**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： 同步互斥和Linux内核模块

学生姓名： 张佳瑶 学号： 3170103240

电子邮件地址： 1531077171@qq.com

实验日期： 2019 年 10 月 23 日

1. 实验环境

处理器：2.4GHz Intel Core i5

macOS 10.14.5

Linux version 5.0.0-29-generic (buildd@lgw01-amd64-039) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1))

处理器：Intel® Core™ i7-6700HQ CPU @ 2.60GHz

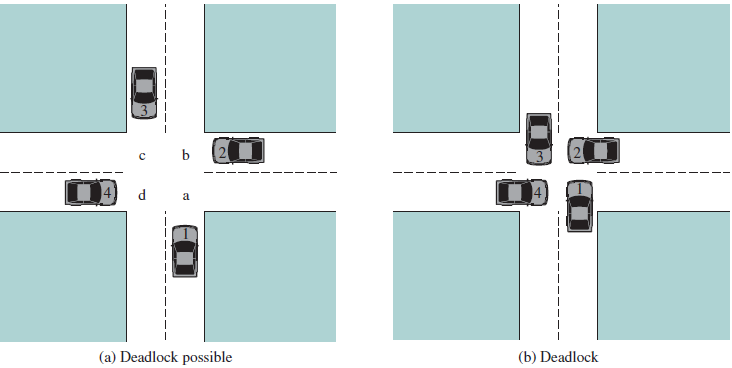
Windows10

Linux version 5.0.0-29-generic (buildd@lgw01-amd64-039) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1))

1. 实验内容和结果及分析
   1. 题目

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**（**每三行代码必须要有注释**）。

* 1. 设计文档

1.2.1 **pthread.h**

1) 线程控制方面的函数有：pthread*attr*init、pthread*create、pthread*join、pthread\_exit

|  |  |
| --- | --- |
| 函数 | 功能 |
| pthread*attr*init | 初始化一个线程对象的属性 |
| pthread\_create | 四个参数：第一个参数是指向线程标识符的指针，type: pthread\_t\*；第二次参数设置线程属性，NULL则使用默认属性；第三个参数是线程运行函数的起始地址，type: (void\*)(\*)(void\*)；第四个参数是运行函数的参数，type: void\* |
| pthread\_join | 用来等待一个线程的结束，线程间同步的操作，共两个参数：第一个参数为线程标识符，即线程id，type: pthread\_t；第二个参数retval为用户定义的指针，用来存储线程的返回值，type:void\*\* |
| pthread\_exit | 显式退出进程 |

2） 互斥锁机制函数：pthread*mutex*init、pthread*mutex*lock、pthread*mutex*unlock、pthread*mutex*destroy

互斥锁保证只有一个进程访问该对象。

锁类型结构ptread*mutex*t

|  |  |
| --- | --- |
| 函数 | 功能 |
| pthread*mutex*init | 在C语言的多线程编程中，实现互斥锁的初始化 |
| pthread*mutex*lock | 获取互斥锁 |
| pthread*mutex*unlock | 释放互斥锁 |
| pthread*mutex*destroy | 注销一个互斥锁，参数为**pthread*mutex*t** *mute*x |

1.2.2 流程说明

1. 记录每一辆车的id和方向，将每一辆车加入到对应方向的车队中等待行驶。创建一个死锁检测线程，检测到死锁，就释放信号让北方车行驶。
2. 为每一辆车创建一个线程。
3. 判断线程对应的车是否为等待车队中的第一辆车，若是，则进入4；若否，则循环等待。
4. 判断这个方向上是否还有车在行驶，若是，则等待条件变量唤醒；若否，则进入5。
5. 该车可以到达路口，获得这个方向上的互斥锁。
6. 判断当前右方向是否有车等待，若是，等待可行驶条件变量；若否，进入7。
7. 行驶当前的车，打印行驶信息。
8. 获得等待车队的互斥锁，将当前车移除等待队列，释放当前车队下一辆车可行驶的条件变量。
9. 所有线程结束后，释放所有的条件变量和互斥量

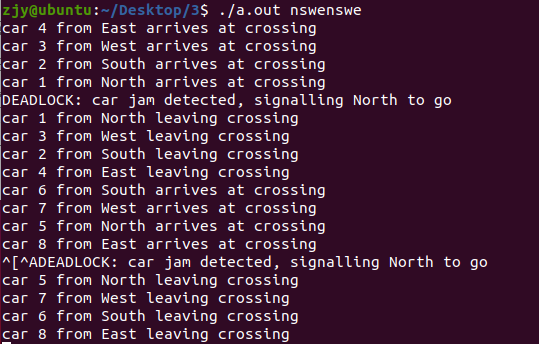
1.2.3 基本变量和函数声明

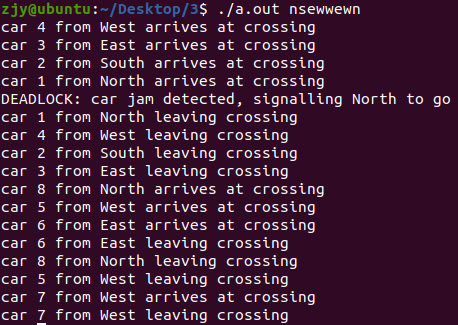
pthread\_cond\_t queue[4]; //各个方向车辆队列的条件变量  
pthread\_mutex\_t lock[4]; //各个方向条件变量互斥锁  
pthread\_cond\_t next[4]; //各个方向设置下次通行车辆的条件变量  
pthread\_mutex\_t cross; //路口通行互斥量  
pthread\_mutex\_t Wait[4]; //设置各个方向上等待队列的互斥量  
struct carNode car[MAX]; //车量队列  
struct carQueue waitCarQueue[4]; //各个方向等待的车的队列  
int isCar[4] = {0}; //各个方向上的路口是否有车正在通行  
void \*carCross(void \*car); //车辆通过线程  
void \*checkDeadlock(void \*item); //死锁检测线程

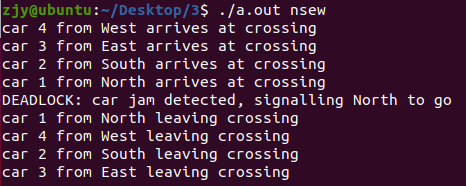
* 1. 程序运行结果截图

编译程序：g++ final.cpp -o thread -lpthread

运行程序：./thread [车队]



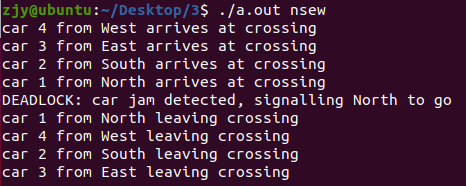




1.4 结果分析

当少于四辆车到达十字路口时，不会发生死锁。

当有四辆车到达十字路口时，会发生死锁，释放北方车辆先行，解除死锁，其他方向车辆接着行驶。



为例：

释放north先行信号。1号车最先行驶。1号车右手边的4号车接着行驶。4号车右手边的2号车接着行驶。最后3号车行驶。

1.5 源程序

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h> #include <pthread.h>  #define MAX 100 #define NORTH 0 #define EAST 1 #define SOUTH 2 #define WEST 3  //一辆汽车 struct carNode {  int id;  int direction; }; //一个汽车队列 struct carQueue {  int size; //队列中车总数  int head; //第一辆车  int tail; //最后一辆车  char direction[6];  int cars[MAX]; }; pthread\_cond\_t queue[4]; //各个方向车辆队列的条件变量 pthread\_mutex\_t lock[4]; //各个方向条件变量互斥锁 pthread\_cond\_t next[4]; //各个方向设置下次通行车辆的条件变量 pthread\_mutex\_t cross; //路口通行互斥量 pthread\_mutex\_t Wait[4]; //设置各个方向上等待队列的互斥量 pthread\_cond\_t deadlockCond; //死锁条件变量 pthread\_mutex\_t deadlockLock; //检测死锁互斥量 struct carNode car[MAX]; //车量队列 struct carQueue waitCarQueue[4]; //各个方向等待的车的队列 int isWait[4] = {0}; //各个方向上是否有车等待 int isCar[4] = {0}; //各个方向上的路口是否有车正在通行  void \*carCross(void \*car);  int main(int argc, char const \*argv[]) {  //初始化条件变量和互斥量  for (int i = 0; i < 4; i++)  {  pthread\_cond\_init(&queue[i], NULL);  pthread\_cond\_init(&next[i], NULL);  pthread\_mutex\_init(&lock[i], NULL);  }  pthread\_cond\_init(&deadlockCond, NULL);  pthread\_mutex\_init(&deadlockLock, NULL);   //初始化等待队列  for (int i = 0; i < 4; i++)  {  waitCarQueue[i].head = 0;  waitCarQueue[i].tail = 0;  waitCarQueue[i].size = 0;  }   //创建和总车数相等的线程  pthread\_t carThread[strlen(argv[1])];  //标记好四个车队的方向  strcpy(waitCarQueue[NORTH].direction, "North");  strcpy(waitCarQueue[EAST].direction, "East");  strcpy(waitCarQueue[SOUTH].direction, "South");  strcpy(waitCarQueue[WEST].direction, "West");  //获得四个方向的车队  for (int i = 0; i < strlen(argv[1]); i++)  {  switch (argv[1][i])  {  case 'n':  //北方  car[i].id = i + 1;  car[i].direction = 0;  //加入北方车队  waitCarQueue[NORTH].size++;  waitCarQueue[NORTH].cars[waitCarQueue[NORTH].tail++] = i + 1;  break;  case 'e':  //东方  car[i].id = i + 1;  car[i].direction = 1;  //加入东方车队  waitCarQueue[EAST].size++;  waitCarQueue[EAST].cars[waitCarQueue[EAST].tail++] = i + 1;  break;  case 's':  //南方  car[i].id = i + 1;  car[i].direction = 2;  //加入南方车队  waitCarQueue[SOUTH].size++;  waitCarQueue[SOUTH].cars[waitCarQueue[SOUTH].tail++] = i + 1;  break;  case 'w':  //西方  car[i].id = i + 1;  car[i].direction = 3;  //加入西方车队  waitCarQueue[WEST].size++;  waitCarQueue[WEST].cars[waitCarQueue[WEST].tail++] = i + 1;  break;  default:  break;  }  }   //创建线程  for (int i = 0; i < strlen(argv[1]); i++)  {  pthread\_create(&carThread[i], NULL, carCross, (void \*)&car[i]);  }  //等待线程停止  for (int i = 0; i < strlen(argv[1]); i++)  {  pthread\_join(carThread[i], NULL);  }   //释放资源  pthread\_mutex\_destroy(&cross);  pthread\_mutex\_destroy(&deadlockLock);  pthread\_cond\_destroy(&deadlockCond);  for (int i = 0; i < 4; i++)  {  pthread\_mutex\_destroy(&lock[i]);  pthread\_cond\_destroy(&queue[i]);  pthread\_cond\_destroy(&next[i]);  pthread\_mutex\_destroy(&Wait[i]);  } }  void \*carCross(void \*car) {  struct carNode \*Car = (struct carNode \*)car;   //如果这辆车不是等待列表中的第一辆车  while (waitCarQueue[Car->direction].cars[waitCarQueue[Car->direction].head] != Car->id)  {  ;  }  //开到路口  printf("car %d from %s arrives at crossing\n", Car->id, waitCarQueue[Car->direction].direction);   //开启路口锁  pthread\_mutex\_lock(&lock[Car->direction]);  //这个方向上有车在开，等待信号量  while (isCar[Car->direction] == 1)  {  pthread\_cond\_wait(&queue[Car->direction], &lock[Car->direction]);  }  //标记本车要开  isCar[Car->direction] = 1;  //标记本车等待  isWait[Car->direction] = 1;  //死锁检测  pthread\_mutex\_lock(&deadlockLock);  //发现死锁  if (isWait[NORTH] && isWait[EAST] && isWait[SOUTH] && isWait[WEST])  {  switch (Car->direction)  {  case NORTH:  //北方车让西方车先行驶  printf("DEADLOCK: car jam detected, signalling West to go\n");  pthread\_cond\_signal(&next[WEST]);  break;  case EAST:  //东方车让北方车先行驶  printf("DEADLOCK: car jam detected, signalling North to go\n");  pthread\_cond\_signal(&next[NORTH]);  break;  case SOUTH:  //南方车让东方车先行驶  printf("DEADLOCK: car jam detected, signalling East to go\n");  pthread\_cond\_signal(&next[EAST]);  break;  case WEST:  //西方车让北方车先行驶  printf("DEADLOCK: car jam detected, signalling South to go\n");  pthread\_cond\_signal(&next[SOUTH]);  break;  default:  break;  }  }  pthread\_mutex\_unlock(&deadlockLock);  //如果右边有车，等待信号量  if (waitCarQueue[(Car->direction + 1) % 4].size != 0)  {  pthread\_cond\_wait(&next[Car->direction], &lock[Car->direction]);  }  //当前没有车在路口等待，本车可以开了  isWait[Car->direction] = 0;  pthread\_mutex\_lock(&cross);  printf("car %d from %s leaving crossing\n", Car->id, waitCarQueue[Car->direction].direction);  pthread\_mutex\_unlock(&cross);  //释放左边车辆通行信号  pthread\_cond\_signal(&next[(Car->direction + 3) % 4]);  pthread\_mutex\_unlock(&lock[Car->direction]);   //将开走的车移除等待队列  pthread\_mutex\_lock(&Wait[Car->direction]);  waitCarQueue[Car->direction].head++;  waitCarQueue[Car->direction].size--;  isCar[Car->direction] = 0;  //释放本车队下一辆车可以行驶的信号量  pthread\_cond\_signal(&queue[Car->direction]);  pthread\_mutex\_unlock(&Wait[Car->direction]);  pthread\_exit(NULL); } |

2.1 题目

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。

2.2 设计文档

2.2.3 内核模块

2.2.3.1 头文件

|  |
| --- |
| #include <linux/module.h> //modules KERN\_INFO #include <linux/sched/signal.h> //struct task\_struct |

头文件linux/module.h包含对模块的结构定义以及模块的版本控制，如KERN\_INFO*。*头文件linux/sched/signal.h包含必要的INIT*\_*TASK宏。

2.2.3.2 主要函数

|  |
| --- |
| int init\_module(void); void cleanup\_module(void); |

函数init\_module的作用是向内核注册模块提供的新功能。

函数cleanup\_module的作用是注销所有模块注册的新功能。

2.2.3.3 init\_module说明

Linux是一个多用户、多任务的系统，会产生很多的进程，每个进程会有不同的状态。Linux进程状态可以为分：可执行状态TASK\_RUNNING\可中断的睡眠状态TASK\_INTERRUPTIBLE\不可中断的睡眠状态TASK\_UNINTERRUPTIBLE\暂停状态TASK\_STOPPED\跟踪状态TASK\_TRACED\退出状态TASK\_DEAD\EXIT\_ZOMBIE。

struct task\_struct中:

1. 成员comm[TASKCOMMLEN]代表正在执行的名字，路径。
2. pid代表进程的id
3. state代表进程状态,-1代表没有运行，0代表可以运行，大于0表示停止。
4. parent代表父进程指针，children代表孩子进程指针，sibling代表兄弟进程指针。
5. exit\_state代表进程是否退出。

首先定义一个指向进程的指针process，然后用这个指针遍历系统中所有的进程。Linux系统中进程的组织数据结构是双向循环链表，其根为init\_task。因此，将process的初始值设为init\_task，通过process=next\_task(process)，将指针process修改为指向当前进程的下一个进程。通过判断process是否重新指向init\_task决定循环是否结束。

对于系统中的每一个进程，首先通过在循环中totalProcess++，统计系统中进程总数。读取结构体成员，在终端打印进程的基本信息，如process->comm,process->pid,process->state,process->parent->pid,process->parent->comm。

首先判断进程是否终止。如果process->exit\_state == 0为true代表进程没有终止，如果process->exit\_state == 0为false代表进程已经终止。

一个进程没有终止，state域能够取5个互为排斥的值。

|  |  |
| --- | --- |
| 状态 | 描述 |
| TASK\_RUNNING | 1. 进程正在被执行 2. 进程就绪 |
| TASK\_INTERRUPTIBLE | 进程因为等待一些条件而被阻塞 |
| TASK\_ININTERRUPTIBLE | 进程因为等待一些条件而被阻塞 |
| TASK\_STOPPED | 进程被停止执行 |
| TASK\_TRACED | 进程被debugger等进程监视 |

一个进程终止了，在exit\_state域可以取2个值。

|  |  |
| --- | --- |
| 状态 | 描述 |
| EXIT\_ZOMBIE | 进程被终止，但父进程未终止，进程成为僵尸进程 |
| EXIT\_DEAD | 进程最终状态 |

用switch-case语句，判断进程状态，记录不同状态下的进程个数。

最后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。

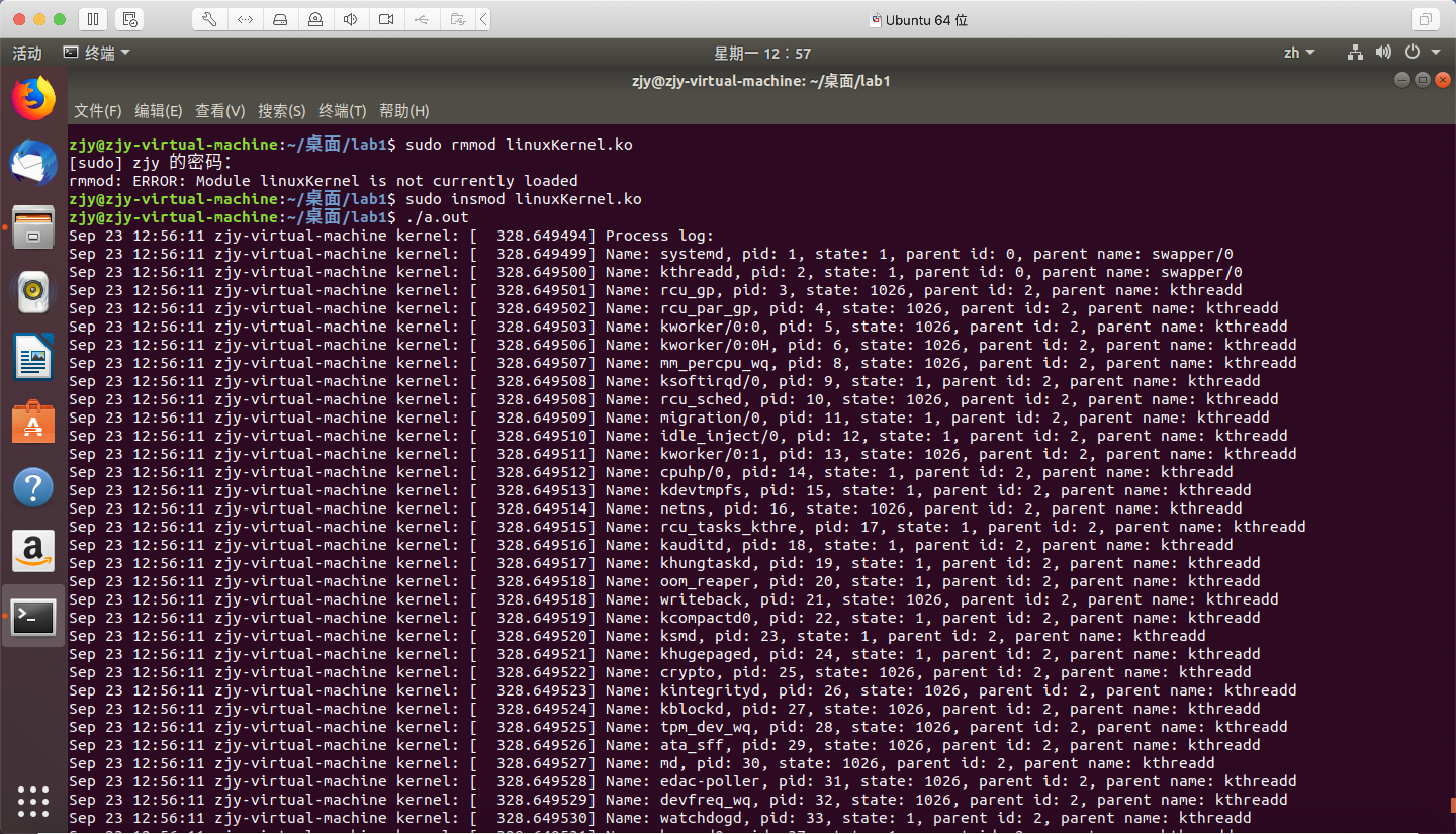
在init\_module中，先打印了一个”Process log:”的标志，为了用户态程序能够从日志中找到需要的进程统计信息。

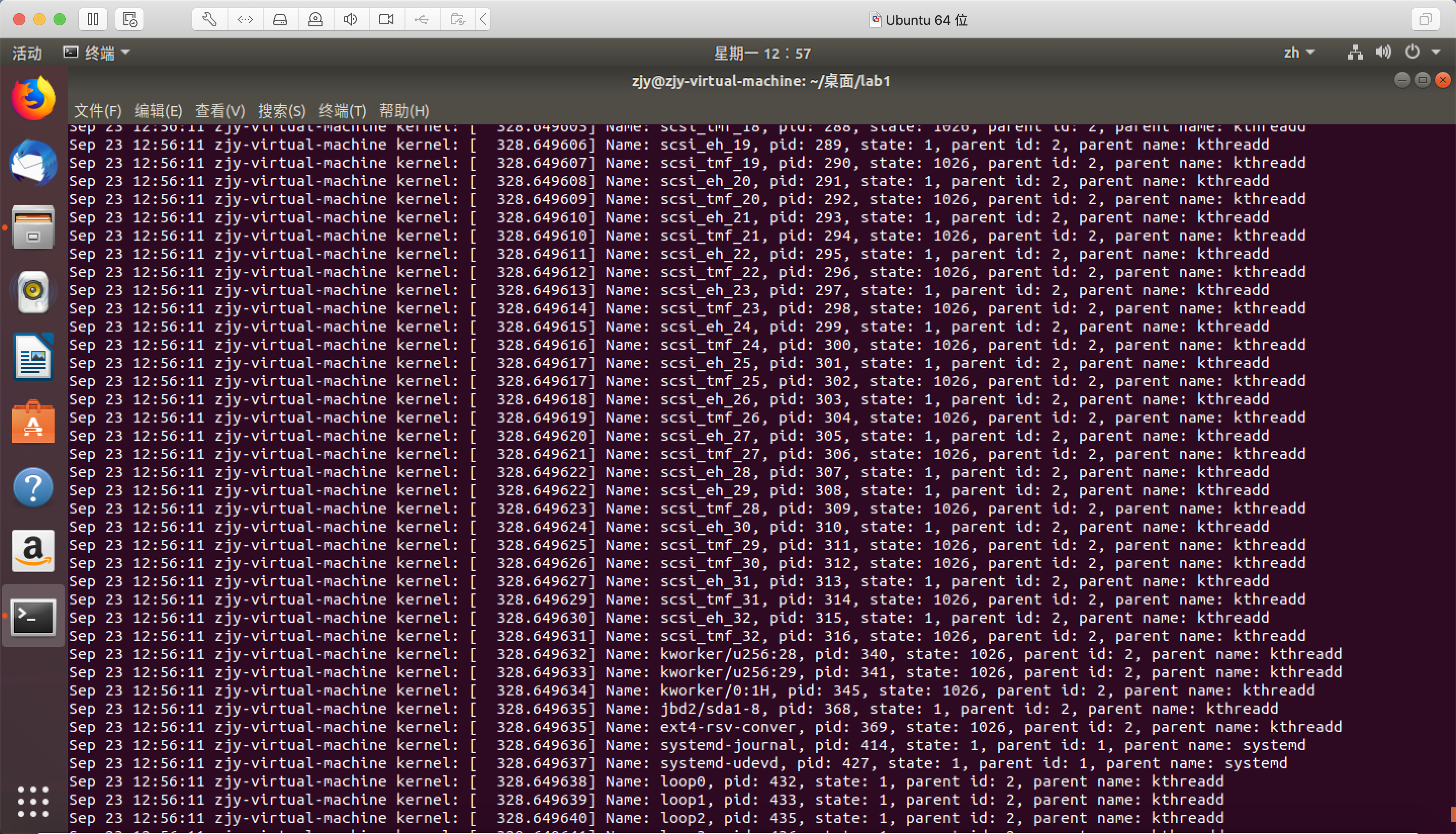
2.2.4 用户程序

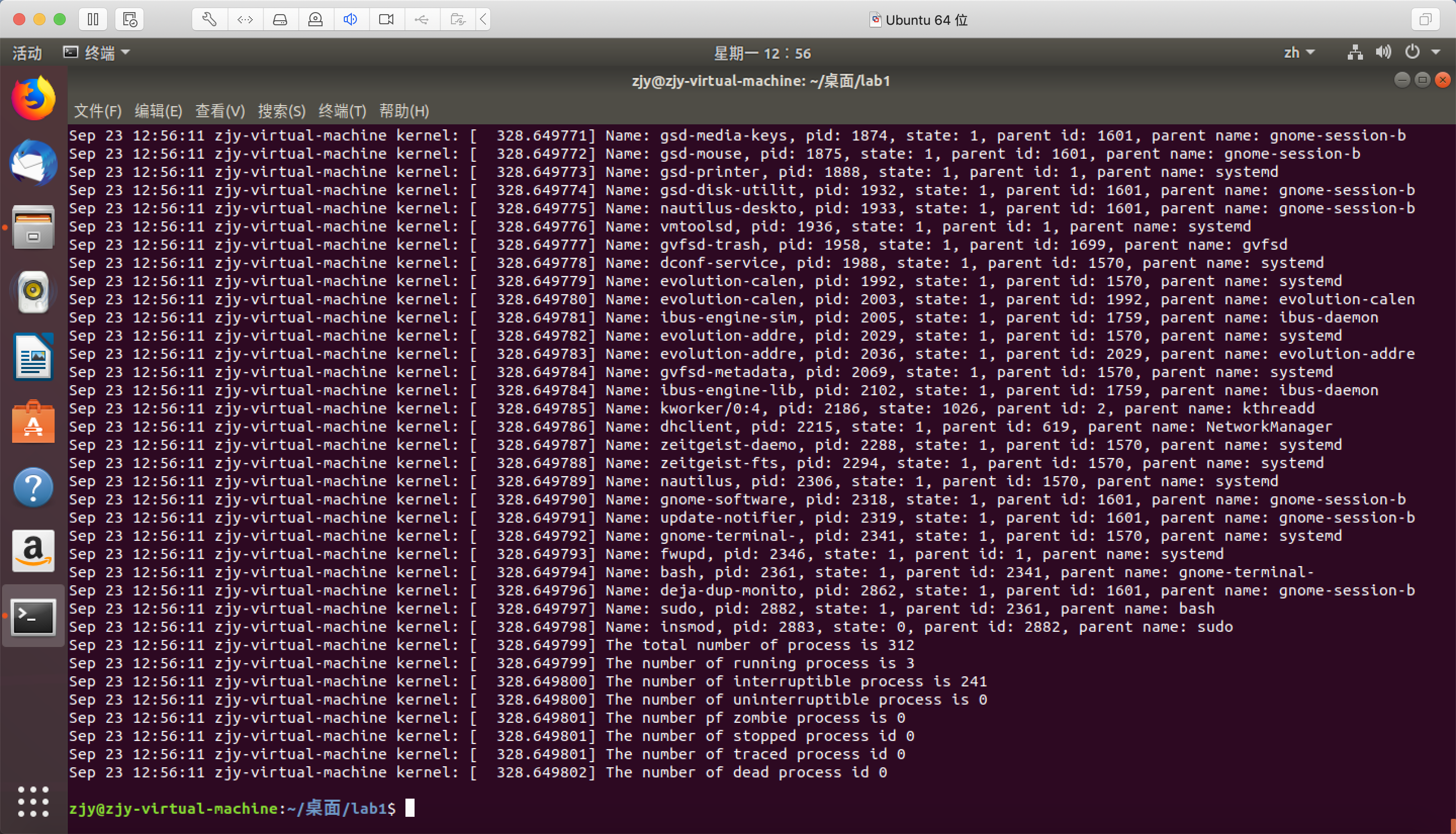
用户态下的程序是从/var/log/ kern.log (ubuntu)文件中读出内核模块输出的内容。首先，打开/var/log/ kern.log (ubuntu)文件。通过getline函数每次读取一行文件中的内容，通过find函数判断是否包含"Process log:"的提示信息，记录这个提示信息一共出现的次数count。再次打开/var/log/ kern.log (ubuntu)文件，每次读取文件中的一行，统计"Process log:"提示信息在当前出现的次数flag。当flag和count相等，说明下面的记录是需要打印的进程统计信息。于是向终端打印最后一次"Process log:"提示信息后出现的文件信息。

2.3 程序运行结果截图

1. $make编译获得内核模块linuxKernel.ko。
2. $sudo insmod linuxKernel.ko加载内核模块。
3. $g++ user.cpp编译获得可执行用户文件a.out。
4. $./a.out在当前目录下执行用户程序，得到运行进程统计结果。







2.4 源程序

2.4.1 内核模块

linuxKernel.c

|  |
| --- |
| #include <linux/module.h> //modules KERN\_INFO #include <linux/sched/signal.h> //struct task\_struct  /\*initilize the module\*/ int init\_module(void) {  printk("Hello!\n"); //mark to start count   int totalProcess = 0; //the total number of process  /\*the number of different state process\*/  int running = 0, interruptible = 0, uninterruptible = 0, zombie = 0, stopped = 0, traced = 0, dead = 0;   struct task\_struct \*process; //pointer pointing to process  for(process=&init\_task;(process=next\_task(process))!=&init\_task;) //iterate over the process list  {  /\*print the basic data\*/  printk(KERN\_INFO"Name: %s, pid: %d, state: %d, parent id: %d, parent name: %s",process->comm,process->pid,process->state,process->parent->pid,process->parent->comm);  if(process->exit\_state == 0) //the process is not dead yet   {  switch(process->state) //count the number of process in different states  {  case TASK\_RUNNING:  running++;  break;  case TASK\_INTERRUPTIBLE:  interruptible++;  break;  case TASK\_UNINTERRUPTIBLE:  uninterruptible++;  break;  case TASK\_STOPPED:  stopped++;  break;  case TASK\_TRACED:  traced++;  break;  default:  break;  }  }  else //the process is dead or zombie  {  switch(process->exit\_state)  {  case EXIT\_ZOMBIE:  zombie++;  break;  case EXIT\_DEAD:  dead++;  break;  default:  break;  }  }  totalProcess++;  }   /\*print the count result\*/  printk(KERN\_INFO"The total number of process is %d\n", totalProcess);  printk(KERN\_INFO"The number of running process is %d\n", running);  printk(KERN\_INFO"The number of interruptible process is %d\n", interruptible);  printk(KERN\_INFO"The number of uninterruptible process is %d\n", uninterruptible);  printk(KERN\_INFO"The number pf zombie process is %d\n", zombie);  printk(KERN\_INFO"The number of stopped process id %d\n", stopped);  printk(KERN\_INFO"The number of traced process id %d\n", traced);  printk(KERN\_INFO"The number of dead process id %d\n", dead);   return 0; }  /\*clean uo the module\*/ void cleanup\_module(void) {  printk("Good-bye!\n"); }  MODULE\_LICENSE("GPL"); |

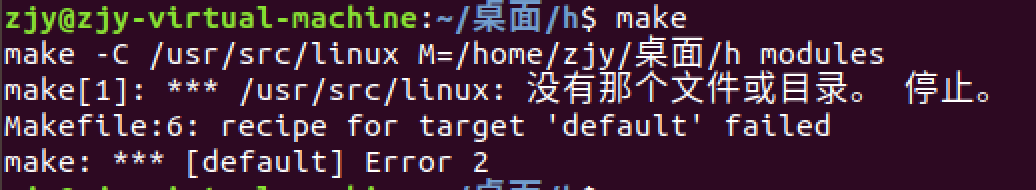
2.4.2用户程序

user.cpp

|  |
| --- |
| #include <iostream> #include <fstream> #include <string>  using namespace std; #define MAXLENGTH 1000  int main() {  /\*The program in user mode reads out the output of the kernel module   \*from the / var / log / kern. log (ubuntu) file  \*/  string file = "/var/log/kern.log";  string mark = "Process log:"; //mark in the module    int count = 0; //count the number of hello occur   char buffer[MAXLENGTH]; //store the one line log information  string temp;   fstream fs;  fs.open(file, ios::in);    while(!fs.eof())  {  fs.getline(buffer,MAXLENGTH,'\n');  temp = buffer;  if(temp.find(mark)!=string::npos)  {  count++; //if hello occurs, count is added by 1  }  }  fs.close();   int flag = 0;  fs.open(file, ios::in);  while(!fs.eof())  {  fs.getline(buffer,MAXLENGTH,'\n');  temp = buffer;  if(temp.find(mark)!=string::npos)  {  flag++;  }  if(flag == count) //if meet the last hello print the information  {  cout<<buffer<<endl;  }  }  fs.close();   return 0;  } |

1. 讨论、心得（20分）
2. 写好helloworld.c的程序后，编写makefile文件。根据《边干边学——Linux内核指导》，KDIR设置为KDIR=/usr/src/linux

但是在编译时发现出现了错误。



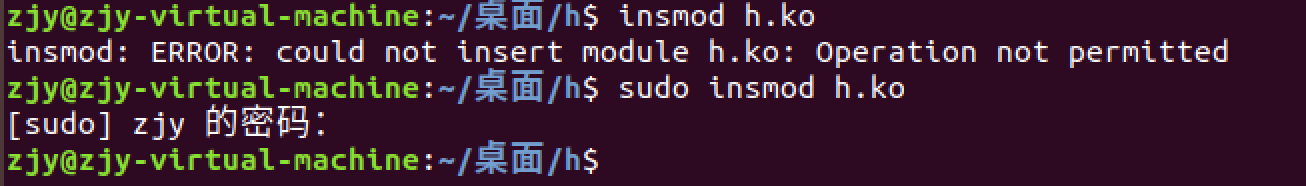
随后在上课听讲后，将KDIR修正为

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

make -C $(KDIR) M=$(PWD) modules

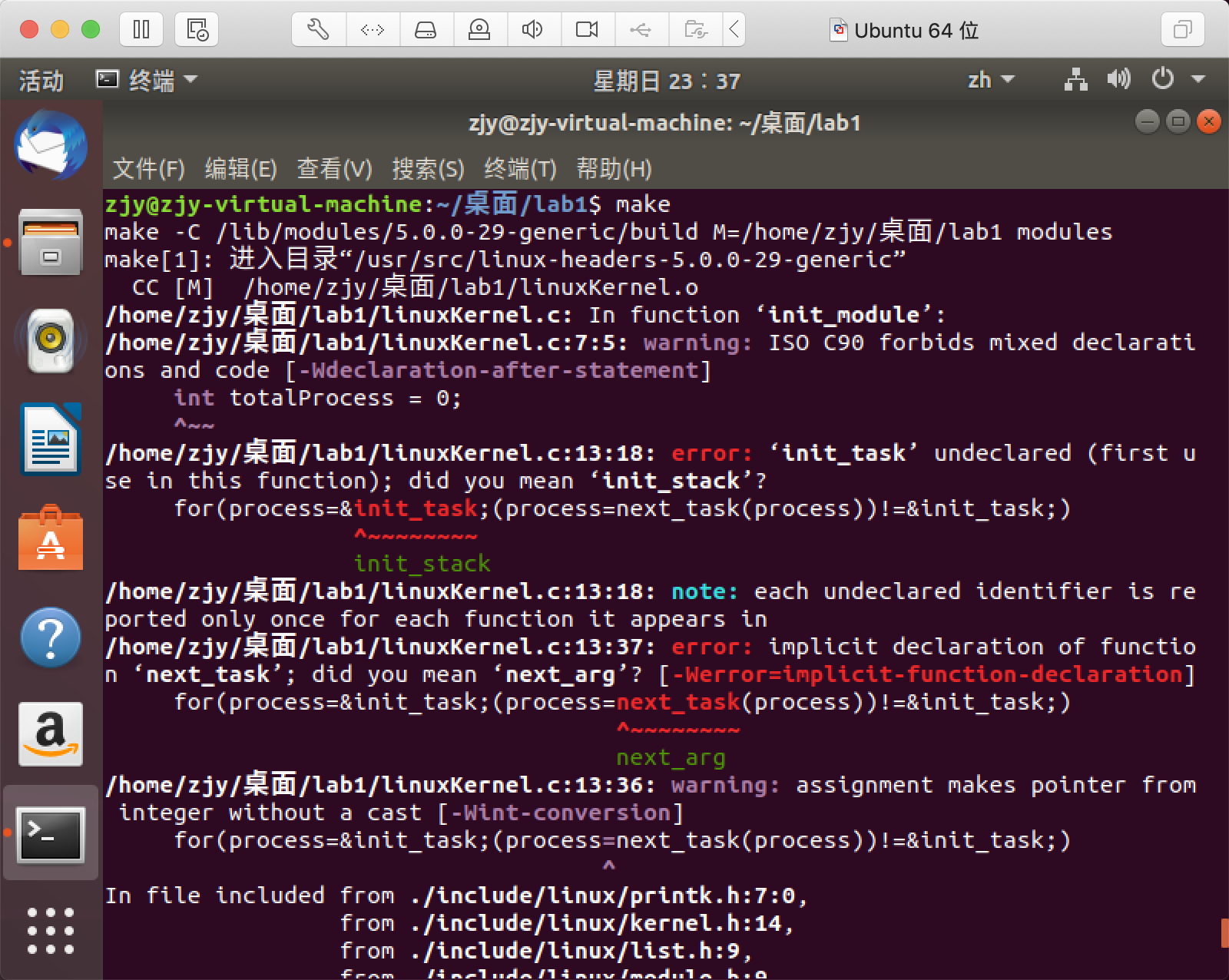
该命令中-C选项的作用是指将当前的工作目录转移到指定的目录（KDIR），程序到当前目录（pwd）查找模块源码，将其编译，生成.ko文件。

1. 直接使用insmod执行内核模块失败。需要在命令前面加上sudo执行超级用户权限。

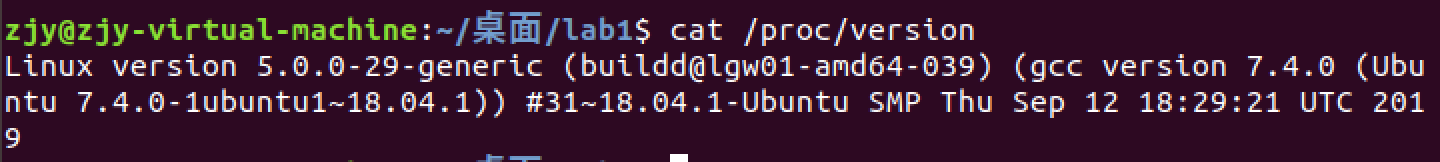


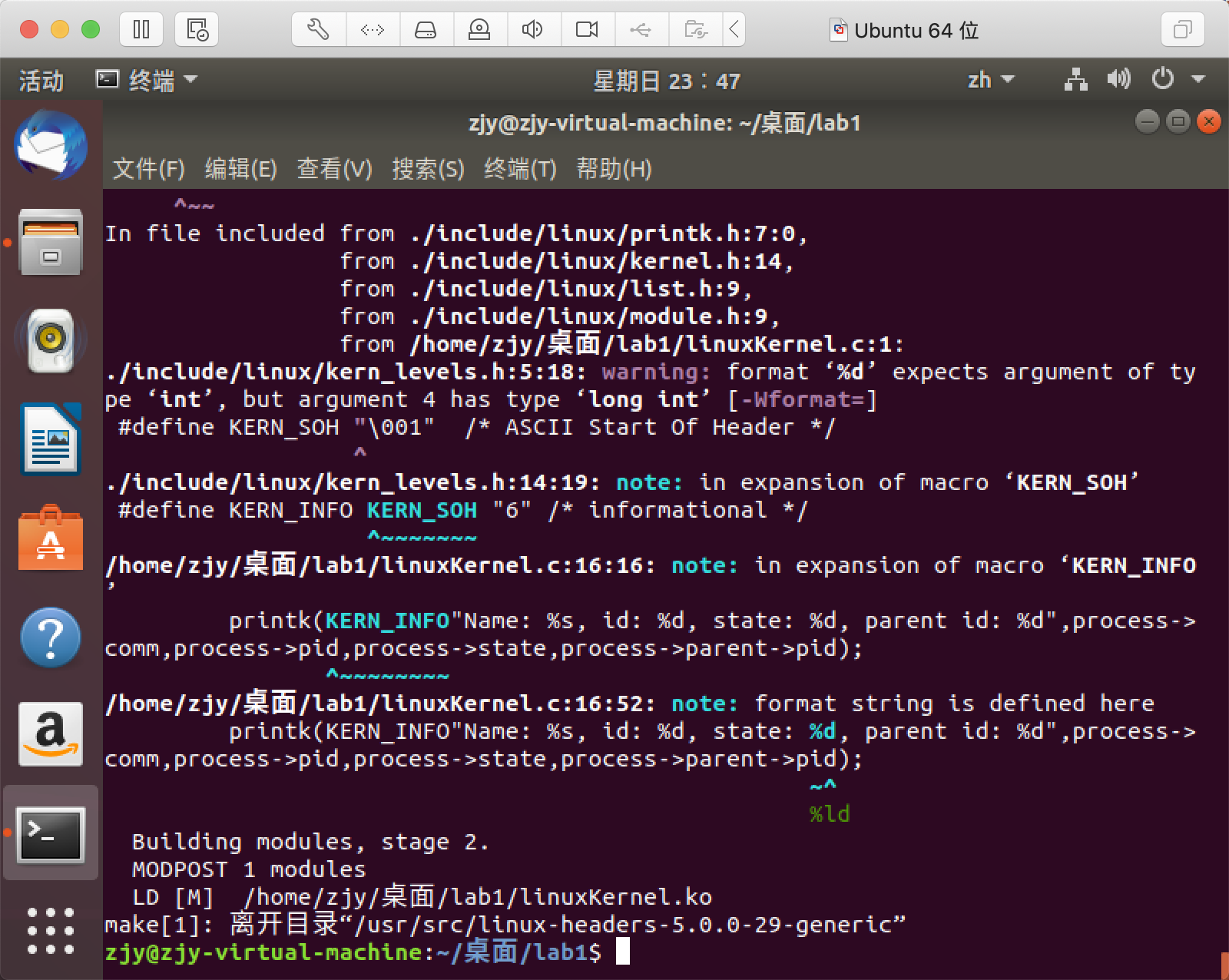
* 在dmesg后看到了fig:的信息。

1. 在执行makefile时，会出现下面的错误。



根据提示，init\_*task*在使用前没有被定义。但*init\_*task属于内核代码段，说明没有将正确的Linux源文件包含进去。查阅发现4.11以后，该方法都放在了include/linux/sched/signal.h中。修改后，编译成功。





1. 一开始以为helloworld.ko模块加载到内核中就会马上在终端打印一个"hello world"，发现并没有，后才才意识到要有用户程序去调用这个内核模块，才会执行这个模块中的代码，在终端上打印"hello world"。
2. 死锁检测要有单独的线程。
3. 编译线程函数最后要加上-lpthread。