# 3170103240-张佳瑶-实验一

# 1 实验要求

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 实验指导

1．如何编写内核模块程序及编译、安装内核模块，可以参考“边干边学—Linux内核指导”教材第13章。

2．每个进程的进程名字、pid、进程状态、父进程的指针等在task-struct结构的字段中。在内核中使用printk函数打印有关变量的值。遍历进程可以使用next*task宏，init*task进程为0号进程。task-struct结构参阅“边干边学—Linux内核指导”教材11.2节；遍历进程方法可以参阅“边干边学—Linux内核指导”教材11.6节。

3．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。

# 3 实验设计

## 3.1 内核模块

### 3.1.1 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。

### 3.1.2 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。

### 3.1.3 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\T退出状态ASK*DEAD\EXIT\_ZOMBIE。

struct task\_struct中:

1. 成员comm[TASK*COMM*LEN]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。

## 3.2 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。

# 4 实验源程序

## 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");

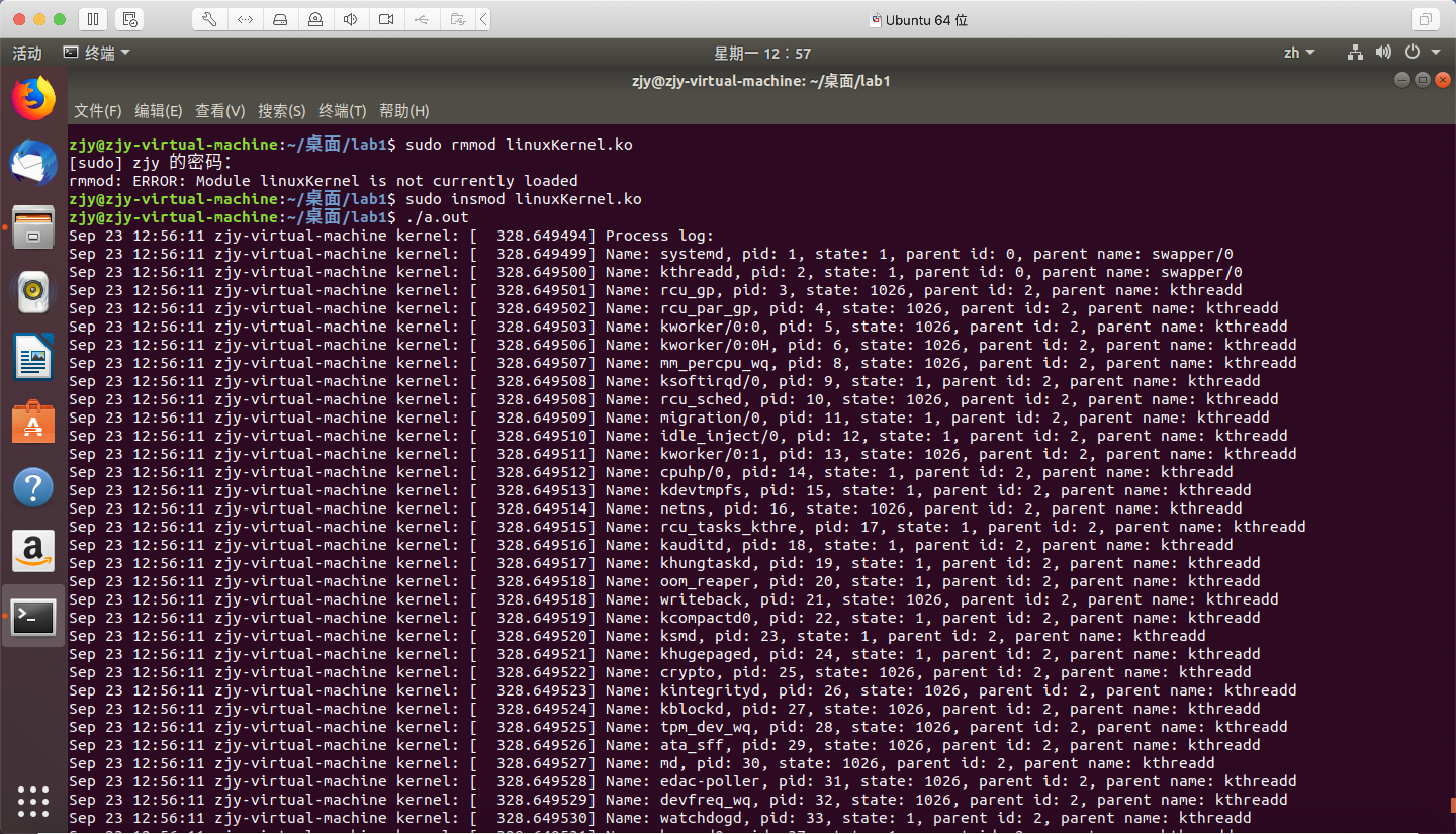
## 4.2 用户程序

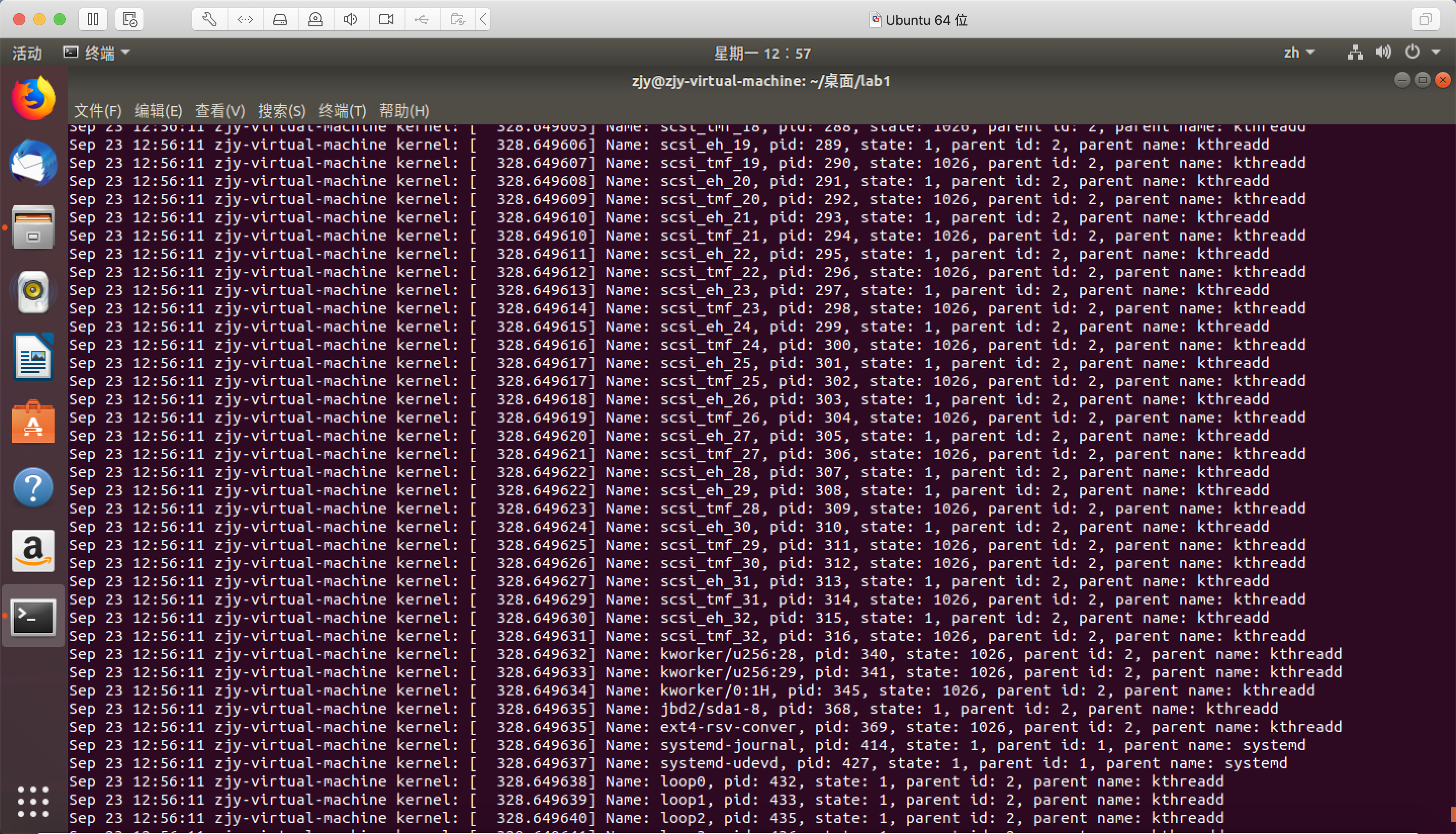
user.cpp

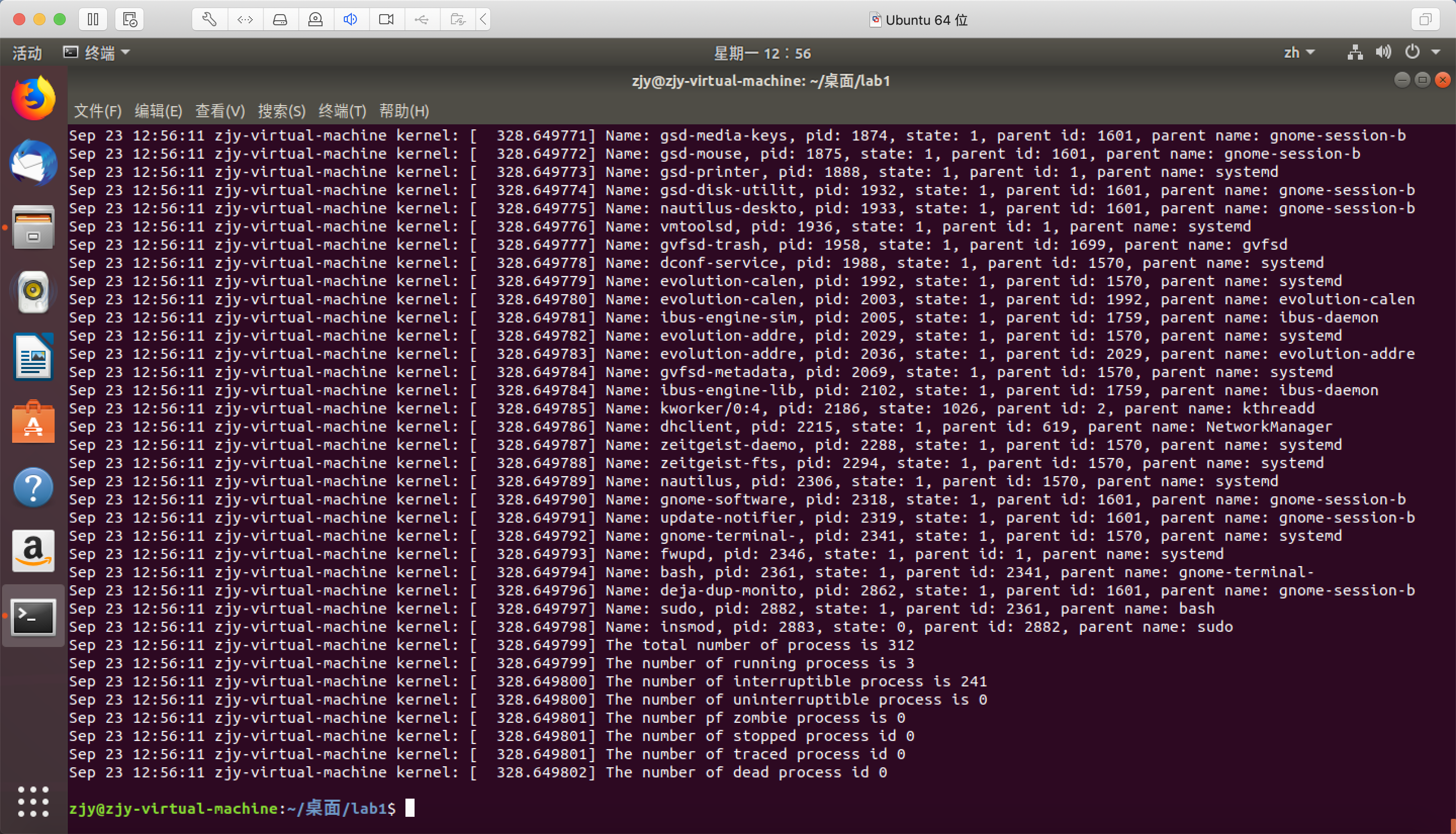
#include <iostream>  
#include <fstream>  
#include <string>  
  
using namespace std;  
#define MAXLENGTH 1000  
  
int main()  
{  
 /\*The program in user mode reads out the output of the kernel module   
 \*from the / var / log / kern. log (ubuntu) file  
 \*/  
 string file = "/var/log/kern.log";  
 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;   
}

# 5 实验运行结果截图

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。





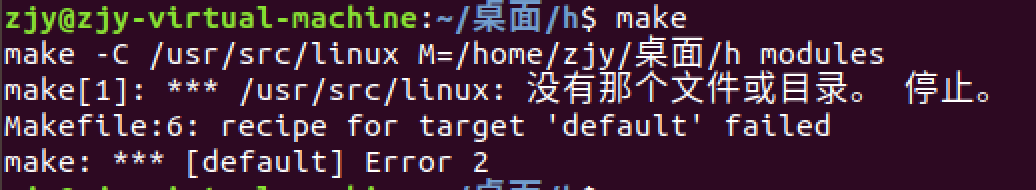


# 6 实验讨论与心得

1. 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。



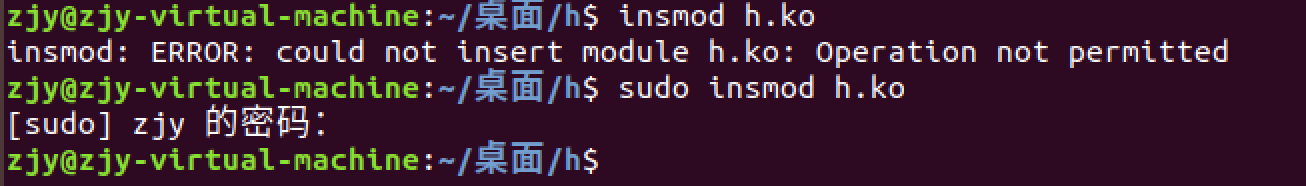
随后在上课听讲后，将KDIR修正为

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

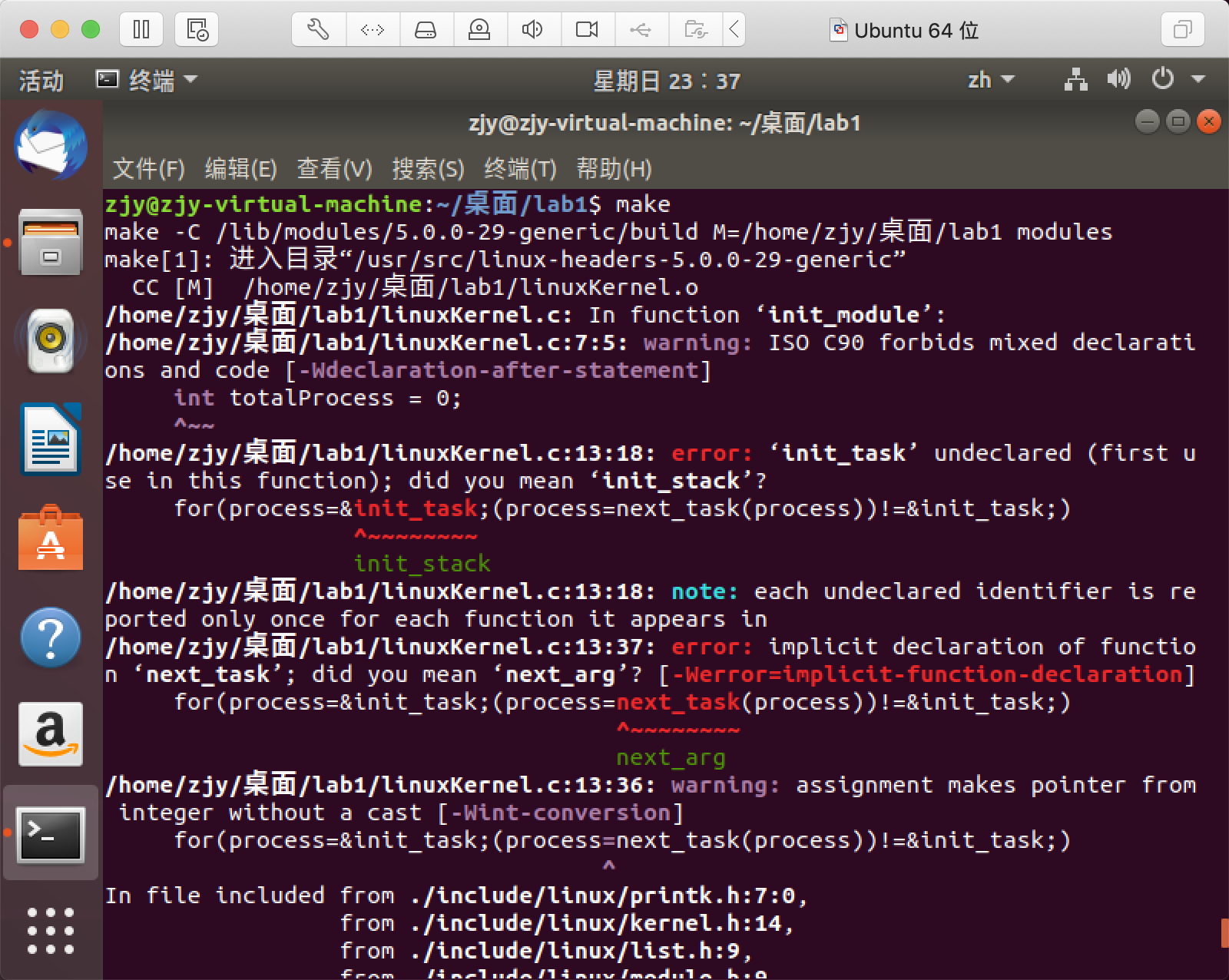
make -C $(KDIR) M=$(PWD) modules

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。

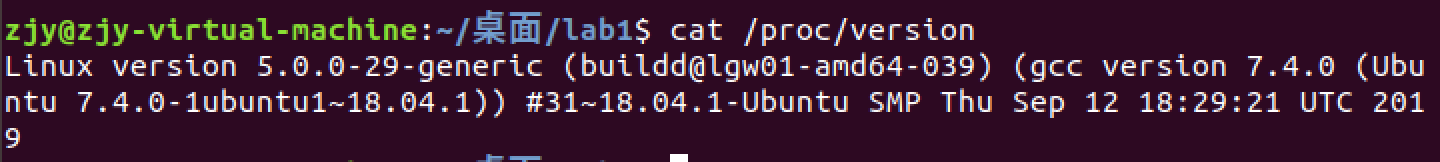
1. 直接使用insmod执行内核模块失败。需要在命令前面加上sudo执行超级用户权限。

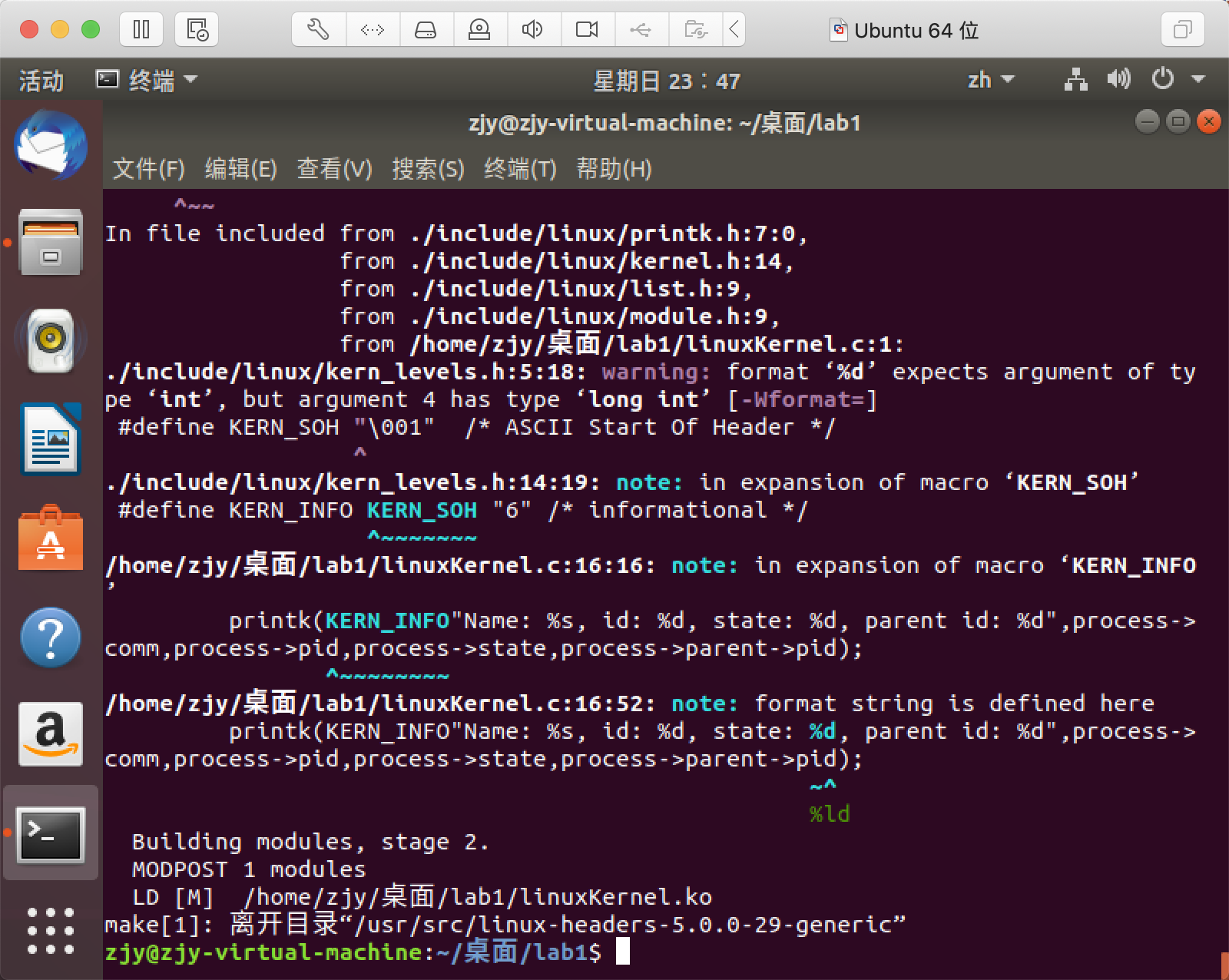
* 
* 在dmesg后看到了fig:的信息。

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"。