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

学生姓名：　杨樾人　　学号：　3160104875

电子邮件地址：　yangyueren@outlook.com

实验日期：　2018　年　11　月　28　日

一、　实验环境

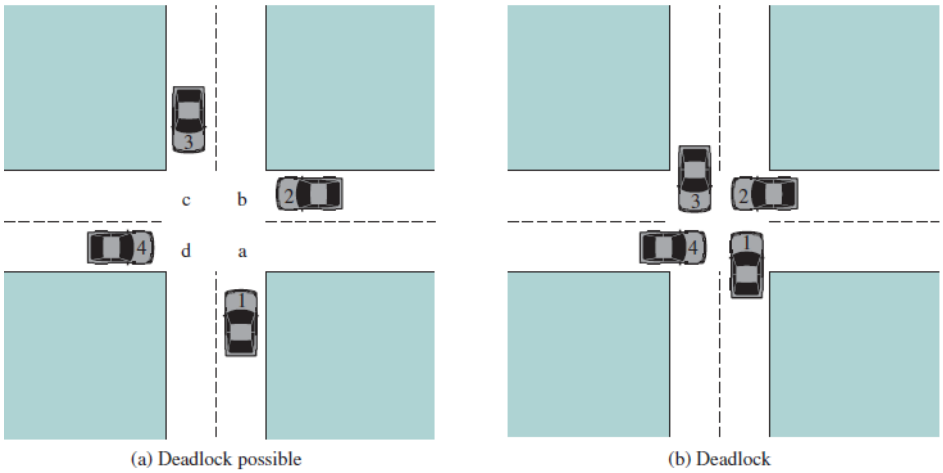
Ubuntu 16.04.3LTS(GNU/Linux 4.4.0-93-generic x86_64)，mac 下虚拟机

二、　实验内容和结果及分析

- 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。

- 设计文档

1. 功能模块

本程序分为初始化互斥锁和条件变量模块，处理字符模块，创建线程模块，线程调度模块和结束模块。

初始化模块：初始化互斥锁和条件变量。

处理字符模块中：主要处理输入的“n/w/s/e“四种字符，按输入顺序生成四个方向的汽车线程并通过锁和条件变量实现汽车通过路口的调度。

创建线程模块：根据字符创建线程 。

线程调度模块：线程的具体实现，避免饥饿和死锁。

结束模块：销毁锁和条件变量。

2. 程序流程概述：

1. 初始化锁和条件变量；
2. 处理字符，创建线程
3. 线程调度，通过路口
4. 线程结束，销毁锁和条件变量

3. 概要设计说明：

- **编写目的：**

  编写实现十字路口车辆调度的程序，并防止汽车通过十字路口时产生死锁和饥饿。

参考资料：

1. Linux 线程创建和同步的编程参考资料：

   操作系统原理教材“Operating System Concepts”的第四章和第六章

   边干边学 Linux 内核指导

2. Linux 的 Iphread 库中的 API 文档说明

包括 Linux 操作系统提供了 pthread 线程库，它是符合 POSIX 标准的函数库。线程控制方面的函数定义在 pthread.h 文件中,信号量控制方面的函数定义在 semaphore.h 文件中。

1) 线程控制方面的函数有：pthread_attr_init、pthread_create、pthread_join、pthread_exit
2) 互斥锁机制函数：pthread_mutex_init、 pthread_mutex_lock、pthread_mutex_unlock、 pthread_mutex_destroy
3) 条件变量函数：pthread_cond_init、int pthread_cond_signal、 int pthread_cond_wait、、int pthread_cond_destroy

- **输入输出：**

输入：用户通过命令行参数的形式输入来自各个方向的车，以 nswe 分别代表北南西东。

输出：

输出每辆车到达路口：car %d from %s arrives at crossing

输出该辆车离开路口：car %d from %s leaving crossing

遇到死锁，调度某一方向的车优先通过：DEADLOCK: car jam detected, signalling %s to go

- 模块的具体实现：

使用变量：

pthread_mutex_t crossingLock，通过路口的锁，每次只能一个线程通过路口。

pthread_mutex_t deadlock，检测死锁，在对路口锁加锁后，要检测是否产生了死锁。

pthread_mutex_t directionLock[4]，到达四个方向的路口的锁。

pthread_cond_t nextDirectionCond[4], followedCarCond[4]，条件变量，给同方向的下一辆车以及左边方向的车发信号。

int isWait[4] = {0}，判断该方向是否有等待。

int counter[4][maxNumOfCar] = {0}，队列，对每个方向的车进行依次排序。

初始化锁：调用 pthread_cond_init 和 pthread_mutex_init 函数初始化条件变量和互斥锁。

处理字符串和线程：

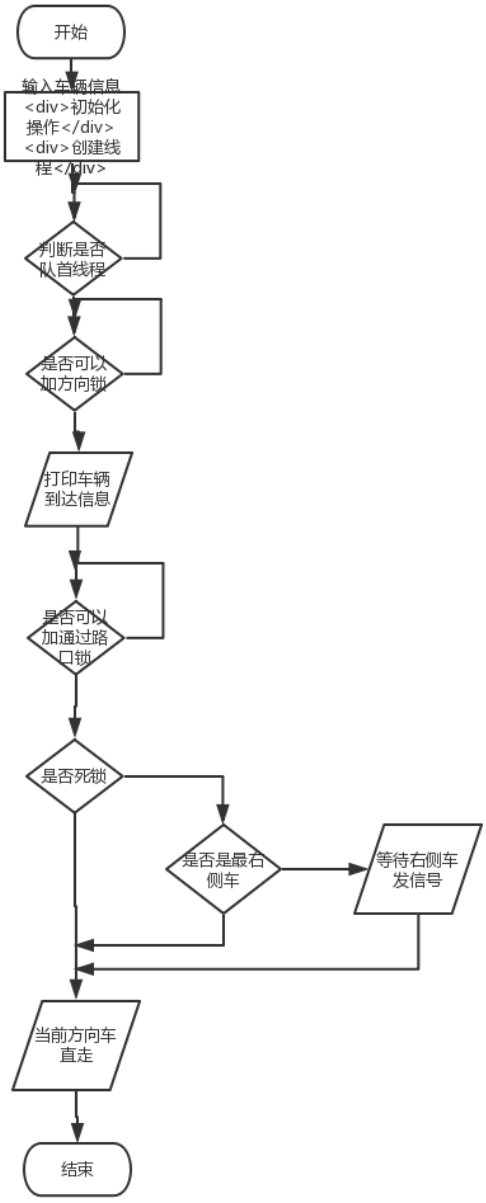
通过 carDirectionMap 来记录字符对应的方向，并将其转为对应的 int。

每辆车用一个 struct car 来存储，id 代表车的序号，direction 代表车的方向

调用 pthread_create 函数，对每辆车生成一个线程

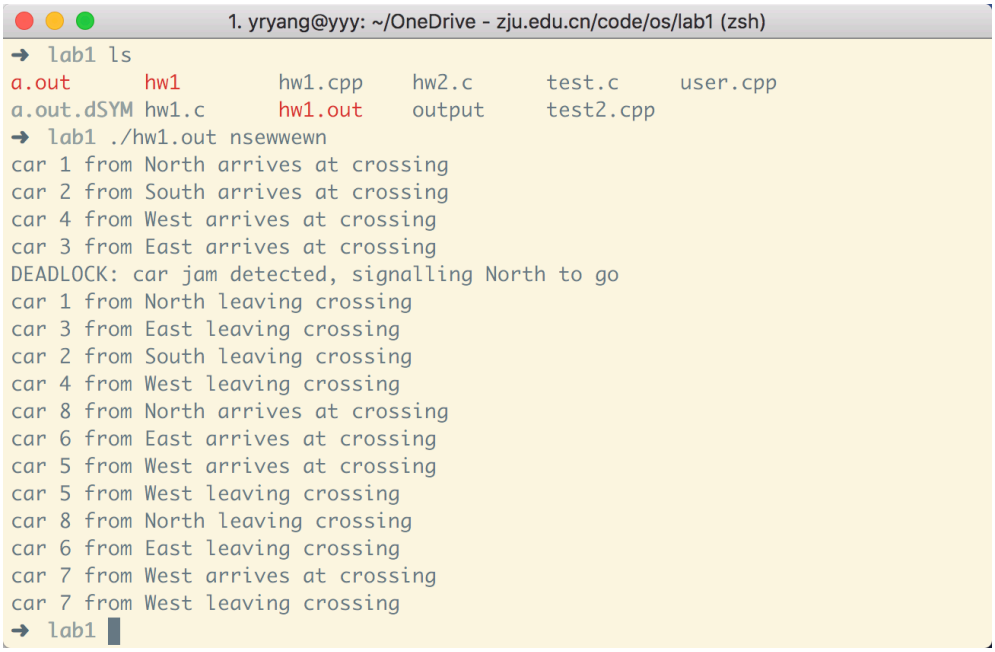
主函数调用 join 函数，等待所有线程结束。

- 具体流程图：

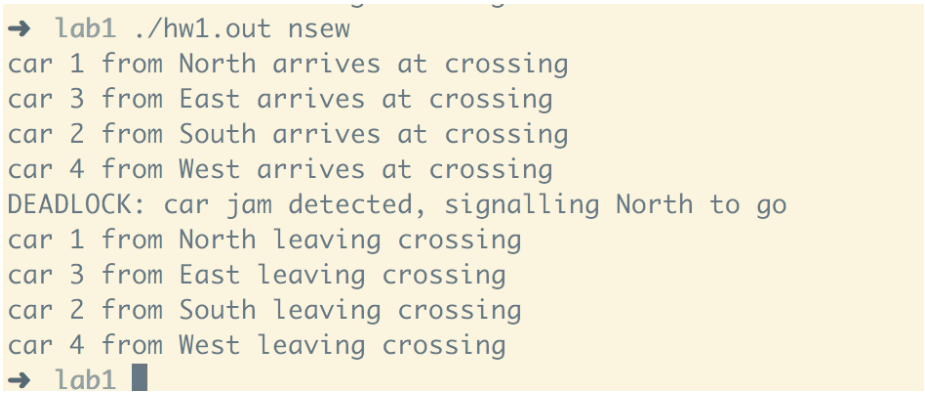


- 程序运行结果截图

nsewwewn



nsew



结果分析

从运行结果来看，程序实现了实验的要求。

在实验要求的 nswewewn 调度中，会产生一次死锁，程序在北 南 西 东四个方向的车同时到达路口后，会检测到死锁，并且让北方向的车开走。

在对之后到达对车进行调度时，不会再出现死锁，每次都让最右侧的车先走，然

后是下一个方向路口的车。同方向的下一辆车会在其他方向都走一辆车之后才会走，避免了饥饿。

4. 源程序（见附录）

三、 讨论、心得（20 分）

遇到的问题：

1. 没有办法检测到死锁。

我一开始认为是线程创建后立刻执行，四个方向的车不会同时到达路口，所以一直不会存在死锁现象，对于样例，也没有办法检测到死锁现象，这个问题一直困扰着我。

解决方法：给每个路口增加了 isWait[4]变量和 direcitonLock 互斥锁，来标记当前路口是否有车到达且只能有一辆到达，正在等待通过路口。这样就可以检测到了死锁。

2. 超车现象，即同一方向后面的一辆车会先到达路口。

解决方法：模拟使用队列，给每个方向的车入队，当该方向的车线程对该方向的锁进行锁操作前，要先判断该线程是否是队首的车 id，如果 id 不符，则该线程不能加锁。

3. 关于条件变量的使用。

每个条件变量必须配合一把锁进行使用，如果有多个同方向的线程进行了条件变量的 wait 操作，当通过路口的车发出了 signal 信号，哪一辆车会被唤醒？

这个问题导致了超车现象的存在，为了解决这个问题，我使用了队列。也就是第二个问题里的解决方法。

4. 饿死现象

为了避免饿死现象，需要先给左边的车发信号，让其通过，然后再给同方向的下一辆车发信号。在 Linux 虚拟机上跑的时候，还是可能会存在饿死的现象，但是在 mac 上面就没有饿死现象。

解决方法：在给同方向的下一辆车发信号之前，进行 sleep 操作，让其等待 1s 中再收到信号。

体会：

1. 多线程的程序难以 debug，复现 bug 也非常难。
2. 多线程的程序对于共同变量的修改读写操作，一定要加锁，来避免各种奇怪问题的出现
3. 有时候可以用共同变量替代某些锁，比如我的队列就没有用队列锁的方式实现，而是采用了其他的实现方式。

四、 附录：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#define maxNumOfCar 1000//max car number
int carNum = 1000;
enum Direction
{
    north,
    west,
    south,
    east
};//direction
typedef struct Car *pCar;
struct Car
{
    int carid;
    int direction;
};
pthread_mutex_t crossingLock;//cross the crossing
pthread_mutex_t deadLock;//check the deadlock
pthread_mutex_t directionLock[4];//direction lock
pthread_cond_t nextDirectionCond[4], followedCarCond[4];//cond
```

```
int counter[4][maxNumOfCar] = {0};//queue
int nowCounter[4] = {0};//initialize the queue
int curCounter[4] = {0};//check the next car
int isWait[4] = {0};
int alreadyHasCarArrived[4] = {0};
char directionMap[4][10] = {"North", "West", "South", "East"};
//thread
void *CarCross(void *car)
{
    pCar nowCar = (pCar)car;
    int id = nowCar->carid + 1;//car id
    int nowDir = nowCar->direction;
    //lock the direction
    pthread_mutex_lock(&directionLock[nowDir]);
    //when the crossing is not locked and this car in the queue is the right
car
    while (alreadyHasCarArrived[nowDir] != 0 ||
counter[nowDir][curCounter[nowDir]] != nowCar->carid)
    {
        pthread_cond_wait(&followedCarCond[nowDir], &directionLock[nowDir]);
    }
    alreadyHasCarArrived[nowDir] = 1;//lock this direction
    isWait[nowDir] = 1;//other cars in this direciotn needs to wait
//this dir is waiting
    printf("car %d from %s arrives at crossing\n", nowCar->carid+1,
directionMap[nowCar->direction]); //到达路口

    pthread_mutex_lock(&crossingLock);
    //lock crossingLock
    int leftCar = (nowDir + 3) % 4;//left car
    int rightCar = (nowDir + 1) % 4;//right car

    int isDead = 1;
    pthread_mutex_lock(&deadLock);

    for (int i = 0; i < 4; i++)
    {
        if (isWait[i] == 0)
        {
            isDead = 0;
        }
    }
```

```
    pthread_mutex_unlock(&deadLock);
    if (isDead == 1)
        printf("DEADLOCK: car jam detected, signalling %s to go\n",
directionMap[nowDir]);

    while (isDead == 0 && isWait[rightCar])
        pthread_cond_wait(&nextDirectionCond[nowDir], &crossingLock); //if
not dead, wait the right siganl

    isWait[nowDir] = 0;//pass the crossing

    pthread_mutex_unlock(&directionLock[nowDir]);//unlock this direction
    printf("car %d from %s leaving crossing\n", nowCar->carid+1,
directionMap[nowCar->direction]); //离开路口
    pthread_mutex_unlock(&crossingLock);

    if (isWait[leftCar] == 1)
        pthread_cond_signal(&nextDirectionCond[leftCar]);//left car to go

    sleep(1);//wait 1s
    alreadyHasCarArrived[nowDir] = 0;
    curCounter[nowDir]++;
    pthread_cond_signal(&followedCarCond[nowDir]);//next car in the queue to
go

    pthread_exit(NULL);
}
int main(int argc, char const *argv[])
{
    pthread_mutex_init(&crossingLock, NULL);//initialize crossing lock
    pthread_mutex_init(&deadLock, NULL);

    for (int i = 0; i < 4; i++)
    {
        pthread_mutex_init(&directionLock[i], NULL);//initialize direciton
lock
        pthread_cond_init(&nextDirectionCond[i], NULL);//initialize cond
        pthread_cond_init(&followedCarCond[i], NULL);

    }
    int numOfCar = strlen(argv[1]);
    // int numOfCar;
```

```
    // scanf("%d", &numOfCar);
    // getchar();

    char ar[numOfCar];
    for (int i = 0; i < numOfCar; i++)
    {
        // scanf("%c", &ar[i]);
        ar[i] = argv[1][i];//scan the direction
    }
    struct Car carArray[numOfCar];
    pthread_t carThread[numOfCar];//pthread array
    for (int i = 0; i < numOfCar; i++)
    {
        carArray[i].carid = i;//car id
        char dir = ar[i];//car direction

        switch (dir)//tranfer the char to int
        {
        case 'n':
            carArray[i].direction = 0;//north, use 0 to represent
            counter[0][nowCounter[0]++] = i;//put the car id into the queue
            break;
        case 'w':
            carArray[i].direction = 1;// use 1 to represent west
            counter[1][nowCounter[1]++] = i;
            break;
        case 's':
            carArray[i].direction = 2;//use 2 to represent south
            counter[2][nowCounter[2]++] = i;
            break;
        case 'e':
            carArray[i].direction = 3;//use 3 to represent east
            counter[3][nowCounter[3]++] = i;
            break;
        default:
            carArray[i].direction = 0;
        }

        pthread_create(&carThread[i], NULL, CarCross, (void
*)&carArray[i]);//create car thread
    }
    for (int i = 0; i < numOfCar; i++)
```

```
        {
            pthread_join(carThread[i], NULL);//wait
        }

        pthread_mutex_destroy(&crossingLock);
        pthread_mutex_destroy(&deadLock);//destroy

        for (int i = 0; i < 4; i++)
        {
            pthread_mutex_destroy(&directionLock[i]);//destroy
            pthread_cond_destroy(&nextDirectionCond[i]);
            pthread_cond_destroy(&followedCarCond[i]);
        }

        return 0;
    }
```