车辆的基本信息，包括车辆在全局队列中的 id，方向（从何处来）以及在本方向队列中的序号（用于在判断条件变量的过程中决定某一辆处于等待状态的车辆是否应当进入就绪态）

queue 与线程数组

```
class queue
{
    public:
        int max_count;
        int now_order = 0;
        pthread_t thread[MAX];
        int thp = 0;
        car_t qi[MAX];
        int front = -1;
        int end = 0;
        queue()
        {
            front = -1;
            end = 0;
            thp = 0;
            now_order = 0;

        }
```

```
        void enqueue(car_t car){
            qi[++front] = car;
        }
        car_t dequeue(){
            car_t err = {-1, NORTH, -1};
            return front >= end ? qi[end++] : err;
        }
        int count()
        {
            return front - end + 1;
        }
};
queue qe;
queue qw;
queue qs;
queue qn;

car_t q[MAX];
pthread_t car_thread[MAX];
int ptr = 0;

```

Queue 类，用于队列的实现。具有 max_count（该队列曾经达到的最长长度，用于判断某一方向的车辆是否完全通过路口），now_order（当前该方向处于或应当处于就绪或活跃状态的车辆的队内编号，用于判断某一方向的车辆是否完全通过路口），thread[MAX]（用于存储该方向所有车辆线程的栈），thp（thread 的栈指针），pi[MAX] （队列数组），front 和 end（队列的头指针和尾指针）这几个成员变量以及构造函数、入队出队方法和 count() 函数（计算当前队列中成员的个数）。

Queue 类用于实现各个方向的队列。与之互作的还有存储所有车辆对象的数组和存储所有车辆线程的数组。Ptr 是车辆对象数组和车辆线程数组共用的下标指针。

互斥锁、条件变量与优先变量

```
pthread_mutex_t mutex_e;
pthread_mutex_t mutex_w;
pthread_mutex_t mutex_s;
pthread_mutex_t mutex_n;

pthread_cond_t cond_e;
pthread_cond_t cond_w;
```

```
pthread_cond_t cond_s;
pthread_cond_t cond_n;

bool firstNorth = false;
bool firstSouth = false;
bool firstWest = false;
bool firstEast = false;

pthread_cond_t c_firstNorth;
pthread_cond_t c_firstSouth;
pthread_cond_t c_firstEast;
pthread_cond_t c_firstWest;

pthread_mutex_t mutex_firstNorth;
pthread_mutex_t mutex_firstSouth;
pthread_mutex_t mutex_firstEast;
pthread_mutex_t mutex_firstWest;
```

上述定义了本题中需要用到的互斥锁和条件变量。mutex_e-mutex_n 四个变量用与给四个方向以及四个队列加锁，cond_s 和 cond_n 是负责阻塞和唤醒四个方向车辆线程的条件变量，c_firstNorth-c_firstWest 是用于阻塞和唤醒因让行（包括右侧车和左侧车）而等待的线程的条件变量，mutex_firstNorth-mutex_firstEast 则是配合其使用的条件变量。至于 firstNorth-firstEast，则是为了让左车低优先级车辆在右侧车辆让行的情况下跳过阻塞判断直接通行的标志变量。

通行状态变量

```
bool west_ready = false;
bool east_ready = false;
bool south_ready = false;
bool north_ready = false;

// 在路口处于已经发车的车辆
bool west_active = false;
bool north_active = false;
bool south_active = false;
bool east_active = false;

car_t now_north = {-1, NORTH, -1};
car_t now_east = {-1, EAST, -1};
car_t now_west = {-1, WEST, -1};
car_t now_south = {-1, SOUTH, -1};
```

```
16.
17. car_t ready_north = {-1, NORTH, -1};
18. car_t ready_east = {-1, EAST, -1};
19. car_t ready_west = {-1, WEST, -1};
20. car_t ready_south = {-1, SOUTH, -1};
21.
```

上述定义了用于表征各方向车辆通行状态的变量。Ready 系列布尔变量用于表征对应方向是否有等待、就绪的车辆。理论上说，除非该方向所有车全部通过路口，ready 函数调用后对应方向的 ready 变量应当始终是 true；Active 系列布尔变量用于表征该方向当前是否有活跃车辆（正在过路口）；now 系列 car_t 变量用于表征当前各方向处于活跃状态的车辆信息；ready 系列 car_t 变量用于表征当前已经进入就绪状态但还未占用路口的车辆信息。

ddl_graph 与死锁检测

```
1.  bool ddl_graph[4][4] =
2.  {
3.  //(waitee)    e     s      w       n
4.              {false, false, false, false}, // e
5.              {false, false, false, false}, // n
6.              {false, false, false, false}, // w
7.              {false, false, false, false}  // s
8.                                            //(waitor)
9.  };
```

Ddl_graph 变量用于检测潜在死锁。它表征线程之间互相等待的关系，行是等待者，列是被等待者，每当一个进程开始等待另一个进程，他就要将对应的项置为 true，当等待结束，原进程重新将对应的项置为 false。它们的对应次序如代码注释所示。考虑本题的情境，因为本题的底图始终是一个环（无向环，因为每个线程最多只等待两个其他的线程）所以原则上说，当 ddl_graph 中出现环，潜在死锁也就产生了。之所以称这种死锁方式为“潜在死锁”，是因为它是由进程间互相等待造成的，并不是由资源占用造成的，四个方向的车并没有开进路口。原则上，本题的算法不会出现事实死锁，即路口四个资源被不同方向的车同时占据。但为了防患于未然，代码也提供了强制解决事实死锁的方案。

```
1.  int test_loop()
2.  {
3.      // 优先环
```

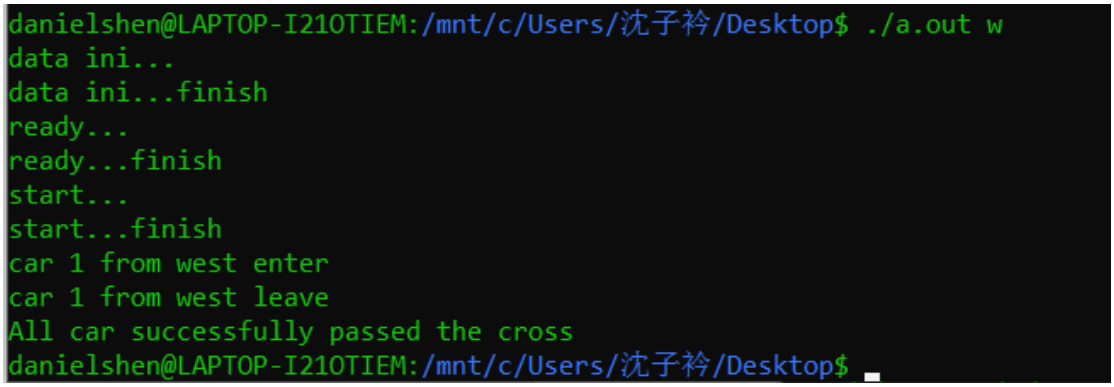
```
    if(ddl_graph[0][3] && ddl_graph[1][2] && ddl_graph[2][1] && ddl_graph[3][0])
        return 1;
    // 饥饿环
    if(ddl_graph[0][1] && ddl_graph[1][0] && ddl_graph[2][3] && ddl_graph[3][2])
        return 2;
    return 0;
}
```

死锁检测线程执行的每一次死锁检测都由这样一个 test_loop()函数实现。它通过遍历 ddl_graph 矩阵检测当前进程中是否存在循环等待的情况，一旦出现循环等待，他将返回一个大于零的值来宣告潜在死锁的产生，以便死锁处理线程进行调度。本题情境中有两种可能的循环等待：“优先环”，即每一辆车都等待其右边的车；以及“饥饿环”，即每一辆车都试图谦让其左边的车。

潜在死锁出现后，死锁处理线程将视循环等待种类的不同给不同的条件变量发信号（因为两种情况下阻塞的位置不同），指示北方的车先通行。

2.1.3 运行结果与分析

样例 1：单方向一辆车



这是最简单的样例，车辆在其他方向没有车辆的情况下跳过阻塞等待直接通过路口，证明代码基本逻辑正确，且后台线程可以正确检测所有车辆通过。

样例 2：单方向车流

```
danielshen@LAPTOP-I21OTIEM:/mnt/c/Users/沈子衿/Desktop$ ./a.out www
data ini...
data ini...finish
ready...
ready...finish
start...
start...finish
car 1 from west enter
car 1 from west leave
car 2 from west enter
car 2 from west leave
car 3 from west enter
car 3 from west leave
All car successfully passed the cross
```

```
danielshen@LAPTOP-I21OTIEM:/mnt/c/Users/沈子衿/Desktop$ ./a.out sssss
data ini...
data ini...finish
ready...
ready...finish
start...
start...finish
car 1 from south enter
car 1 from south leave
car 2 from south enter
car 2 from south leave
car 3 from south enter
car 3 from south leave
car 4 from south enter
car 4 from south leave
car 5 from south enter
car 5 from south leave
```

该样例为了验证同方向的同步处理。可以看出，同向的各个线程虽然是同时启动的，但依然严格按照入队次序依次通过了路口，可以初步验证同方向各进程同步处理的正确性。

样例 3：对向交替车流

```
danielshen@LAPTOP-I21OTIEM:/mnt/c/Users/沈子衿/Desktop$ ./a.out snsnsnssnn
data ini...
data ini...finish
ready...
ready...finish
start...
start...finish
car car 2 from north enter
1 from south enter
car 2 from north leave
car 1 from south leave
car 4 from north enter
car 3 from south enter
car 4 from north leave
car 3 from south leave
car 6 from north enter
car 5 from south enter
car 6 from north leave
car 5 from south leave
car 9 from north enter
car 7 from south enter
car 9 from north leave
car 7 from south leave
car 10 from north enter
car 8 from south enter
car 10 from north leave
car 8 from south leave
All car successfully passed the cross
```

本样例是为验证对向交替车流的调度正确性。可以看出同向车辆依然是依次通过路口，对向车辆则可能同时通过路口，所以在时间片上出现了争抢（从第一行混乱的输出结果能看出）。这并不是异常情况，相反验证了调度情境的真实性和算法的正确性。

样例 4：侧向交替车流（无死锁）

```
danielshen@LAPTOP-I21OTIEM:/mnt/c/Users/沈子衿/Desktop$ ./a.out neneeew
data ini...
data ini...finish
ready...
ready...finish
start...
start...finish
car 7 from west enter
car 7 from west leave
car 1 from north enter
car 1 from north leave
car car 3 from north enter2 from east enter

car 3 from north leave
car 2 from east leave
car 4 from east enter
car 4 from east leave
car 5 from east enter
car 5 from east leave
car 6 from east enter
car 6 from east leave
All car successfully passed the cross
```

由于最右优先。西侧的车先行。西侧车通行过后，由于饥饿让进策略，北侧的第一辆车通过。北侧车辆通过后同时向左侧东面的车以及下一辆北侧车辆发送信号，因此两者几乎同时到达路口，并开始竞争资源。可以看出北侧和东侧车辆首先占据各自的第一片区域，然后北侧车辆首先抢到共用区域的锁，先离开路口，东侧车辆后离开。最后剩下的东侧车辆陆续通过路口。这一样例验证了优先调度、饥饿处理和资源互斥相关算法的正确性。

样例 5：四方向交替长车流（存在潜在死锁）

```
danielshen@LAPTOP-I21OTIEM:/mnt/c/Users/沈子衿/Desktop$ ./a.out  nsewn
data ini...
data ini...finish
ready...
ready...finish
start...
start...finish
 protential deadlock occurs, signal north to go
car 1 from north enter
car 1 from north leave
car 3 from east enter
car 3 from east leave
car 2 from south enter
car 2 from south leave
car 4 from west enter
car 4 from west leave
car 8 from north enter
car 5 from west enter
car 5 from west leave
car 8 from north leave
car 7 from west enter
car 6 from east enter
car 7 from west leave
car 6 from east leave
All car successfully passed the cross
```

可以看出，死锁检测线程在就绪状态时就已经检测到了潜在死锁，因此在死锁事实上发生前首先命令北边的车先走，解除了死锁，北车走后，按照让进规则东车随其后，随后各车有序服从调度通过路口。该样例验证了死锁检测算法的正确性。

2.1.4 源代码

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <pthread.h>
4.  #include <time.h>
5.  #include <string.h>
```

```
#include <iostream>
#define MAX 1000

using namespace std;


// 向北行驶的车 1 需要象限 a 和 b
// 向西行驶的车 2 需要象限 b 和 c
// 向南行驶的车 3 需要象限 c 和 d
// 向东行驶的车 4 需要象限 d 和 a

enum dir_t {NORTH, WEST, SOUTH, EAST};

/*
car_t 类型结构变量，用于表征车辆的基本信息，
包括车辆在全局队列中的 id，方向（从何处来）以及在本方向队列中的
序号（用于在判断条件变量的过程中决定某一辆处于等待状态的车辆是
否应当进入就绪态）
*/

struct car_t
{
    int id;
    dir_t dir;
    int order;
};
/*
Queue 类，用于队列的实现。具有 max_count（该队列曾经达到的最长长度，用于判断某一方向的车辆是否完全通过路口），now_order（
当前该方向处于或应当处于就绪或活跃状态的车辆的队内编号，用于判断某一方向的车辆是否完全通过路口），thread[MAX]
（用于存储该方向所有车辆线程的栈），thp（thread 的栈指针），pi[MAX] （队列数组），front 和 end（队列的头指针和尾指针）
这几个成员变量以及构造函数、入队出队方法和 count()函数（计算当前队列中成员的个数）。
Queue 类用于实现各个方向的队列。与之互作的还有存储所有车辆对象的数组和存储所有车辆线程的数组。
Ptr 是车辆对象数组和车辆线程数组共用的下标指针。

*/
class queue
{
    public:
        int max_count; // 该队列曾经达到的最长长度，用于判断某一方向的车辆是否完全通过路口
```

```
45.         int now_order = 0; // 当前该方向处于或应当处于就绪或活跃状态的车辆的队内编号，用于判断某一方向的车辆是否完全通过路口
46.         pthread_t thread[MAX]; // 用于存储该方向所有车辆线程的栈
47.         int thp = 0; // 栈指针
48.         car_t qi[MAX];
49.         int front = -1;
50.         int end = 0;
51.         //pi[MAX] （队列数组），front 和 end（队列的头指针和尾指针）
52.         //这几个成员变量以及构造函数、入队出队方法和 count()函数（计算当前队列中成员的个数）。
53.         queue()
54.         {
55.             front = -1;
56.             end = 0;
57.             thp = 0;
58.             now_order = 0;
59.
60.         }
61.         // 入队函数
62.         void enqueue(car_t car){
63.             qi[++front] = car;
64.         }
65.         // 出队函数
66.         car_t dequeue(){
67.             car_t err = {-1, NORTH, -1};
68.             return front >= end ? qi[end++] : err;
69.         }
70.         // 计数函数
71.         int count()
72.         {
73.             return front - end + 1;
74.         }
75. };
76.
77. // 为路口四个资源加锁
78.
79. int resource = 4;
80. pthread_mutex_t mutex_resource;
81. pthread_mutex_t mutex_a;
82. pthread_mutex_t mutex_b;
83. pthread_mutex_t mutex_c;
84. pthread_mutex_t mutex_d;
85.
86.
```

```
// 四个方向的队列
queue qe;
queue qw;
queue qs;
queue qn;

// 存储所有车辆信息的数组
car_t q[MAX];
pthread_t car_thread[MAX];
int ptr = 0;

// mutex_e-mutex_n 四个变量用与给四个方向以及四个队列加锁
pthread_mutex_t mutex_e;
pthread_mutex_t mutex_w;
pthread_mutex_t mutex_s;
pthread_mutex_t mutex_n;

// cond_s 和 cond_n 是负责阻塞和唤醒四个方向车辆线程的条件变量
pthread_cond_t cond_e;
pthread_cond_t cond_w;
pthread_cond_t cond_s;
pthread_cond_t cond_n;

pthread_mutex_t mutex_ce;
pthread_mutex_t mutex_cw;
pthread_mutex_t mutex_cs;
pthread_mutex_t mutex_cn;


// firstNorth-firstEast，则是为了让左车低优先级车辆在右侧车辆让行的情况下跳过阻塞判断直接通行的标志变量
bool firstNorth = false;
bool firstSouth = false;
bool firstWest = false;
bool firstEast = false;
// c_firstNorth-c_firstWest 是用于阻塞和唤醒因让行（包括右侧车和左侧车）而等待的线程的条件变量
pthread_cond_t c_firstNorth;
pthread_cond_t c_firstSouth;
pthread_cond_t c_firstEast;
pthread_cond_t c_firstWest;


// mutex_firstNorth-mutex_firstEast 则是配合其使用的条件变量。
```

```
pthread_mutex_t mutex_firstNorth;
pthread_mutex_t mutex_firstSouth;
pthread_mutex_t mutex_firstEast;
pthread_mutex_t mutex_firstWest;

pthread_cond_t area_e;
pthread_cond_t area_s;
pthread_cond_t area_w;
pthread_cond_t area_n;

// Ready 系列布尔变量用于表征对应方向是否有等待、就绪的车辆。
// 理论上说，除非该方向所有车全部通过路口，ready 函数调用后对应方向的 ready 变量应当始终是 true
// 在队列中有处于就绪状态的车辆
bool west_ready = false;
bool east_ready = false;
bool south_ready = false;
bool north_ready = false;

// 在路口处于已经发车的车辆
// Active 系列布尔变量用于表征该方向当前是否有活跃车辆（正在过路口）
bool west_active = false;
bool north_active = false;
bool south_active = false;
bool east_active = false;
// now 系列 car_t 变量用于表征当前各方向处于活跃状态的车辆信息
car_t now_north = {-1, NORTH, -1};
car_t now_east = {-1, EAST, -1};
car_t now_west = {-1, WEST, -1};
car_t now_south = {-1, SOUTH, -1};
// ready 系列 car_t 变量用于表征当前已经进入就绪状态但还未占用路口的车辆信息。
car_t ready_north = {-1, NORTH, -1};
car_t ready_east = {-1, EAST, -1};
car_t ready_west = {-1, WEST, -1};
car_t ready_south = {-1, SOUTH, -1};


/*
Ddl_graph 变量用于检测潜在死锁。它表征线程之间互相等待的关系，行是等待者，列是被等待者，
每当一个进程开始等待另一个进程，他就要将对应的项置为 true，当等待结束，原进程重新将对应的项置为 false。
它们的对应次序如代码注释所示。考虑本题的情境，因为本题的底图始终是一个环（无向环，因为每个线程最多只等待两个其他的线程）
```

```
所以原则上说，当 ddl_graph 中出现环，潜在死锁也就产生了。
之所以称这种死锁方式为“潜在死锁”，是因为它是由进程间互相等待造成的，并不是由资源占用造成的，四个方向的车并没有开进路口。
*/

bool ddl_graph[4][4] =
{
//(waitee)    e     s      w       n
            {false, false, false, false}, // e
            {false, false, false, false}, // n
            {false, false, false, false}, // w
            {false, false, false, false}  // s
                                          //(waitor)
};
/*
它通过遍历 ddl_graph 矩阵检测当前进程中是否存在循环等待的情况，一旦出现循环等待，他将返回一个大于零的值来宣告潜在死锁的产生，
以便死锁处理线程进行调度。本题情境中有两种可能的循环等待：“优先环”，即每一辆车都等待其右边的车；以及“饥饿环”，即每一辆车都
试图谦让其左边的车。
*/
int test_loop()
{
    // 优先环
    if(ddl_graph[0][3] && ddl_graph[1][2] && ddl_graph[2][1] && ddl_graph[3][0])
        return 1;
    // 饥饿环
    if(ddl_graph[0][1] && ddl_graph[1][0] && ddl_graph[2][3] && ddl_graph[3][2])
        return 2;
    return 0;
}

void data_ini(char* s)
{
    // 数据初始化
    cout << "data ini..."<< endl;
    pthread_mutex_init(&mutex_resource, NULL);
    pthread_mutex_init(&mutex_a, NULL);
    pthread_mutex_init(&mutex_b, NULL);
    pthread_mutex_init(&mutex_c, NULL);
    pthread_mutex_init(&mutex_d, NULL);
    pthread_mutex_init(&mutex_e, NULL);
```

```
    pthread_mutex_init(&mutex_w, NULL);
    pthread_mutex_init(&mutex_s, NULL);
    pthread_mutex_init(&mutex_n, NULL);
    pthread_cond_init(&cond_e, NULL);
    pthread_cond_init(&cond_w, NULL);
    pthread_cond_init(&cond_s, NULL);
    pthread_cond_init(&cond_n, NULL);
    pthread_mutex_init(&mutex_ce, NULL);
    pthread_mutex_init(&mutex_cw, NULL);
    pthread_mutex_init(&mutex_cs, NULL);
    pthread_mutex_init(&mutex_cn, NULL);
    pthread_cond_init(&c_firstNorth, NULL);
    pthread_cond_init(&c_firstSouth, NULL);
    pthread_cond_init(&c_firstWest, NULL);
    pthread_cond_init(&c_firstEast, NULL);
    pthread_mutex_init(&mutex_firstEast, NULL);
    pthread_mutex_init(&mutex_firstNorth, NULL);
    pthread_mutex_init(&mutex_firstSouth, NULL);
    pthread_mutex_init(&mutex_firstWest, NULL);
    pthread_cond_init(&area_e, NULL);
    pthread_cond_init(&area_w, NULL);
    pthread_cond_init(&area_s, NULL);
    pthread_cond_init(&area_n, NULL);
    int ww = 0;
    int ss = 0;
    int nn = 0;
    int ee = 0;
    for(int i = 0; i < strlen(s); i++)
    {
        // 西方的车辆入队
        if(s[i] == 'w')
        {
            car_t temp = {i+1, WEST, ww++};
            qw.enqueue(temp);
            q[ptr++] = temp;
        }
        // 南方的车辆入队
        else if(s[i] == 's')
        {
            car_t temp = {i+1, SOUTH, ss++};
            qs.enqueue(temp);
            q[ptr++] = temp;
        }
        // 北方的车辆入队
```

```
252.         if(s[i] == 'n')
253.         {
254.             car_t temp = {i+1, NORTH, nn++};
255.             qn.enqueue(temp);
256.             q[ptr++] = temp;
257.         }
258.         // 东方的车辆入队
259.         if(s[i] == 'e')
260.         {
261.             car_t temp = {i+1, EAST, ee++};
262.             qe.enqueue(temp);
263.             q[ptr++] = temp;
264.         }
265.     }
266.     // 计算 max_count
267.     qw.max_count = qw.count();
268.     qe.max_count = qe.count();
269.     qs.max_count = qs.count();
270.     qn.max_count = qn.count();
271.     cout << "data ini...finish"<< endl;
272. }
273. // ready 函数。将所有存在车辆的方向 ready 变量全部置为 true，表示这个方向还有车
274. void ready()
275. {
276.     cout << "ready..."<< endl;
277.     if(qs.count())
278.     {
279.         // 南方 ready 置为 true
280.         south_ready = true;
281.     }
282.     if(qw.count())
283.     {
284.         // 西方 ready 置为 true
285.         west_ready = true;
286.     }
287.     if(qn.count())
288.     {
289.         // 北方 ready 置为 true
290.         north_ready = true;
291.     }
292.     if(qe.count())
293.     {
294.         // 东方 ready 置为 true
295.         east_ready = true;
```

```
296.     }
297.     cout << "ready...finish"<< endl;
298. }
299. // start 函数，通过向所有方向的小车发送启动信号来启动系统
300. void start()
301. {
302.     cout << "start..."<< endl;
303.     pthread_cond_signal(&cond_s);
304.     pthread_cond_signal(&cond_w);
305.     pthread_cond_signal(&cond_n);
306.     pthread_cond_signal(&cond_e);
307.     cout << "start...finish"<< endl;
308. }
309.
310. // 死锁检测函数
311. void* check_ddl(void* para)
312. {
313.     int flag;
314.     while(1)
315.     {
316.         // 如果 test_loop 为真
317.         if(flag = test_loop()) // 这里确实是赋值！！！
318.         {
319.             cout << " protential deadlock occurs, signal north to go " << e
    ndl;
320.         }
321.         // 首先将所有数据强行归位，让北边先过
322.         firstEast = firstNorth = firstSouth = firstWest = false;
323.         if(flag == 1) // 优先环
324.         {
325.             pthread_cond_signal(&c_firstWest);
326.         }
327.         else if(flag == 2) // 饥饿环
328.         {
329.             pthread_cond_signal(&c_firstEast);
330.         }
331.         sleep(1);
332.         // 如果四个方向都没有车，程序结束
333.         if(north_ready == false && south_ready == false && west_ready == fa
    lse && east_ready == false)
334.         {
335.             cout << "All car successfully passed the cross"<< endl;
336.             exit(0);
337.         }
```

```
338.     }
339. }
340. 
341. void* west_thread(void *para)
342. {
343.     sleep(1);
344.     car_t temp_ready;
345.     // 选择下一个就绪车辆
346.     if((temp_ready = qw.dequeue()).id != -1)
347.     {
348.         west_ready = true;
349.     }
350.     // 进入就绪态，队列上锁
351.     //if(west_ready == false) {cout << "No car from west" << endl; return N
    ULL;}
352.     pthread_mutex_t tplock;
353.     pthread_mutex_init(&tplock, NULL);
354.     pthread_mutex_lock(&tplock);
355.     west_ready = true;
356.     // 等待发车信号启动，不属于自己的 order 不发车
357.     do{
358.         pthread_cond_wait(&cond_w, &tplock);
359.     }while(qw.now_order != temp_ready.order);
360.     pthread_mutex_unlock(&tplock);
361.     pthread_mutex_lock(&mutex_w);
362.     ready_west = temp_ready;
363.     // 如果之前赋予了左车先决的条件，要等 n 左车走完
364.     pthread_mutex_lock(&mutex_firstNorth);
365.     if(north_ready == true && firstNorth == true)
366.     {
367.         ddl_graph[2][3] = true;
368.         pthread_cond_wait(&c_firstNorth, &mutex_firstNorth);
369.         pthread_mutex_unlock(&mutex_firstNorth);
370.         ddl_graph[2][3] = false;
371.     }
372.     else
373.     {
374.         pthread_mutex_unlock(&mutex_firstNorth);
375.     }
376.     // 然后判断右边有没有车在就绪或者行驶,注意要在自己没有被优先授权的情况下
377.     pthread_mutex_lock(&mutex_firstWest);
378.     if(south_ready == true && !firstWest)
379.     {
380.         pthread_mutex_unlock(&mutex_firstWest);
```

```
381.         //如果右边有车在等待让他们先走,等走完了解锁
382.         ddl_graph[2][1] = true;
383.         // pthread_cond_signal(&cond_s);
384.         pthread_mutex_lock(&mutex_firstSouth);
385.         pthread_cond_wait(&c_firstSouth, &mutex_firstSouth);
386.         pthread_mutex_unlock(&mutex_firstSouth);
387.         ddl_graph[2][1] = false;
388.     }
389.     else{
390.         pthread_mutex_unlock(&mutex_firstWest);
391.     }
392.     // 优先模式
393.     // 进入运行态，进入路口
394.     pthread_mutex_lock(&mutex_d);
395.     west_active = true;
396.     now_west = ready_west;
397.     // 如果没有就绪车辆
398.     
399.     cout << "car " << now_west.id << " from west enter" << endl;
400.     sleep(1);
401.     // 尝试进入下一个区域
402.     pthread_mutex_lock(&mutex_a);
403.     // 成功进入下一区域后立刻离开
404.     cout << "car " << now_west.id << " from west leave" << endl;
405.     pthread_mutex_unlock(&mutex_d);
406.     sleep(1);
407.     pthread_mutex_unlock(&mutex_a);
408.     // 防止饥饿
409.     if(north_ready == true)
410.     {
411.         // 设置北方为优先
412.         firstNorth = true;
413.         pthread_cond_signal(&cond_n);
414.         sleep(1);
415.     }
416.     // 退出运行态
417.     west_active = false;
418.     pthread_mutex_unlock(&mutex_w);
419.     // 向下一辆车发送
420.     qw.now_order++;
421.     pthread_cond_broadcast(&cond_w);
422.     pthread_mutex_lock(&mutex_firstWest);
423.       // 取消优先级
424.     if(firstWest) firstWest = false;
```

```
425.     pthread_mutex_unlock(&mutex_firstWest);
426.     // 所有线程全部结束
427.     if(qw.max_count == temp_ready.order + 1)
428.     {
429.         west_ready = false;
430.     }
431.     // 解锁所有等待本车的线程
432.     pthread_cond_signal(&c_firstWest);
433.     return NULL;
434. }
435.
436. void* east_thread(void *para)
437. {
438.     car_t temp_ready;
439.     if((temp_ready = qe.dequeue()).id != -1)
440.     {
441.         east_ready = true;
442.     }
443.     // 进入就绪态，队列上锁
444.     //if(east_ready == false) {cout << "No car from east" << endl; return N
    ULL;}
445.     pthread_mutex_t tplock;
446.     pthread_mutex_init(&tplock, NULL);
447.     east_ready = true;
448.     // 等待发信号启动
449.     pthread_cond_wait(&cond_e, &tplock);
450.     pthread_mutex_unlock(&tplock);
451.     // 等待发车信号启动，不属于自己的 order 不发车
452.     do{
453.     pthread_mutex_lock(&mutex_e);
454.     }while(qe.now_order != temp_ready.order);
455.     ready_east = temp_ready;
456.     // 如果之前赋予了左边先决的条件，要等左边走完
457.     pthread_mutex_lock(&mutex_firstSouth);
458.     if(south_ready == true && firstSouth == true)
459.     {
460.         ddl_graph[0][1] = true;
461.         pthread_cond_wait(&c_firstSouth, &mutex_firstSouth);
462.         pthread_mutex_unlock(&mutex_firstSouth);
463.         ddl_graph[0][1] = false;
464.     }
465.     else
466.     {
467.         pthread_mutex_unlock(&mutex_firstSouth);
```

```
    }
    // 然后判断右边有没有车在就绪或者行驶,在自己没有被优先授权的情况下
    pthread_mutex_lock(&mutex_firstEast);
    if(north_ready == true && !firstEast)
    {
        pthread_mutex_unlock(&mutex_firstEast);
        ddl_graph[0][3] = true;
        //如果右边有车在等待让他们先走,等走完了解锁
        // pthread_cond_signal(&cond_n);
        pthread_mutex_lock(&mutex_firstNorth);
        pthread_cond_wait(&c_firstNorth, &mutex_firstNorth);
        pthread_mutex_unlock(&mutex_firstNorth);
        ddl_graph[0][3] = false;
    }
    else{
        pthread_mutex_unlock(&mutex_firstEast);
    }
    // 优先模式
    // 进入运行态，进入路口
    pthread_mutex_lock(&mutex_b);
    east_active = true;
    now_east = ready_east;
    // 选择下一个就绪车辆
    // 如果没有就绪车辆
    cout << "car " << now_east.id << " from east enter" << endl;
    // 尝试进入下一个区域
    pthread_mutex_lock(&mutex_c);
    sleep(1);
    // 成功进入下一区域后立刻离开
    cout << "car " << now_east.id << " from east leave" << endl;
    pthread_mutex_unlock(&mutex_b);
    sleep(1);
    pthread_mutex_unlock(&mutex_c);
    // 防止饥饿
    if(south_ready == true)
    {
        // 设置北方为优先
        firstSouth = true;
        pthread_cond_signal(&cond_s);
        sleep(1);
    }
    // 退出运行态
    east_active = false;
    pthread_mutex_unlock(&mutex_e);
```

```
512.     // 向下一辆车发送信号
513.     qe.now_order++;
514.     pthread_cond_broadcast(&cond_e);
515.     pthread_mutex_lock(&mutex_firstEast);
516.       // 取消优先级
517.     if(firstEast) firstEast = false;
518.     pthread_mutex_unlock(&mutex_firstEast);
519.     // 同方向所有线程全部结束
520.     if(qe.max_count == temp_ready.order + 1)
521.     {
522.         east_ready = false;
523.     }
524.     // 解锁所有等待本车的线程
525.     pthread_cond_signal(&c_firstEast);
526.     return NULL;
527. }
528. 
529. void* north_thread(void *para)
530. {
531.     car_t temp_ready;
532.     if((temp_ready = qn.dequeue()).id != -1)
533.     {
534.         north_ready = true;
535.     }
536.     // 进入就绪态，队列上锁
537.     //if(north_ready == false) {cout << "No car from north" << endl; return
    NULL;}
538.     pthread_mutex_t tplock;
539.     pthread_mutex_init(&tplock, NULL);
540.     pthread_mutex_lock(&tplock);
541.     north_ready = true;
542.     // 等待发车信号启动，不属于自己的 order 不发车
543.     do{
544.     pthread_cond_wait(&cond_n, &tplock);
545.     }while(qn.now_order != temp_ready.order);
546.     pthread_mutex_unlock(&tplock);
547.     pthread_mutex_lock(&mutex_n);
548.     ready_north = temp_ready;
549.     // 如果之前赋予了左方先决的条件，要等左方走完
550.     pthread_mutex_lock(&mutex_firstEast);
551.     if(east_ready == true && firstEast == true)
552.     {
553.         ddl_graph[1][0] = true;
554.         pthread_cond_wait(&c_firstEast, &mutex_firstEast);
```

```
555.         pthread_mutex_unlock(&mutex_firstEast);
556.         ddl_graph[1][0] = false;
557.     }
558.     else
559.     {
560.         pthread_mutex_unlock(&mutex_firstEast);
561.     }
562.     // 然后判断右边有没有车在就绪或者行驶,在自己没有被优先授权的情况下
563.     pthread_mutex_lock(&mutex_firstNorth);
564.     if(west_ready == true && !firstNorth)
565.     {
566.         pthread_mutex_unlock(&mutex_firstNorth);
567.         //如果右边有车在等待让他们先走,等走完了解锁
568.         ddl_graph[1][2] = true;
569.         // pthread_cond_signal(&cond_w);
570.         pthread_mutex_lock(&mutex_firstWest);
571.         pthread_cond_wait(&c_firstWest, &mutex_firstWest);
572.         pthread_mutex_unlock(&mutex_firstWest);
573.         ddl_graph[1][2] = false;
574.     }
575.     else{
576.         pthread_mutex_unlock(&mutex_firstNorth);
577.     }
578.     // 优先模式
579.     // 进入运行态，进入路口
580.     pthread_mutex_lock(&mutex_c);
581.     north_active = true;
582.     now_north = ready_north;
583.     // 选择下一个就绪车辆
584.     // 如果没有就绪车辆
585.     cout << "car " << now_north.id << " from north enter" << endl;
586.     // 尝试进入下一个区域
587.     sleep(1);
588.     pthread_mutex_lock(&mutex_d);
589.     // 成功进入下一区域后立刻离开
590.     cout << "car " << now_north.id << " from north leave" << endl;
591.     pthread_mutex_unlock(&mutex_c);
592.     sleep(1);
593.     pthread_mutex_unlock(&mutex_d);
594.     // 防止饥饿
595.     if(east_ready == true)
596.     {
597.         // 设置北方为优先
598.         firstEast = true;
```

```
599.         pthread_cond_signal(&cond_e);
600.         sleep(1);
601.     }
602.     // 退出运行态
603.     north_active = false;
604.     pthread_mutex_unlock(&mutex_n);
605.     // 向同方向下一辆车发送信号
606.     qn.now_order++;
607.     pthread_cond_broadcast(&cond_n);
608.     pthread_mutex_lock(&mutex_firstNorth);
609.     if(firstNorth) firstNorth = false;
610.     pthread_mutex_unlock(&mutex_firstNorth);
611.     if(qn.max_count == temp_ready.order + 1)
612.     {
613.         north_ready = false;
614.     }
615.     // 解锁所有等待本车的线程
616.     pthread_cond_signal(&c_firstNorth);
617.     return NULL;
618. }
619. 
620. void* south_thread(void *para)
621. {
622.     car_t temp_ready;
623.     if((temp_ready = qs.dequeue()).id != -1)
624.     {
625.         south_ready = true;
626.     }
627.     // 进入就绪态，队列上锁
628.     //if(south_ready == false) {cout << "No car from south" << endl; return
    NULL;}
629.     pthread_mutex_t tplock;
630.     pthread_mutex_init(&tplock, NULL);
631.     pthread_mutex_lock(&tplock);
632.     south_ready = true;
633.    // 等待发车信号启动，不属于自己的 order 不发车
634.     do{
635.         pthread_cond_wait(&cond_s, &tplock);
636.     }while(qs.now_order != temp_ready.order);
637.     pthread_mutex_unlock(&tplock);
638.     pthread_mutex_lock(&mutex_s);
639.     ready_south = temp_ready;
640.     // 如果之前赋予了左先决的条件，要等左走完
641.     pthread_mutex_lock(&mutex_firstWest);
```

```
642.     if(west_ready == true && firstWest == true)
643.     {
644.         ddl_graph[3][2] = true;
645.         pthread_cond_wait(&c_firstWest, &mutex_firstWest);
646.         pthread_mutex_unlock(&mutex_firstWest);
647.         ddl_graph[3][2] = false;
648.     }
649.     else
650.     {
651.         pthread_mutex_unlock(&mutex_firstWest);
652.     }
653.     // 然后判断右边有没有车在就绪或者行驶,在自己没有被优先授权的情况下
654.     pthread_mutex_lock(&mutex_firstSouth);
655.     if(east_ready == true && !firstSouth)
656.     {
657.         pthread_mutex_unlock(&mutex_firstSouth);
658.         //如果右边有车在等待让他们先走,等走完了解锁
659.         ddl_graph[3][0] = true;
660.         // pthread_cond_signal(&cond_e);
661.         pthread_mutex_lock(&mutex_firstEast);
662.         pthread_cond_wait(&c_firstEast, &mutex_firstEast);
663.         pthread_mutex_unlock(&mutex_firstEast);
664.         ddl_graph[3][0] = false;
665.     }
666.     else{
667.         pthread_mutex_unlock(&mutex_firstSouth);
668.     }
669.     // 优先模式
670.     // 进入运行态，进入路口
671.     pthread_mutex_lock(&mutex_a);
672.     south_active = true;
673.     now_south = ready_south;
674.     // 选择下一个就绪车辆
675.
676.     // 如果没有就绪车辆
677.     cout << "car " << now_south.id << " from south enter" << endl;
678.     sleep(1);
679.     // 尝试进入下一个区域
680.     pthread_mutex_lock(&mutex_b);
681.     // 成功进入下一区域后立刻离开
682.     cout << "car " << now_south.id << " from south leave" << endl;
683.     pthread_mutex_unlock(&mutex_a);
684.     sleep(1);
685.     pthread_mutex_unlock(&mutex_b);
```

```
686.     // 防止饥饿
687.     if(west_ready == true)
688.     {
689.         // 设置北方为优先
690.         firstWest = true;
691.         pthread_cond_signal(&cond_w);
692.         sleep(1);
693.     }
694.     // 退出运行态
695.     south_active = false;
696.     pthread_mutex_unlock(&mutex_s);
697.     // 向下一辆车发送信号
698.     qs.now_order++;
699.     pthread_cond_broadcast(&cond_s);
700.     pthread_mutex_lock(&mutex_firstSouth);
701.     // 取消优先级
702.     if(firstSouth) firstSouth = false;
703.     pthread_mutex_unlock(&mutex_firstSouth);
704.     if(qs.max_count == temp_ready.order + 1)
705.     {
706.         south_ready = false;
707.     }
708.     // 解锁所有等待本车的线程
709.     pthread_cond_signal(&c_firstSouth);
710.     return NULL;
711. }
712. 
713. 
714. int main(int argc, char** argv)
715. {
716.     data_ini(argv[1]);
717.     ready();
718.     for(int i = 0; i < ptr; i++)
719.     {
720.         if(q[i].dir == NORTH)
721.         {
722.             // 北方线程启动
723.             car_thread[i] = qn.thread[qn.thp++];
724.             pthread_create(&qn.thread[qn.thp++], NULL, north_thread, NULL);
725.         }
726.         if(q[i].dir == SOUTH)
727.         {
728.              // 南方线程启动
```

```
729.             car_thread[i] = qs.thread[qs.thp++];
730.             pthread_create(&qs.thread[qs.thp++], NULL, south_thread, NULL);

731.         }
732.         if(q[i].dir == EAST)
733.         {
734.             // 东方线程启动
735.             car_thread[i] = qe.thread[qe.thp++];
736.             pthread_create(&qe.thread[qe.thp++], NULL, east_thread, NULL);

737.         }
738.         if(q[i].dir == WEST)
739.         {
740.              // 西方线程启动
741.             car_thread[i] = qw.thread[qw.thp++];
742.             pthread_create(&qw.thread[qw.thp++], NULL, west_thread, NULL);

743.         }
744.         sleep(1);
745.     }
746.     start();
747.     pthread_create(&car_thread[ptr], NULL, check_ddl, NULL);
748.     for(int i = 0; i <= ptr; i++)
749.     {
750.         // 连 check_ddl 都 join
751.         pthread_join(car_thread[i], NULL);
752.     }
753. }
```

实验二

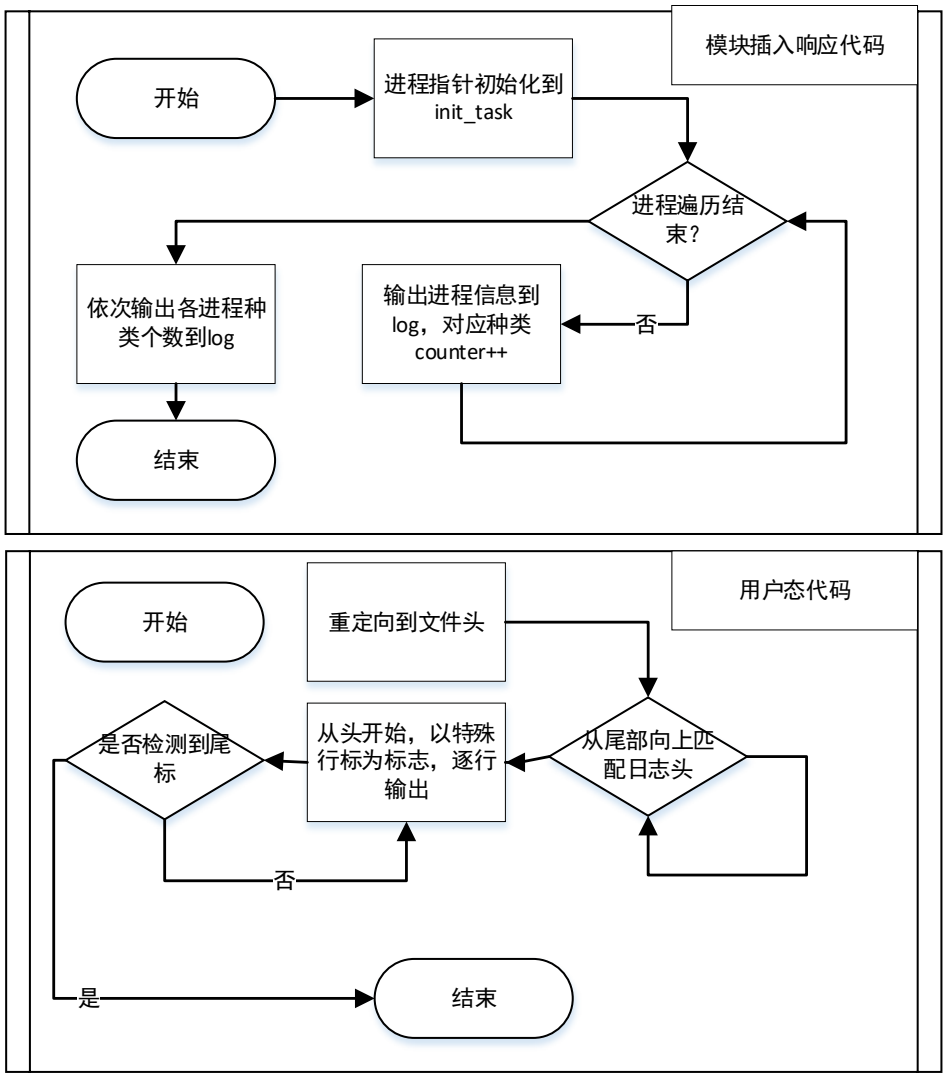
2.1.1 题干

编写一个 Linux 的内核模块，其功能是遍历操作系统所有进程。该内核模块输出系统中每个进程的：名字、进程 pid、进程的状态、父进程的名字等；以及统计系统中进程个 数，包括统计系统中 TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、

TASK_ZOMBIE、TASK_STOPPED 等（还有其他状态） 状态进程的个数。同时还需要编写一个用户态下执行的程序，格式化输出（显示）内核模块输出的内容。

2.2.2 设计文档

2.2.2.1 总体逻辑流程图



2.2.3 运行结果与分析

```
danielshen@ubuntu:~$ sudo rmmod module1
[sudo] danielshen 的密码:
danielshen@ubuntu:~$
```

首先将原模块移除

```
danielshen@ubuntu:~/Desktop$ make
make -C /lib/modules/4.15.0-39-generic/build M=/home/danielshen/Desktop modules

make[1]: 进入目录“/usr/src/linux-headers-4.15.0-39-generic”
Makefile:975: "Cannot use CONFIG_STACK_VALIDATION=y, please install libelf-dev,
libelf-devel or elfutils-libelf-devel"
  CC [M]  /home/danielshen/Desktop/module1.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/danielshen/Desktop/module1.mod.o
  LD [M]  /home/danielshen/Desktop/module1.ko
make[1]: 离开目录“/usr/src/linux-headers-4.15.0-39-generic”
```

重新 make 模块代码

```
danielshen@ubuntu:~$ cd Desktop/
danielshen@ubuntu:~/Desktop$ sudo insmod module1.ko
danielshen@ubuntu:~/Desktop$
```

插入新模块

```
danielshen@ubuntu:~/Desktop$ lsmod
Module                  Size  Used by
module1                16384  0
```

模块已经成功插入

```
danielshen@ubuntu:~/Desktop$ gcc usermew.c
danielshen@ubuntu:~/Desktop$ sudo ./a.out
HEAD

name            pid     state   pirorty p_pid
kthreadd        2       1       120     0
kworker/0:0H    4       1026    100     2
mm_percpu_wq    6       1026    100     2
ksoftirqd/0     7       1       120     2
rcu_sched       8       1026    120     2
rcu_bh  9       1026    120     2
migration/0     10      1       0       2
watchdog/0      11      1       0       2
cpuhp/0 12      1       120     2
cpuhp/1 13      1       120     2
watchdog/1      14      1       0       2
migration/1     15      1       0       2
ksoftirqd/1     16      1       120     2
kworker/1:0H    18      1026    100     2
cpuhp/2 19      1       120     2
watchdog/2      20      1       0       2
migration/2     21      1       0       2
```

```
TOTAL_NUMBER:           164
TASK_RUNNING:           0
TASK_INTERRUPTIBLE:     81
TASK_UNINTER:           0
TASK_STOPPED:           0
TASK_TRACED:            0
EXIT_DEAD:              0
EXIT_ZOMBIE:            0
EXIT_TRACE:             0
TASK_PARKED:            0
TASK_DEAD:              1
TASK_WAKEKILL:          0
TASK_WAKING:            0
TASK_NOLOAD:            0
TASK_NEW:               0
TASK_STATEMAX:          0
TASK_KILLABLE:          0
TASK_STOPPED:           0
TASK_TRACED:            0
TASK_IDLE:              82
TASK_NORMAL:            0
OTHERS:         0
```

编译运行用户态程序，成功输出结果

2.2.4 源代码

Module1.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/init_task.h>
// 初始化函数
// 计算总数的变量
int count1 = 0;
int temp;
// 计算各进程数量的变量
static int count[50];
int init_module(void)
{
    struct task_struct *p;
    p = NULL;
    // 从 init_task 开始遍历
    p = &init_task;
```

```
    printk("#$#\n");
    // 表头
    printk("@name\t\tpid\tstate\tpirorty\tp_pid\t");
    for_each_process(p)
    {
    // 对于每一个进程
        if(p->mm == NULL){ //内核线程的 mm 成员为空
        // 输出相关信息
            printk(KERN_EMERG"@%s\t%d\t%ld\t%d\t%d\n",p->comm,p->pid, p->state,p->normal_prio,p->parent->pid);
        count1++;
        // 按照 state 统计各种进程的个数
        if(p->state == 0x0000) count[0]++;
        else if(p->state == 0x0001) count[1]++;
        else if(p->state == 0x0002) count[2]++;
        else if(p->state == 0x0004) count[3]++;
        else if(p->state == 0x0008) count[4]++;
        else if(p->state == 0x0010) count[5]++;
        else if(p->state == 0x0020) count[6]++;
        else if(p->state == (0x0020 | 0x0010)) count[7]++;
        else if(p->state == 0x0040) count[8]++;
        else if(p->state == 0x0080) count[9]++;
        else if(p->state == 0x0100) count[10]++;
        else if(p->state == 0x0200) count[11]++;
        else if(p->state == 0x0400) count[12]++;
        else if(p->state == 0x0800) count[13]++;
        else if(p->state == 0x1000) count[14]++;
        else if(p->state == (0x0100 | 0x0002)) count[15]++;
        else if(p->state == (0x0100 | 0x0004)) count[16]++;
        else if(p->state == (0x0100 | 0x0008)) count[17]++;
        else if(p->state == (0x0002 | 0x0400)) count[18]++;
        else if(p->state == (0x0001 | 0x0002)) count[19]++;
        }
    }
    // 使用 temp 变量计算 other 进程
    // 将各种进程的信息输出到系统日志中
    temp = count1;
    printk("@TOTAL_NUMBER:\t\t%d\n",count1);
    printk("@TASK_RUNNING:\t\t%d\n",count[0]);
    temp -= count[0];
    printk("@TASK_INTERRUPTIBLE:\t%d\n",count[1]);
    temp -= count[1];
    printk("@TASK_UNINTER:\t\t%d\n",count[2]);
    temp -= count[2];
```

```
61.     printk("@TASK_STOPPED:\t\t%d\n",count[3]);
62.     temp -= count[3];
63.     printk("@TASK_TRACED:\t\t%d\n",count[4]);
64.     temp -= count[4];
65.     printk("@EXIT_DEAD:\t\t%d\n",count[5]);
66.     temp -= count[5];
67.     printk("@EXIT_ZOMBIE:\t\t%d\n",count[6]);
68.     temp -= count[6];
69.     printk("@EXIT_TRACE:\t\t%d\n",count[7]);
70.     temp -= count[7];
71.     printk("@TASK_PARKED:\t\t%d\n",count[8]);
72.     temp -= count[8];
73.     printk("@TASK_DEAD:\t\t%d\n",count[9]);
74.     temp -= count[9];
75.     printk("@TASK_WAKEKILL:\t\t%d\n",count[10]);
76.     temp -= count[10];
77.     printk("@TASK_WAKING:\t\t%d\n",count[11]);
78.     temp -= count[11];
79.     printk("@TASK_NOLOAD:\t\t%d\n",count[12]);
80.     temp -= count[12];
81.     printk("@TASK_NEW:\t\t%d\n",count[13]);
82.     temp -= count[13];
83.     printk("@TASK_STATEMAX:\t\t%d\n",count[14]);
84.     temp -= count[14];
85.     printk("@TASK_KILLABLE:\t\t%d\n",count[15]);
86.     temp -= count[15];
87.     printk("@TASK_STOPPED:\t\t%d\n",count[16]);
88.     temp -= count[16];
89.     printk("@TASK_TRACED:\t\t%d\n",count[17]);
90.     temp -= count[17];
91.     printk("@TASK_IDLE:\t\t%d\n",count[18]);
92.     temp -= count[18];
93.     printk("@TASK_NORMAL:\t\t%d\n",count[19]);
94.     temp -= count[19];
95.     printk("@OTHERS:\t\t%d\n",temp);
96.     printk("$#$\n");
97.     return 0;
98. }
99. // 清理函数
100. void cleanup_module(void)
101. {
102.     printk("goodbye!\n");
103. }
104. // 模块许可申明
```

```
105. MODULE_LICENSE("GPL");
```

User.c

```
1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  int main(){
5.      int ch;
6.      int output=0;
7.      FILE *fp;
8.
9.      //打开日志文件
10.     fp=fopen("/var/log/kern.log","r");
11.     if(fp==NULL){//若打不开则输出错误信息 返回-1
12.         printf("Can't open log file!\n");
13.         return -1;
14.     }
15.
16.     //读取日志文件
17.     fseek(fp,0,SEEK_SET);//文件指针重定位到文件头
18.     ch=fgetc(fp);
19.     //找到内核模块输出记录开头
20.     while(ch!=EOF){
21.         //输出记录开头的特殊标记为"#$#"
22.         if(ch=='#'){
23.             ch=fgetc(fp);
24.             if(ch=='$'){
25.                 ch=fgetc(fp);
26.                 if(ch=='#'){
27.                     printf("HEAD\n");
28.                     break;
29.                 }
30.             }
31.         }
32.         ch=fgetc(fp);
33.     }
34.     //打印出内核模块的输出记录
35.     while(ch!=EOF){
36.         if(ch == '@'){
37.             output = 1;
38.         }
39.         else if(ch == '\n'){
40.             printf("\n");
41.             output = 0;
```

```
        }
        else if(ch =='$'){//输出记录结尾的特殊标记为"$#$"
            ch=fgetc(fp);
            if(ch=='#'){
                ch=fgetc(fp);
                if(ch=='$'){
                    printf("EOF!\n");
                    break;
                }
            }
        }if(output == 1 && ch != '@'){
            printf("%c",ch);
        }
        ch=fgetc(fp);
    }
    //关闭日志文件
    fclose(fp);
    return 0;
}
```

Makefile

```
obj-m:=module1.o
KDIR:= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) M=$(PWD) cleanobj-m:=module1.o
```

#讨论、心得（20 分）

本次实验我学到了 Linux 线程库、Linux 系统调用程序编写技术，也熟悉了 Linux 环境，为之后的实验打下了基础。

本次实验的挑战还是很多的，比如互斥这一块我经常脑子转不过来，在编写 makefile 的时候也遇到了无数坑，所幸我都一一客服了下来。我曾经求助于 ppr，但是发现 ppt 上的内容并不全，我相信这是老师的意志，是为了让我们看书。所以我认认真真的看了书。以后我会更加重视书本的重要性。

因为是第一次尝试使用一种新的环境去做东西，各种莫名其妙的坑再所难免，所以最终在日志中调出实验结果的时候，那种熟悉的成就感油然而生。以后我也会继续好好学习。