浙江大学实验报告

课程名称：操作系统分析及实验　　实验类型：设计性

实验项目名称：实验 1 同步互斥和 Linux 内核模块

学生姓名：蒋仕彪　专业：求是科学班 1701　学号：3170102587

电子邮件地址：969519450@qq.com　手机：18888921530

实验日期：2019 年 11 月 21 日

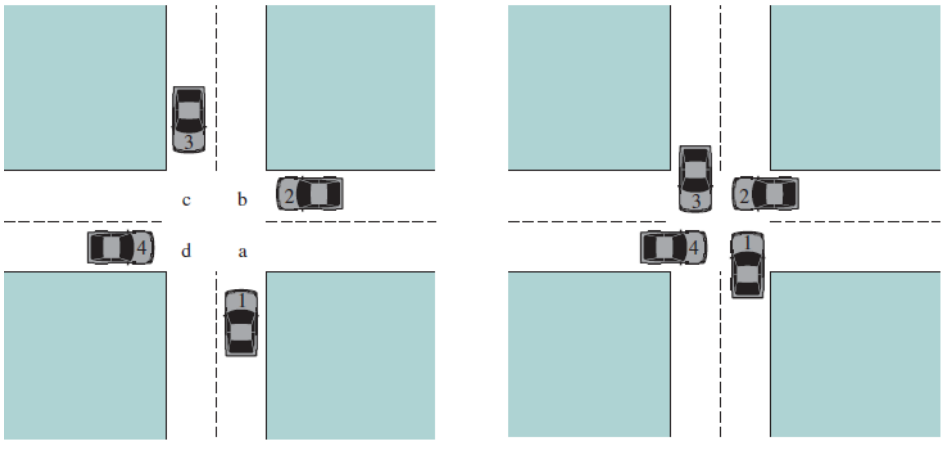
一、实验目的

- 学习使用 Linux 的系统调用和 pthread 线程库编写程序。
- 充分理解对共享变量的访问需要原子操作。
- 进一步理解、掌握操作系统进程和线程概念，进程或线程的同步与互斥。
- 学习编写多线程程序，掌握解决多线程的同步与互斥问题。

二、实验内容

1. 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。



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。

编写程序实现避免产生死锁和饥饿的车辆通过十字路口方案，并给出详细的设计方案，程序中要有详细的注释。

2. 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；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，显示内核模块输出的内容。要求：**程序中每行代码都要有注释**

三、实验器材

安装了 Vmware 的 Windows10 计算机。虚拟机为 Linux 4.15.0-generic。

四、小车实验的操作方法和实验步骤

4.1 基本架构

◆ 小车的表示

```
struct Car{
    int id;
    int queueNumber;
    int dir;
};
```

- 一个唯一的编号 id，用来识别和输出。
- queueNumber 表示小车在当前方向上是第几辆。
- dir 表示小车所属的方向（我们用 0,1,2,3 对四个方向进行标号）。

- 线程设置
  - 根据题目要求，对每一辆小车单独开一个线程 carThread[i]，如下所示：

```
for (int i = 0; i < len; i++){
    pCar carUnit = (pCar) malloc(sizeof(struct Car));
    carUnit->id = i + 1;
    carUnit->dir = dirDecoder(carQueue[i]);
    carUnit->queueNumber = ++dirTotal[carUnit->dir];
    int ret = pthread_create(&carThread[i], NULL, carHandle, carUnit);
    if (ret){
        printf("Error! Thread for car %d can not be created.\n", carUnit->id);
    }
}
```

  - 调用 pthread.h 的接口 pthread_create() 来创建线程，用法如下：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*),void *arg);
    //thread          指向线程标识符的指针，使用这个标识符来引用新线程
    //attr            设置线程属性，设为 NULL 则生成默认属性的线程
    //start_routine   线程运行函数的起始位置
    //arg             线程运行函数的参数
```

  - 所以本实验的终点是：**设计 carHandle() 函数，来解决每辆小车的调度**。

- 四个方向的状态设计
  - 有时候一个方向上可能车流滚滚，有时候可能一辆车都没有。
  - 我们要对某个方向设计一些状态来合理表示小车的通过情况。

```
#define FREE 0          //There is no car in this direction.
#define READY 1         //A car is in front of the crossing, waiting to pass.
#define CROSSING 2      //A car is crossing the intersection and then leaving.
```

  - 开一个数组 dirState[i]，用以上三个值来表示方向 i 的当前状态。

- 如何表示“小车的等待”
  - 我们知道，小车线程 t 到达路口的时候，如果左边或者右边有小车正在通过，该线程需要被阻塞。我们用 pthread_cond_t 来控制进程之间的阻塞和唤醒。

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

  - Cond 表示一个条件（它的类型是 pthread_cond_t），mutex 表示一个互斥锁（它的类型是 pthread_mutex_t）。
  - pthread_cond_wait() 是要放在 mutex 的 lock() 和 unlock() 之间被保护的。**每当执行 pthread_cond_wait() 进入睡眠后，对应的锁会被暂时释放**。具体步骤：
    1. mutex 会被解锁。
    2. 该线程会一直休眠直到 cond 被激活。
    3. 重新对 mutex 上锁。
  - cond 会在 pthread_cond_signal() 或者 pthread_cond_broadcast()（前者是激活单个线程，后者是广播**全部**线程）时被激活，从而使线程恢复运行。

4.2 核心逻辑

◆ **基本：每个方向上的线程控制**

- Ready 表示一个方向上有车已经就位了。如果有多辆车从该方向来，我们就要考虑一个结构，能让小车一辆一辆顺序通过。即：在前一辆没有完全通过前，后一辆不能进入 Ready 状态。
- 我们要找到一个合适的线程，然后将 dirState[currect_direction] 置为 Ready。注意到，在别的线程运行的时候可能会用到当前方向的状态，所以我们需要用互斥锁将 criterion section 保护起来。
- 那么如何找到最近一辆小车所代表的线程呢？我们只需在此方向上维护一个 dirDone[dir]，表示此方向已经通过的小车数量；那么现在的小车标号必须是该数量 +1。如果不是，我们就要睡眠该进程，等正确的小车通过后，触发 dirLeaving[dir] 条件再唤醒它。注意，可能有老远的小车线程很早就进入了队列，我们需要套一个 while 来不断将其睡眠，直至满足要求。详见下面的代码：

```
pthread_mutex_lock(&dirLock[car->dir]);
while (dirDone[car->dir] + 1 != car->queueNumber){
    pthread_cond_wait(&dirLeaving[car->dir], &dirLock[car->dir]);
}
dirState[car->dir] = READY;
printf("car %d from %s arrives at crossing\n", car->id, name[car->dir]);
pthread_mutex_unlock(&dirLock[car->dir]);
```

◆ **核心：设计防止饥饿的机制**

- 如果相邻两个方向同时有小车抵达路口，靠右边的先行。但是题目又要求有防饥饿的机制，即如果右侧一直有小车来，左侧的小车必须在某个时刻能通行。
- 有一个注意点是：经过上述每个方向的线程控制后，**任何时刻最多只有四个线程正在运行**。即我们只需考虑不同方向上线程的并发，这样问题就简化了。
- 一个很简单自然的想法是：设置条件变量数组 trigger[4]（四个方向各设置一个）。当某方向一辆车 Ready 的时候，我们先观察它右侧的小车：如果右侧小车是 Ready 或者 Crossing 状态，我们就得睡眠此方向的线程，等右侧小车完全通过后，发射一个 trigger 信号来激活此方向的小车。
- **遗憾的是，上述做法是有问题的。**考虑这样一个场景：当右侧小车开走后，右侧立刻又有一辆小车就位。此时当前小车也被激活，它和右侧的新车没有任何信息交流，可能会一起开而撞在一起。
- 为了修补上面的做法，需要再开设一个布尔变量 leftFirst[4]。当右侧小车被放行后，右侧的 leftFirst 需要置为 1，表示右方向正在**无条件让行本方向**（每当右侧有新小车来的时候，如果 leftFirst 置为 1 就要强制等待）。等本方向的小车通过十字路口后，才能将右方向 leftFirst 置为 0。
- 这里就**引入了第二个问题**：如上段所述，“当右侧小车被放行后，右侧的 leftFirst 需要置为 1”，是为了给本方下一辆小车优先权，防止饥饿。但如果右侧新车到达路口的时候，本方依然还没来车呢？显然，为了资源的合理利用，我们肯定要继续批准右侧新车进入十字路口。即，**leftFirst 仅仅是一个优先权标记，而不是一个强制等待的限制**。
- 于是又**产生了第三个问题**。考虑这样一种场景：右方新车刚抵达路口，发现有 leftFirst 标记是 1，于是它查看了本方的行车状态准备让行；结果本方显示是

FREE（无车）状态，于是右车线程被批准可以 CROSSING（但还未执行 CROSSING 语句）；巧的是，**在右车线程查看本方状态刚结束、释放本方的状态锁后，本方向突然来了一辆小车，立即获取该锁并将状态改成 READY 了！**然后本方小车开始过马路判定：他发现右侧小车目前还是 READY 状态（还没来得及 CROSSING），且 leftfirst 标记是 1（它会让咱们）——于是本方小车也义无反顾地冲进十字路口。右方小车刚才已经通过了所有 check，它也大摇大摆地 CROSSING，就导致了撞车。

- 解决以上问题的关键是：**leftFirst 标记要随着不同的情况动态修改。**当右车线程发现本方暂时 FREE 后，它会无视 leftFirst 的优先权而直接冲过去。此时，右车线程趁着掌握了本方向 state 状态的互斥锁（本方还不能把状态设置为 READY），不妨将 leftFirst 直接改成 0，表示“我查询你的时候发现你是 FREE 的，我可以强制不让你”。至此，同步和互斥问题都解决了。
- 详细代码如下。注意一个方向既要处理它和右侧的关系（leftFirst[R]），也要处理它和左侧的关系（leftFirst[now]），所以会类似地写两遍。经过了这两遍 check 后，就可以放心地将状态设成 CROSSING。

```
pthread_mutex_lock(&dirLock[Right(car->dir)]);
if (dirState[Right(car->dir)] == READY && !leftFirst[Right(car->dir)] || dirS
tate[Right(car->dir)] == CROSSING){
    //Wait right car cross the street if it doesn't have leftFirst tag.
    pthread_cond_wait(&dirLeftGo[Right(car->dir)],&dirLock[Right(car->dir)]);
    //After this thread being waken up, the right car must have leftFirst tag
, so it will wait for this thread.
}
leftFirst[Right(car->dir)] = 1;
//leftFirst[Right(car->dir)] may be false. We must put it to true to stop the
 right car who will come at any time.
pthread_mutex_unlock(&dirLock[Right(car->dir)]);

pthread_mutex_lock(&dirLock[Left(car->dir)]);
if (dirState[Left(car->dir)] == CROSSING || dirState[Left(car->dir)] == READY
 && leftFirst[car->dir]){
    //If leftFirst tag is true, we must wait for the left car to go first
    pthread_cond_wait(&dirRightGo[Left(car->dir)], &dirLock[Left(car->dir)]);
    //After this thread being waken up, leftFirst[dir] must be false
}
leftFirst[car->dir] = 0;//leftFirst[car->dir] may be true. We must put it to
false  to stop the left car who will come at any time.
pthread_mutex_unlock(&dirLock[Left(car->dir)]);


pthread_mutex_lock(&dirLock[car->dir]);
dirState[car->dir] = CROSSING;
pthread_mutex_unlock(&dirLock[car->dir]);
usleep(SLEEP_AFTER_CROSSING);
```

◆ **思考：死锁的检测和解开**

- **死锁的检测**相对比较简单：我们另开一个线程 deadLockDectection，以某种手段做到每隔一段时间运行一次。每次检测时，我们依次获取 dirState[4] 这四个方向的互斥锁，**如果它们同时是 READY 状态，就发生了死锁**。
- 这里需要注意一个问题：同时获取这四个方向的互斥锁，会不会导致线程之间的获取锁而不得的死锁局面？可以证明是不会的。除了死锁检测代码，在代码的任何一处我都保证：最多只有一个互斥锁被拿到。所以不会出现“A 线程拥有 x 想要 y，B 线程拥有 y 想要 x”的僵持情况。

- **死锁的解开**需要重新设计结构。本来我们只需在每个方向设计一个条件变量 dirLeaving[4]，表示目前方向 i 的车是否已经通过这个式子路口（如果通过了，就 broadcast 这个条件变量来唤醒一些睡眠线程，包括它身后的车以及左右的车）。但是死锁被发现后，我们很难去唤醒我们想要的线程：不能盲目将北边小车的 dirLeaving 手动触发，这样该北方小车的线程并不能被正确执行。为此，我重新加了两组条件变量 dirLeftGo[4] 和 dirRightGo[4]。在左右方向车互相阻塞时，用 dirLeftGo 和 dirRightGo 来睡眠和激活。这样当死锁被发现后，我们只需将与北方对应的这两组变量激活，意思是**“北方车左右的小车均激活它，表示可以让它先行”**。这样北方车可以畅通无阻地通过十字路口。
- 还有一个有趣的事情是，如何设置死锁检车线程，使其不间断地工作？我想到了两种方法：①不断（死循环式地）运行该线程，每当发现死锁后启动解开程序。但这里有个问题：被激活的线程还未完全执行完，死锁程序又在循环，就重复监测了。我们可以设一个布尔变量 find ，只有 find =1 时才监测死锁，一旦监测到标记为 0；每当任何一辆车通过后，将 find 标记为 1，意为解开程序运行成功，可以继续监测死锁。②设置一个时间片常量 SLEEP_WHEN_DEADLOCK，每隔这段时间后再监测。我采用了方法②，因为这样可以节省 CPU 资源。

- **代码如下：**

```
void *deadLockDetection(void *empty){
    while (allDone < needDone){
        #ifdef DEBUG
            printf("Start in deadLockDetection\n");
        #endif
        for (int i = 0; i < 4; i++)
            pthread_mutex_lock(&dirLock[i]);
        if (dirState[0] == READY && dirState[1] == READY && dirState[2] ==
READY && dirState[3] == READY){
            printf("DEADLOCK: car jam detected, signalling %s to go\n", nam
e[0]);
            pthread_cond_broadcast(&dirLeftGo[1]);
            pthread_cond_broadcast(&dirRightGo[3]);
        }
        for (int i = 0; i < 4; i++)
            pthread_mutex_unlock(&dirLock[i]);
        usleep(SLEEP_WHEN_DEADLOCK);
    }
}
```

4.3 验收和反思

本来以为自己的做法毫无漏洞，验收的时候还是被找出破绽。李老师的评价是：**这的确是一个无饥饿、无死锁的小车调度系统，但是不满足题目要求**。他举出了一个例子：

Time 0 东方来了一辆小车

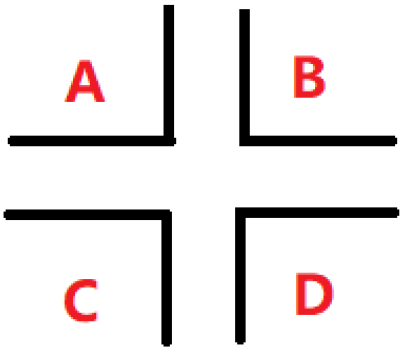
Time 1 南方来了一辆小车

Time 2 西方来了一辆小车

当东方小车判断能否通过的时候，需要取左方（也就是南方）的状态锁。假设锁住的时间过长，Time 1 南方来的小车在排队等锁，来不及更新自己的状态。之后西方小车到达，排队等待南方状态锁。由于**锁的竞争是随机的**，如果西方小车先拿到南方锁，它会认为没有车直接开过去，这**违背了题目的要求**。按照题目的描述，通过路口的顺序必须是东、南、西。

4.4 最终改进

上述问题产生的原因是：每个方向可以取右方的状态锁，**但是不能取左方的状态锁**。那如何来获取左方的状态信息呢？在李老师的提示下，我最终设计出了如下方案：



在十字路口的四个角设四个信号量 A、B、C、D，用来处理和左侧交互的信息。这些信号量的初值是 1，即左右至多一个能获得。**以南方小车为例**，它的判定操作如下：

Wait (C)
获取右侧的锁。如果右侧 Ready，睡眠直到右车通过。
Wait (D)　（占用了 D 的资源，告诉右方小车必须等我通行）
---------本方小车通行---------
Signal (D)　（告诉右方小车，它们又变成了优先状态）
Signal (C)　（释放 C 信号量；如果左侧有小车等着，它会迅速获得 C 信号量）

巧妙地利用信号量，让不想饥饿的左方小车阻塞当前小车。

五、模块加载实验的操作方法和实验步骤

5.1 编写内核模块

内核模块的核心是，我们要正确地遍历每个进程，并统计一些信息。

为了标记这个内核模块，我在初始化时，先输出一个 Hello World 进行标识。

遍历内核需要用到 task_struct 这个类型。我们可以用 p->comm, p->pid, p->state 分别来取它的进程名字、进程编号和进程状态。主程序如下：

```
int init_module(void){
    printk(KERN_INFO "Hello World!\n");
    const int maxState = 1<<12;
    int count[maxState] = {0};
    struct task_struct *p;
    for (p = &init_task; (p = next_task(p)) != &init_task; ){
        count[p->state]++;
        printk(KERN_INFO "%s %s ", p->comm, p->pid);
        printString(p->state);
        printk(KERN_INFO " %s\n", p->real_parent->comm);
    }
    for (int state = 0; state < maxState; state++)
        if (count[state]){
            printString(state);
            printk(KERN_INFO " %d\n", count[state]);
```

```
        }
}
```

注意题目还要求统计各种进程状态的个数，所以我们还得知道有哪些状态。我在互联网上找到了很多很多状态，它们往往是 2 的幂次（方便状态的融合和计算），而且最大不超过 211. 所以我开了一个大小为 212 的桶装数组计数。状态情况见下：

```
void printString(int state){
    switch (state){
        case TASK_RUNNING: printk(KERN_INFO "TASK_RUNNING"); break;
        case TASK_INTERRUPTIBLE: printk(KERN_INFO "TASK_INTERRUPTIBLE"); break;
        case TASK_UNINTERRUPTIBLE: printk(KERN_INFO "TASK_UNINTERRUPTIBLE"); br
eak;
        case TASK_ZOMBIE : printk(KERN_INFO "TASK_ZOMBIE"); break;
        case TASK_STOPPED : printk(KERN_INFO "TASK_STOPPED"); break;
        case TASK_TRACED: printk(KERN_INFO "TASK_TRACED"); break;
        case TASK_DEAD : printk(KERN_INFO "TASK_DEAD"); break;
        case EXIT_ZOMBIE : printk(KERN_INFO "EXIT_ZOMBIE"); break;
        case EXIT_DEAD : printk(KERN_INFO "EXIT_DEAD"); break;
        default: printk(KERN_INFO "UNDEFINED STATE"); break;
    }
}
```

5.2 编写用户态程序

题目还要求把日志里的文本输出到控制台，所以需要写一个用户态程序。

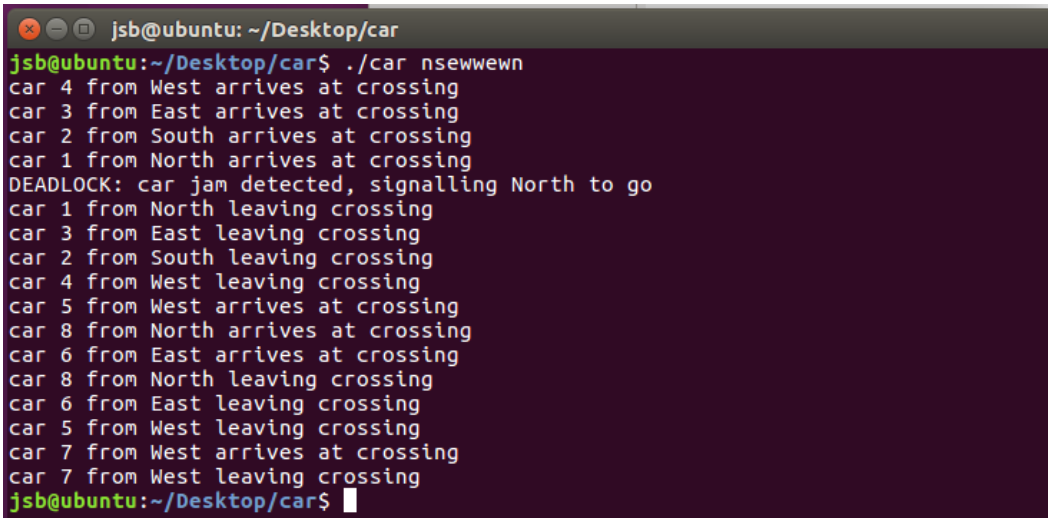
上述内核模块的信息会输出在 `/var/log/kern.log` 里，所以我们要把那个文件读进来逐行扫描。每当读到 Hello World 时，我们就要把结果开始展示在控制台上了。

为了美观，我把每个状态都进行了一定的缩进表示，代码如下：

```
    FILE *fp = fopen("/var/log/kern.log", "r");
    if (fp == NULL){
        printf("open file failed!\n");
        return 0;
    }
    int doing = 0;
    while(!feof(fp)){
        char buffer[205];
        fgets(buffer, 200, fp);
        char message[205];
        int p = 0, cnt = 0;
        for (p = 0; buffer[p]; p++) if (buffer[p] == ']') break;
        if (!buffer[p]) p = -1;
        for (++p; buffer[p]; p++)
            message[cnt++] = buffer[p];
        int len = strlen(message);
        if (message[0] == 'H' && message[1] == 'e' && message[2] == 'l' && mess
age[3] == 'l' && message[4] == 'o'){
            doing = 1;
            printf("%20s%5s%20s%20s\n", "Process Name", "Pid", "State", "Father
 Name");
        }
        else if (message[0] == 'G' && message[1] == 'o' && message[2] == 'o' &&
 message[3] == 'd') doing = 0;
        else if (doing)
        {
            printf("%s\n", message);
        }
    }
```

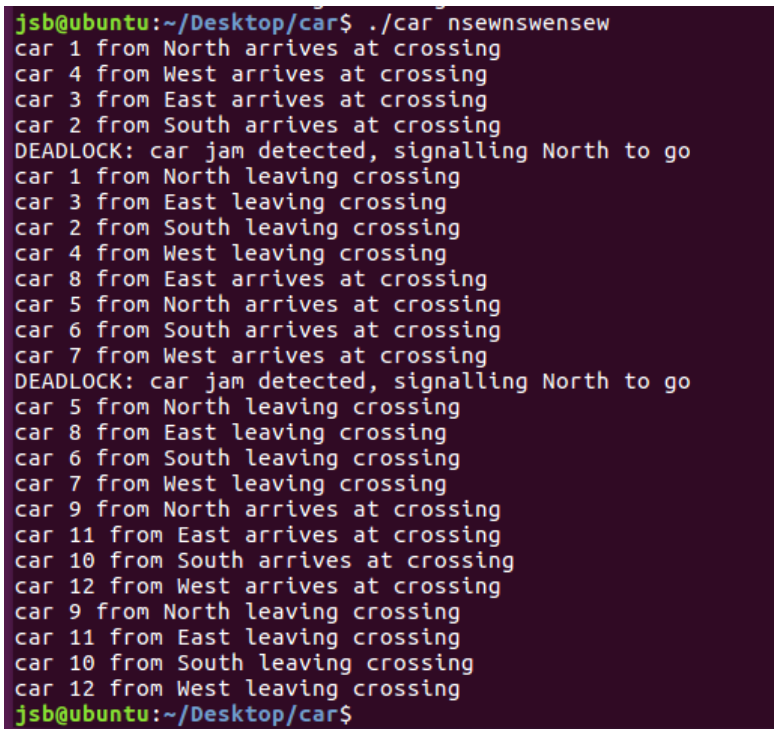
六、实验结果和分析

6.1 小车样例检测



系统安全地检测到了死锁，并安排车辆通过

6.2 两次死锁的检测



多次死锁也能检测出来

6.3 随机大数据测试

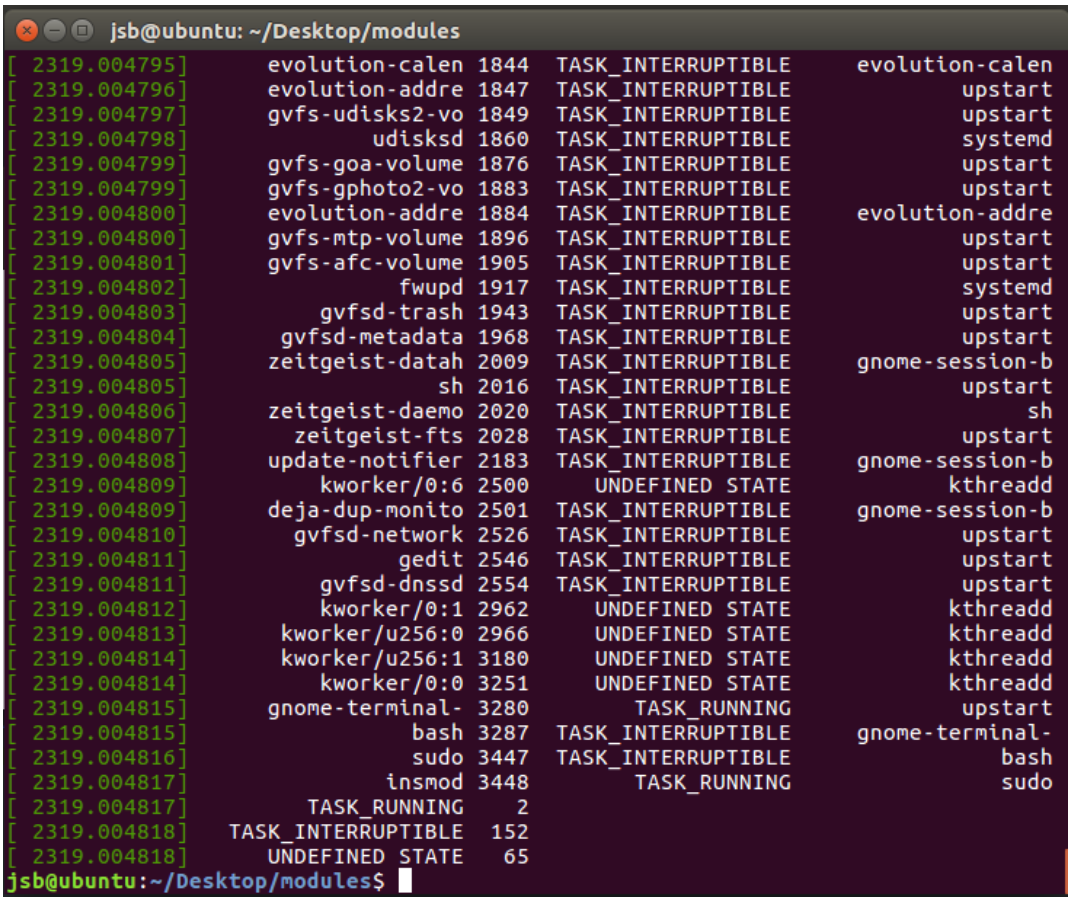
```
car 47 from East arrives at crossing
car 45 from South leaving crossing
car 48 from South arrives at crossing
car 47 from East leaving crossing
car 48 from South leaving crossing
car 50 from South arrives at crossing
car 49 from West arrives at crossing
car 51 from East arrives at crossing
car 49 from West leaving crossing
car 51 from East leaving crossing
car 52 from East arrives at crossing
car 50 from South leaving crossing
car 52 from East leaving crossing
car 55 from North arrives at crossing
car 56 from South arrives at crossing
car 54 from East arrives at crossing
car 53 from West arrives at crossing
DEADLOCK: car jam detected, signalling North to go
car 55 from North leaving crossing
car 53 from West leaving crossing
car 56 from South leaving crossing
car 54 from East leaving crossing
car 58 from East arrives at crossing
car 57 from North arrives at crossing
car 60 from West arrives at crossing
car 57 from North leaving crossing
car 58 from East leaving crossing
```

./1000 的部分截图。

6.4 内核模块的编译

```
jsb@ubuntu: ~/Desktop/modules
jsb@ubuntu:~/Desktop/modules$ make clean
rm -rf *.o *.ko *.mod.c .*.cmd *.markers *.order *.symvers .tmp_versions
jsb@ubuntu:~/Desktop/modules$ make
make -C /usr/src/linux-headers-4.15.0-70-generic M=/home/jsb/Desktop/modules mod
ules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-70-generic'
  CC [M]  /home/jsb/Desktop/modules/getstate.o
/home/jsb/Desktop/modules/getstate.c: In function ‘init_module’:
/home/jsb/Desktop/modules/getstate.c:31:2: warning: ISO C90 forbids mixed declar
ations and code [-Wdeclaration-after-statement]
  int i;
  ^
/home/jsb/Desktop/modules/getstate.c:35:2: warning: ISO C90 forbids mixed declar
ations and code [-Wdeclaration-after-statement]
  struct task_struct *p;
  ^
/home/jsb/Desktop/modules/getstate.c:39:3: warning: ISO C90 forbids mixed declar
ations and code [-Wdeclaration-after-statement]
   char message[30]; printString(message, p->state);
   ^
/home/jsb/Desktop/modules/getstate.c:43:2: warning: ISO C90 forbids mixed declar
ations and code [-Wdeclaration-after-statement]
  int state;
  ^
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jsb/Desktop/modules/getstate.mod.o
  LD [M]  /home/jsb/Desktop/modules/getstate.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-70-generic'
jsb@ubuntu:~/Desktop/modules$ sudo rmmod getstate
[sudo] password for jsb:
jsb@ubuntu:~/Desktop/modules$ sudo insmod getstate.ko
jsb@ubuntu:~/Desktop/modules$ dmesg
```

6.5 内核模块的结果



调用用户态程序后返回的结果如图

七、讨论、心得

1. 小车的程序搞了我很长的时间，相当有难度。最开始接触这道题的时候，我觉得直接看一下左右方向的状态，随便判断一下就行。但是每次实现一个想法之后，我都能找到反例来 hack 自己。**多线程的编程太容易出漏洞了**。
2. 此外，**多线程编程的调试也特别麻烦**。其实除了输出调试，我并没有别的调试手段了。输出调试也有一定的瓶颈，即使是同样的输入，多次运行结果也可能不同。每当遇到难以解决的错误时，我只能一遍遍地 check 代码，或者在互联网上确认函数的调用方法。Cond_wait，mutex 等函数第一次接触时用得好累。
3. 小车实验还有一个麻烦的地方在于：如何测试数据？如果直接把小车丢入线程队列，CPU 会直接一个一个让小车通过，违背了我的本意。所以我尝试在各种地方加入 sleep，希望让前面的小车线程“慢下来”，这样才会有“卡住”和“堵塞”的效果。
4. 当我把精雕细琢的程序给李老师验收的时候，他认为我的系统不符合题目要求，让我很受打击。不过他给了我一个小提示：用信号量来解决向左看的问题。于是我回去思考了一周，得到了一个更精妙的做法，下一次验收的时候终于李老师说服了哈哈。
5. 内核模块编写的任务相对于小车实验来说，没什么技术难度。只需按照说明书一步一步来就行，代码量也非常小。验收的时候我还听同学说，可以用 for_each_process （这是一个自带的宏定义）代替进程的遍历，代码会更加简单。