# 浙江大学实验报告

课程名称： 操作系统 实验类型： 综合

实验项目名称： 同步互斥和Linux内核模块

学生姓名： 专业： 计算机 学号：

电子邮件地址： [3160103176@zju.edu.cn](mailto:3160103176@zju.edu.cn) 手机：

实验地点 玉泉曹光彪2期503 实验日期： 2018-11-20

# 一、实验目的和要求

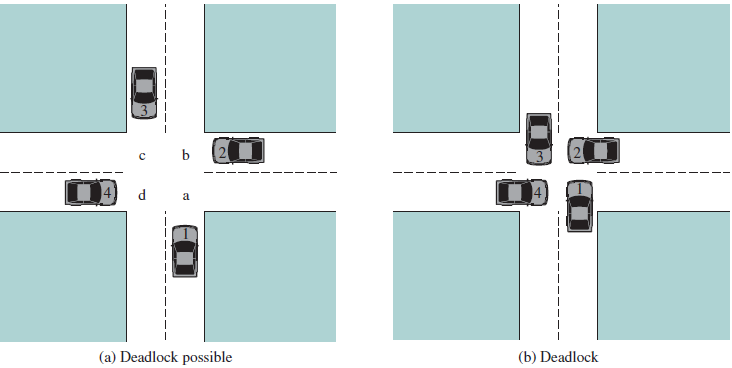
* 学习使用Linux的系统调用和pthread线程库编写程序。
* 充分理解对共享变量的访问需要原子操作。
* 进一步理解、掌握操作系统进程和线程概念，进程或线程的同步与互斥。
* 学习编写多线程程序，掌握解决多线程的同步与互斥问题。
* 学习Linux模块的实现机理，掌握如何编写Linux模块。
* 通过对Linux系统中进程的遍历，进一步理解操作系统进程概念和进程结构。

# 实验内容和原理

**多线程编程：**

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。**

**编写Linux内核模块：**

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、主要仪器设备

* Host Machine
  + Windows 10 家庭版
  + Intel Core i5
  + 12GB RAM
* Guest Machine
  + Ubuntu 16.04
  + Kernel 4.19.5
  + 4GB RAM

# 四、操作方法与实验步骤

## A. 多线程编程

定义方向变量direction：

typedef enum

{

north = 0,

east = 1,

south = 2,

west = 3

} direction;

每一个方向的车辆的信息都将存在一个结构中：

struct queue

{

int num; //the number of the car in this direction

direction dir; //the direction

int carID[MAXNUM]; //the ID of the car in this direction

int front; //the front pointer of the queue

int rear; //the rear pointer of the queue

};

其中4个不同的方向的车由其中的direction成员作区分。本程序一共4个这样的队列：

struct queue dirQueue[4];

总共用到了2种互斥锁，一个叫做方向锁，另一个叫做资源锁，方向锁的作用是：每一次该方向只能有一辆车在通过；资源锁顾名思义：对于十字路口的4个资源是互斥的，每一个资源只能被一辆车占用。

pthread\_mutex\_t mutex\_source[4];

pthread\_mutex\_t mutex\_dir[4];

其中的4表示4个方向。

总共用到了一种条件变量，来表示右边的车是否可以让左边的车先开：

pthread\_cond\_t first\_dir[4];

它们的初始化我都写成了单独的函数，在主线程一开始的时候将其初始化。

主函数的作用为：将用户的输入全部读入，并且将相应的结构体创建出来，当所有结构体创建完毕后，再创建对应的线程。

创建线程并且等待进程退出：

for(int i=0;i<carNum;i++)

{

pthread\_create(&car\_thread[i], NULL, car\_thread\_handle, car[i]);

}

//wait for the threads to quit

for (int i = 0; i < carNum; i++)

{

pthread\_join(car\_thread[i], NULL);

}

为了不让代码太过冗长，我没有采用每个方向都写一个线程函数，每一个方向事实上是等价的，只要传入的参数不一致，那么就代表不同的线程。

处理线程的函数为：

void \*car\_thread\_handle(void \*pParam)

程序的关键是这个函数。在这个函数中：

* 每个方向都有一个队列，如果当前线程不是该队列中的第一个元素，那么将会被阻止在“路口”：

while (dirQueue[car->dir].carID[dirQueue[car->dir].front] != car->id)

;

* 如果是第一个元素，将进入路口，获取方向锁，并打印车辆到达路口的信息：

pthread\_mutex\_lock(&mutex\_dir[car->dir]);

usleep(2000);

printf("Car %d from %s arrives at crossing\n", car->id, dir\_s);

* 到达十字路口后，我不能立刻占用我眼前的资源，它需要判断我占用眼前的资源后是否会有deadlock。如果已经有3辆车等在路口，那么就不能占用眼前的资源，而是示意左边的车辆先走。并且自己需要等在原地。

if (wait\_num == 3) // Now The car want to occupy the source, but if there will be a deadlock, it can not occupy

{

printf("DEADLOCK: car jam detected, signaling %s to go\n", dir\_left\_s);

usleep(4000);

pthread\_cond\_signal(&first\_dir[(car->dir + 1) % 4]);

}

while (wait\_num == 3);

* 如果不会发生交通堵塞，那么我将希望能占用该资源，眼前的资源具有资源锁，模拟车辆开上去的时间，可以设置一个微小的delay：

wait\_num++;

pthread\_mutex\_lock(&mutex\_source[car->dir]);

usleep(2000);

* 占用了眼前的资源后，需要判断右边有没有车，判断方式为右边的队列是否非空。如果右边有车，我们需要等待让右边的车先走。

if (dirQueue[right\_dir].num) //There is at least a car at right

{

pthread\_cond\_wait(&first\_dir[car->dir], &mutex\_dir[car->dir]);

}

* 如果收到了右边车发来的先走的信号，那么可以准备占用下一个资源了，占用2个资源后也就意味这这辆车将离开十字路口。需要将队列元素减1，将等待车辆数减1，并打印信息：

pthread\_mutex\_lock(&mutex\_source[(car->dir - 1) % 4]);

dirQueue[car->dir].num--;

usleep(2000);

wait\_num--;

printf("Car %d from %s leaving crossing\n", car->id, dir\_s);

* 准备解锁两个资源，先把占用的第一个资源解放，再给左边的车辆发通行信号，同时让自己的方向的队列元素指向下一位：

pthread\_mutex\_unlock(&mutex\_source[car->dir]);

pthread\_cond\_signal(&first\_dir[(car->dir + 1) % 4]);

usleep(4000);

dirQueue[car->dir].front++;

* 最后解除第二个资源锁和方向锁：

pthread\_mutex\_unlock(&mutex\_source[(car->dir - 1) % 4]);

usleep(2000);

pthread\_mutex\_unlock(&mutex\_dir[car->dir]);

处理完成，准备退出该线程。

## 内核模块程序编译

变量定义：

int num\_task = 0; //number of all task

int num\_task\_running = 0; //number of running task

int num\_task\_interruptible = 0; //number of interruptible task

int num\_task\_uninterruptible = 0; //number of uninterruptible task

int num\_task\_zombie = 0; //number of zombie task

int num\_task\_stopped = 0; //number of stoppend task

int num\_task\_traced = 0; //number of traced task

int num\_task\_dead = 0; //number of dead task

int num\_task\_unknown = 0; //number of unknown task

int task\_state; //the running state of the task

int task\_exit\_state; //the exit state of the task

struct task\_struct \*p = &init\_task; //the pointer to travel all the process

主要的循环部分采用：

for (p = &init\_task; (p = next\_task(p)) != &init\_task;)

遍历每一个进程，使用prink函数打印出相应的信息。比如打印出总进程数的写法如下：

printk("MicroJerry@Total number = %d\n", num\_task);

前面的信息表示这是我打印出来的，可以和系统作一些区分，便于后面用户态程序的编写。

对于删除模块，使用cleanup\_module函数：

void cleanup\_module(void)

{

////print the info :MODULE CLEANUP

printk("MicroJerry@MODULE CLEANUP\n");

}

用户态程序主要是从/var/log/ kern.log中读出相应的信息并打印到命令行上面，我将它写在testPrint.c中。

这里需要去掉一些系统自带的名字，否则打印出来的消息显得过于冗长。

Makefile文件为：

TARGET = os\_project1\_2

KDIR = /lib/modules/$(shell uname -r)/build

PWD = $(shell pwd)

obj-m += $(TARGET).o

all:

    make -C $(KDIR) M=$(PWD) modules

clean:

    make -C $(KDIR) M=$(PWD) clean

程序实现方式，打开kern.log文件，查找有没有MicroJerry@字样，如果有，说明是我们打印的消息。将消息简化后输出：

const char\* mystring = "MicroJerry@";

if((fp=fopen("/var/log/kern.log","r"))==NULL)return -1;

while(1)

{

if(fgets(info,1024,fp)==NULL)break;

//judge if the line includes the string :"MicroJerry@"

if(is\_string\_inline(info,mystring))

{

printf\_info(info);

}

}

# 实验结果与分析

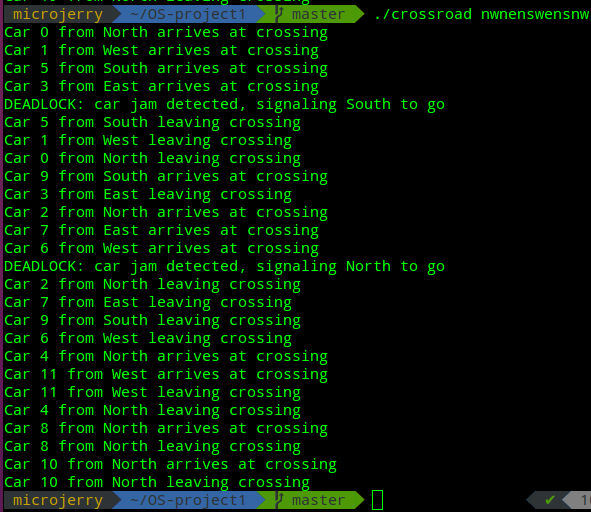
## A. 多线程编程

编译：



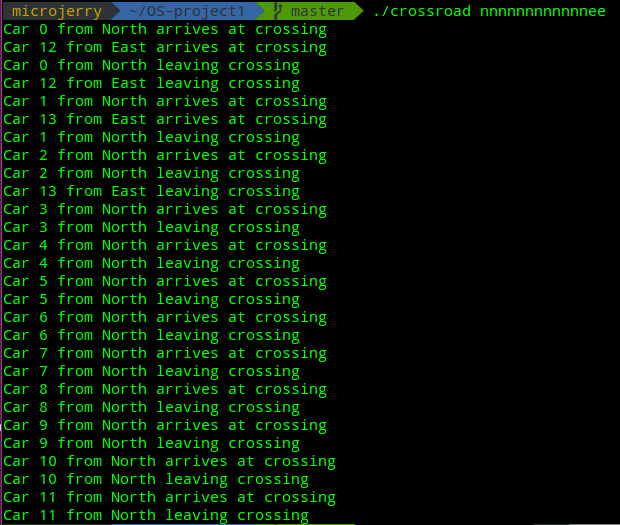
运行：

随机输入比较多的车辆：



可以看到发生了deadlock并及时处理，最后成功运行完所有的线程后退出。

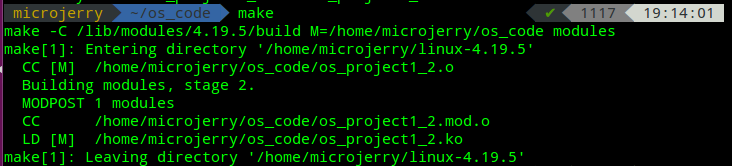
多个同方向车辆，看是否有饥饿情况发生：



可以看到东边的车辆并没有被饥饿，而是正常运行。

## 内核模块程序编译

Makefile：



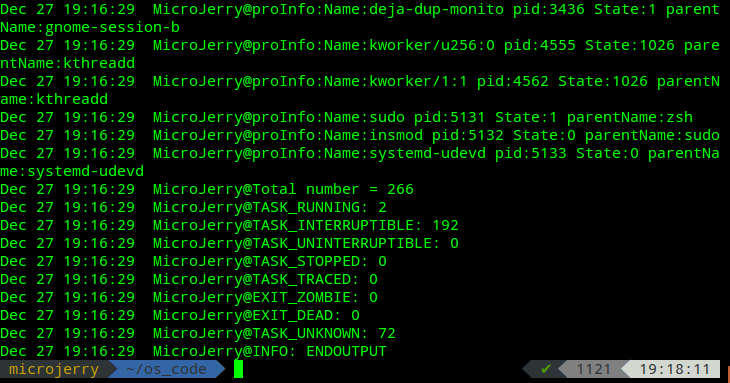
安装模块：



编译testPrint.c：



运行后结果为：



显示时间和打印的内容。显示正确