**武汉大学国家网络安全学院**

**实验报告**

**课程名称 操作系统**

**专业年级 信安21级**

**姓名学号 赵伯俣2021302181156**

**张宇哲2021302181162**

**协 作 者 无**

**实验学期 2022-2023　 学年 第二 学期**

**课堂时数 32 课外时数**

**填写时间 2023 年 4 月 14 日**

**项目一**

**UNIX外壳和历史特征**

一、实验目的

设计一个C语言的外壳接口，可以接受用户的命令并在单独的进程中执行用户的命令。

二、实验环境

在本次实验中使用到了虚拟机进行程序的编写和运行。虚拟机的配置如下。

Vmware16

ubuntu-22.04.2-desktop-amd64.iso

使用的Linux内核为linux-6.2.8

三、实验源代码

本实验的源代码内容如下。

#include "unistd.h"

#include <fcntl.h>

#include <pwd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h> // 提供类型pid\_t的定义

#include <sys/wait.h>

#define MAX\_LINE 80 // 每一个命令的最多字符数为80个

char \*args[MAX\_LINE / 2 + 1]; // 用户输入标记最多为40个

int should\_run = 1;           // 值为1时main一直循环，知道为0时退出

struct History

{

    char \*args[MAX\_LINE / 2 + 1];

};

History h[10]; // 存放历史命令

int num = 1;

int assume(char \*a[]);

void process(int i);

void play(int i) // 输出需要调用的命令

{

    int count2 = 0;

    while (h[i].args[count2])

    {

        printf("%s ", h[i].args[count2++]);

    }

    printf("\n");

}

int main(void)

{

    while (should\_run)

    {

        printf("osh>"); // 设置提示符为osh>

        fflush(stdout);

        int i = 0, count = 1;

        char c;

        do

        {

            char \*s = (char \*)malloc(128);

            scanf("%s", s); // 输入空格或者回车换行结束

            c = getchar();

            args[i] = s;

            h[num % 10].args[i] = s;

            i++;

            if (strcmp(args[0], "exit") == 0)

            {

                should\_run = 0;

            } // 用户在提示符后面输入exit后，should\_run为0并且终止；

            if (strcmp(args[i - 1], "&") == 0)

            {

                i--;

            }                                    // 输入& 显示waitpid：success

            if (strcmp(args[0], "history") == 0) // 列出历史指令

            {

                int count2 = 0;

                if (!h[count].args[0])

                {

                    printf("No commands in history");

                }

                while (h[count].args[count2])

                {

                    count++;

                }

                count--;

                while (h[count].args[count2])

                {

                    while (h[count].args[count2])

                    {

                        printf("%d %s ", count, h[count].args[count2]);

                        count2++;

                    }

                    printf("\n");

                    count2 = 0;

                    count--;

                }

            }

            if (strcmp(args[0], "!!") == 0) // 列出上一条指令并执行

            {

                if (!h[num - 1].args[0])

                {

                    printf("No such command in history");

                }

                else

                {

                    play(num - 1);

                    process(num - 1);

                }

            }

        } while (c == ' ');

        args[i] = NULL;

        h[num % 10].args[i] = NULL; // 数组的结束位置

        assume(args);

        process(num);

        num++;

    }

    return 0;

}

void process(int i) // 通过父子进程的方式执行相关的指令

{

    pid\_t id = fork(); // 创建子进程，父进程和子进程之间共享代码段（建立相同的程序副本）

    if (id < 0)

    {

        perror("fork"); // 抛出异常

    }

    if (id == 0)

    {

        execvp(h[i].args[0], h[i].args); // exec系列的一种，用于运行新程序

        exit(1);                         // 调用exit（）来终止程序

    }

    else

    {

        int status = 0;

        pid\_t ret = waitpid(id, &status, 0); // 等待子进程结束

        if (ret > 0 && WIFEXITED(status))

        {

        }

        else

        {

            perror("waitpid");

        }

    }

}

void row(int i) // 列出对应行的指令并执行

{

    if (!h[i].args[0])

    {

        printf("No such command in history");

    }

    else

    {

        play(i);

        process(i);

    }

}

int assume(char \*args[]) // 判断是列出哪一行的指令

{

    pid\_t id;

    if (strcmp(args[0], "!1") == 0)

    {

        row(1);

    }

    if (strcmp(args[0], "!2") == 0)

    {

        row(2);

    }

    if (strcmp(args[0], "!3") == 0)

    {

        row(3);

    }

    if (strcmp(args[0], "!4") == 0)

    {

        row(4);

    }

    if (strcmp(args[0], "!5") == 0)

    {

        row(5);

    }

    if (strcmp(args[0], "!6") == 0)

    {

        row(6);

    }

    if (strcmp(args[0], "!7") == 0)

    {

        row(7);

    }

    if (strcmp(args[0], "!8") == 0)

    {

        row(8);

    }

    if (strcmp(args[0], "!9") == 0)

    {

        row(9);

    }

    if (strcmp(args[0], "!10") == 0)

    {

        row(0);

    }

    return 0;

}

四、实验原理

1.main函数。

该程序中main函数的作用是在should\_run值为1的情况下不断进行循环读取用户输入的命令，在用户输入时定义args数组存储用户输入的指令，每一个数组存取一条指令，数组中的每一个变量存取一条指令的各个部分，例如指令cat prog.c在args数组中保存为args[0]=cat,args[1]=prog.c。然后将指令存入History型变量h数组中，h数组保存着历史指令，方便后续读取历史操作。

之后对args中的指令进行识别，如果指令为exit退出指令，则将should\_run置为0，使得程序退出，不再接受新的指令。

如果指令为history读取历史指令，则将h数组中保存的全部指令以及其序号进行输出（play函数）。从而实现显示历史指令的作用。

如果指令为！！，则将h中的最近一条指令输出（play函数）并且执行（process函数），实现执行最近历史指令的作用。

如果args中的指令均不满足上面的操作则将args数组的末尾指针指向NULL，调用assume函数实现！加上整数N实现执行最近的第N条指令的操作。

2.process函数

process函数的作用是创建子进程，实现父子进程之间的共享代码段，在子进程中调用execvp函数来执行当前需要执行的命令，令父进程等待子进程结束后再退出。

3.assume函数

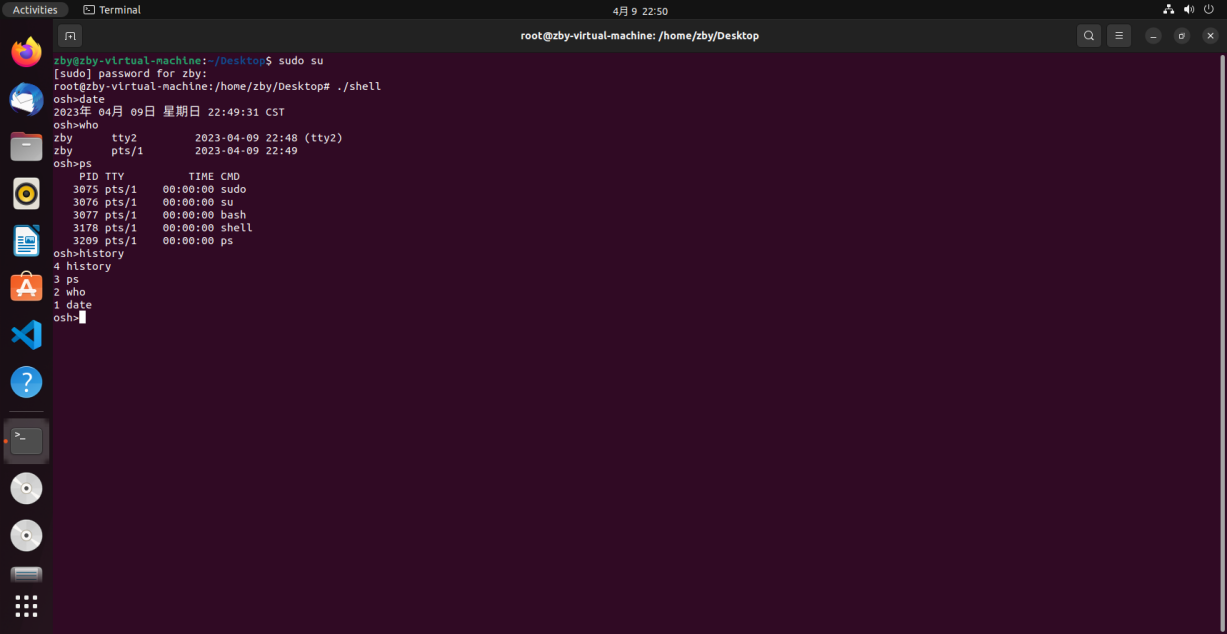
sssume函数的作用是判断传递进来的命令是否为！加上整数N的组合，并根据N的不同调用不同的row将需要调用第几条指令传递给row函数。

4.row函数

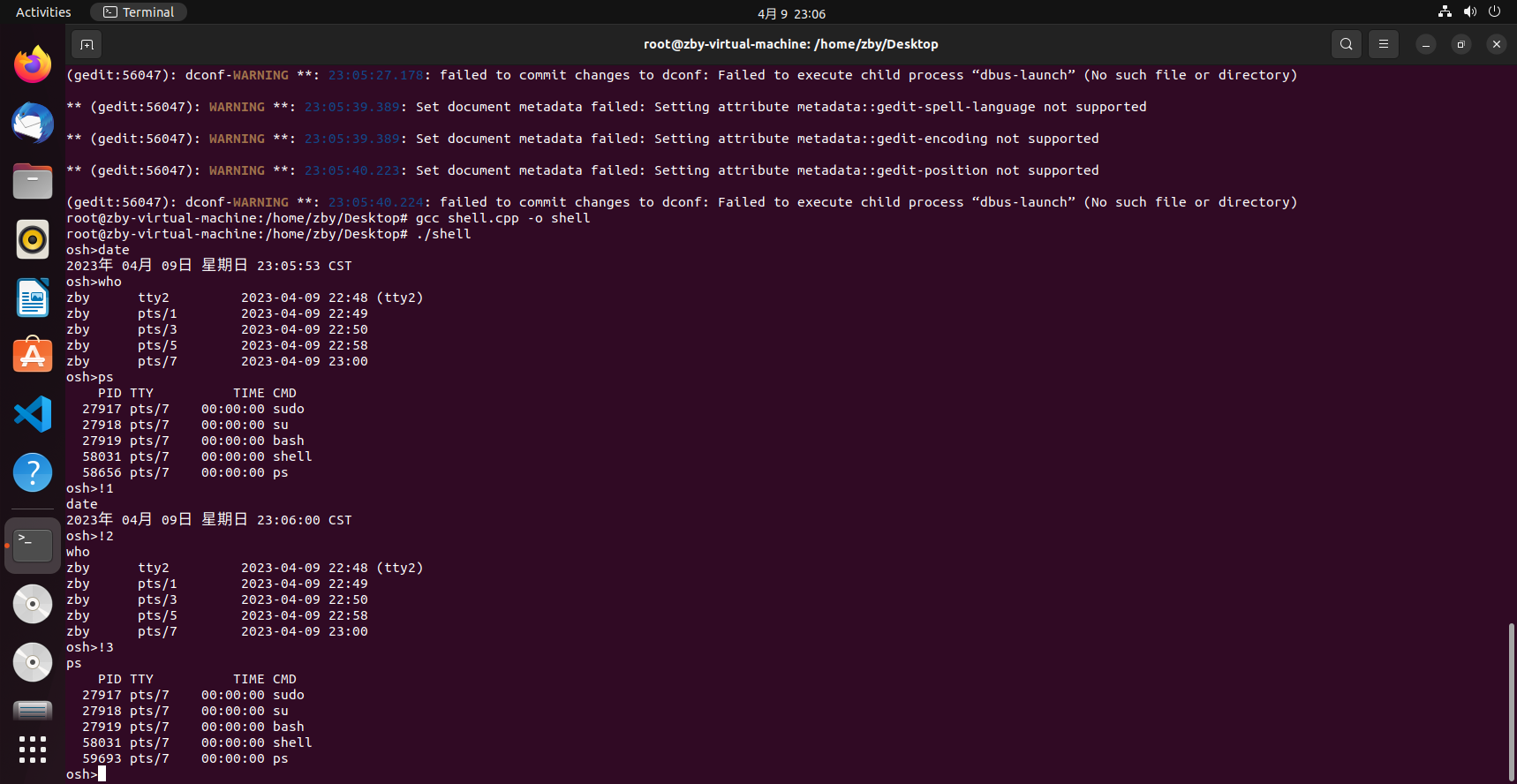
row函数的作用是输出需要调用的指令，并且执行该指令。

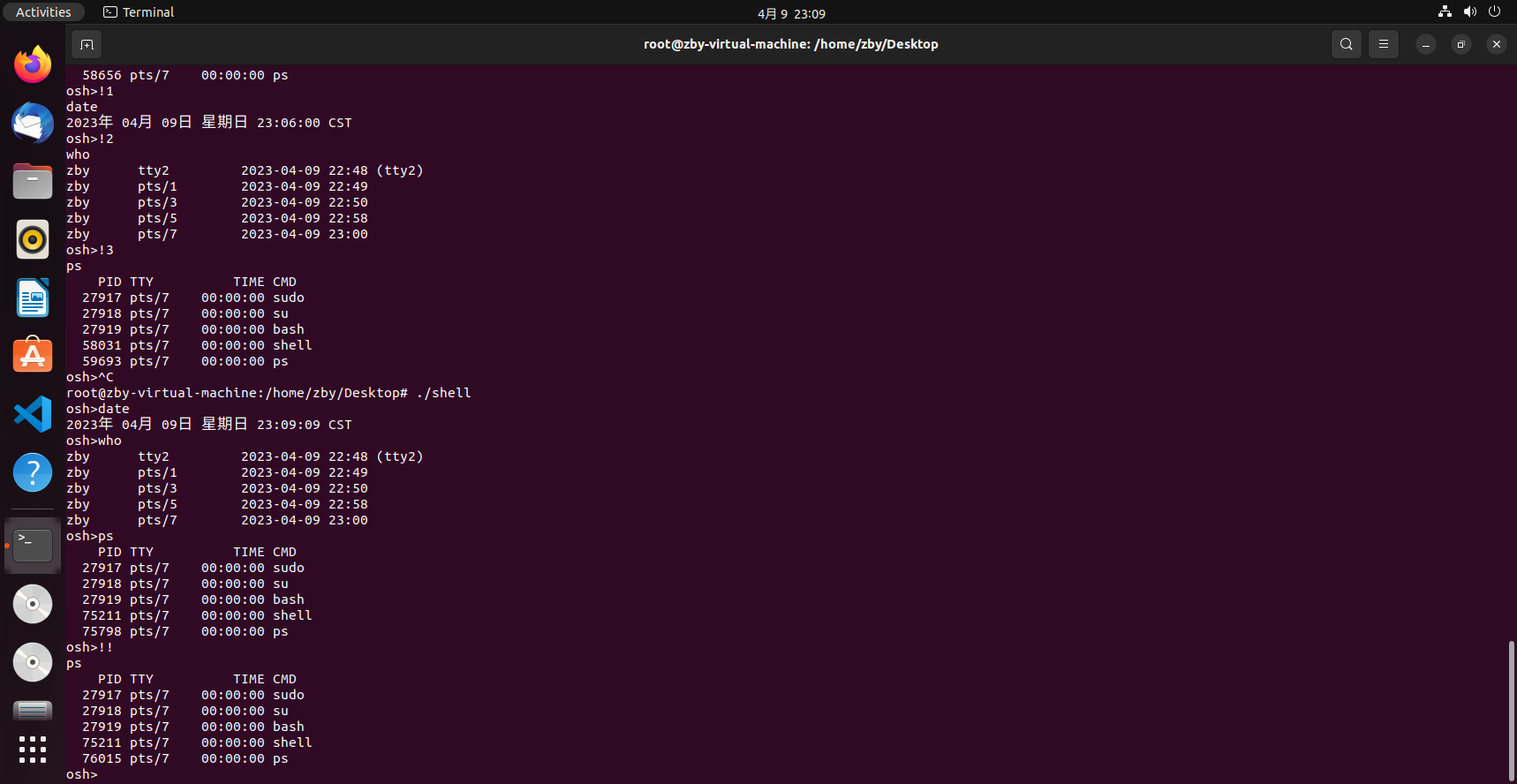
五、实验结果

1.按照顺序执行命令date、who、ps、history结果如下图所示，程序能够正确执行各个命令并且在使用history命令时可以按照题目要求的顺序将历史指令显示出来。



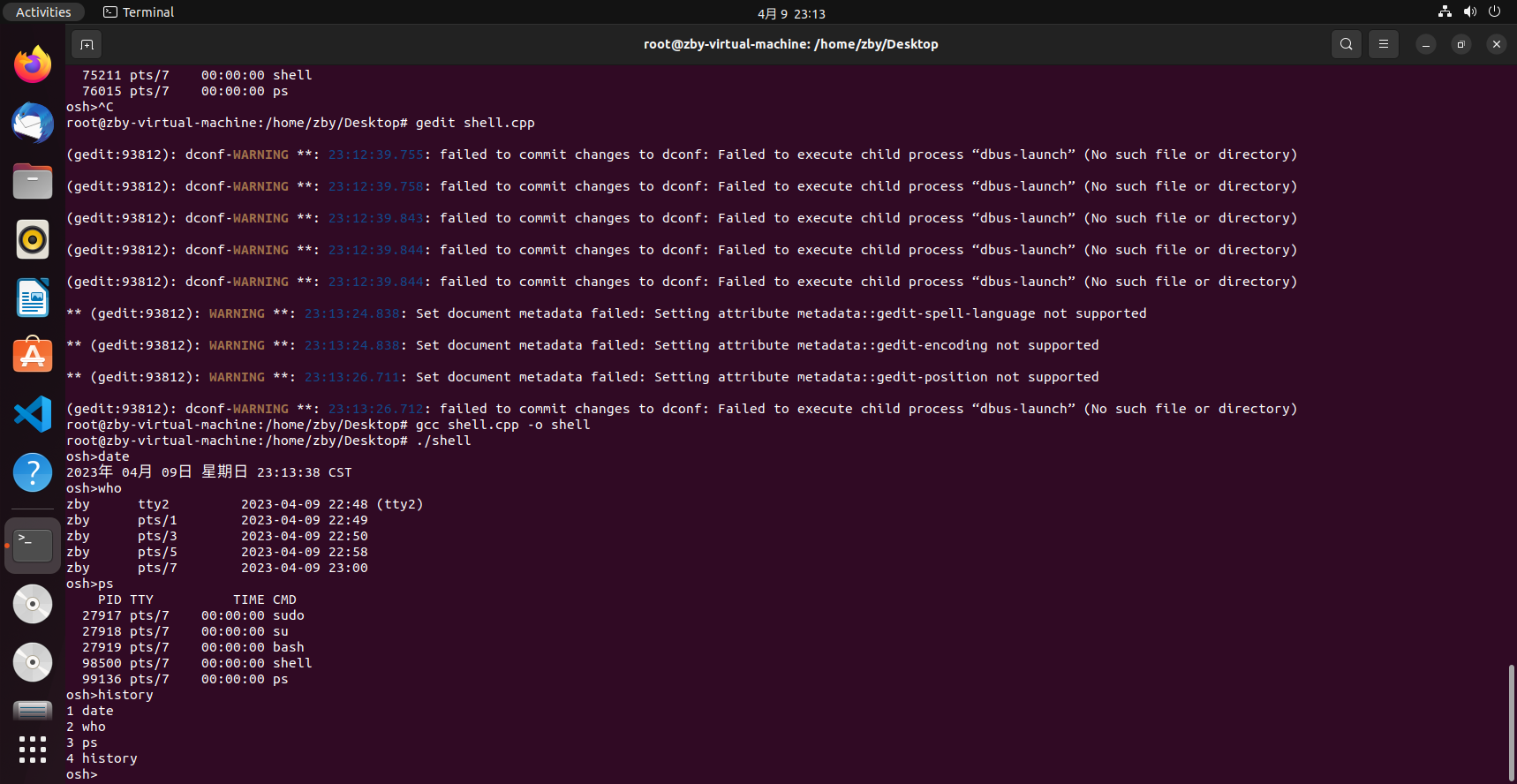
2.按照顺序输入命令date、who、ps后输入命令！1，！2，！3得到的输出如下图所示，该程序能够正确的执行！加上一个整数N的命令，并且能够将正在执行的命令回显到屏幕上。



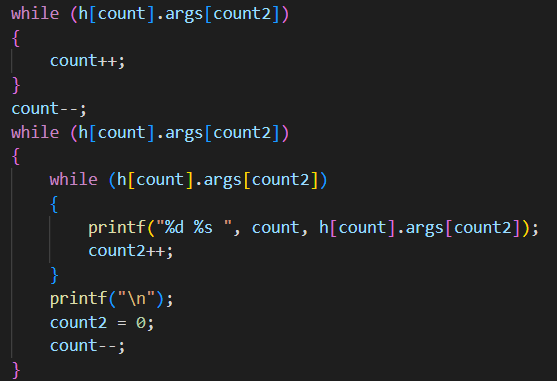
3.按照顺序输入命令date、who、ps、再输入命令！！进行执行上一条指令的操作，操作结果如下图所示，该程序能够正确的执行！！命令，实现执行上一条指令的命令。

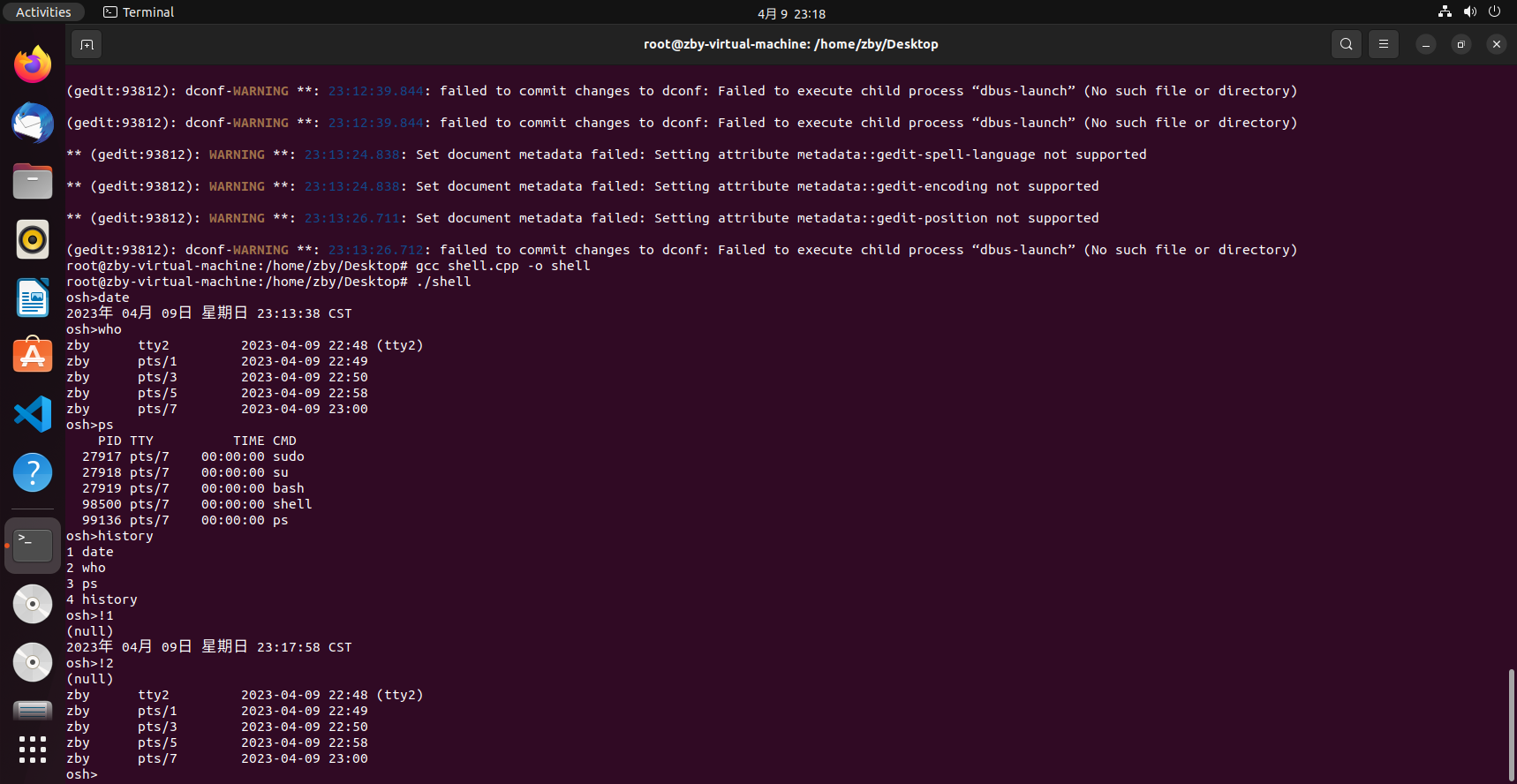
六、实验中出现的bug及其修改方式

1.在使用history命令时，虽然指令的顺序逆序，应当按照从最近到最远的顺序排列。错误结果如下图所示。

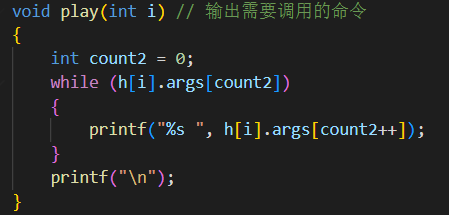


解决方法为在main函数中添加一个模块，首先计数用户一共输入了多少条指令，再逆序输出h数组即可满足题目中history命令的要求。



2.在执行！加上整数N的指令时回显不出正确的指令只能回显（null），错误输出如下所示。

解决方法为更正play函数中count2的自加次数，在错误程序中count2自加两次导致无法输出正确的结果。



**实验二**

**用于生成任务列表的linux内核模块**

一、实验目的

1.设计一个内核模块，通过宏for\_each\_process()，迭代系统的所有任务。

2.设计一个内核模块，采用DFS树，从任务init开始遍历系统内的所有任务。

二、实验环境

在本次实验中使用到了虚拟机进行程序的编写和运行。虚拟机的配置如下。

Vmware16

ubuntu-22.04.2-desktop-amd64.iso

使用的Linux内核为linux-6.2.8

三、实验原理

1.模块1。

在my\_module\_init内核模块加载函数中定义task\_struct类型的变量task,直接调用for\_each\_process宏输出每一个系统进程的任务名称name、状态status、进程标识符pid、任务名称调用task中的变量comm，任务的状态调用task变量中的stats，进程的标识符pid调用task变量中的pid。

在my\_module\_exit内核卸载模块中只需要输出Exiting my\_module.即可。

2.模块2。

（1）实现DFS。在改程序中，使用堆栈来实现DFS，定义堆栈为st通过list\_for\_each来得到该子节点的所有子节点，将这些得到的节点继续压栈，那么出栈的就是子节点的子节点，以此实现DFS。

（2）实现双向链表。通过使用类型list\_head定义双向链表，并生成list1和list2两个双向链表类型的变量。

（3）my\_module\_init设计思路。首先定义堆栈，在栈不空时进行循环，每一次循环都执行宏list\_for\_each\_prev利用list\_entry得到该链表节点的父数据结构，并将该节点入栈，然后使用宏list\_for\_each在得到当前节点的父数据结构之后再将其进程名，进程标识符还有进程的状态进行输出

3.实验过程。

（1）首先将实验所需的两个文件代码文件和Makefile文件放入同一文件夹中，在该目录下直接执行make指令，完成编译过程。

（2）之后检查在文件夹中是否生成了.ko文件，之后进行命令insmod 代码文件名.ko，对内核模块进行加载，在终端中不会看到输出，使用命令dmesg查看内核缓冲日志即可得到该模块的正确输出。

（3）之后对内核模块进行卸载，执行命令rmmod 代码文件名最后检查内核缓冲日志查看是否输出句子“Exiting my\_module.”。

四、实验源码

1.实验中采用的内核模块1的源码如下所示。

#include <linux/init.h>

#include <linux/init\_task.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched.h>

MODULE\_LICENSE("GPL");

static int \_\_init my\_module\_init(void)//模块的加载

{

    struct task\_struct \*p;

    p = NULL;

    p = &init\_task;

    printk(KERN\_INFO "ListTask module loaded\n");

    printk(KERN\_INFO "PID\tStatus                    Name\n");//列出表头方便显示

    for\_each\_process(p)

    {

        printk(KERN\_INFO "%d\t%ld\t%s\n", p->pid, p->stats, p->comm);//依次列出每一个进程的进程号，状态以及进程名称

    }

    return 0;

}

static void \_\_exit my\_module\_exit(void)//模块的卸载

{

    printk(KERN\_INFO "Exiting my\_module.\n");

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

2.在模块一中编译过程中使用的Makefile文件源码如下所示。

obj-m += mod\_1.o

all:

 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

3.模块2的源码如下所示。

#include <linux/init.h>

#include <linux/init\_task.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched.h>

#include <linux/types.h>

#define MAXN 256

MODULE\_LICENSE("GPL");

struct stack

{

    struct task\_struct \*data[MAXN];

    int head;

} st; // 定义堆栈结构变量

int st\_size(void)

{

    return st.head;

}

bool st\_empty(void)

{

    return st.head == 0;

}

bool st\_full(void)

{

    return st.head == MAXN;

}

struct task\_struct \*st\_pop(void) // 定义出栈函数

{

    if (st\_empty())

        return NULL;

    st.head -= 1;

    return st.data[st.head];

}

bool st\_push(struct task\_struct \*t) // 定义入栈函数

{

    if (st\_full())

        return false;

    st.data[st.head++] = t;

    return true;

}

static int my\_module\_init(void) // 加载模块函数

{

    st.head = 0;

    st\_push(&init\_task);

    struct task\_struct \*tsk, \*p;

    struct list\_head \*list, \*list2;

    while (!st\_empty()) // 当每次栈不空时进行循环

    {

        tsk = st\_pop();

        if ((&tsk->children)->next != (&tsk->children))

            printk(KERN\_INFO "The child processes of %s (with pid:%d) is:\n", tsk->comm, tsk->pid);

        list\_for\_each\_prev(list, &tsk->children)

        {

            p = list\_entry(list, struct task\_struct, sibling);

            st\_push(p);

        }

        list\_for\_each(list2, &tsk->children)

        {

            p = list\_entry(list2, struct task\_struct, sibling);

            printk(KERN\_INFO "name [%s] status [%d] pid [%d]\n", p->comm, p->stats, p->pid); // 输出进程的名称，状态以及进程号

        }

    }

    return 0;

}

static void \_\_exit my\_module\_exit(void) // 进程的卸载函数

{

    printk(KERN\_INFO "Exiting my\_module.\n");

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

4.模块2在编译过程中所使用的Makefile文件如下所示。

obj-m += mod\_2.o

all:

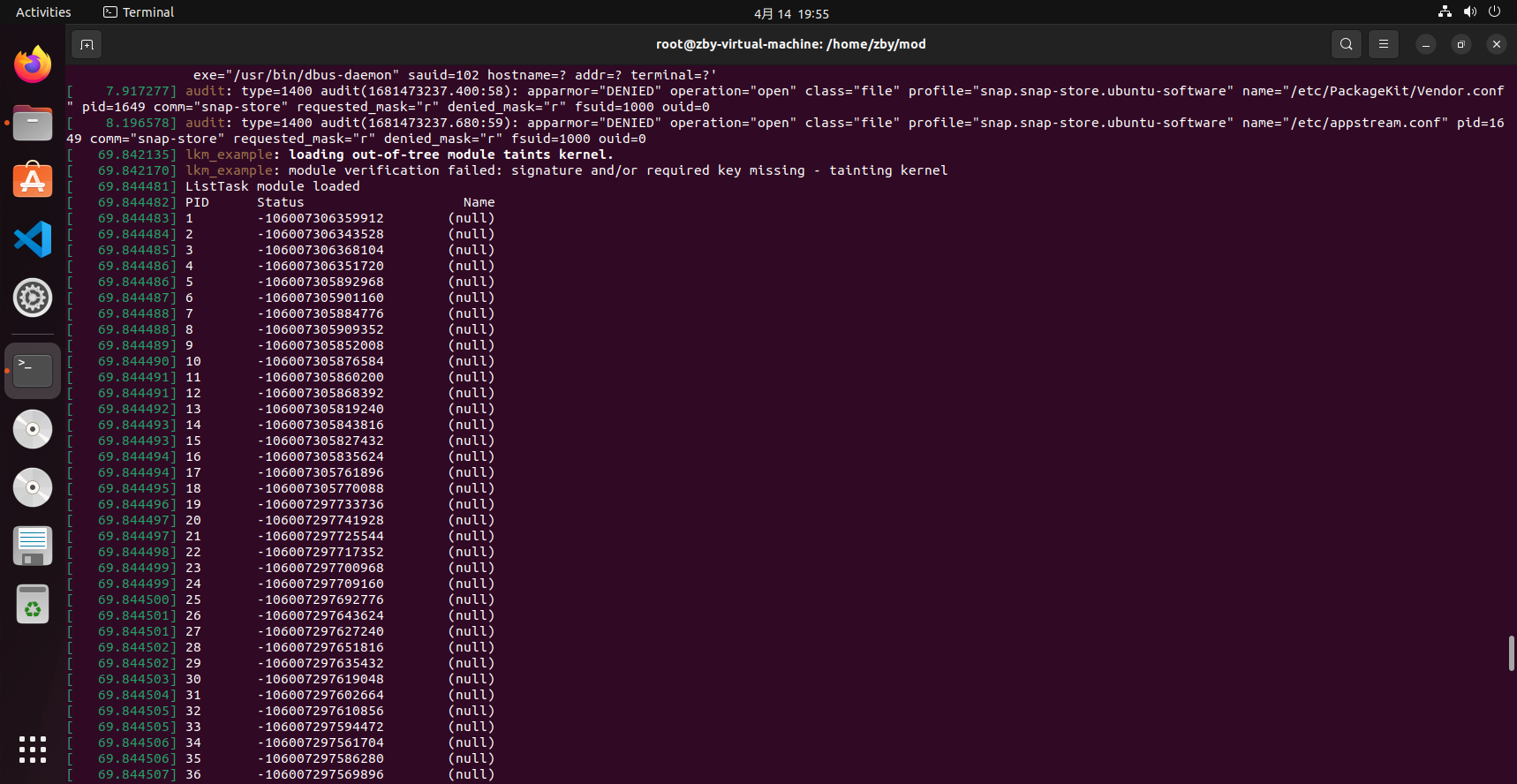
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

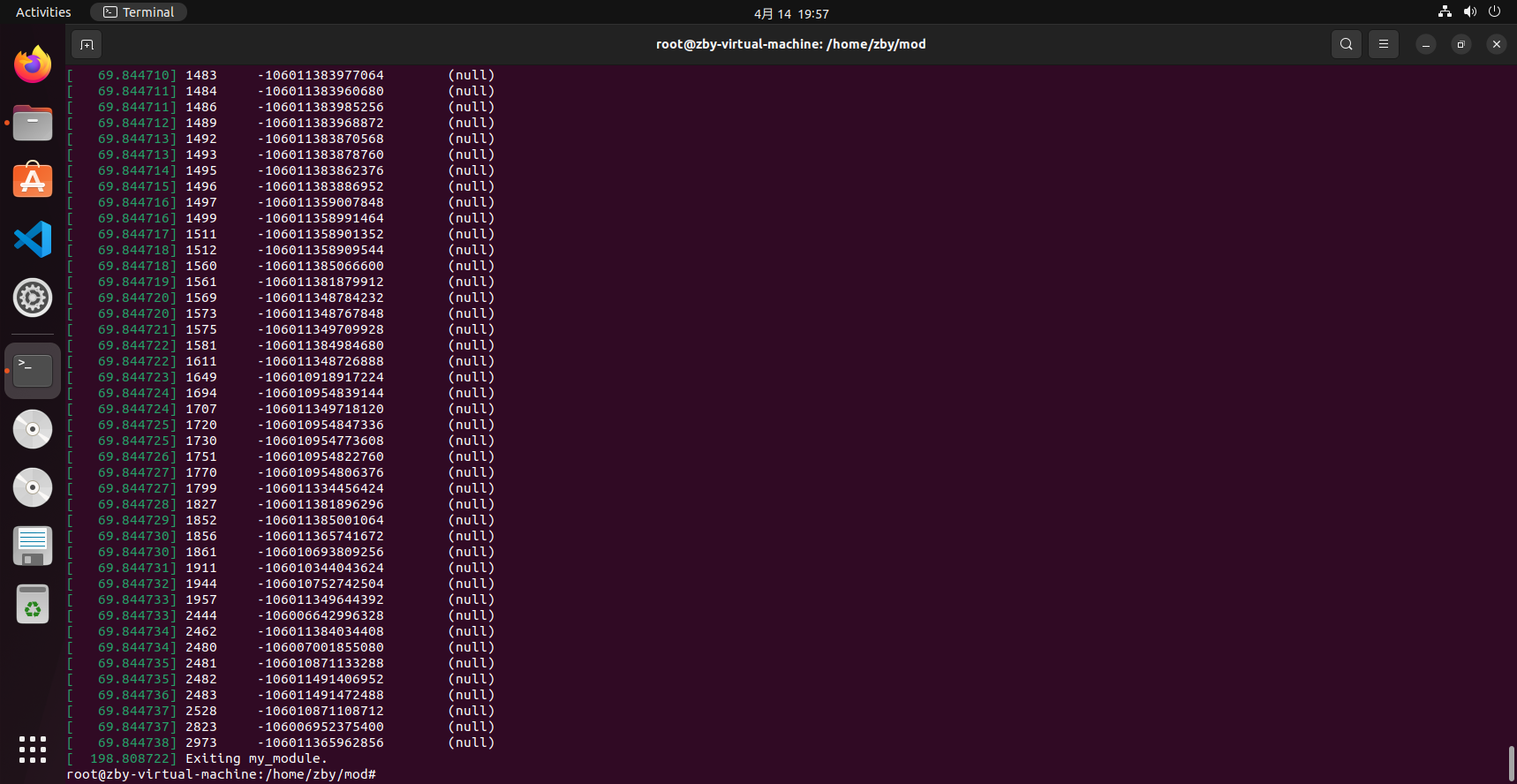
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

五、实验结果

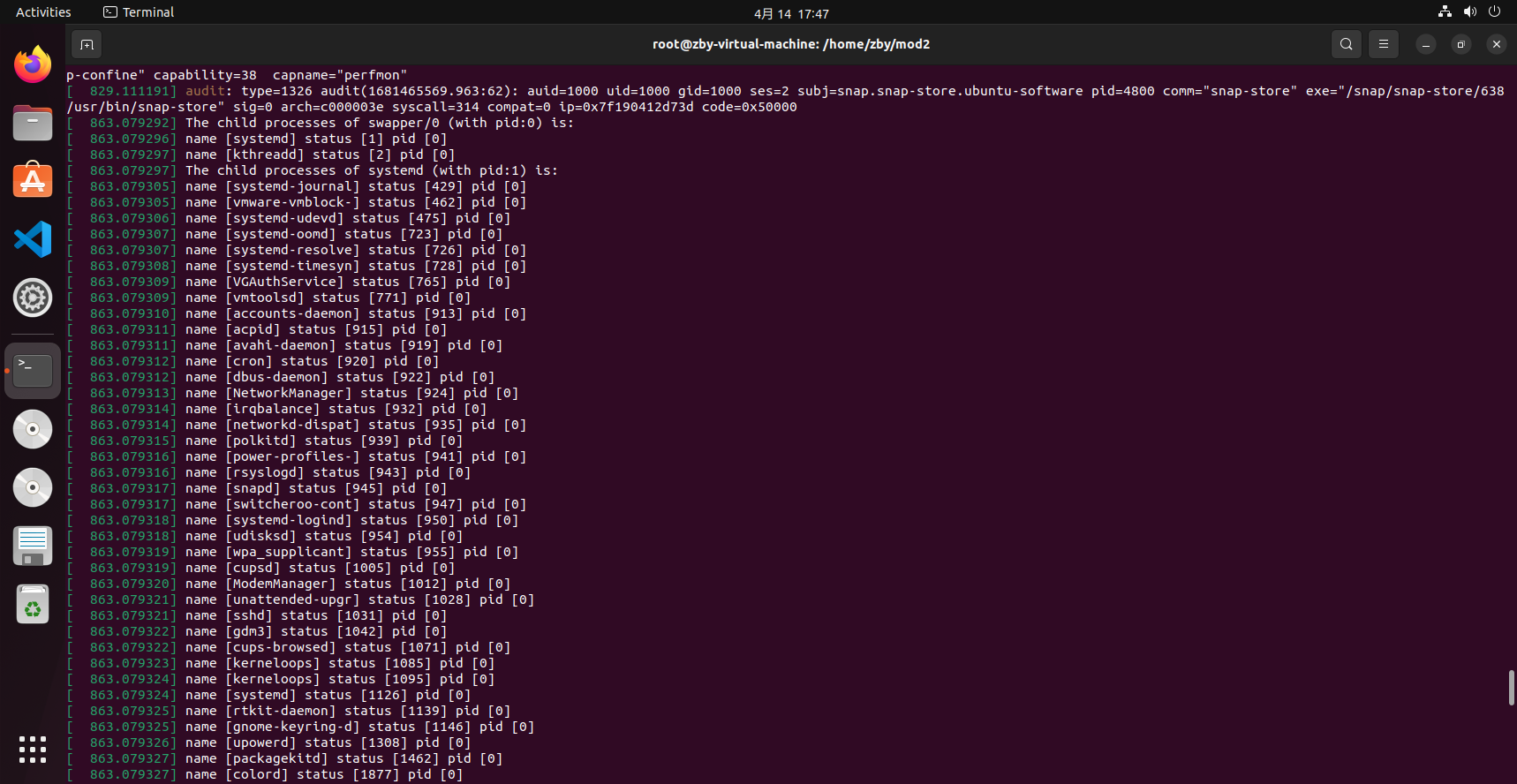
1.将模块1进行编译之后加载查看内核运行日志之后得到结果如下图所示，正确的显示了进程的标识符，状态还有名称。



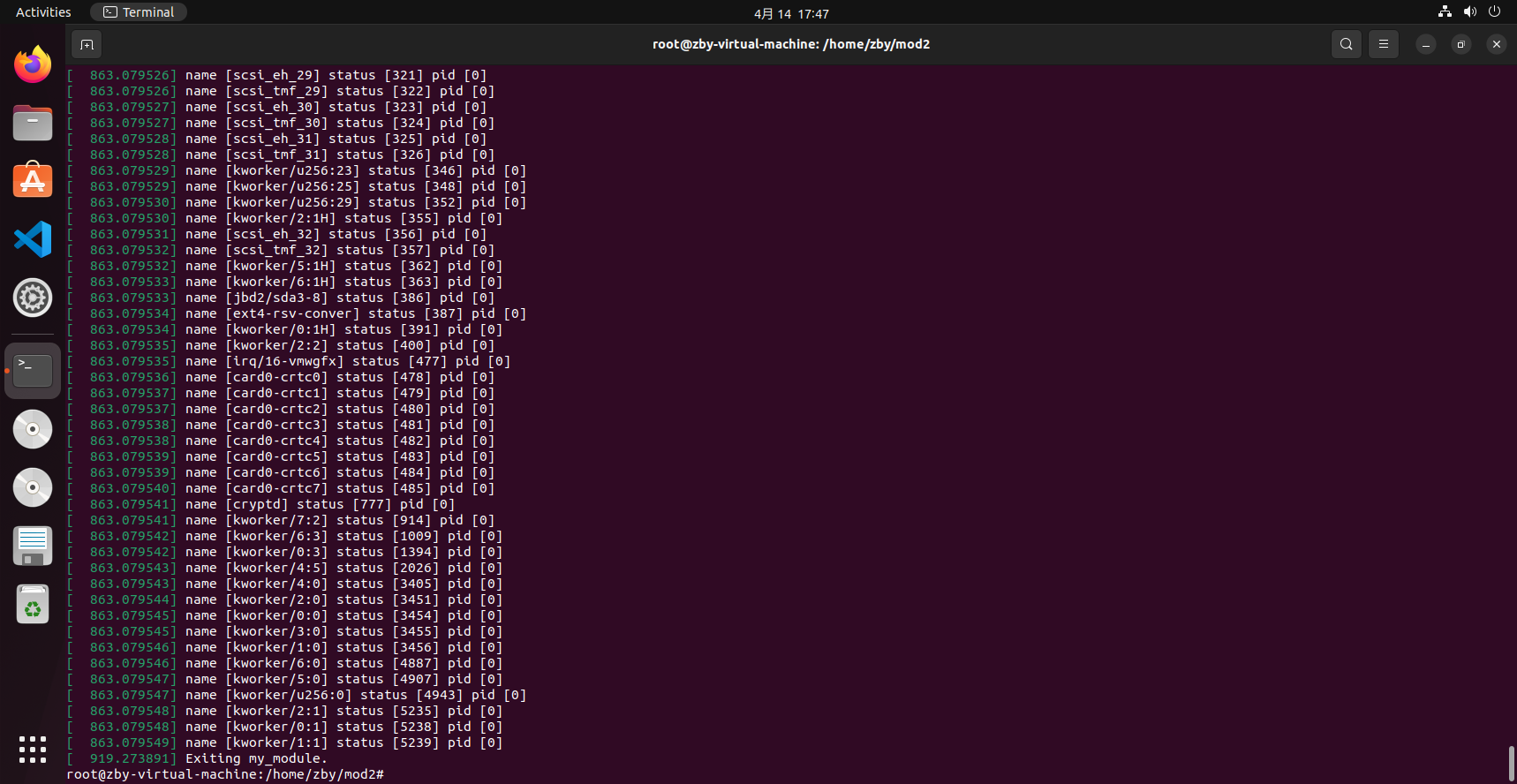
2.对模块1使用命令rmmod将内核模块进行卸载之后检查内核运行日志之后显示如下，内核能够正常卸载。



3.对模块2进行加载之后检查内核运行日志得到的输出如下所示，内核能够以DFS的形式实现对于系统内所有任务的迭代。

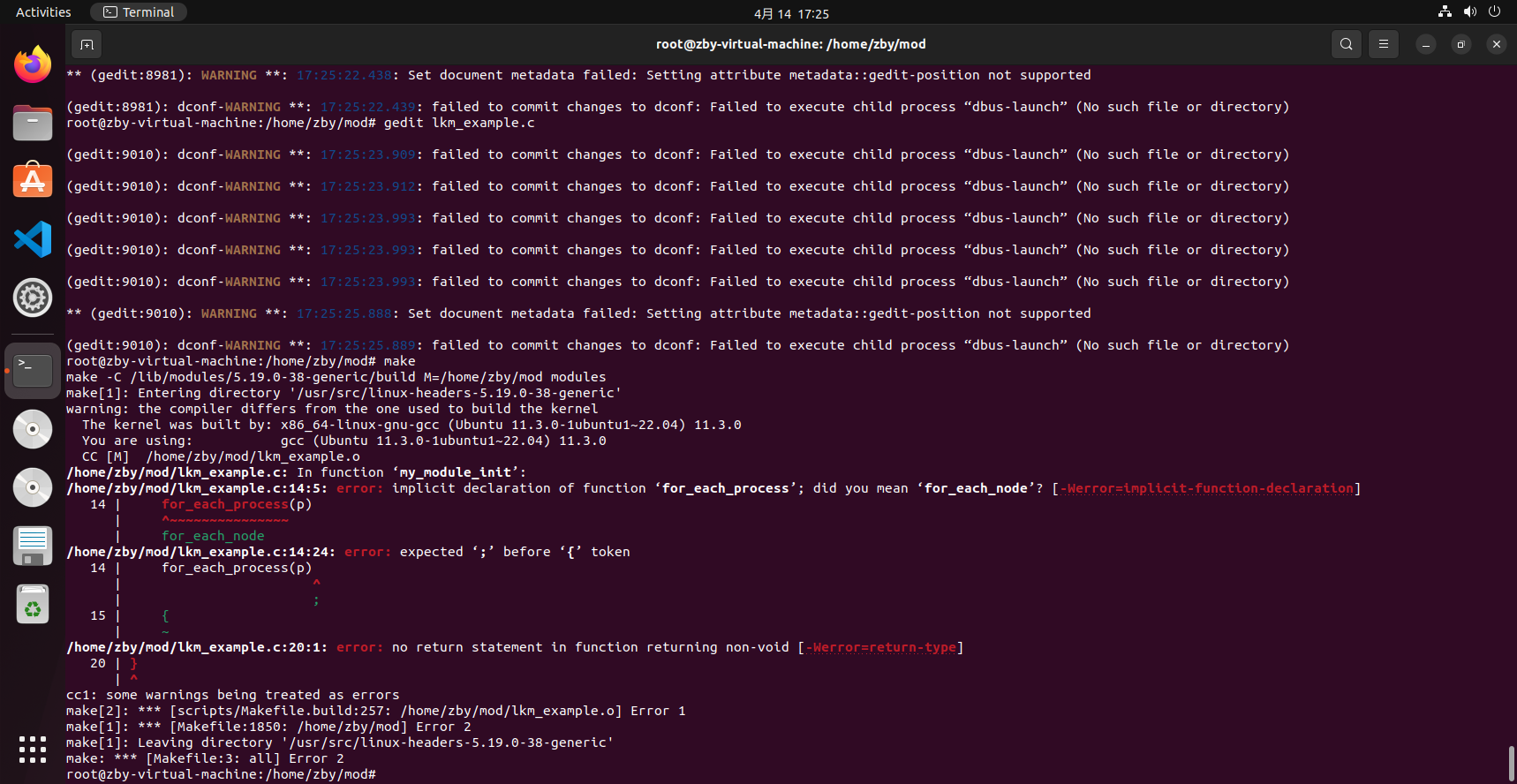


4.对内核进行卸载之后检查内核运行日志能够看到正确输出语句Exiting my\_module。卸载成功。



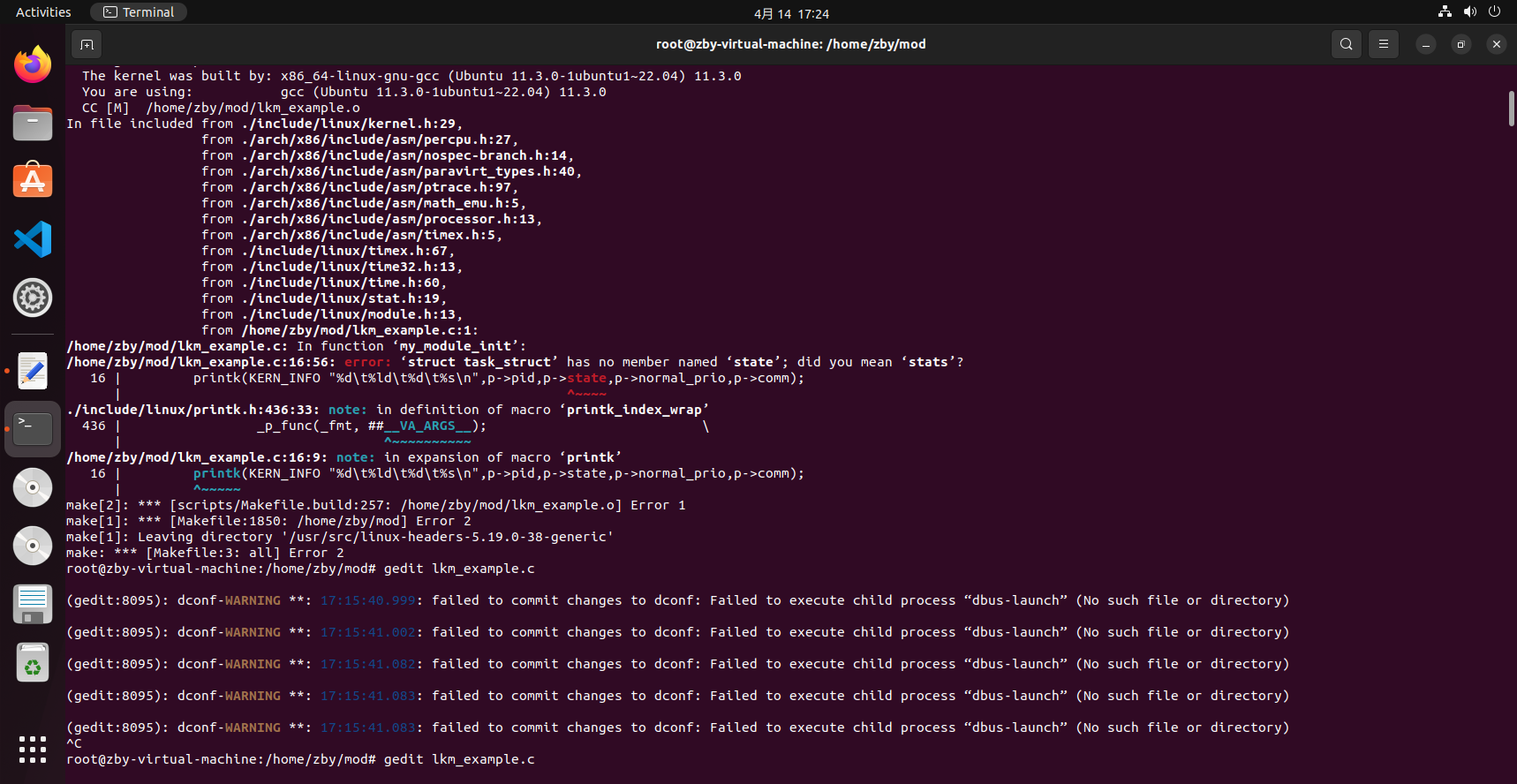
六、实验中bug与解决方法

1.报错如下所示。显示缺少头文件不能正常调用宏for\_each\_process。



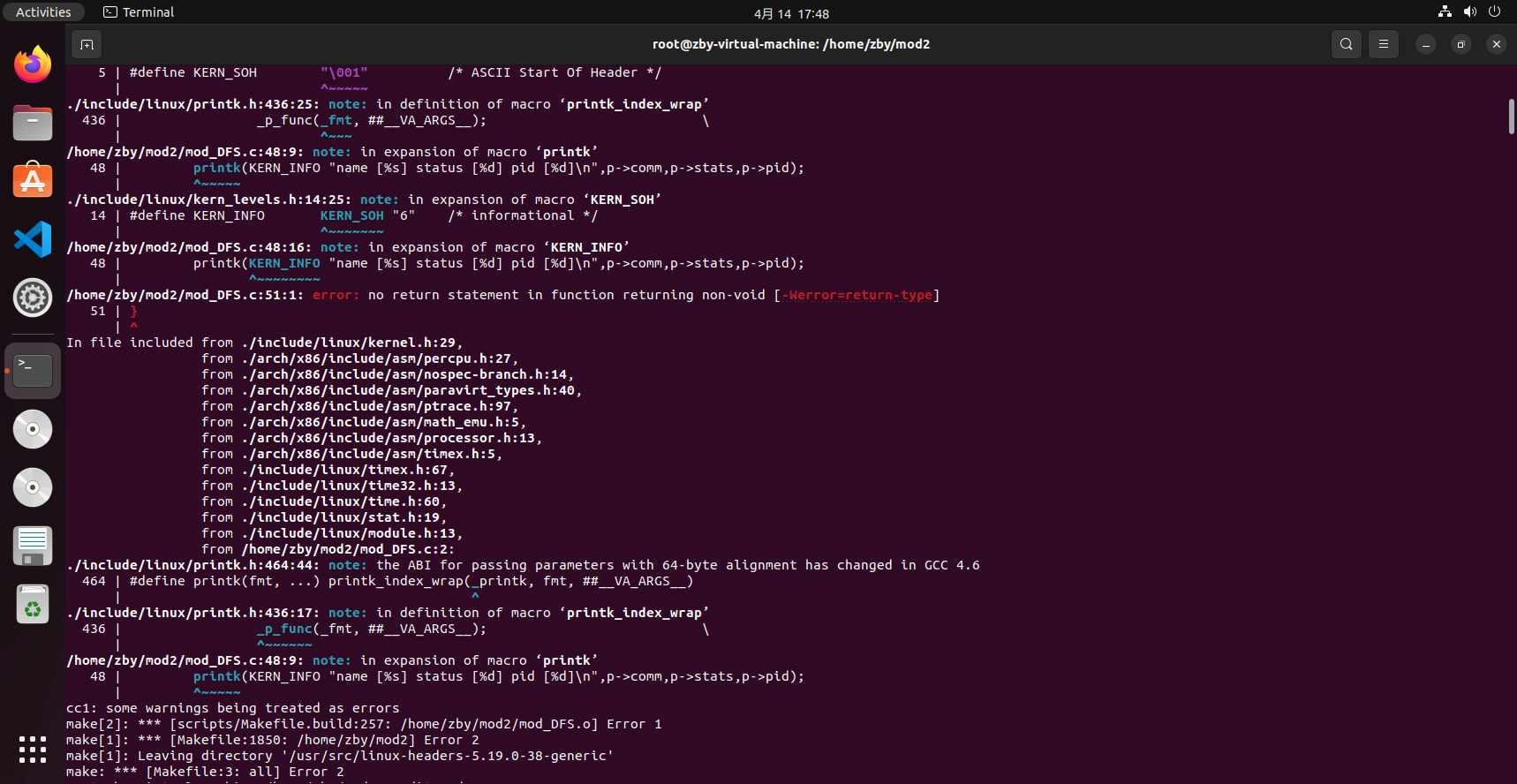
解决方法：在查阅相关资料后发现代码缺少头文件linux/init\_task.h,在文件开头补上头文件即可消除报错。

2.文件报错如下所示，不能正确识别struct\_task中的变量status。



解决办法：通过查阅资料得出struct\_task类型中的变量会发生修改，采取编译器的建议将status变量改为stats即可消除报错。

3.在编译过程中报错no ruturn



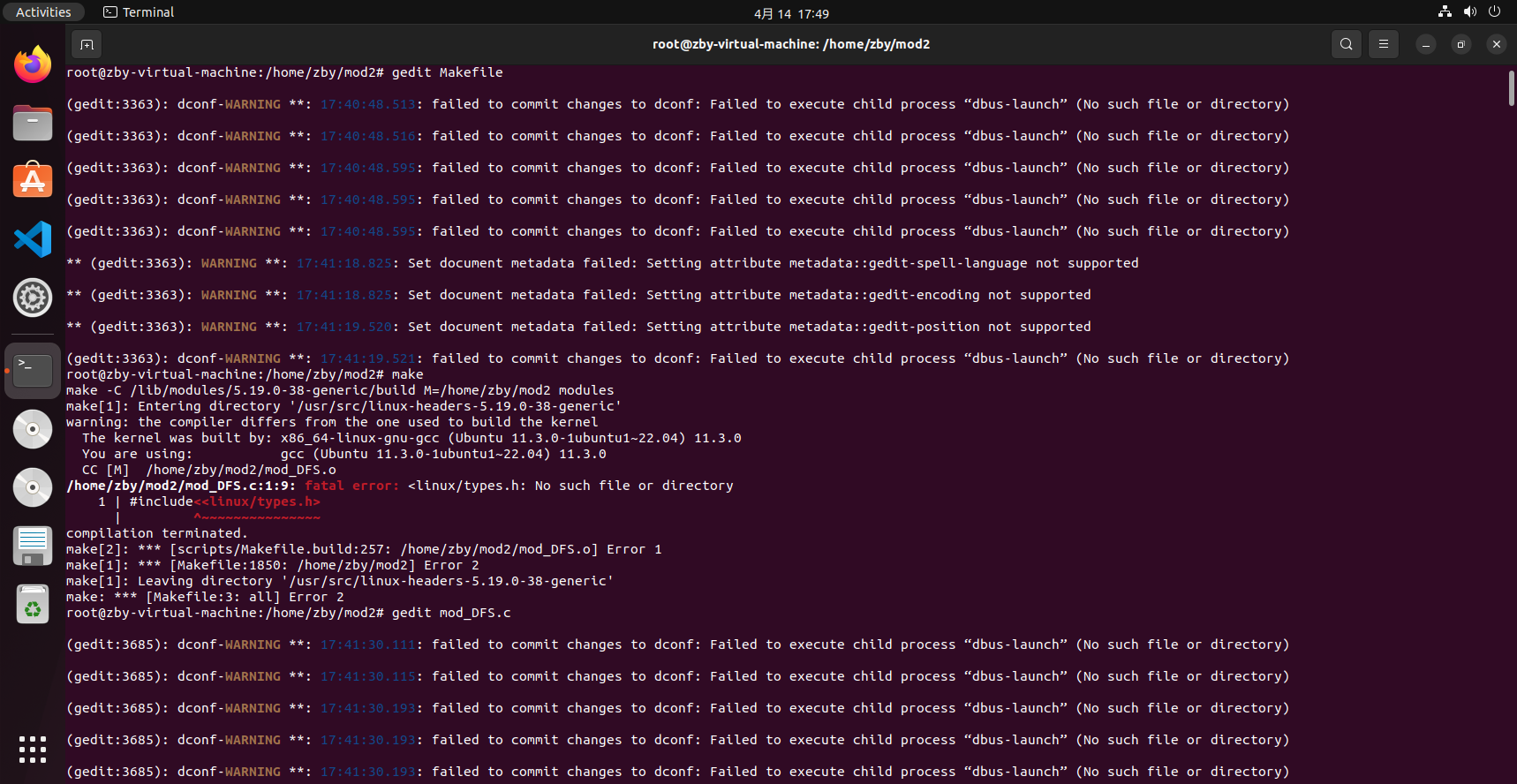
解决方法：在检查代码后发现在my\_module\_init函数调用中省略了return0才会出现报错，在该函数的最后加上return0即可解决报错。

4.在编译过程中出现报错如下图所示。



解决方法：经过观察发现所有的报错均围绕着init\_module,在检查代码文件后发现头文件中缺失了linux/init\_task.h,补上该头文件之后即可消除报错。

5.报错如下图所示。



解决方法：在检查函数的头文件之后发现代码中调用了错误的头文件，将头文件中的linux/types.h删去即可解决报错。