同步互斥和Linux内核模块实验报告

刘轩铭 3180106071 软件工程

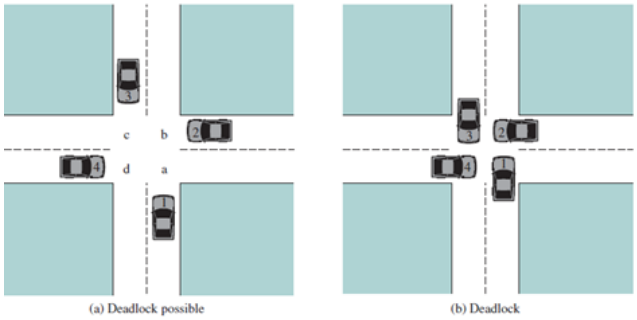
实验环境

- Linux版本：Linux version 4.15.0-117-generic (buildd@lgw01-amd64-032)
- GCC版本：gcc version 5.4.0
- Ubuntu版本：Ubuntu 5.4.0-6ubuntu1~16.04.12
- A virtual machine version under Windows 10

实验内容和结果及分析

Part 1 同步互斥实验

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。
>
> 
>
>
> - 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：
> - 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（每三行代码必须要有注释）。

设计文档

- 由于每个方向的车辆都有被阻塞的可能，所以设计四个mutex类型锁，来对各个方向的第一部车进行阻塞；此外，每个方向还需要有一个条件变量，来告诉该方向的车辆穿过十字路口。（此外还设计了一个通用锁占位使用）。
- 主要原理如下：

```
    // lock this thread if this direction has one car at crossing.
    pthread_mutex_lock(&waitToGo[dir]);
    // the car is the first one in this direction, at crossing.
    printf("car %d from %s arrives at crossing.\n", id, direction);
    // tell others this direction has car.
    isWaiting[dir] = TRUE;
    sleep(1); // avoid program runs too fast.

    pthread_mutex_t temp;
    pthread_mutex_init(&temp, NULL);
    pthread_mutex_lock(&temp);
    if(isWaiting[right(dir)]) { // if right direction has car
        if(isWaiting[right(right(dir))] && isWaiting[left(dir)] && dir == NORTH) { // deadlock
            // deadlock and this car is from north
            // detect the car jam situation and let this car go first.
            printf("DEADLOCK: car jam detected, signalling North to go\n");
        }
        else {
            // else,
            // wait for left direction to let this direction cross.
            pthread_cond_wait(&firstToGo[dir], &temp);
        }
    }
    pthread_mutex_unlock(&temp);
    pthread_mutex_destroy(&temp);

    // this car go first.
    printf("car %d from %s leaving crossing.\n", id, direction);
    // dequeue this car.
    pop(&Cars[dir]);
    // this direction has not been waiting yet.
    isWaiting[dir] = FALSE;
    // tell the left direction to cross.
    pthread_cond_signal(&firstToGo[left(dir)]);
    sleep(0);
    pthread_mutex_unlock(&waitToGo[dir]);
```

对于每个方向而言，首先用waitToGo锁阻塞该方向的车辆，如果这辆车不是第一辆，就不能够运行其余的代码。只有当第一辆车运行完后，才能打开waitToGo锁让后面的车辆运行。

然后对当前的路况进行判断，如果右边没有车，那么可以直接通行；如果有车的话，判断此时是否为死锁，如果是死锁，那么让北边的车辆通行：在程序中，这一思想被转化为，如果该车辆来自北方，那么可以直接通行，否则等待其他车辆的信号。

每辆车通行后，都要向左边的车辆发送信号，保证不发生饥饿。

- 在程序中，设计了四个车辆队列来储存每辆车的信息。用户手动输入车辆来自的方向，然后用随机数的方法生成车辆的编号。

运行结果及截图

- 题目示例的运行情况：

```
lbruyne@ubuntu:~/os/labs/lab-1/car$ gcc main.c -o main -pthread
lbruyne@ubuntu:~/os/labs/lab-1/car$ ./main
Input the cars number:
8
Input the direction of each:
nsewwewn
car 5172 from WEST arrives at crossing.
car 5612 from NORTH arrives at crossing.
car 3806 from EAST arrives at crossing.
car 7090 from SOUTH arrives at crossing.
DEADLOCK: car jam detected, signalling North to go
car 5612 from NORTH leaving crossing.
car 3806 from EAST leaving crossing.
car 7090 from SOUTH leaving crossing.
car 5172 from WEST leaving crossing.
car 3905 from WEST arrives at crossing.
car 2983 from EAST arrives at crossing.
car 3426 from NORTH arrives at crossing.
car 3905 from WEST leaving crossing.
car 3426 from NORTH leaving crossing.
car 2983 from EAST leaving crossing.
car 1114 from WEST arrives at crossing.
car 1114 from WEST leaving crossing.
```

- 大规模随机数据运行情况：

```
lbruyne@ubuntu:~/os/labs/lab-1/car$ ./main
Input the cars number:
20
Input the direction of each:
newswseewwswnewsness
car 6119 from EAST arrives at crossing.
car 7897 from SOUTH arrives at crossing.
car 4507 from WEST arrives at crossing.
car 524 from NORTH arrives at crossing.
DEADLOCK: car jam detected, signalling North to go
car 524 from NORTH leaving crossing.
car 6119 from EAST leaving crossing.
car 7897 from SOUTH leaving crossing.
car 4507 from WEST leaving crossing.
car 1601 from EAST arrives at crossing.
car 1351 from WEST arrives at crossing.
car 6377 from SOUTH arrives at crossing.
car 8464 from NORTH arrives at crossing.
DEADLOCK: car jam detected, signalling North to go
car 8464 from NORTH leaving crossing.
car 1601 from EAST leaving crossing.
car 6377 from SOUTH leaving crossing.
car 1351 from WEST leaving crossing.
car 4254 from WEST arrives at crossing.
car 7642 from SOUTH arrives at crossing.
car 3287 from EAST arrives at crossing.
car 4677 from NORTH arrives at crossing.
DEADLOCK: car jam detected, signalling North to go
car 4677 from NORTH leaving crossing.
car 3287 from EAST leaving crossing.
car 7642 from SOUTH leaving crossing.
car 4254 from WEST leaving crossing.
car 4506 from WEST arrives at crossing.
```

```
car 6892 from WEST arrives at crossing.
car 8236 from SOUTH arrives at crossing.
car 7346 from EAST arrives at crossing.
car 7346 from EAST leaving crossing.
car 8236 from SOUTH leaving crossing.
car 6892 from WEST leaving crossing.
car 5127 from WEST arrives at crossing.
car 2581 from SOUTH arrives at crossing.
car 2581 from SOUTH leaving crossing.
car 5127 from WEST leaving crossing.
```

结果分析

- 可以看出，示例程序运行的结果和题目给出的结果一样，大规模数据也能够正确完成。
- 我们的程序避免了死锁和饥饿的发生，可以有效完成对车辆的调度。

源程序c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include <pthread.h>    // thread library.
#include <semaphore.h>  // signal control library.

#define TRUE 1
#define FALSE 0
#define left(x) ((x+3) % 4)  // left car
#define right(x) ((x+1) % 4) // right car
#define DIRECTION_NUM 4
#define MAX_CAR_NUM 100 // max number of one direction
```

```c
#define MAX_CID 10000
#define random(x) (rand() % x)

// enum class to define the direction where the cars come from.
typedef enum {
    EAST = 0,
    NORTH,
    WEST,
    SOUTH,
} Direction;

// struct for cars record.
struct _Car {
    int cid;
    pthread_t cThread;
    Direction cDirection;
};
typedef struct _Car  Car;
typedef struct _Car* pCar;

// data structrue for car queue from one direction.
struct Queue {
    pCar cars[MAX_CAR_NUM]; // a series of pointer of struct _Car
    Direction direction; // enum to indicate the direction
    int front;  // front pointer
    int rear;   // rear pointer
    int count;  // the number of car in the queue
};
typedef struct Queue CarsQueue;

// initialize queue.
void initializeCarsQueues();

// push operation for cars queue.
void push(CarsQueue* queue, pCar newCarRecord);

// pop function for cars queue, return the cid.
pCar pop(CarsQueue* queue);

// create new car record.
pCar generateNewCarRecord();

// initialize.
void initialLocks();
void initialConditionalVars();

// destroy.
void destroyLocks();
void destroyConditionalVars();

// cars thread function.
void carThread(pCar thisCarReocrd);

// pointer for cars from four direction.
CarsQueue Cars[DIRECTION_NUM];

// direction directory.
char directionDir[][10] = {
```

```
        "EAST",
        "NORTH",
        "WEST",
        "SOUTH",
};

// bool to represent whether this direction has car waiting.
int isWaiting[DIRECTION_NUM] = {FALSE, FALSE, FALSE, FALSE};

// mutex of car waiting, promising that there is only the first car from each direction to cross.
pthread_mutex_t waitToGo[DIRECTION_NUM];

// cond of each direction, indicates the next car to cross.
pthread_cond_t firstToGo[DIRECTION_NUM];

// INPUT DATA:
int directions[MAX_CAR_NUM];
int currentCar = 0;
int carsNum;

int main() {
    // set the random seeds.
    srand((int)time(0));
    // initialize for four cars queues.
    initializeCarsQueues();

    // initiaize the locks and conditional variables.
    initialLocks();
    initialConditionalVars();

    // Input information of cars directions.
    int i;
    printf("Input the cars number:\n");
    scanf("%d", &carsNum); // Input the cars number to run
    printf("Input the direction of each:\n");
    fflush(stdin);
    for(i = 0; i < carsNum; i++) {
        char d;
        scanf("%c", &d); // Input each car's direction
        // change the input char to enum direction.
        switch(d) {
            case 'e': directions[i] = EAST; break;
            case 'w': directions[i] = WEST; break;
            case 's': directions[i] = SOUTH; break;
            case 'n': directions[i] = NORTH; break;
            default: printf("Illegal direction!\n"); exit(-1);
        }
    }

    // run the program.
    pCar newCarRecord;
    pthread_t* threads[MAX_CAR_NUM];
    for(i = 0; i < carsNum; i++) {
        // generate the car record. ID is randomly set.
        newCarRecord = generateNewCarRecord();
        // push the record into this direction's cars queue.
        push(&Cars[newCarRecord->cDirection], newCarRecord);
```

```
        // sequentially record the car's thread.
        threads[i] = &newCarRecord->cThread;
        // create the thread.
        pthread_create(&newCarRecord->cThread, NULL, (void*)carThread,
newCarRecord);
    }
    for(i = 0; i < carsNum; i++) {
        // wait for each thread to join.
        pthread_join(*threads[i], NULL);
    }

    // destroy the locks and conditional variables.
    destroyLocks();
    destroyConditionalVars();

    return 0;
}

void carThread(pCar thisCarReocrd)
{
    char* direction = directionDir[thisCarReocrd->cDirection];
    // obtain the id and direction of this car.
    int id = thisCarReocrd->cid;
    int dir = thisCarReocrd->cDirection;

    // lock this thread if this direction has one car at crossing.
    pthread_mutex_lock(&waitToGo[dir]);
    // the car is the first one in this direction, at crossing.
    printf("car %d from %s arrives at crossing.\n", id, direction);
    // tell others this direction has car.
    isWaiting[dir] = TRUE;
    sleep(1); // avoid program runs too fast.

    pthread_mutex_t temp;
    pthread_mutex_init(&temp, NULL);
    pthread_mutex_lock(&temp);
    if(isWaiting[right(dir)]) { // if right direction has car
        if(isWaiting[right(right(dir))] && isWaiting[left(dir)] && dir ==
NORTH) { // deadlock
            // deadlock and this car is from north
            // detect the car jam situation and let this car go first.
            printf("DEADLOCK: car jam detected, signalling North to
go\n");
        }
        else {
            // else,
            // wait for left direction to let this direction cross.
            pthread_cond_wait(&firstToGo[dir], &temp);
        }
    }
    pthread_mutex_unlock(&temp);
    pthread_mutex_destroy(&temp);

    // this car go first.
    printf("car %d from %s leaving crossing.\n", id, direction);
    // dequeue this car.
    pop(&Cars[dir]);
    // this direction has not been waiting yet.
```

```
186     isWaiting[dir] = FALSE;
187     // tell the left direction to cross.
188     pthread_cond_signal(&firstToGo[left(dir)]);
189     sleep(1); // avoid program runs too fast.
190     pthread_mutex_unlock(&waitToGo[dir]);
191     pthread_exit(NULL);
192 }
193
194
195 void initializeCarsQueues()
196 {
197     // initialize the cars queue.
198     int i;
199     for(i = 0; i < DIRECTION_NUM; i++) {
200         CarsQueue* queue = &Cars[i];
201         queue->direction = i; // Direction
202         queue->rear = -1; // the last car in the queue.
203         queue->front = 0; // the first car in the queue.
204         queue->count = 0; // the number of car in the queue
205         for(int i = 0; i < MAX_CAR_NUM; i++)
206             queue->cars[i] = malloc(sizeof(Car)); // initialize the car
    struct.
207     }
208 }
209
210 void initialLocks()
211 {
212     int i;
213     // initialize all locks.
214     for(i = 0; i < DIRECTION_NUM; i++)
215         pthread_mutex_init(&waitToGo[i], NULL);
216 }
217
218 void initialConditionalVars()
219 {
220     int i;
221     // initialize all conditional variables.
222     for(i = 0; i < DIRECTION_NUM; i++)
223         pthread_cond_init(&firstToGo[i], NULL);
224 }
225
226 void destroyLocks()
227 {
228     int i;
229     // destroy all locks.
230     for(i = 0; i < DIRECTION_NUM; i++)
231         pthread_mutex_destroy(&waitToGo[i]);
232 }
233
234 void destroyConditionalVars()
235 {
236     int i;
237     // destroy all conditional variables.
238     for(i = 0; i < DIRECTION_NUM; i++)
239         pthread_cond_destroy(&firstToGo[i]);
240 }
241
242 void push(CarsQueue* queue, pCar newCarRecord)
```

```
{
    // judge if the queue is full.
    if(queue->count == MAX_CAR_NUM) {
        printf("The queue %d is full, push failed.\n", queue->direction);
        exit(-1);
    }
    queue->count++;

    // car enqueue.
    queue->rear = (queue->rear + 1) % MAX_CAR_NUM; // loop
    queue->cars[queue->rear] = newCarRecord;
}

pCar pop(CarsQueue* queue)
{
    // judge if the queue is empty.
    if(queue->count == 0) {
        printf("The queue %d is empty, pop failed.", queue->direction);
        exit(-1);
    }
    queue->count--;

    // car dequeue.
    pCar popCar = queue->cars[queue->front];
    queue->front = (queue->front + 1) % MAX_CAR_NUM; // loop
    return popCar;
}

pCar generateNewCarRecord()
{
    pthread_t cp;
    Direction cd;
    // printf("Input a dirction:\n");
    // scanf("%d", &cd);
    // cd = (int)random(DIRECTION_NUM);
    // the id is randomly generated.
    int cid = (int)random(MAX_CID);

    // the direction is what user input.
    cd = directions[currentCar++] % DIRECTION_NUM;
    pCar newCarRecord = (void*)malloc(sizeof(Car));
    // set the car record.
    newCarRecord->cid = cid;
    newCarRecord->cDirection = cd;
    newCarRecord->cThread = cp;
    return newCarRecord;
}
```

Part 2 Linux内核模块

实验内容

> - 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。

设计文档

内核模块编写

- 本模块主要实现：遍历系统中当前的所有进程，按照一定的格式打印出相关的信息
- 通过阅读linux源文件 sched.h 部分源码，可以发现：进程控制块PCB主要由 TASK_STRUCT 这一个结构体构成，其内部有许多成员变量，进程的名字，ID，状态和父进程的名字分别由 ptr->comm, ptr->pid, ptr->state 和 ptr->parent->comm 保存。其中，state有许多个取值，如下：

```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */

/* Used in tsk->state: */
#define TASK_RUNNING            0x0000
#define TASK_INTERRUPTIBLE      0x0001
#define TASK_UNINTERRUPTIBLE        0x0002
#define __TASK_STOPPED          0x0004
#define __TASK_TRACED           0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD           0x0010
#define EXIT_ZOMBIE         0x0020
#define EXIT_TRACE          (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED         0x0040
#define TASK_DEAD           0x0080
#define TASK_WAKEKILL           0x0100
#define TASK_WAKING         0x0200
#define TASK_NOLOAD         0x0400
#define TASK_NEW            0x0800
#define TASK_STATE_MAX          0x1000

/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED         (TASK_WAKEKILL | __TASK_TRACED)

#define TASK_IDLE           (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)

/* Convenience macros for the sake of wake_up(): */
#define TASK_NORMAL         (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL            (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)

/* get_task_state(): */
#define TASK_REPORT         (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                     TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
                     __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
                     TASK_PARKED)
```

- 于是，我们只需要遍历每一个进程，然后分别打印出上述信息即可。对于不同状态的统计，只需要根据上面的值进行判断，然后用计数器累加即可。

- ```c
  int init_module(void)
  {
      struct task_struct *p = NULL;
      p = &init_task;
      for_each_process(p){
          ...
      }
  }

  void cleanup_module(void)
  {
      ...
  }
  ```

  用该循环来遍历各个进程，然后打印相关信息。

  具体的源代码见附录。

- 在编写了 print_processes.c 文件后，编写Makefile文件如下，用于编译该源文件：

  ```
  TARGET=print_processes
  KDIR=/lib/modules/$(shell uname -r)/build
  PWD=$(shell pwd)
  obj-m += $(TARGET).o
  default:
      make -C $(KDIR) M=$(PWD) modules
  clean:
      make -C $(KDIR) M=$(PWD) clean
  ```

- 接下来需要用下列的指令将模块加载，然后查看相关信息

  ```
  $ make
  $ sudo insmod print_processes.ko
  $ dmesg
  ```

- 之后执行：

  ```
  $ sudo rmmod print_processes
  ```

  来卸载模块。

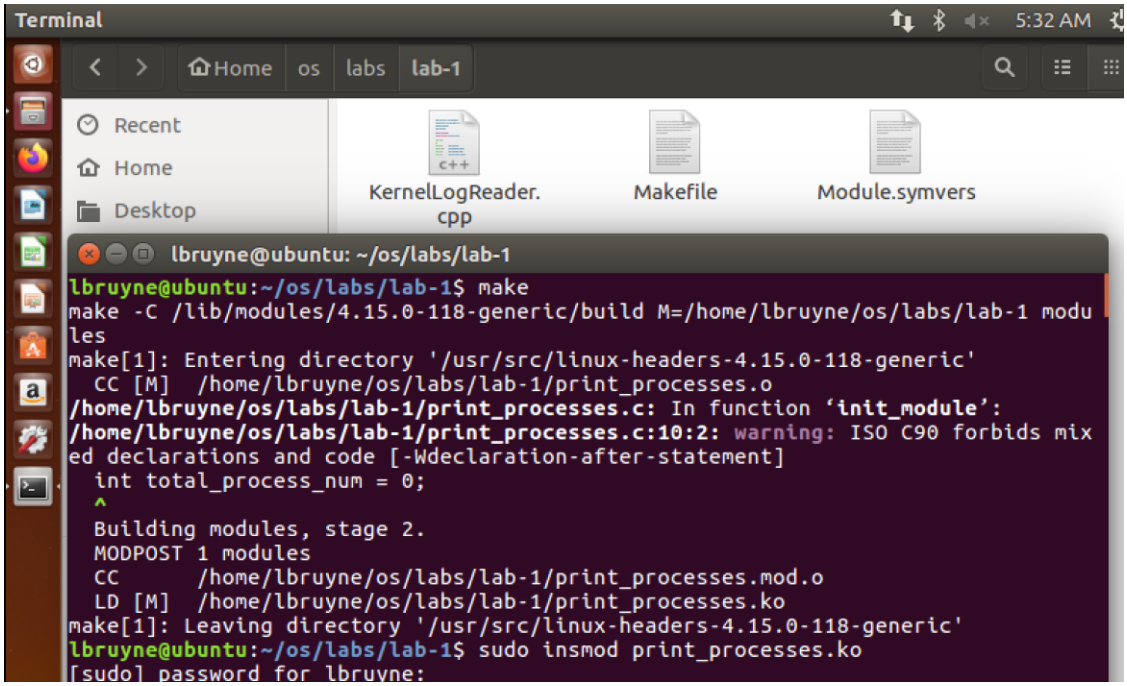
用户态程序编写

- 由于需要将内核日志的文件输出到标准输入输出端，所以编写了 KernelLogReader.cpp 文件。该文件定义了 KernelLogReader类，用户通过输入文件路径和标识符，来对对应的内核模块信息进行打印。
- 在Linux中，内核日志文件被记录在 /var/log/kern.log 文件中，所以只需要用文件流访问该文件，然后找到对应的文件段打印即可。为了进行识别，我在内核模块中打印信息的前后，都加入了标识符 *PROCESS_STATE_INFO* 进行标志。
- 这样，只需要遍历每一行记录，找到该标识符，然后打印之间的内容就可以了。由于用户只需要最新的打印内容，所以我们需要过滤掉之前的内容，找到最后一组标识符，然后打印其间的信息。详细代码见附录。
- 编写完用户程序后，运行下列脚本，输出日志内容：
  ```

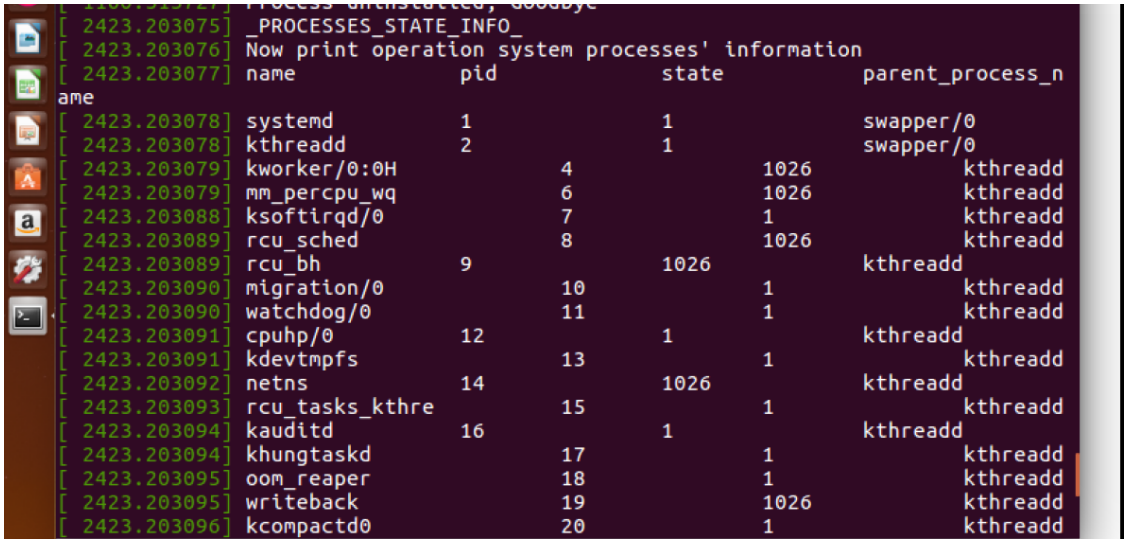
```
$ g++ KernelLogReader.cpp -o KernelLogReader
$ ./KernelLogReader
```

运行结果及截图

- 编译和安装模块过程如下：



- 用 dmesg 命令显示结果如下：



- 各种进程的统计数据结果如下：

```
[ 2423.203208] The total number of process is: 221
[ 2423.203208] The number of running process is: 1
[ 2423.203209] The number of interruptible process is: 156
[ 2423.203209] The number of uninterruptible process is: 0
[ 2423.203209] The number of stopped process is: 0
[ 2423.203209] The number of traced process is: 0
[ 2423.203210] The number of parked process is: 0
[ 2423.203210] The number of dead process is: 0
[ 2423.203210] The number of wakekill process is: 0
[ 2423.203210] The number of waking process is: 0
[ 2423.203210] The number of noload process is: 0
[ 2423.203211] The number of new process is: 0
[ 2423.203211] The number of exit dead process is: 0
[ 2423.203211] The number of exit zombie process is: 0
[ 2423.203211] The number of exit trace process is: 0
[ 2423.203212] The number of idle  process is: 64
[ 2423.203212] The number of normal process is: 0
[ 2423.203212] The number of killable process is: 0
[ 2423.203212] The number of all(state) process is: 0
[ 2423.203213] The number of report process is: 0
[ 2423.203213] _PROCESSES_STATE_INFO_
```

- 用户态输出结果：

```
lbruyne@ubuntu:~/os/labs/lab-1$ ./KernelLogReader
Start print the log of identified module.
Oct 12 05:31:48 ubuntu kernel: [ 2423.203076] Now print operation system processes'
 information
Oct 12 05:31:48 ubuntu kernel: [ 2423.203077] name              pid             stat
e            parent_process_name
Oct 12 05:31:48 ubuntu kernel: [ 2423.203078] systemd           1               1swa
pper/0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203078] kthreadd          2               1swa
pper/0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203079] kworker/0:0H              4       1026
             kthreadd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203079] mm_percpu_wq              6       1026
             kthreadd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203088] ksoftirqd/0               7       1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203089] rcu_sched         8               1026
             kthreadd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203089] rcu_bh            9               1026
             kthreadd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203090] migration/0               10      1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203090] watchdog/0                11      1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203091] cpuhp/0           12              1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203091] kdevtmpfs         13              1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203092] netns             14              1026
             kthreadd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203093] rcu_tasks_kthre           15      1kth
readd
Oct 12 05:31:48 ubuntu kernel: [ 2423.203094] kauditd           16              1kth
```

```
Oct 12 05:31:48 ubuntu kernel: [ 2423.203207] bash             3164          1gno
me-terminal-
Oct 12 05:31:48 ubuntu kernel: [ 2423.203207] sudo             3313          1bas
h
Oct 12 05:31:48 ubuntu kernel: [ 2423.203208] insmod           3314          0sud
o
Oct 12 05:31:48 ubuntu kernel: [ 2423.203208] The total number of process is: 221
Oct 12 05:31:48 ubuntu kernel: [ 2423.203208] The number of running process is: 1
Oct 12 05:31:48 ubuntu kernel: [ 2423.203209] The number of interruptible process i
s: 156
Oct 12 05:31:48 ubuntu kernel: [ 2423.203209] The number of uninterruptible process
 is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203209] The number of stopped process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203209] The number of traced process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203210] The number of parked process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203210] The number of dead process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203210] The number of wakekill process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203210] The number of waking process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203210] The number of noload process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203211] The number of new process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203211] The number of exit dead process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203211] The number of exit zombie process is:
 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203211] The number of exit trace process is:
0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203212] The number of idle  process is: 64
Oct 12 05:31:48 ubuntu kernel: [ 2423.203212] The number of normal process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203212] The number of killable process is: 0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203212] The number of all(state) process is:
0
Oct 12 05:31:48 ubuntu kernel: [ 2423.203213] The number of report process is: 0
Print ends.
```

结果分析

- 如图所示，我们打印出了系统内各个进程的相关信息和对于状态的统计结果。
- 可以看出，大部分进程都处于等待状态，有一些进程处于空闲中，而只有一个进程正在运行。和上课所学不同，真实的Linux环境中，状态的数量远远多于课上所说的5-6个。
- 此外，我们也通过用户态程序，打印出了内核模块的信息。

源程序

- 内核模块 print_processes.c

```c
#include<linux/module.h>
#include<linux/init.h>
#include<linux/kernel.h>
#include<linux/sched.h>
#include<linux/init_task.h>

// run when install the module.
int init_module(void)
{
    /*
        #define TASK_RUNNING            0x0000
        #define TASK_INTERRUPTIBLE      0x0001
        #define TASK_UNINTERRUPTIBLE        0x0002
        #define __TASK_STOPPED          0x0004
        #define __TASK_TRACED           0x0008
        #define EXIT_DEAD           0x0010
        #define EXIT_ZOMBIE         0x0020
        #define EXIT_TRACE          (EXIT_ZOMBIE | EXIT_DEAD)
        #define TASK_PARKED         0x0040
        #define TASK_DEAD           0x0080
```

```
        #define TASK_WAKEKILL           0x0100
        #define TASK_WAKING         0x0200
        #define TASK_NOLOAD         0x0400
        #define TASK_NEW            0x0800
        #define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
        #define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
        #define TASK_TRACED         (TASK_WAKEKILL | __TASK_TRACED)
        #define TASK_IDLE           (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
        #define TASK_NORMAL         (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
        #define TASK_ALL            (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)
        #define TASK_REPORT         (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                                TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \

                                __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
                                TASK_PARKED)

     */

    // print header identifier
    printk(KERN_INFO"_PROCESSES_STATE_INFO_");

    // below are states suitable for p->state
    int total_process_num = 0; // total process number
    int task_running_num = 0; // running process number
    int task_int_num = 0; // interruptible process number
    int task_uint_num = 0; // uninterruptible process number
    int task_stopped_num = 0; // stopped process number
    int task_traced_num = 0; // traced process number
    int task_parked_num = 0; // parked process number
    int task_dead_num = 0; // dead process number
    int task_wakekill_num = 0; // wakekill process number
    int task_waking_num = 0; // waking process number
    int task_noload_num = 0; // noload process number
    int task_new_num = 0; // new process number
    int exit_dead_num = 0; // dead process number
    int exit_zombie_num = 0; // exited zombie process number
    int exit_trace_num = 0; // exited trace process number
    int task_killable_num = 0; // killable process number
    int task_idle_num = 0; // idle process number
    int task_normal_num = 0; // normal process number
    int task_all_num = 0; // all process number
    int task_report_num = 0; // report process number

    struct task_struct *p = NULL; // TASK_STRUCT pointer
    p = &init_task;
    printk(KERN_INFO"Now print operation system processes' information\n");
    printk(KERN_INFO"name\t\tpid\t\tstate\t\tparent_process_name\n"); // table header

    // using a process to traversal all processes.
```

```
    for_each_process(p)
    {
        // print the information for each process.
        printk(KERN_INFO"%s\t\t%d\t\t%ld\t\t%s\n", p->comm, p->pid, p->state, p->parent->comm);

        total_process_num++;

        // using switch to differ each state.
        switch(p->state) {
            case TASK_RUNNING: task_running_num++; break;
            case TASK_INTERRUPTIBLE: task_int_num++; break;
            case TASK_UNINTERRUPTIBLE: task_uint_num++; break;
            case __TASK_STOPPED:
            case TASK_STOPPED: task_stopped_num++; break;
            case __TASK_TRACED:
            case TASK_TRACED: task_traced_num++; break;
            case EXIT_DEAD: exit_dead_num++; break;
            case EXIT_ZOMBIE: exit_zombie_num++; break;
            case EXIT_TRACE: exit_trace_num++; break;
            case TASK_PARKED: task_parked_num++; break;
            case TASK_DEAD: task_dead_num++; break;
            case TASK_WAKEKILL: task_wakekill_num++; break;
            case TASK_WAKING: task_waking_num++; break;
            case TASK_NOLOAD: task_noload_num++; break;
            case TASK_NEW: task_new_num++; break;
            case TASK_IDLE: task_idle_num++; break;
            case TASK_KILLABLE: task_killable_num++; break;
            case TASK_NORMAL: task_normal_num++; break;
            case TASK_ALL: task_all_num++; break;
            case TASK_REPORT: task_report_num++; break;
            // if our state classes are not overall, this signal will be printed.
            // Fortunately, it didn't appear.
            default: printk(KERN_INFO"There is other state!\n"); break;
        }
    }

    // print all statistics.
    printk(KERN_INFO"The total number of process is: %d\n", total_process_num);
    printk(KERN_INFO"The number of running process is: %d\n", task_running_num);
    printk(KERN_INFO"The number of interruptible process is: %d\n", task_int_num);
    printk(KERN_INFO"The number of uninterruptible process is: %d\n", task_uint_num);
    printk(KERN_INFO"The number of stopped process is: %d\n", task_stopped_num);
    printk(KERN_INFO"The number of traced process is: %d\n", task_traced_num);
    printk(KERN_INFO"The number of parked process is: %d\n", task_parked_num);
    printk(KERN_INFO"The number of dead process is: %d\n", task_dead_num);
    printk(KERN_INFO"The number of wakekill process is: %d\n", task_wakekill_num);
```

```
115     printk(KERN_INFO"The number of waking process is: %d\n",
    task_waking_num);
116     printk(KERN_INFO"The number of noload process is: %d\n",
    task_noload_num);
117     printk(KERN_INFO"The number of new process is: %d\n",
    task_new_num);
118     printk(KERN_INFO"The number of exit dead process is: %d\n",
    exit_dead_num);
119     printk(KERN_INFO"The number of exit zombie process is: %d\n",
    exit_zombie_num);
120     printk(KERN_INFO"The number of exit trace process is: %d\n",
    exit_trace_num);
121     printk(KERN_INFO"The number of idle  process is: %d\n",
    task_idle_num);
122     printk(KERN_INFO"The number of normal process is: %d\n",
    task_normal_num);
123     printk(KERN_INFO"The number of killable process is: %d\n",
    task_killable_num);
124     printk(KERN_INFO"The number of all(state) process is: %d\n",
    task_all_num);
125     printk(KERN_INFO"The number of report process is: %d\n",
    task_report_num);
126 
127     // print a tail identifier.
128     printk(KERN_INFO"_PROCESSES_STATE_INFO_");
129     printk(KERN_INFO"print_processes module runs ending");
130     return 0;
131 }
132 
133 // run when uninstalling the module.
134 void cleanup_module(void)
135 {
136     printk(KERN_INFO"Process uninstalled, Goodbye\n");
137 }
138 
139 MODULE_LICENSE("GPL");
140 
```

- Makefile

```
1 TARGET=print_processes
2 KDIR=/lib/modules/$(shell uname -r)/build
3 PWD=$(shell pwd)
4 obj-m += $(TARGET).o
5 default:
6     make -C $(KDIR) M=$(PWD) modules
7 clean:
8     make -C $(KDIR) M=$(PWD) clean
```

- 用户态程序 KernelLogReader.cpp

```
1 #include<iostream>
2 #include<fstream>
3 #include<string>
4 #include<vector>
5 #include<cstdlib>
```

```cpp
using namespace std;

// A class: OOP required.
class KernelLogReader
{
private:
    string filePath; // log file root path.
    fstream f; // file i/o obj.
    vector<int> posVec; // a vector to store the line number of each identifier.

public:
    KernelLogReader(string filePath) {
        this->filePath = filePath;
    }

    // when exits, close the file.
    ~KernelLogReader() {
        closeTheLogFile();
    }

    void printModuleLog(string moduleIdentifier) {
        openTheLogFile();
        // traversal the file line by line and fill the vector with line number.
        findPosOfIdentifier(moduleIdentifier);
        if(posVec.empty()) {
            // if didn't find the identifier, exits.
            cout << "Identifier not exists." << endl;
            exit(0);
        }
        // redirect the file pointer to the head.
        f.clear();
        f.seekg(0);
        // lpos is the last position of identifier pair.
        // cnt is the line number that the infomation we need occupies.
        int lpos = posVec.at(posVec.size()-2);
        int cnt = posVec.at(posVec.size()-1) - lpos - 1;
        // print the information from lpos, and continues for cnt line.
        printFromLastPosLine(lpos, cnt);
    }

private:
    // open the file
    void openTheLogFile() {
        f.open(filePath.c_str(), ios::in);
        // exits if not exists.
        if(!f) {
            cout << "File not exists." << endl;
            exit(0);
        }
    }

    // close the file.
    void closeTheLogFile() {
        f.close();
```

```
60      }
61
62      void findPosOfIdentifier(string id) {
63          int pos = 0;
64          string buf;
65          // traversal every line.
66          while(getline(f, buf)) {
67              // total line read.
68              pos++;
69              // if this line contains identifier, add the line number
    into vector.
70              if(buf.find(id) != string::npos)
71                  posVec.push_back(pos);
72          }
73      }
74
75      void printFromLastPosLine(int lpos, int cnt) {
76          int tmp = 0;
77          string buf;
78          // traversal the file again.
79          while(getline(f, buf)) {
80              tmp++;
81              // not meet the position we need.
82              if(tmp < lpos) continue;
83              // meets, print a prompt infomation.
84              if(tmp == lpos) cout << "Start print the log of identified
    module." << endl;
85              // if jumps the last information we need, break;
86              if(tmp > lpos + cnt) {
87                  cout << "Print ends." << endl;
88                  break;
89              }
90              // between the identifier pair, we print the information
    into standard I/O screen.
91              if(tmp > lpos) cout << buf.c_str() << endl;
92          }
93      }
94  };
95
96  int main()
97  {
98      // User give the road for log file and the identifier.
99      string filePath = "/var/log/kern.log";
100     string id = "_PROCESSES_STATE_INFO_";
101
102     // create a new obj.
103     KernelLogReader r(filePath); // set a file path.
104     // this method print the log into standard i/o screen.
105     r.printModuleLog(id);
106     return 0;
107 }
108
```

讨论与心得

模块一

心得

在这一实验中，比较困难的是实现汽车线程的同步。主要的要点在于以下几点：

- 前车通知后车的机制

  最终考虑的方案是使用为每一个方向设置一个互斥锁，在每辆车进入路口时获取并将其锁住，在锁住的期间后车是无法获取到锁的，所以只能等待；而通过路口时再解开锁。这样的操作比较简单可行，同时也能保护isWaiting变量。

- 面对死锁时的调度策略

  最终采取的办法是每辆车都检查死锁，发生死锁时固定由北边车辆先行，再使用条件变量通知其他车辆。这一实现相比其他方式更为简单易行，而且每个线程不需要其他人通知，自己就会明白“应该做什么”。

- 关于条件变量的使用
  我对条件变量的使用存在一些疑惑：为什么条件变量的wait函数一定要配一个互斥锁使用?我查阅资料后发现：条件变量的使用经常搭配了一个互斥锁，出于编程的简化性以及原子性考虑，将加锁-修改值-解锁-发信号的操作整合到一个函数中。互斥锁只是一种编程习惯，那么没有互斥锁的条件变量也是可以单独用于收发信号的。于是我做了一个尝试，用一个临时变量来作为通知右车的条件变量等待所需要的互斥锁，随用随销毁，对程序本身没有任何影响，同样可以完成任务。

体会

- 我发现多线程编程还是较为困难的，因为线程之间需要小心谨慎，一不小心就有可能触发死锁。
- 一些编程范式是可以改变的，而不是一成不变的。
- 注释写多还是有好处的，像这个题目，我编程的跨时较长，如果没有注释，我很容易忘记自己之前的思路。

模块二

心得

- 内核模块的 Makefile 文件包含目标文件，目标文件依赖的文件，更新（或生成）目标文件的命令，并且每个命令之前必须是 TAB 键，否则会编译错误。我一开始 TAB 默认设置的是空格，所以会一直报编译错误，后来将其重新设置为 TAB 键，编译成功。
- 由于每次加载模块都会在/var/log/kern.log 中输出信息，而我只想输出最后一次的信息，所有在内核模块加载时会先输出标志字符串，用来做标记。在用户态程序读取了 log 文件后，会先查找标记字符串出现的最后一次的位置，然后程序从该位置开始写出 log 文件的内容。

体会

- 对内核模块的加载和卸载过程了解更深
- 对 Makefile 文件的编写更加深入了解
- 对 printk 的打印级别做了了解