**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： 实验1同步互斥和Linux内核模块

学生姓名： 专业： 自动化 学号：

电子邮件地址： 手机：

实验地点： 玉泉曹光彪2期503 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。
* 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。

**二、实验内容**

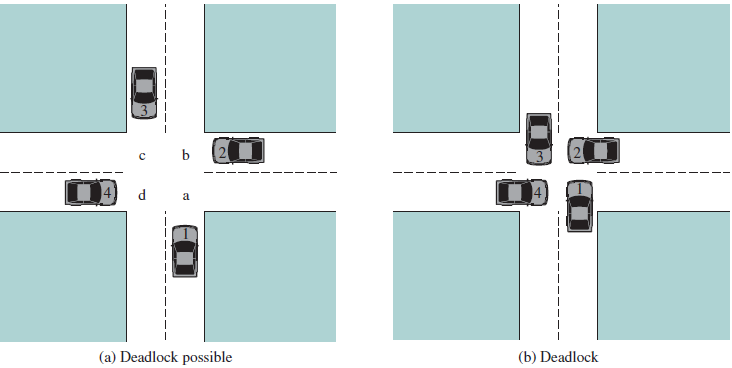
1. 有两条道路双向两个车道，即每条路每个方向只有一个车道，两条道路十字交叉。假设车辆只能向前直行，而不允许转弯和后退。如果有4辆车几乎同时到达这个十字路口，如图（a）所示；相互交叉地停下来，如图（b），此时4辆车都将不能继续向前，这是一个典型的死锁问题。从操作系统原理的资源分配观点，如果4辆车都想驶过十字路口，那么对资源的要求如下：

* 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；

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。

**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**。

**实验提示：**

1. 每一辆车的行为设计为一个单独的线程。由于有4个不同方向的车辆，需要4种不同类型的线程。
2. 使用pthread的互斥锁和条件变量解决车辆的同步与互斥。
3. 对4个不同方向的车辆，要设置车辆队列条件变量如： queueNorth、queueEast、queueSouth、queueWest。比如说，当一辆车从北方来的时候已经在过十字路口，另一辆从北方驶来的车就要等在queueNorth队列中。每一个方向都需要一个计数器来跟踪等待排队的车辆数量。
4. 按照右边车辆优先通行规则，当一辆车在等待通过路口而它右边不断有车辆到达时，这辆车及这个方向车辆队列会导致饥饿。为了防止饥饿，我们要让刚刚通过路口的A车辆发一个信号给它左边等待的B车辆，接下去让B车辆通行。需要设置下次通行车辆的条件变量firstNorth， firstEast， firstSouth， firstWest
5. 每一车辆到达十字路口时，要检测是否有死锁发生，当发生死锁时，死锁检测线程必须发出一种信号，例如：从北方来的车辆先行。
6. 假设我们设计的可执行程序文件名为p1-1，可以用'e'、'w'、's'、'n'来标识东西南北4个方向驶来的车辆，程序p1-1运行时有如下显示（你的程序不一定是这样相同的输出）：

$ ./ p1-1 nsewwewn

car 4 from West arrives at crossing

car 2 from South arrives at crossing

car 1 from North arrives at crossing

car 3 from East arrives at crossing

DEADLOCK: car jam detected, signalling North to go

car 1 from North leaving crossing

car 3 from East leaving crossing

car 2 from South leaving crossing

car 4 from West leaving crossing

car 6 from East arrives at crossing

car 5 from West arrives at crossing

car 8 from North arrives at crossing

car 5 from West leaving crossing

car 6 from East leaving crossing

car 8 from North leaving crossing

car 7 from West arrives at crossing

car 7 from West leaving crossing

1. 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（必填）**

我的计算机配置和操作系统环境：

Windows 10

Intel Core i5 7300HQ

8GB RAM

LINUX版本：

Ubuntu 18.04.3 LTS

7.7 GiB Memory

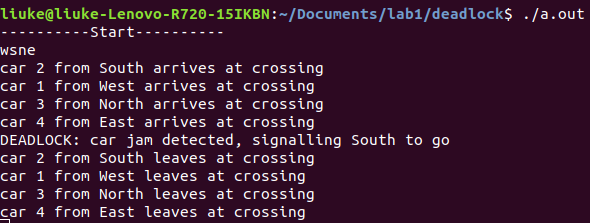
**四、操作方法与实验步骤**

1. 配置clion集成开发环境。
   * 1. 从jetbrain官网下载linux版clion，解压缩即可使用。
     2. 下载、安装、编译cmake与gdb，并在clion中配置其路径。
2. 分析并编写“同步互斥问题”代码，运行，测试，debug。
3. 分析并编写“遍历操作系统所有进程”的内核模块，运行，测试，debug。

**五、实验结果和分析**

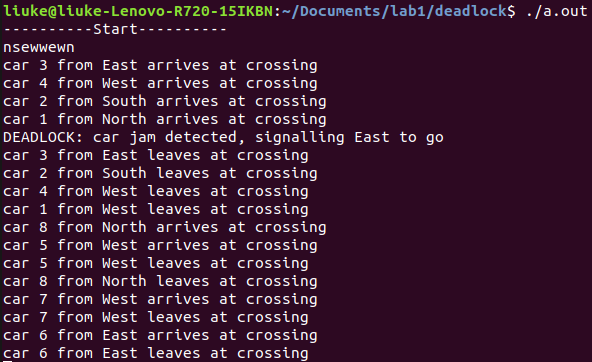
**同步互斥问题：**

* 1. 输入wsne，即四个方向各来一辆车：



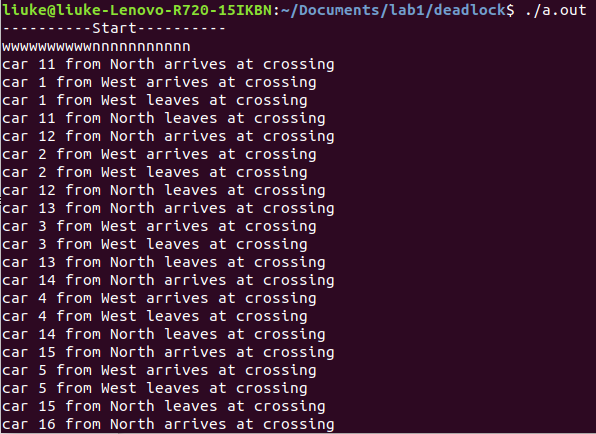
在此情况下，wsne四个方向均有来车，造成死锁，然后按照swne的顺序离开。当创建了car1、car2、car3、car4四个线程后，四个线程分别被放置在其对应的West、South、North、East队列中，当死锁检测线程被创建后，唤醒每个队列的第一辆车，即第一个线程，四个线程依次按照south、west、north、east的顺序唤醒，因此，依次到达路口，分别占据a、d、c、b路段。此时发生死锁，每个线程需要的下一个资源均被占据。接下来，按照最后达到的车的左边一辆车先走的策略解锁，在现实场景中，相当于在最后一辆车进入造成死锁前，使其停止，让其左边的车先走，然后就能够释放出资源。

* 1. 输入nsewwewn，即一种很复杂的来车情况（共8辆车）：



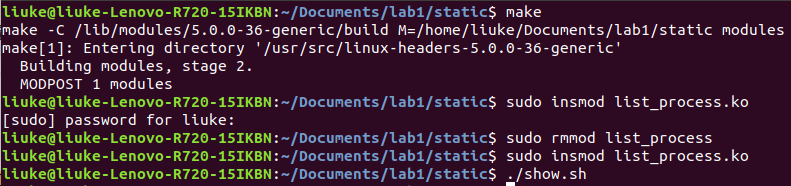
按照第一种情况下的思路，进行复杂来车情况的测试，发现不会饥饿不会死锁，车辆全部通过路口，程序通过。

3. 输入wwwwwwwwwwnnnnnnnnnn（共20辆车），用于检测是否存在饥饿的情况:。篇幅有限，贴上关键部分结果：



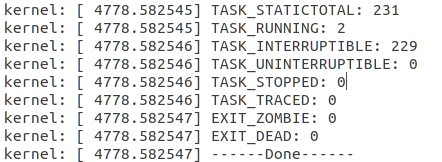
可以看到，west方向和north方向的车辆到达路口，首先是右边的先行，即west首先离开路口。为了避免饥饿，当左手边的车辆让行之后将会提升优先级，不再让行下一辆右手边的车，其通行的优先级提高，即下一个通行的应为north方向车辆（具体实现方法是，当west方向的车离开后，他唤醒的不是自己方向的下一辆车，而是左手边让行过的车）。观察程序运行结果可以发现，最终达到了west和north交替通行的结果，避免了饥饿的情况。

**遍历操作系统所有进程：**

（鉴于输出过多，我截取一部分）

 在加载好内核模块后，通过shell程序show.sh，将输出结果重定向，输入result文件中，可以看出，列出了系统的所有231个进程的状态信息。

下图为最终的统计结果，总共有231个进程，其中运行中的进程有2个，可中断进程为229个，没有其他的进程。

**六、讨论、心得**

**同步互斥问题：**

**设计思路：**

**1） 整体思路（以South方向的车S1为例）：**

South方向车辆被上一辆south方向车辆或者让行过的右手边East方向车辆唤醒，进入路口a，给路口a上锁。进入路口a后，empty（初始化为4）减去1，（表示此时资源少了一个），如果此时empty为0，则给死锁线程发信号，唤醒死锁线程。死锁线程给最后到的车的左边车发信号，让其直接通行。对于S1线程：

i）如果empty为0，则说明该车是发生死锁的最后一辆车，给死锁线程发完信号后，等待路口d以及死锁解锁信号，完成后则依次占用路口b，释放路口a的锁，释放路口b的锁。然后唤醒下一辆车，如果S1左手边的车让行过S1，则唤醒west方向车，否则唤醒下一辆south方向车。退出。

ii）empty不为0，则等待下一个路口b可用。如果S1车的右边，即east方向有车，则等待右车给S1通行的信号，并做一个标记flag，表示其等待过右手边的车；如果没有车，则该车辆进入路口b并上锁，释放路口a的锁，empty++，如果S1之前由右车唤醒，即等待过右车，则需要唤醒右边队列的车，然后唤醒south方向等待队列中的车；如果是从死锁中恢复过来的，则S1除了进入路口b上锁，释放路口a的锁，empy++外，还需要唤醒左边的车，然后释放路口b的锁。退出。

**2）死锁检测线程：**

在小车线程创建好了之后，创建了一个死锁检测线程，该线程创建后，首先开始唤醒4个方向的等待车辆，开始执行。然后进入一个检测死锁的循环，等待empty为0时，小车线程给它发送一个信号，收到信号后，判断是哪一个小车给它发送的信号，则该小车是最后一个进入造成死锁的小车，所以给其左手边的小车发信号，让其通行。

**3）小车按顺序通行及避免饥饿**

当所有小车线程都创建后，其并没有开始通行，而是排列在等待队列中， 等待一个唤醒信号，然后开始执行。以此保证顺序通行。

其中，唤醒信号有两个来源：一个是前方的车辆离开，并且其左边的车 没有等待，则它唤醒自己方向的下一辆车；另一个来源是，如果当前方向 的车离开后，左边有车等待，则唤醒其左边的车，并由左边的车来唤醒该 方向的下一辆车。避免饥饿也是由该策略完成，等待右方一次后，右方的 车不会唤醒自己方向的车，而是唤醒等待过它的车。

**心得体会：**

我认为使用多线程编程最大的困难是debug。其实理清楚思路完成代码并不困难，困难的就是在写程序过程中，由于逻辑不缜密，多少出现了一些逻辑先后上的bug。而在多线程编程中，一点点不一样都会有巨大差别。

而在处理这些bug的时候，调试是很麻烦的，因为多线程很多error是由于并发产生的，而使用调试的时候，因为等待时间改变了，也许并发产生的错误就没有并发出现，就调试不出来。

所以我最后只能够用大量的输出去调试。即运行一遍完整的程序，同时过程中增添很多输出语句，来看看是否达到预期，是否有地方不一样，来逐步缩小bug范围，进而debug。其实也是在尽可能的用肉眼，去不断理清楚自己的逻辑，去进行debug。虽然过程很艰难，但在这个过程中，我更理解了并行和多线程。这是一个对我的挑战，而我也受益良多。

此次实验花费了较多的时间，也让我从实际编程中更深入地理解了信号量和条件变量的使用。最主要的还是理清一路车的流程，然后别的车就几乎一样了。

**遍历操作系统所有进程：**

**设计思路：**

设计一个指针指向所有进程的列表，通过指针遍历所有进程，printk打印信息到缓冲区。遍历到每个进程时根据进程类型进行计数，最后输出各种类型进程数即可。

编写一个shell程序，show.sh。将输出重定向到result文件中，得到printk输出到内核缓冲区的结果。

使用教材上的方法会出现一个函数没有库的问题，因此，我在网上找到了一个使用链表直接构造的方法。

**心得体会：**

该实验相对前一个实验来说比较简单，更多是让我们学习内核模块的编程。在了解内核模块编程方法和学会使用关于查看进程信息的一系列函数，就可以进行编程了。

在加载内核模块的过程中，要注意如果是更改代码重新加载的话。要rmmod删除原来的模块，否则无法insmod进行加载。另外insmod要在root权限下进行。