Project1 - 同步互斥和 Linux 内核模块

1. 同步互斥

1.1 程序设计思路：

对于车辆线程:

- 车辆分为 4 种，表示来自 4 个不同的方向;
- 当有一辆车被创建的时候，根据车辆来自的方向把它添加到相应队列的队尾;
- 当队列里的车成为队列头的时候就可以过马路了，来到路口;
- 通过判断右边方向的车辆队列是否为空来判断右边是否也有车辆需要过马路，如果有，就等待，让右边的车辆先过，这时候如果发生死锁，则把控制权交给死锁处理线程，由其发信号让来自北方的车先过;
- 为了防止饿死，右边的车辆过去后应该发一个信号告诉它左边的车立即通过（左边的车最多等一辆）;
- 车辆过马路后检查左侧是否有已经到达路口的车辆在等待，如果有，则发信号让其立即通过，然后告诉自己后边的车可以上路了，并把自己从队列里踢出;

对于死锁处理线程:

- 死锁处理线程一直等待死锁发生，然后接管十字路口，让来自北方的车先走，然后继续等待下一次死锁发生;

1.2 详细设计说明

用 4 个队列来表示来自 4 个方向的车辆

```
Queue *qFromEast, *qFromWest, *qFromSouth, *qFromNorth;
```

用 4 个互斥量限制对队列的访问，确保队列数据的正确性（防止有多个对队列的操作同时发生）:

```
pthread_mutex_t mtx_qE, mtx_qS, mtx_qW, mtx_qN;
```

用 4 个互斥量表示 4 条路，车辆拥有一条路的互斥锁表示正处于该条路上

```
pthread_mutex_t mtx_south, mtx_east, mtx_north, mtx_west;
```

用 4 个条件量表示让某个方向的车辆立即通过：

```
pthread_cond_t firstSouth, firstEast, firstNorth, firstWest;
```

用 4 个条件量表示某个方向后面的车是否可以上路了：

```
pthread_cond_t come2a, come2b, come2c, come2d;
```

用 4 个 bool 量表示 4 个路口是否被占用：

```
bool a_occupied, b_occupied, c_occupied, d_occupied;
```

用 4 个 bool 量表示是否有左边的车辆在等待路口：

```
bool ext_wait_a, ext_wait_b, ext_wait_c, ext_wait_d;
```

用一个条件量表示死锁处理线程是否可以开始处理死锁：

```
pthread_cond_t cond_deadlock;
```

用一个互斥量表示死锁处理程序的资源，拥有该互斥量的锁表示拥有死锁处理的资源，进行与死锁有关的操作必须要先获得该互斥量的锁：

```
pthread_mutex_t mtx_deadlock;
```

用一个互斥量表示对整个路口的控制权：

```
pthread_mutex_t mtx_crossing;
```

来自 4 个方向的车辆的行为函数：

```
void* fromWest(void *car_id_ptr);
void* fromNorth(void *car_id_ptr);
void* fromEast(void *car_id_ptr);
void* fromSouth(void *car_id_ptr);
```

死锁处理线程的行为函数：

```
void* deadlock_proc();
```

程序运行流程：

1. 初始化各种变量，创建死锁处理线程；
2. 接受输入，根据输入创建车辆进程；
3. 等待所有车辆进程结束；

车辆线程的具体行为：

1. 根据自己来的方向将自己添加进相应车辆队列；
2. 等待前方车辆通行（等待成为队列头）；

3. 试图上路(获取路的互斥锁),如果路口被占用,则等待可以上路的条件量,上路后,就将对应路口标记为被占用;
4. 检查右侧车辆队列是否为空，为空则准备直接通过，不为空则等待，并将等待标志标记为有;
5. 如果 4 个等待标志全为有，说明发生了死锁，这时候发信号给死锁处理程序，交给其处理;
6. 先暂时释放路的资源（互斥锁），等待右侧车辆的立即通行的信号;
7. 收到立即通行的信号后，将等待标志清空，调用系统调用让线程暂停一段时间，表示正在通过路口;
8. 通过路口后检查左边是否有车辆在等待（检查等待标志），如果有就发信号让其立即通行;
9. 将自己移出车辆队列，发信号告诉后面的车可以上路了，并离开（释放该方向路的互斥锁);

死锁处理线程的具体行为:

1. 死锁处理线程从被创建起应该一直存在直到整个程序结束;
2. 在每个阶段（每次循环）中，先试图获取死锁互斥量的互斥锁，然后等待死锁发生的信号;
3. 收到死锁发生的信号后进行处理，获取北方的路的资源（互斥锁），然后发信号让来自北方的车先通过，释放北方的路的互斥锁;
4. 释放死锁互斥量的互斥锁，结束一个阶段（一次循环）后，再次进入等待状态，等待下一次死锁发生;

1.3 源代码及注释

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

#define MAXN 100
#define CRXTIME 1
```

```c
typedef enum{false=0, true=1} bool;
// typedef enum{EAST, WEST, SOUTH, NORTH} Direction;
// typedef int car;

typedef struct carsQueue{
    int *cars;
    int front;
    int rear;
    int size;
    int capacity;
}Queue;

Queue* newQueue(int cap){
    Queue *q = (Queue*)malloc(sizeof(Queue));
    q->cars = (int*)malloc(sizeof(int)*(1+cap));
    q->front = 0;
    q->rear = 0;
    q->size = 0;
    q->capacity = cap;
    return q;
}
void enQueue(Queue* q, int car_id){
    if(q->size == q->capacity){
        printf("The queue is full!\n");
        return;
    }
    q->cars[q->rear] = car_id;
    q->rear = (q->rear+1)%(q->capacity+1);
    q->size++;
}
void deQueue(Queue* q){
    if(q->size==0){
        printf("The queue is empty.\n");
        return;
    }
    q->front = (q->front+1)%(q->capacity+1);
    q->size--;
}
int getFront(Queue* q){
    if(q->size==0) {
        printf("Error: fetch an empty queue!\n");
        return -1;
    }
    return q->cars[q->front];
```

```
}
void destroyQueue(Queue* q){
    free(q);
}

Queue *qFromEast, *qFromWest, *qFromSouth, *qFromNorth;  // cars queue
for 4 directions
bool ext_wait_a, ext_wait_b, ext_wait_c, ext_wait_d;  // whether exist
waiting right
bool a_occupied, b_occupied, c_occupied, d_occupied;  // whether a road
is occupied

pthread_cond_t come2a, come2b, come2c, come2d; //condtions for
following cars to go forward
pthread_cond_t cond_deadlock;   // condtion for deadlock occuring
pthread_cond_t firstSouth, firstEast, firstNorth, firstWest; //
conditions for left to go first

pthread_mutex_t mtx_south, mtx_east, mtx_north, mtx_west;// mutexes
for 4 roads
pthread_mutex_t mtx_qE, mtx_qS, mtx_qW, mtx_qN;              //
mutexes for 4 queues
pthread_mutex_t mtx_deadlock;   // mutex for deadlock
pthread_mutex_t mtx_crossing;   // mutex for the crossing

void* deadlock_proc(){
    while(1){
        // printf("Deadlock: waiting...\n");
        pthread_mutex_lock(&mtx_crossing);
        // wait for a deadlock occuring
        pthread_cond_wait(&cond_deadlock, &mtx_crossing);
        printf("Deadlock: Car jam detected, processing...\nDeadlock:
Signalling North to go...\n");
        pthread_mutex_lock(&mtx_north);
        // signal the car from North to go first
        pthread_cond_signal(&firstNorth);
        pthread_mutex_unlock(&mtx_north);
        pthread_mutex_unlock(&mtx_crossing);
        // printf("Deadlock: A car jam has been processed.\n");
    }
}

void* fromWest(void *car_id_ptr){
    int car_id = *(int*)car_id_ptr;
```

```
    pthread_mutex_lock(&mtx_qW);
    enQueue(qFromWest, car_id);
    // printf("Car %d in queue West.\n", car_id);
    pthread_mutex_unlock(&mtx_qW);

    while(getFront(qFromWest) != car_id);

    pthread_mutex_lock(&mtx_west);
    // wait for the front car on the road
    while(d_occupied) pthread_cond_wait(&come2d, &mtx_west);
    d_occupied = true;
    printf("Car %d from West arrives at the crossing.\n", car_id);

    if(qFromSouth->size > 0){
        ext_wait_a = true;
        // printf("Car %d: wait for the right.\n", car_id);
        if(ext_wait_a && ext_wait_b && ext_wait_c && ext_wait_d){
            // printf("Car %d: DEADLOCK: car jam detected, signalling
North to go.\n", car_id);
            pthread_mutex_lock(&mtx_deadlock);
            pthread_cond_signal(&cond_deadlock);
            pthread_mutex_unlock(&mtx_deadlock);
            // printf("Car %d: DEADLOCK: solved.\n", car_id);
        }
        pthread_cond_wait(&firstWest, &mtx_west);
        // printf("Car %d: right finished, go first.\n", car_id);
    }
    ext_wait_a = false;
    // printf("Car %d: ready to go. \n", car_id);
    // pthread_mutex_lock(&mtx_crossing);
    sleep(CRXTIME);
    // pthread_mutex_unlock(&mtx_crossing);
    printf("Car %d from West leaves the crossing.\n", car_id);

    if(ext_wait_d){
        pthread_mutex_lock(&mtx_north);
        // let the left to go first
        pthread_cond_signal(&firstNorth);
        pthread_mutex_unlock(&mtx_north);
    }

    pthread_mutex_lock(&mtx_qW);
    deQueue(qFromWest);
```

```
    // printf("Car %d: out of the queue West.\n", car_id);
    pthread_mutex_unlock(&mtx_qW);

    d_occupied = false;
    // let following cars to go forward
    pthread_cond_signal(&come2d);

    pthread_mutex_unlock(&mtx_west);
}

void* fromNorth(void *car_id_ptr){
    int car_id = *(int*)car_id_ptr;

    pthread_mutex_lock(&mtx_qN);
    enQueue(qFromNorth, car_id);
    // printf("Car %d in queue North.\n", car_id);
    pthread_mutex_unlock(&mtx_qN);

    while(getFront(qFromNorth)!=car_id);

    pthread_mutex_lock(&mtx_north);
    while(c_occupied) pthread_cond_wait(&come2c, &mtx_north);
    c_occupied = true;
    printf("Car %d from North arrives at the crossing.\n", car_id);

    if(qFromWest->size > 0){
        ext_wait_d = true;
        // printf("Car %d: wait for the right.\n", car_id);
        if(ext_wait_a && ext_wait_b && ext_wait_c &&ext_wait_d){
            // printf("Car %d: DEADLOCK: car jam detected, signalling
North to go.\n", car_id);
            pthread_mutex_lock(&mtx_deadlock);
            pthread_cond_signal(&cond_deadlock);
            pthread_mutex_unlock(&mtx_deadlock);
            // printf("Car %d: DEADLOCK: solved.\n", car_id);
        }
        pthread_cond_wait(&firstNorth, &mtx_north);
        // printf("Car %d: right finished, go first.\n", car_id);
    }
    ext_wait_d = false;
    // printf("Car %d: ready to go.\n", car_id);
    // pthread_mutex_lock(&mtx_crossing);
    sleep(CRXTIME);
    // pthread_mutex_unlock(&mtx_crossing);
```

```
    printf("Car %d from North leaves the crossing.\n", car_id);

    if(ext_wait_c){
        pthread_mutex_lock(&mtx_east);
        // let the left to go first
        pthread_cond_signal(&firstEast);
        pthread_mutex_unlock(&mtx_east);
    }

    pthread_mutex_lock(&mtx_qN);
    deQueue(qFromNorth);
    // printf("Car %d: out of the queue North.\n", car_id);
    pthread_mutex_unlock(&mtx_qN);

    c_occupied = false;
    // let following cars to go forward
    pthread_cond_signal(&come2c);

    pthread_mutex_unlock(&mtx_north);
}

void* fromEast(void *car_id_ptr){
    int car_id = *(int*)car_id_ptr;

    pthread_mutex_lock(&mtx_qE);
    enQueue(qFromEast, car_id);
    // printf("Car %d in queue East.\n", car_id);
    pthread_mutex_unlock(&mtx_qE);

    while(getFront(qFromEast)!=car_id);

    pthread_mutex_lock(&mtx_east);
    // wait for the front car on the road
    while(b_occupied) pthread_cond_wait(&come2b, &mtx_east);
    b_occupied = true;
    printf("Car %d from East arrives at the crossing.\n", car_id);

    if(qFromNorth->size > 0){
        ext_wait_c = true;
        // printf("Car %d: wait for the right.\n", car_id);
        if(ext_wait_a && ext_wait_b && ext_wait_c &&ext_wait_d){
            // printf("Car %d: DEADLOCK: car jam detected, signalling
North to go.\n", car_id);
            pthread_mutex_lock(&mtx_deadlock);
```

```
            pthread_cond_signal(&cond_deadlock);
            pthread_mutex_unlock(&mtx_deadlock);
            // printf("Car %d: DEADLOCK: solved.\n", car_id);
        }
        pthread_cond_wait(&firstEast, &mtx_east);
        // printf("Car %d: right finished, go first.\n", car_id);
    }
    ext_wait_c = false;
    // printf("Car %d: ready to go. \n", car_id);
    // pthread_mutex_lock(&mtx_crossing);
    sleep(CRXTIME);
    // pthread_mutex_unlock(&mtx_crossing);
    printf("Car %d from East leaves the crossing.\n", car_id);

    if(ext_wait_b){
        pthread_mutex_lock(&mtx_south);
        // let the left to go first
        pthread_cond_signal(&firstSouth);
        pthread_mutex_unlock(&mtx_south);
    }

    pthread_mutex_lock(&mtx_qE);
    deQueue(qFromEast);
    // printf("Car %d: out of the queue East.\n", car_id);
    pthread_mutex_unlock(&mtx_qE);

    b_occupied = false;
    // let following cars to go forward
    pthread_cond_signal(&come2b);

    pthread_mutex_unlock(&mtx_east);
}

void* fromSouth(void *car_id_ptr){
    int car_id = *(int*)car_id_ptr;

    pthread_mutex_lock(&mtx_qS);
    enQueue(qFromSouth, car_id);
    // printf("Car %d in queue South.\n", car_id);
    pthread_mutex_unlock(&mtx_qS);

    while(getFront(qFromSouth) != car_id);

    pthread_mutex_lock(&mtx_south);
```

```
    // wait for the front car on the road
    while(a_occupied) pthread_cond_wait(&come2a, &mtx_south);
    a_occupied = true;
    printf("Car %d from South arrives at the crossing.\n", car_id);

    // pthread_mutex_lock(&mtx_east);
    if(qFromEast->size > 0){
        ext_wait_b = true;
        // printf("Car %d: wait for the right.\n", car_id);
        if(a_occupied && b_occupied && c_occupied && d_occupied){
            // printf("Car %d: DEADLOCK: car jam detected, signalling
North to go.\n", car_id);
            pthread_mutex_lock(&mtx_crossing);
            pthread_cond_signal(&cond_deadlock);
            pthread_mutex_unlock(&mtx_crossing);
            // printf("Car %d: DEADLOCK: solved.\n", car_id);
        }
        pthread_cond_wait(&firstSouth, &mtx_south);
        // printf("Car %d: right finished, go first.\n", car_id);
    }
    ext_wait_b = false;
    // b_occupied = true;
    // printf("Car %d: ready to go. \n", car_id);
    // pthread_mutex_lock(&mtx_crossing);
    sleep(CRXTIME);
    // pthread_mutex_unlock(&mtx_crossing);
    printf("Car %d from East leaves the crossing.\n", car_id);

    // b_occupied = false;
    // pthread_mutex_unlock(&mtx_east);
    if(ext_wait_a){
        pthread_mutex_lock(&mtx_west);
        // let the left to go first
        pthread_cond_signal(&firstWest);
        pthread_mutex_unlock(&mtx_west);
    }

    pthread_mutex_lock(&mtx_qS);
    deQueue(qFromSouth);
    // printf("Car %d: out of the queue South.\n", car_id);
    pthread_mutex_unlock(&mtx_qS);

    a_occupied = false;
    // let the following to go forward
```

```
    pthread_cond_signal(&come2a);
    pthread_mutex_unlock(&mtx_south);
}

int main(int argc, char **argv){
    int n = 10;
    char dircs[MAXN];
    // char dircs[] = "senwssnwse";
    printf("directions string:\n");
    scanf("%s", dircs);
    n = strlen(dircs);

    // initilization
    pthread_cond_init(&firstSouth, NULL);
    pthread_cond_init(&firstEast, NULL);
    pthread_cond_init(&firstNorth, NULL);
    pthread_cond_init(&firstWest, NULL);

    pthread_cond_init(&come2a, NULL);
    pthread_cond_init(&come2b, NULL);
    pthread_cond_init(&come2c, NULL);
    pthread_cond_init(&come2d, NULL);

    pthread_mutex_init(&mtx_south, NULL);
    pthread_mutex_init(&mtx_east, NULL);
    pthread_mutex_init(&mtx_north, NULL);
    pthread_mutex_init(&mtx_west, NULL);

    pthread_mutex_init(&mtx_qS, NULL);
    pthread_mutex_init(&mtx_qE, NULL);
    pthread_mutex_init(&mtx_qN, NULL);
    pthread_mutex_init(&mtx_qW, NULL);

    pthread_mutex_init(&mtx_crossing, NULL);
    pthread_mutex_init(&mtx_deadlock, NULL);
    pthread_cond_init(&cond_deadlock, NULL);

    ext_wait_a = ext_wait_b = ext_wait_c = ext_wait_d = false;
    a_occupied = b_occupied = c_occupied = d_occupied = false;

    qFromSouth = newQueue(n);
    qFromEast = newQueue(n);
    qFromNorth = newQueue(n);
    qFromWest = newQueue(n);
```

```
    // create the deadlock processing thread
    pthread_t deadlock;
    if(pthread_create(&deadlock, NULL, deadlock_proc, NULL)!=0)
        printf("cannot create the deadlock thread.\n");

    // create car threads
    int car_no = 0;
    pthread_t cars[MAXN];
    // pthread_t *cars = (car*)malloc(sizeof(pthread_t)*n);
    int car_ids[MAXN];
    for(int i=0; i<n; i++){ car_ids[i] = i+1; }
    for(int i=0; i<n; i++){
        car_no++;
        if(dircs[i]=='s'){
            if(pthread_create(&cars[i], NULL, fromSouth, car_ids+i)!=0)
                printf("Creating car thread failed.\n");
        }else if(dircs[i]=='e'){
            if(pthread_create(&cars[i], NULL, fromEast, car_ids+i)!=0)
                printf("Creating car thread failed.\n");
        }else if(dircs[i]=='n'){
            if(pthread_create(&cars[i], NULL, fromNorth, car_ids+i)!=0)
                printf("Creating car thread failed.\n");
        }else if(dircs[i]=='w'){
            if(pthread_create(&cars[i], NULL, fromWest, car_ids+i)!=0)
                printf("Creating car thread failed.\n");
        }
    }
    for(int i=0; i<n; i++){ pthread_join(cars[i], NULL); }
    destroyQueue(qFromSouth);
    destroyQueue(qFromEast);
    destroyQueue(qFromNorth);
    destroyQueue(qFromWest);

    return 0;
}
```

1.4 运行结果

```
Car 1 from South arrives at the crossing.
Car 2 from East arrives at the crossing.
Car 3 from North arrives at the crossing.
Car 4 from West arrives at the crossing.
Deadlock: Car jam detected, processing...
Deadlock: Signalling North to go...
Car 3 from North leaves the crossing.
Car 7 from North arrives at the crossing.
Car 2 from East leaves the crossing.
Car 6 from East arrives at the crossing.
Car 1 from East leaves the crossing.
Car 5 from South arrives at the crossing.
Deadlock: Car jam detected, processing...
Deadlock: Signalling North to go...
Car 4 from West leaves the crossing.
Car 7 from North leaves the crossing.
Car 8 from West arrives at the crossing.
Car 6 from East leaves the crossing.
Car 5 from East leaves the crossing.
Car 9 from South arrives at the crossing.
Car 9 from East leaves the crossing.
Car 8 from West leaves the crossing.
Car 10 from West arrives at the crossing.
Car 10 from West leaves the crossing.
PS ~/code>
```

2. Linux 内核模块

2.1 程序设计思路

直接遍历一遍操作系统的所有进程，并统计所需要的数据，包括进程总数，TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_ZOMBIE、TASK_TRACED、TASK_STOPPED、以及其他（unknown）类的进程的个数，对于遍历到的每个进程，立即输出进程的详细信息，包括进程名，pid，状态，父进程的名字，统计结果在遍历结束后输出。

2.2 详细设计说明

一个用于遍历的进程指针：

```
struct task_struct *p = &init_task;
```

用遍历进程的宏进行遍历：

```
for_each_process(p){...}
```

在进程信息统计输出前输出“###”作为开始标记，在进程统计信息输出完后输出“###”作为结束标记。

用户态程序根据开始标记和结束标记定位日志中的进程统计信息，然后输出。

（此程序逻辑较为简单，更详细的部分直接参见源码）

2.3 源代码及注释

内核模块程序

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched/task.h>   // init_task
#include <linux/sched/signal.h> // for_each_process

static void traverse_count(void){
    int running = 0;         // number of TASK_RUNNING
    int interruptible = 0;       // number of TASK_INTERRUPTIBLE
    int uninterruptible = 0; // number of TASK_UNINTERRUPTIBLE
    int zombie = 0;              // number of EXIT_ZOMBIE
    int stopped = 0;         // number of TASK_STOPPED
    int traced = 0;              // number of TASK_TRACED
    int dead = 0;            // number of EXIT_DEAD
    int unknown = 0;         // number of unknown (other)
    int total = 0;           // number of all processes

    // int exit_state;
    // int state;
    struct task_struct *p = &init_task; // process pointer
    // read_lock(&tasklist_lock);       // get the lock of tasklist
    // for(p=&init_task; (p = next_task(p))!=&init_task;)
    for_each_process(p){
        // output the info of a process
        printk("Name:%s, pid:%d, state:%ld, dad_name:%s\n", p->comm,
p->pid, p->state, p->real_parent->comm);
        total++; // count the total number of processes
        // state = p->state;
        // exit_state = p->exit_state;

        switch(p->exit_state){
            case EXIT_ZOMBIE: // zombie process
                zombie++; break;
            case EXIT_DEAD:   // dead process
                dead++; break;
            default:          // other
                break;
        }

        // if the process has exited, traverse the next directly
        if(p->exit_state != 0) continue;
```

```
        switch(p->state){
            case TASK_RUNNING:              // running process
                running++; break;
            case TASK_INTERRUPTIBLE:     // interruptible process
                interruptible++; break;
            case TASK_UNINTERRUPTIBLE:      // uninterruptible process
                uninterruptible++; break;
            case TASK_STOPPED:              // stopped process
                stopped++; break;
            case TASK_TRACED:            // traced process
                traced++; break;
            default:                     // other (unknown) process
                unknown++; break;
        }
    }
    // read_unlock(&tasklist_lock); // unlock tasklist

    // output the count result
    printk("total: %d\n", total);          // output the total number
of processes
    printk("running: %d\n", running);      // output the number of
TASK_RUNNING
    printk("interruptible: %d\n", total);  // output the number of
TASK_INTERRUPTIBLE
    printk("uninterruptible: %d\n", total);// output the number of
TASK_UNINTERRUPTIBLE
    printk("traced: %d\n", traced);        // output the number of
TASK_TRACED
    printk("stopped: %d\n", stopped);      // output the number of
TASK_STOPPED
    printk("zombie: %d\n", zombie);        // output the number of
EXIT_ZOMBIE
    printk("dead: %d\n", dead);            // output the number of
EXIT_DEAD
    printk("unknown: %d\n", unknown);      // output the number of
other processes
}

static int proc_init(void){
    printk("<proc_traverse> Hello~\n"); // initialization msg
    printk("###\n");  // symbol of beginning
    traverse_count(); // run the traversing function
    printk("###\n");  // symbol of ending
```

```
    return 0;

}

static void proc_exit(void){
    printk("<proc_traverse> Bye~\n");   // exiting msg
}

module_init(proc_init);  // register initialization module
module_exit(proc_exit);  // register exiting module
MODULE_LICENSE("GPL");   // GNU General Public License
```

用户态程序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(){
    FILE *fp;
    if((fp=fopen("/var/log/kern.log", "r")) == NULL){
        printf("Error: cannot open the log file.\n");
        exit(1);
    }
    fseek(fp, 0, SEEK_SET);
    char c = fgetc(fp);
    int matched = 0;
    int find = 0;
    while(c != EOF){
        if(c=='#') matched++;
        else matched=0;

        if(matched==3){find = 1; break;}
        c = fgetc(fp);
    }
    if(find){
        printf("Find!\n");
    }else{
        printf("Error: cannot find log info. :(\n");
        exit(1);
    }
    c = fgetc(fp);
    matched=0;
    int out = 0;
    while(c != EOF){
```

```
        if(out==0 && c==']') out++;
        else if(out==1 && c==' ') out++;
        else if(out==2){if(c!='#') printf("%c", c);}

        if(c=='\n') out = 0;
        else if(c=='#') matched++;

        if(matched==3) break;
        c = fgetc(fp);
    }
    printf("End\n");
    fclose(fp);
    return 0;
}
```

2.4 运行结果

内核模块编译：

```
maswell@ubuntu-vm:linux_core$ make
make -C /lib/modules/5.4.0-52-generic/build M=/home/maswell/Desktop/Codes/linux_core modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-52-generic'
  CC [M]  /home/maswell/Desktop/Codes/linux_core/proc_traverse.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/maswell/Desktop/Codes/linux_core/proc_traverse.mod.o
  LD [M]  /home/maswell/Desktop/Codes/linux_core/proc_traverse.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-52-generic'
maswell@ubuntu-vm:linux_core$
```

内核模块安装：

```
maswell@ubuntu-vm:linux_core$ sudo insmod proc_traverse.ko
[sudo] password for maswell:
maswell@ubuntu-vm:linux_core$ dmesg
[    0.000000] Linux version 5.4.0-52-generic (buildd@lcy01-amd64-022) (gcc version 7.5.0 (Ub
04)) #57~18.04.1-Ubuntu SMP Thu Oct 15 14:04:49 UTC 2020 (Ubuntu 5.4.0-52.57~18.04.1-generic
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.4.0-52-generic root=UUID=a8ad9f1f-13c
e ro quiet splash
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
```

内核模块安装时的输出：

```
[  311.991594] proc_traverse: loading out-of-tree module taints kernel.
[  311.991625] proc_traverse: module verification failed: signature and/or required key missing
[  311.992120] <proc_traverse> Hello~
[  311.992120] ###
[  311.992122] Name:systemd, pid:1, state:1, dad_name:swapper/0
[  311.992136] Name:kthreadd, pid:2, state:1, dad_name:swapper/0
[  311.992137] Name:rcu_gp, pid:3, state:1026, dad_name:kthreadd
[  311.992138] Name:rcu_par_gp, pid:4, state:1026, dad_name:kthreadd
[  311.992139] Name:kworker/0:0, pid:5, state:1026, dad_name:kthreadd
[  311.992140] Name:kworker/0:0H, pid:6, state:1026, dad_name:kthreadd
```

用户态程序运行：

```
maswell@ubuntu-vm:linux_core$ gcc user.c
maswell@ubuntu-vm:linux_core$ ./a.out
Find!
Name:systemd, pid:1, state:1, dad_name:swapper/0
Name:kthreadd, pid:2, state:1, dad_name:swapper/0
Name:rcu_gp, pid:3, state:1026, dad_name:kthreadd
Name:rcu_par_gp, pid:4, state:1026, dad_name:kthreadd
Name:kworker/0:0H, pid:6, state:1026, dad_name:kthreadd
Name:kworker/0:1, pid:7, state:1026, dad_name:kthreadd
Name:mm_percpu_wq, pid:9, state:1026, dad_name:kthreadd
```

内核模块卸载时的输出：

```
[  311.992390] stopped: 0
[  311.992390] zombie: 0
[  311.992391] dead: 0
[  311.992391] unknown: 106
[  311.992391] ###
[  612.345758] <proc_traverse> Bye~
maswell@ubuntu-vm:linux_core$
```

3. 心得与体会

本次实验分为两个部分，一个是线程的同步互斥实验，主要考察对于操作系统中进程或线程同步的理解，以及对死锁、饿死问题的初步处理，要求有基本的 pthread 编程能力，对 pthread 库有一定的了解；第二个部分是写一个遍历操作系统所有进程的内核模块，然后用一个用户态程序读取日志文件中的输出，主要考察 Linux 内核编程。

作为操作系统课程的第一次实验，实验的两个部分都比较简单和基础，在实验的过程中学到了很多，包括对进程（线程）同步问题的理解，对 pthread 库的使用以及对 Linux 内核编程的了解，对理论课上学到的内容也有了更深的体会和思考。

希望可以在接下来的实验和课程中做的更好。