浙江大学

本科实验报告

课程名称：　操 作 系 统

实验名称：　同步互斥和内核模块实验

姓　　名：　张 溢 弛

学　　院：　计算机科学与技术学院

系：　软件工程

专　　业：　软件工程

学　　号：　3180103772

指导教师：　季江民

2020 年　12 月　6 日

目录

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：同步互斥和内核模块实验
学生姓名：张溢弛 学号：3180103772
电子邮件地址：3180103772@zju.edu.cn
实验日期：2020 年 11 月 20 日

一、实验环境

操作系统：Windows 10
虚拟机：VMware Workstation Pro 16
Linux 版本：Ubuntu 18.04
内核版本：5.3.0-64-generic
代码编辑器：Visual Studio Code

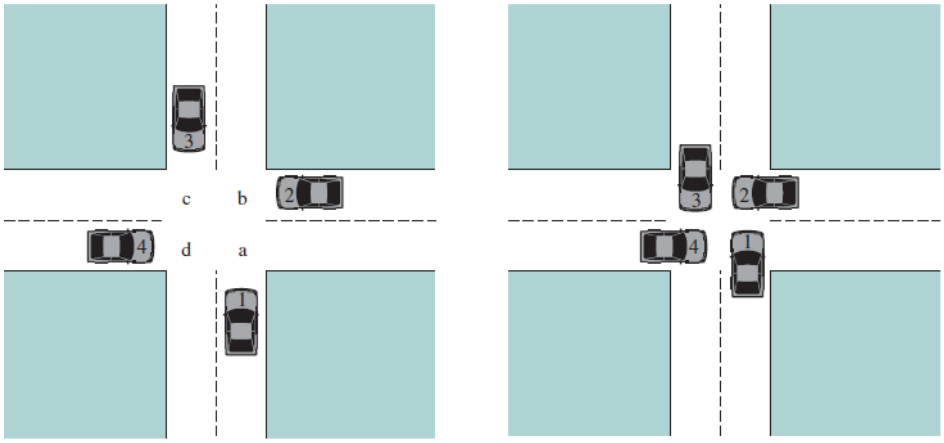
二、实验内容和结果及分析

第一部分：同步互斥

1.1 实验内容和预备知识

- 实验内容：

  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，那么对资源有一定的要求。



(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。车辆通行需要遵守如下规则

- 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；
- 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**；
- 避免产生死锁；
- 避免产生饥饿；
- 任何一个线程（车辆）不得采用单点调度策略；
- 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

- 实验的预备知识
  - 进程的同步、死锁等操作系统基本知识。本题中四个方向的车辆就好比是需要访问临界资源的临界代码段，而十字路口四个方向的可通过路径就是需要访问的临界资源，而根据题意，每个临界区需要访问两个临界资源，因此就会造成访问的冲突
  - 我们的目标就是编程来解决每个方向上车辆的对资源的访问，使得车与车之间同步关于访问资源的信息，并且需要检测是否出现死锁，以及出现死锁时的调度策略。
  - Linux 环境下的 POSIX 线程库用法，包括线程的创建和信号量的使用

1.2 实验设计

- 程序总体设计思路：
  - 首先需要读入控制台中输入的车辆序列(四个方向分别用 nwse 来表示)，并初始化一系列小车的结构体(这个结构体在程序中定义，包含小车的行驶方向和编号)
  - 然后我们为每个方向的每一辆小车都创建一个线程，并开启这个线程进行运行
  - 每个线程中小车都被放置在一个队列中，只有每个队列顶部的小车需要进行通过十字路口的调度，因此小车如果不是处于队列的顶部或者前面依然有小车在等待通信，则该小车就需要等待一个条件信号量释放之后才可以进入十字路口，参与通行的调度
  - 当一辆小车进入十字路口准备通行的时候，程序对其进行如下调度：
    - 观察右边有没有车辆正在准备通过马路，如果没有就直接通过马路并打印通过的信息
    - 如果右边有车也准备过马路，根据题目规则，应该是右边的车优先级更高，应该让右边的车先通行，因此小车进入等待状态，等待右边的小车通过之后给左边的小车发送信号量
    - 此时如果出现了死锁，即四个方向都有车辆要通过马路，死锁检测线程会检测到这一情况，并给北方的小车发送信号量让北方的小车先通行
  - 小车通行之后会打印小车通过路口的信息，同时从队列中移除，然后分别给左边方向的小车和同方向的队列发送信号，让下一辆小车继续通行
  - 程序运行结束之后系统会销毁之前使用的所有信号量和互斥锁，防止内存泄漏等异常发生

- 关键信号量和全局数组
  - 互斥信号量：
    - cross 和 dead_lock_mutex，direction_mutex[4],queue_mutex[4]分别代表对十字路口、死锁检测调度、小车方向、同方向队列的访问的互斥限制，即同一个资源不能被两个临界区同时访问
  - 条件信号量：queue_cond[4]和 next_car[4]分别用于给同一个方向的队列和下一个方向的小车发送信号告诉它可以通行
  - 我们用四个队列 queue<int> waiting_car[4]来表示小车的队列，在读入小车的时候就将小车放入队列中

- is_arrived [4]小车是否到达了路口，is_waiting[4]表示十字路口处是否有车在等待正在准备通行

- 程序结构：

  主要由五个部分组成，一个是信号量和数组的初始化模块，第二个是输入信息读取和线程创建模块，第三个是小车线程运行模块，第四个是信号量销毁的模块，第五个是单独的一个死锁检测线程，该线程处于一直运行的状态。

  其中小车的线程是一个单独的函数，传入小车的地址作为参数，对每辆小车进行调度。

  死锁检测线程也是单独的一个函数，在程序的一开始就创建并运行

1.3 程序运行结果

- 一辆小车，最简单的情况

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
n
Car 1 from north arrives at crossing.
Car 1 from north leaving crossing.

Process finished with exit code 0
```

- 三辆车，无死锁的情况

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
nws
Car 1 from north arrives at crossing.
Car 3 from south arrives at crossing.
Car 3 from south leaving crossing.
Car 2 from west arrives at crossing.
Car 2 from west leaving crossing.
Car 1 from north leaving crossing.

Process finished with exit code 0
```

- 四辆车，最经典的一次死锁，检测到出现死锁之后会按照既定的调度策略来进行调度

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
nwse
Car 2 from west arrives at crossing.
Car 1 from north arrives at crossing.
Car 3 from south arrives at crossing.
Car 4 from east arrives at crossing.
DEADLOCK: car jam detected.Deadlock schedule wake up and signal north.
Car 1 from north leaving crossing.
Car 4 from east leaving crossing.
Car 3 from south leaving crossing.
Car 2 from west leaving crossing.

Process finished with exit code 0
```

- 八辆车，一次死锁

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
nwsesewn
Car 2 from west arrives at crossing.
Car 3 from south arrives at crossing.
Car 4 from east arrives at crossing.
Car 1 from north arrives at crossing.
DEADLOCK: car jam detected.Deadlock schedule wake up and signal north.
Car 1 from north leaving crossing.
Car 8 from north arrives at crossing.
Car 4 from east leaving crossing.
Car 6 from east arrives at crossing.
Car 3 from south leaving crossing.
Car 5 from south arrives at crossing.
Car 2 from west leaving crossing.
Car 7 from west arrives at crossing.
Car 8 from north leaving crossing.
Car 6 from east leaving crossing.
Car 5 from south leaving crossing.
Car 7 from west leaving crossing.

Process finished with exit code 0
```

- 多辆车检测到两次死锁，内容比较多，截图不完整

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
nwsewnweeswesnew
Car 1 from north arrives at crossing.
Car 2 from west arrives at crossing.
Car 3 from south arrives at crossing.
Car 4 from east arrives at crossing.
DEADLOCK: car jam detected.Deadlock schedule wake up and signal north.
Car 1 from north leaving crossing.
Car 6 from north arrives at crossing.
Car 4 from east leaving crossing.
Car 3 from south leaving crossing.
Car 8 from east arrives at crossing.
Car 10 from south arrives at crossing.
Car 2 from west leaving crossing.
Car 6 from north leaving crossing.
Car 5 from west arrives at crossing.
DEADLOCK: car jam detected.Deadlock schedule wake up and signal north.
```

- 大量同方向的车

```
/Users/randomstar/Desktop/oslab1/cmake-build-debug/oslab1
nnnnnnn
Car 1 from north arrives at crossing.
Car 1 from north leaving crossing.
Car 2 from north arrives at crossing.
Car 2 from north leaving crossing.
Car 3 from north arrives at crossing.
Car 3 from north leaving crossing.
Car 4 from north arrives at crossing.
Car 4 from north leaving crossing.
Car 5 from north arrives at crossing.
Car 5 from north leaving crossing.
Car 6 from north arrives at crossing.
Car 6 from north leaving crossing.
Car 7 from north arrives at crossing.
Car 7 from north leaving crossing.

Process finished with exit code 0
```

1.4 结果分析

- 根据上面的程序运行和测试结果，该程序实现了题目的所有要求。
- 关于死锁时的调度策略，我采取的是先发现的先通行，给先发现的车发送一个信号让其立即通行，然后再按照左右的优先级让剩下的三辆车轮流通行
- 如何防止饥饿：我采用的策略是四个方向轮流通行，即每次一辆车通

过之后先给左边的路口处的车发送信号让其通行，再让同方向队列中的下一辆车通过

- 如何防止死锁：单独开一个线程来检测可能出现的死锁的情况，每次出现死锁的时候，就按照规则给北方发送信号量让北方先走，然后就会按照北东南西的顺序顺次通过，因为小车每次通过之后都会先给左边的小车发送信号
- 实验的源代码和心得体会见第三部分和第四部分

第二部分：编写一个自己的内核模块

2.1 实验内容与预备知识

- 实验要求：
  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
- 实验所需预备知识：Linux 内核源码中的进程的数据结构
  Linux 内核中进程用一个结构体 struct task_struct 来表示，这个结构中包含了每个进程的进程名(正在执行的可执行文件的名称)，进程的唯一标识号 pid，进程的状态(分为若干种)，父进程和子进程的指针等相关信息，其中比较特殊的是头进程 init_task，在头文件 linux/init_task.h 中进行了定义

2.2 实验总体设计思路

实验的代码分为内核模块部分和用户态程序部分，其中内核模块部分的设计思路如下：

- 定义一个 Linux 系统下进程结构的指针 p 并将其初始化为 init_task
- 用 for_each_process(p) 来遍历当前系统中的所有进程
- 对于系统中的每个进程 p，打印出进程的信息，包括进程的名字，进程的 pid，进程的父进程 pid 以及进程所处的状态
- 并通过 p->state 来获取进程当前的状态，并记录到数组 count 对应的位置
- 在遍历结束之后，输出各个状态的进程的统计信息，并结束内核模块的一次运行
- 内核模块在卸载的时候会用 printk 输出 `See you next time!!`

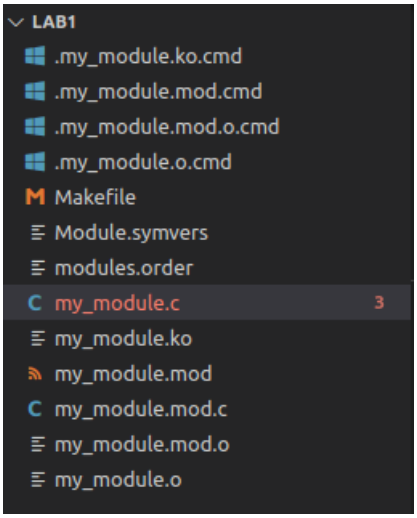
2.3 实验运行结果与分析

完成内核模块代码的编写之后，需要进行如下操作

- 编写 Makefile 并进行源代码的编译，我使用的编辑器是 Visual Studio Code，代码编译的过程如下(Makefile 可以见实验源代码)

```
randomstar@ubuntu:~/Desktop/lab1$ make
make -C /lib/modules/5.3.0-64-generic/build M=/home/randomstar/Desktop/lab1 modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-64-generic'
  CC [M]  /home/randomstar/Desktop/lab1/my_module.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/randomstar/Desktop/lab1/my_module.mod.o
  LD [M]  /home/randomstar/Desktop/lab1/my_module.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-64-generic'
randomstar@ubuntu:~/Desktop/lab1$
```

- 之后会在源代码所在目录下生成一系列文件，如下图所示，忽略 VScode 产生的莫名其妙的错误提示，这是因为 VScode 找不到一些内核模块需要的头文件的路径，但是在编译的时候 GCC 会帮我们找到这些头文件，因此上面也可以成功编译，我认为可能是因为 VScode 只能找到用户态下的 C/C++头文件而找不到内核态下需要的一些头文件



- 然后我们装载这个模块并查看是否装载成功，发现装载成功了

```
randomstar@ubuntu:~$ cd ./Desktop/lab1
randomstar@ubuntu:~/Desktop/lab1$ ls -l
total 40
-rw-r--r-- 1 randomstar randomstar  178 Oct 16 23:12 Makefile
-rw-r--r-- 1 randomstar randomstar   43 Oct 17 01:48 modules.order
-rw-r--r-- 1 randomstar randomstar    0 Oct 16 23:14 Module.symvers
-rw-r--r-- 1 randomstar randomstar 2258 Oct 17 01:48 my_module.c
-rw-r--r-- 1 randomstar randomstar 6032 Oct 17 01:48 my_module.ko
-rw-r--r-- 1 randomstar randomstar   43 Oct 17 01:48 my_module.mod
-rw-r--r-- 1 randomstar randomstar  646 Oct 17 01:48 my_module.mod.c
-rw-r--r-- 1 randomstar randomstar 2800 Oct 17 01:48 my_module.mod.o
-rw-r--r-- 1 randomstar randomstar 4128 Oct 17 01:48 my_module.o
randomstar@ubuntu:~/Desktop/lab1$ sudo insmod my_module.ko
[sudo] password for randomstar:
randomstar@ubuntu:~/Desktop/lab1$ lsmod
Module                  Size  Used by
my_module              16384  0
helloworld             16384  0
rfcomm                 81920  4
intel_rapl_msr         20480  0
bnep                   24576  2
intel_rapl_common      24576  1 intel_rapl_msr
crct10dif_pclmul       16384  1
```

- 可以在/var/log/kern.log 文件中查看内核模块的输出内容(截图不全，只有一部分内容)

```
Oct 17 02:27:47 ubuntu kernel: [  217.599465] my_module: loading out-of-tree module taints kernel.
Oct 17 02:27:47 ubuntu kernel: [  217.599506] my_module: module verification failed: signature and/or required key missing - tainting kernel
Oct 17 02:27:47 ubuntu kernel: [  217.600477]  my_module: p_name        p_id    p_parent        p_state
Oct 17 02:27:47 ubuntu kernel: [  217.600479]  my_module: systemd       1       0       1
Oct 17 02:27:47 ubuntu kernel: [  217.600480]  my_module: kthreadd      2       0       1
Oct 17 02:27:47 ubuntu kernel: [  217.600482]  my_module: rcu_gp        3       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600483]  my_module: rcu_par_gp    4       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600484]  my_module: kworker/0:0   5       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600485]  my_module: kworker/0:0H  6       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600486]  my_module: kworker/0:1   7       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600487]  my_module: kworker/u256:0        8       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600488]  my_module: mm_percpu_wq  9       2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600490]  my_module: ksoftirqd/0   10      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600491]  my_module: rcu_sched     11      2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600492]  my_module: migration/0   12      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600493]  my_module: idle_inject/0 13      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600494]  my_module: cpuhp/0       14      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600495]  my_module: cpuhp/1       15      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600496]  my_module: idle_inject/1 16      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600497]  my_module: migration/1   17      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600498]  my_module: ksoftirqd/1   18      2       1
Oct 17 02:27:47 ubuntu kernel: [  217.600499]  my_module: kworker/1:0   19      2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600501]  my_module: kworker/1:0H  20      2       1026
Oct 17 02:27:47 ubuntu kernel: [  217.600502]  my_module: kdevtmpfs     21      2       1
```

```
Oct 17 02:27:47 ubuntu kernel: [  217.600948]  my_module: TOTAL NUMBER: 373
Oct 17 02:27:47 ubuntu kernel: [  217.600949]  my_module: TASK_RUNNING: 3
Oct 17 02:27:47 ubuntu kernel: [  217.600949]  my_module: TASK_INTERRUPTIBLE: 257
Oct 17 02:27:47 ubuntu kernel: [  217.600950]  my_module: TASK_UNINTERRUPTIBLE: 0
Oct 17 02:27:47 ubuntu kernel: [  217.600950]  my_module: TASK_STOPEED: 0
Oct 17 02:27:47 ubuntu kernel: [  217.600951]  my_module: EXIT_DEAD: 0
Oct 17 02:27:47 ubuntu kernel: [  217.600951]  my_module: EXIT_ZOMBIE: 0
Oct 17 02:27:47 ubuntu kernel: [  217.600952]  my_module: TASK_DEAD: 0
Oct 17 02:27:47 ubuntu kernel: [  217.600952]  my_module: TASK_IDLE: 113
Oct 17 02:27:47 ubuntu kernel: [  217.600953]  my_module: OTHERS: 0
```

- 之后我们需要编写一个用户态的程序 user_mode.cpp，来格式化地输出这个文件中的内容，为了方便用户态程序的独写，我在输出的结果开始处加了一些标志字符，用来简化读写，因为输出的内容较多因此只选择一部分来截图

```
randomstar@ubuntu:~/Desktop/lab1$ g++ user_mode.cpp
randomstar@ubuntu:~/Desktop/lab1$ ./a.out
loading out-of-tree module taints kernel.
module verification failed: signature and/or required key missing - tainting kernel
 p_name p_id    p_parent        p_state
 systemd        1       0       1
 kthreadd       2       0       1
 rcu_gp 3       2       1026
 rcu_par_gp     4       2       1026
 kworker/0:0    5       2       1026
 kworker/0:0H   6       2       1026
 kworker/0:1    7       2       1026
 kworker/u256:0 8       2       1026
 mm_percpu_wq   9       2       1026
 ksoftirqd/0    10      2       1
 rcu_sched      11      2       1026
 migration/0    12      2       1
 idle_inject/0  13      2       1
 cpuhp/0        14      2       1
 cpuhp/1        15      2       1
 idle_inject/1  16      2       1
 migration/1    17      2       1
 ksoftirqd/1    18      2       1
 kworker/1:0    19      2       1026
 kworker/1:0H   20      2       1026
 kdevtmpfs      21      2       1
 netns  22      2       1026
 rcu_tasks_kthre        23      2       1
 kauditd        24      2       1
 kworker/0:2    25      2       1026
 khungtaskd     26      2       1
 oom_reaper     27      2       1
 writeback      28      2       1026
```

```
loop8  2910    2       1
code   3036    1513    1
code   3038    3036    1
code   3039    3036    1
code   3073    3061    1
code   3081    3038    1
code   3093    3061    1
bash   3124    3093    1
code   3132    3093    1
code   3147    3093    1
code   3160    3061    1
cpptools       3218    3132    1
cpptools-srv   3252    3219    1
deja-dup-monito        3298    1705    1
gnome-terminal-        3767    1513    1
bash   3777    3767    1
sudo   3793    3777    1
insmod 3794    3793    0
TOTAL NUMBER: 373
TASK_RUNNING: 3
TASK_INTERRUPTIBLE: 257
TASK_UNINTERRUPTIBLE: 0
TASK_STOPEED: 0
EXIT_DEAD: 0
EXIT_ZOMBIE: 0
TASK_DEAD: 0
TASK_IDLE: 113
OTHERS: 0
```

2.4 实验结果分析

- 本实验完成了所有的实验要求，通过添加内核模块遍历了操作系统中的所有进程并进行了信息的输出和统计，总的来说是一个难度比较低的实验
- 具体的在实验中遇到的坑和经验总结可以看最后一部分内容

第三部分：实验源代码

3.1 同步互斥实验

```
1.  #include <cstdio>
2.  #include <cstdlib>
3.  #include <iostream>
4.  #include <pthread.h>
5.  #include <unistd.h>
6.  #include <string>
7.  #include <queue>
8.
9.
10. using namespace std;
11.
```

```
// 四个方向北西南东分别用 0123 来表示
#define north 0
#define west 1
#define south 2
#define east 3
// 最大的小车数量
#define MAX 50

// 小车信息的结构体，包含方向和编号
typedef struct pCar * Car;
struct pCar{
    int car_direction;//车辆来自的方向
    int car_id;//车辆的编号
};

// 存储方向的字符串数组 便于打印
const string direction_in_string[4] = { "north", "west", "south", "east"};
// 表示小车的队列，存储小车的编号
std::queue<int> waiting_car[4];

// 用于每个方向中车辆队列发送信号的条件变量
pthread_cond_t queue_cond[4];
// 用于给下一个通过路口的方向的小车发送信号量
pthread_cond_t next_car[4];
// 各个方向的互斥锁
pthread_mutex_t direction_mutex[4];
// 四个小车队列的访问互斥锁
pthread_mutex_t queue_mutex[4];
// 通过马路时的互斥锁
pthread_mutex_t cross;
// 唤醒死锁监测调度线程的条件变量
pthread_cond_t dead_lock;
// 死锁访问的互斥变量
pthread_mutex_t dead_lock_mutex;
// 一个死锁检测线程
pthread_t check_dead_lock;
// 表示本方向前面是否有车辆（有车辆为 1，否则为 0）
int is_arrived[4];
// 表示每个方向是否有车处于等待状态
int is_waiting[4];

void* car_thread(void* car_pointer);
void* dead_lock_schedule(void* );
void pthread_init();
```

```
56. void pthread_destroy();
57. 
58. int main(){
59.     int number = 1;
60.     string input_car;
61.     // 若干个车辆的线程，每辆车单独创建一个线程
62.     pthread_t car[MAX];
63.     // 初始化一些定义的互斥锁，条件变量和数组
64.     pthread_init();
65.     //创建死锁检测线程
66.     pthread_create(&check_dead_lock, NULL, dead_lock_schedule, NULL);
67.     // 读取输入的小车序列，并进行初始化
68.     cin >> input_car;
69.     for(int i = 0; i < input_car.size(); i++) {
70.         // 创建一辆新的小车，并初始化它的方向和编号
71.         Car single_car = new pCar;
72.         if(input_car[i] == 'n') {
73.             single_car->car_direction = north;
74.         } else if(input_car[i] == 's') {
75.             single_car->car_direction = south;
76.         } else if(input_car[i] == 'e') {
77.             single_car->car_direction = east;
78.         } else if(input_car[i] == 'w') {
79.             single_car->car_direction = west;
80.         }
81.         // 编号每次增加 1
82.         single_car->car_id = number++;
83.         // 将小车放入所在方向的队列中
84.         waiting_car[single_car->car_direction].push(single_car->car_id);
85.         // 为每辆车单独创建一个线程，并启动
86.         pthread_create(&car[i], NULL, car_thread, single_car);
87.     }
88.     for(int i = 0; i < input_car.size(); i++) {
89.         // 等待所有的小车线程结束
90.         pthread_join(car[i],NULL);
91.     }
92.     // 销毁所有锁
93.     pthread_destroy();
94.     return 0;
95. }
96. 
97. 
98. /**
99.  * 每辆车的单独线程，包含车辆进入路口之后通过的整个过程
```

```
 * @param car_pointer 小车的地址
 * @return
 */
void* car_thread(void* car_pointer) {
    // 获取当前线程小车的基本信息
    Car i=(Car)car_pointer;
    int direct = i->car_direction, id = i->car_id;
    // 如果本车辆不在等待队列最前方则等待
    while (waiting_car[direct].front() != id) {
        // 陷入死循环，一直等待
    }
    // 车进入路口，打印车辆信息

    // sleep(1);
    // 给当前方向的路口上锁
    pthread_mutex_lock(&direction_mutex[direct]);
    // 等待本方向正在过马路的车辆发信号，让本车通过马路
    while (is_arrived[direct]) {
        pthread_cond_wait(&queue_cond[direct], &direction_mutex[direct]);
    }
    // 打印车辆信息
    printf("Car %d from %s arrives at crossing.\n", id, direction_in_string
[direct].c_str());
    // 表示本方向有车已经到达路口
    is_arrived[direct] = 1;
    // 获取本方向的左右方向对应的编号
    int right = (direct + 1) % 4, left = (direct + 3) % 4;
    // 先观察右边的车是否也要过马路
    if (!waiting_car[right].empty()) {
        // 设置本方向车辆正在等待过马路
        is_waiting[direct] = 1;

        /** 已弃用的旧代码
         * // 小车发现四个路口都有车，出现了死锁，此时需要死锁线程进行调度
         * if(is_waiting[0] && is_waiting[1] && is_waiting[2] && is_waiting
[3]) {
            // 打印死锁信息
            printf("DEADLOCK: car jam detected.\n");
            pthread_mutex_lock(&dead_lock_mutex);
            // 发信号给死锁检测线程，让该线程进行死锁的调度
            pthread_cond_signal(&dead_lock);
            pthread_mutex_unlock(&dead_lock_mutex);
            }
         */
```

```
142.
143.         // 右边有车，需要等待右边的车辆通过后给自己发信号
144.         pthread_cond_wait(&next_car[direct], &direction_mutex[direct]);
145.     }
146.     // 轮到本车过马路，将正在准备过马路的标志设置为 0，然后通过马路
147.     is_waiting[direct] = 0;
148.     pthread_mutex_lock(&cross);
149.     // 打印过马路的信息
150.     printf("Car %d from %s leaving crossing.\n", id, direction_in_string[di
    rect].c_str());
151.     // 过马路时间设置为 1s，防止程序运行过快
152.     sleep(1);
153.     pthread_mutex_unlock(&cross);
154.     // 给本方向上锁
155.     pthread_mutex_lock(&direction_mutex[left]);
156.     // 发出信号给左边的车辆通行
157.     pthread_cond_signal(&next_car[left]);
158.     pthread_mutex_unlock(&direction_mutex[left]);
159.     // 给本方向的队列上锁
160.     pthread_mutex_lock(&queue_mutex[direct]);
161.     // 从队列中将第一辆车 pop 出
162.     waiting_car[direct].pop();
163.     // 此时没有到路口的车
164.     is_arrived[direct]=0;
165.     // 然后给后一辆车发送信号，让其通行
166.     pthread_cond_signal(&queue_cond[direct]);
167.     pthread_mutex_unlock(&queue_mutex[direct]);
168.     pthread_mutex_unlock(&direction_mutex[direct]);
169.     // 线程结束，退出线程
170.     pthread_exit(NULL);
171. }
172.
173. /**
174.  * 死锁发生时候的调度线程，向北方发送一个信号量让北方的车先通行
175.  * @return 没有返回值
176.  */
177. void* dead_lock_schedule(void* ) {
178.     while(1) {
179.         // 检测到路口产生了死锁
180.         if (is_waiting[0] && is_waiting[1] && is_waiting[2] && is_waiting[3
    ]) {
181.             pthread_mutex_lock(&cross);
182.             // 出现死锁的时候被唤醒
183.             // pthread_cond_wait(&dead_lock, &cross);
```

```
184.             // 打印死锁的信息
185.             printf("DEADLOCK: car jam detected.Deadlock schedule wake up an
    d signal north.\n");
186.             pthread_mutex_lock(&direction_mutex[north]);
187.             // 给北方发送信号，让北方先走，之后按照北东南西的顺序进行
188.             pthread_cond_signal(&next_car[north]);
189.             pthread_mutex_unlock(&direction_mutex[north]);
190.             pthread_mutex_unlock(&cross);
191.             // 每次检测完休息 1s，防止检测过快
192.             sleep(1);
193.         }
194.     }
195. }
196. 
197. /**
198.  * 初始化所有定义的互斥锁和条件变量以及数组
199.  */
200. void pthread_init() {
201.     // 初始化定义的一系列互斥锁和条件变量
202.     for(int i = 0;i < 4; i++) {
203.         pthread_cond_init(&queue_cond[i], NULL);
204.         pthread_cond_init(&next_car[i], NULL);
205.         pthread_mutex_init(&direction_mutex[i], NULL);
206.         pthread_mutex_init(&queue_mutex[i], NULL);
207.         // 将数组按照定义初始化为 0
208.         is_arrived[i]=0;
209.         is_waiting[i]=0;
210.     }
211.     pthread_cond_init(&dead_lock, NULL);
212.     pthread_mutex_init(&dead_lock_mutex, NULL);
213.     pthread_mutex_init(&cross,NULL);
214. 
215. }
216. 
217. /**
218.  * 销毁所有使用过的互斥锁和条件变量以及数组
219.  */
220. void pthread_destroy() {
221.     // 销毁之前定义的所有条件变量和互斥锁
222.     for(int i = 0;i < 4; i++) {
223.         pthread_cond_destroy(&queue_cond[i]);
224.         pthread_cond_destroy(&next_car[i]);
225.         pthread_mutex_destroy(&direction_mutex[i]);
226.         pthread_mutex_destroy(&queue_mutex[i]);
```

```
227.     }
228.     pthread_cond_destroy(&dead_lock);
229.     pthread_mutex_destroy(&dead_lock_mutex);
230.     pthread_mutex_destroy(&cross);
231. }
```

2 内核模块代码

```
1.  #include <linux/init.h>
2.  #include <linux/module.h>
3.  #include <linux/kernel.h>
4.  #include <linux/sched.h>
5.  #include <linux/init_task.h>
6.
7.
8.  // module initial function
9.  // process traversal and statistic
10. int init_module(void)
11. {
12.     int num = 0;
13.     int count[10] = {0};
14.     struct task_struct *p = NULL;
15.     p = &init_task;
16.     printk(" my_module: p_name\tp_id\tp_parent\tp_state\t\n");
17.     for_each_process (p) {
18.         if (p != NULL) {
19.             printk(" my_module: %s\t%d\t%d\t%ld\n", p->comm, p->pid, p->pare
    nt->pid, p->state);
20.             num++;
21.             // TASK_RUNNING
22.             if (p->state == 0) {
23.                 count[0]++;
24.             } else if (p->state == 1) {
25.                 // TASK_INTERRUPTIBLE
26.                 count[1]++;
27.             } else if (p->state == 2) {
28.                 // TASK_UNINTERRUPTIBLE
29.                 count[2]++;
30.             } else if (p->state == 4) {
31.                 // TASK_STOPEED
```

```
                count[3]++;
            } else if (p->state == 16) {
                //EXIT_DEAD
                count[4]++;
            } else if (p->state == 32) {
                //EXIT_ZOMBIE
                count[5]++;
            } else if (p->state == 128) {
                //TASK_DEAD
                count[6]++;
            } else if (p->state == 1026) {
                // TASK_IDLE
                count[7]++;
            } else {
                // other process
                count[8]++;
            }
            //total number of the process
        }
    }
    // print the infomation
    printk(" my_module: TOTAL NUMBER: %d", num);
    printk(" my_module: TASK_RUNNING: %d\n", count[0]);
    printk(" my_module: TASK_INTERRUPTIBLE: %d\n", count[1]);
    printk(" my_module: TASK_UNINTERRUPTIBLE: %d\n", count[2]);
    printk(" my_module: TASK_STOPEED: %d\n", count[3]);
    printk(" my_module: EXIT_DEAD: %d\n", count[4]);
    printk(" my_module: EXIT_ZOMBIE: %d\n", count[5]);
    printk(" my_module: TASK_DEAD: %d\n", count[6]);
    printk(" my_module: TASK_IDLE: %d\n", count[7]);
    printk(" my_module: OTHERS: %d\n", count[8]);
    return 0;
}

// when the module is cleanup
void cleanup_module(void)
{
    printk("See you next time!!\n");
}

// the license of the module
MODULE_LICENSE("GPL");
```

3.内核模块的用户态程序代码

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    ifstream fin("/var/log/kern.log", ios::in);
    if(fin.fail()) {
        cout << "Fail to open the log file!!!" << endl;
        return 0;
    }
    string s;
    while(getline(fin, s)) {
        int pos = s.find("my_module:");
        if(pos != -1) {
            cout << s.substr(57) << endl;
        }
    }
    fin.close();
    return 0;
}
```

四、讨论、心得（20 分）

本次实验是操作系统课程的第一个大型试验，个人认为本实验的难度比较大，两个任务中碰到的问题和解决方法分别如下：

在同步互斥编程中碰到的问题有：

- 我尝试用 C++来完成代码，但是输入输出的时候发现，cout 是线程不安全的，因此使用 cout 输出会出现同一行有好几条不同线程的输出结果这样的情况，因此后面我改成了用 printf 输出
- 多线程程序的代码 debug 非常困难，因为我无法通过输出的内容顺序知道究竟各个线程的运行情况，因为输出的内容比较混乱，因此我采用的方法是一个线程中小车通过路口后线程 sleep 1s 代表通过马路所需要的时间，这样一来结果就比较清楚了，不容易互相冲突
- 死锁检测线程需要单独设置，并且要设置一定的机制防止一次死锁出现的时候线程多次检测到同一个死锁，具体的办法就是检测到一次死锁之后进行调度，然后休息 1s 之后继续检测，这其中的原因是死锁调度之后四个方向的小车需要按照一定顺序通过十字路口，而这一过程不可能在 1s 内完成，因为上面提到小车通过马路的时间设定成了 1s，即用线程 sleep 1s 来模拟

在内核模块中碰到的问题有：

- 不会编写 Makefile，在网上的资料中学习了半天还是云里雾里，最后按照《边干边学：Linux 内核指导》教材中提供的 Makefile 例子依样画葫芦写出了正确的 Makefile
- Makefile 编译内核模块的时候会报出一些莫名其妙的错误，最后一一排查才可以通过编译。
- Linux 环境下的 VScode 对内核模块的编写非常不友好，很多头文件和声明都会报“异常”，但事实上是可以通过内核模块的编译的，我推测是因为 VScode 主要支持对用户态程序的编写，因此碰到内核模块编写的时候，VScode 还是把它当作用户态程序在分析其代码上下文了，因此产生了一系列不应该有的异常提示。

本次实验我认为难度较大，也很有收获，我掌握了基本的 POSIX 线程库编程和 Linux 下代码编写、调试、Makefile 编译等基本方法，花费了不少时间的同时感觉收获颇丰。