西安交通大学

操作系统专题实验报告

班级： 计算机002

学号： 2206113896

姓名： 赵泽宇

2022 年 12 月 15 日

目录

[操作系统专题实验报告 1](#_Toc5197)

[1 openEuler 系统环境实验 7](#_Toc9245)

[1.1 实验目的 7](#_Toc12704)

[1.2 实验内容 7](#_Toc13424)

[1.3 实验思想 7](#_Toc17256)

[1.4 实验步骤 7](#_Toc17398)

[1.5 测试数据设计 8](#_Toc2461)

[1.6 程序运行初值及运行结果分析 8](#_Toc25447)

[1.7 实验总结 13](#_Toc8869)

[1.7.1 实验中的问题与解决过程 13](#_Toc28424)

[1.7.2 实验收获 14](#_Toc30653)

[1.7.3 意见与建议 14](#_Toc7868)

[1.8 附件 14](#_Toc2796)

[1.8.1 附件 1 程序 14](#_Toc25923)

[1.8.2 附件 2 Readme 20](#_Toc29243)

[2 进程通信与内存管理 20](#_Toc19789)

[2-1 进程的软中断通信 20](#_Toc29784)

[2.1.1 实验目的 20](#_Toc23520)

[2.1.2 实验内容 20](#_Toc16128)

[2.1.3 实验思想 21](#_Toc31080)

[2.1.4 实验步骤 21](#_Toc19459)

[1实验前准备：学习MAN命令的用法 21](#_Toc1833)

[2实验内容：编程实现软中断进程 22](#_Toc21381)

[1. 使用alarm进行中断： 22](#_Toc28920)

[2. 使用delete(ctrl+c)中断： 25](#_Toc23871)

[3. 讨论sleep(5)对结果的影响： 26](#_Toc9722)

[2-2进程的管道通信 26](#_Toc20682)

[2.2.1 实验目的 26](#_Toc30790)

[2.2.2 实验内容 26](#_Toc9529)

[2.1.3 实验思想 27](#_Toc8854)

[2.2.4 实验步骤 27](#_Toc4751)

[1实验前准备： 27](#_Toc27749)

[2 实验过程 27](#_Toc31697)

[2-3实验回答问题 29](#_Toc16041)

[2.3.1软中断通信 29](#_Toc12730)

[2.3.2管道通信 31](#_Toc6214)

[2-4实验总结 31](#_Toc8138)

[2.4.1 实验中的问题与解决过程 31](#_Toc2312)

[2.4.2 实验收获 32](#_Toc21496)

[2-5 源码 32](#_Toc29192)

[3 页面置换算法 47](#_Toc15392)

[3.1 实验目的 47](#_Toc26417)

[3.2 实验内容 47](#_Toc10084)

[3.3 实验思想 47](#_Toc19015)

[3.3.1 FIFO算法 47](#_Toc21)

[3.3.2 LRU算法 47](#_Toc26741)

[3.4 实验步骤 48](#_Toc8809)

[1 FIFO算法 48](#_Toc10753)

[2 LRU算法 49](#_Toc1740)

[3.4 源码 49](#_Toc6627)

[3.5 扩展要求 59](#_Toc6059)

[a) kill 和 signal 怎样实现进程之间的通信？ 59](#_Toc7898)

[b) 在管道通信实验中怎样实现同步与互斥？ 59](#_Toc22185)

[c) 如果是FIFO，观察BLEADY现象。 59](#_Toc6746)

[4 动态模块与设备驱动 59](#_Toc21593)

[4.1 实验目的 59](#_Toc978)

[4.2 实验内容 59](#_Toc22263)

[4.3 实验步骤（与问题解决） 60](#_Toc2828)

[4.4实验总结 63](#_Toc31626)

[4.4.1 实验中的问题与解决过程 63](#_Toc7308)

[4.4.2 实验收获 64](#_Toc21410)

[4.5 源码 64](#_Toc29223)

[5 字符设备驱动 65](#_Toc2652)

[5.1 实验目的 65](#_Toc19821)

[5.2实验内容 66](#_Toc2844)

[5.3 实验步骤 66](#_Toc16411)

[1. 文件准备 66](#_Toc3814)

[2.编译安装过程 68](#_Toc21727)

[4. 聊天功能展示 69](#_Toc25420)

[5.4 实验总结 69](#_Toc16422)

[5.4.1 实验中的问题与解决过程 69](#_Toc14439)

[5.4.2 实验收获 70](#_Toc28069)

[5.5 源码 70](#_Toc21658)

# 1 openEuler 系统环境实验

## 实验目的

在华为云上openEuler操作系统环境，本实验旨在熟悉基于Kunpeng架构弹性云服务器ECS上openEuler操作系统基本系统环境。通过运行shell命令查看系统信息以达到了解openEuler操作系统的目的。进程相关编程实验，通过编写和运行简单的进程、线程相关程序，体会进程调度、进程间变量管理、进程调用其他程序、线程等方面在操作系统中的实际操作。

## 实验内容

a) 观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别是什么

b) 观察进程调度中的全局变量改变，输出父子进程共享变量地址了解物理地址与虚地址概念

c) 在子进程中调用system函数

d) 在子进程中调用exec族函数

e)创建两个线程运行后体会线程共享进程信息、线程对共享变量操作中同步与互斥的知识

## 实验思想

通过实际操作云服务器，实际编译运行代码熟悉云服务器的环境配置与使用，通过亲自写出进程创建代码，理解进程的运行过程与操作系统在这个过程中的作用。

## 实验步骤

1. 理解进程创建与fork()函数
2. 熟悉操作命令、编辑、编译、运行程序。
3. 完成操作系统原理课程教材P103作业 3.7 （采用图3-32所示的程序）的运行验证，多运行程序几次观察结果；去除wait后再观察结果并进行理论分析。
4. 扩展图3-32的程序：
5. 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释，同时输出两种变量的地址观察并分析；
6. 在return前增加对全局变量的操作并输出结果，观察并解释；
7. 修改程序体会在子进程中调用system函数和在子进程中调用exec族函数执行自己写的一段程序，在此程序中输出进程PID进行比较并说明原因；

线程相关编程实验：

创建两个线程运行后体会线程共享进程信息、线程对共享变量操作中同步与互斥的知识。

1、在进程中给一变量赋初值并创建两个线程；

2、在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果 ；

3、多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；

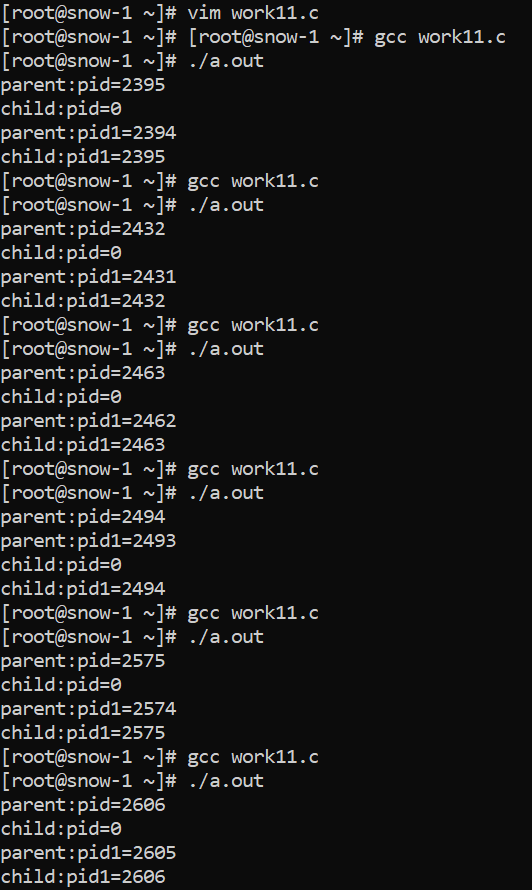
4、 将任务一中第一个实验调用system函数和调用exec族函数改成在线程中实现，观察运行结果输出进程PID与线程TID进行比较并说明原因。

## 测试数据设计

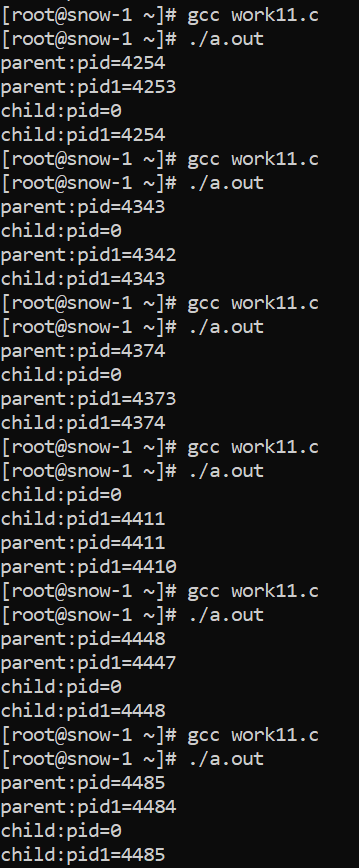
通过在一、2）b）中设置全局变量，在两个进程中分别对进程进行简单的“++”“--”，确定进程与全局变量的关系

通过在二、1.2.中对进程中变量进行5000次“++”“--”操作，进行对进程互斥性的检测

## 程序运行初值及运行结果分析

分析：

使用fork()函数后，父进程中返回的值是子进程的进程号，在子进程中返回的是0；每次输出不一样是因为系统给父子进程分的进程号一直在变化

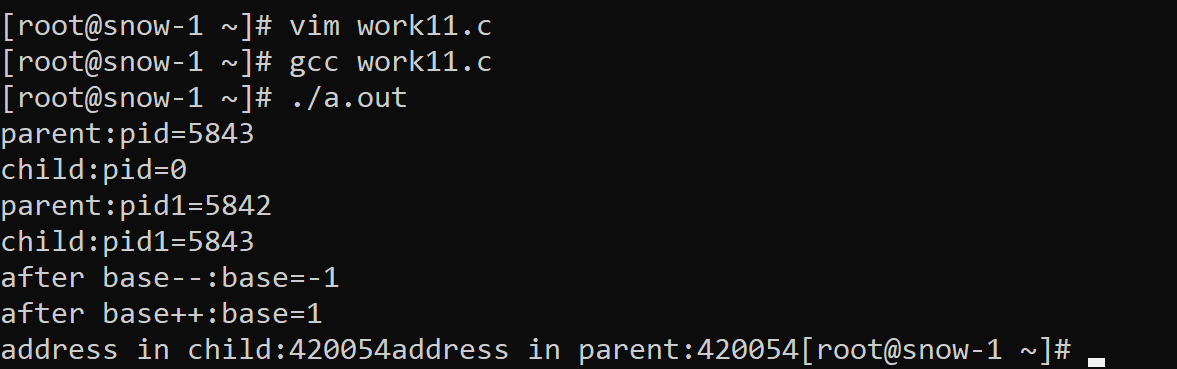


分析：

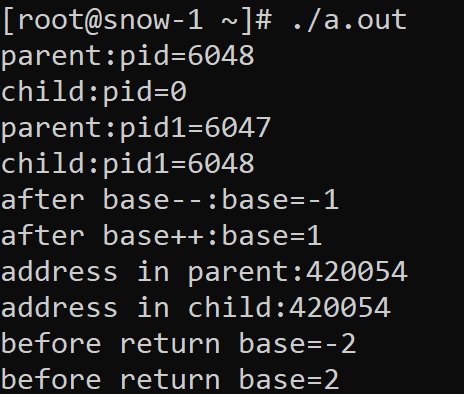
通过程序运行结果可以看出，fork()函数被调用后，在当前进程中创建了一个子进程与父进程同步运行，因此返回了两次结果。pid值为0时，表明子进程在运行，getpid()返回子进程的pid赋给pid1，为子进程的地址；pid值大于0时，表明父进程在运行，pid值为子进程的地址， getpid()返回父进程的pid赋给pid1，为父进程的地址。由于子进程为父进程的拷贝，地址为父进程地址+1。综上所述，parent:pid值与child:pid1值相同，都为子进程的地址，parent:pid1的值为父进程的地址，child:pid的值为0。

fork()函数并不确定父子进程的执行顺序，不同版本的操作系统，不同系统环境、系统负载下的执行顺序都不尽相同，这也导致了上述多次执行结果顺序的不同，由操作系统的调度算法决定。

加入全局变量后的输出：



在return前加上全局变量的操作和输出：



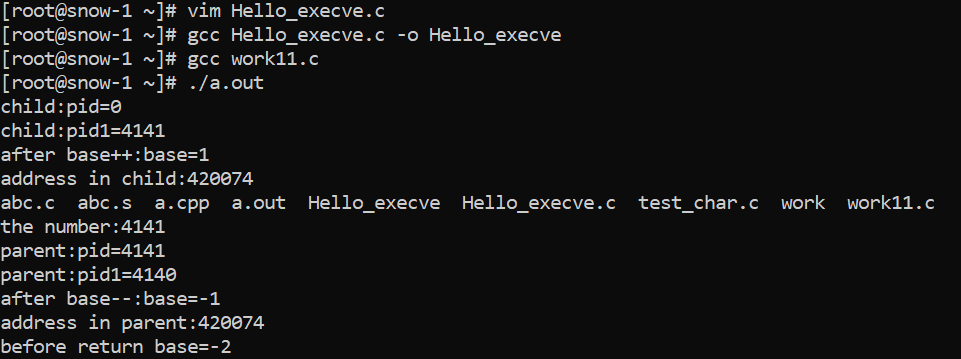
结论：

1.父子进程不能共享全局变量。

2.但是如果父子进程只是对全局变量做读操作，则父子进程共享同一份全局变量。

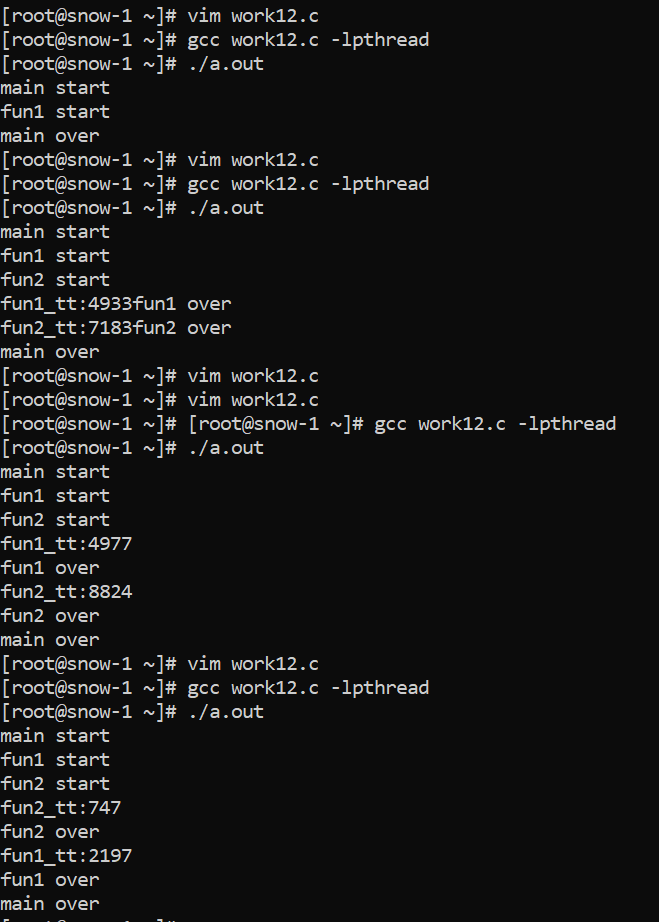
3.如果父子进程中的任何一个对变量做修改操作，会在内存中拷贝一份副本，然后在这个副本上进行修改，修改完成后再通过MMU映射回去。

在子程序中使用system函数和execve()函数输出子进程的进程识别码：



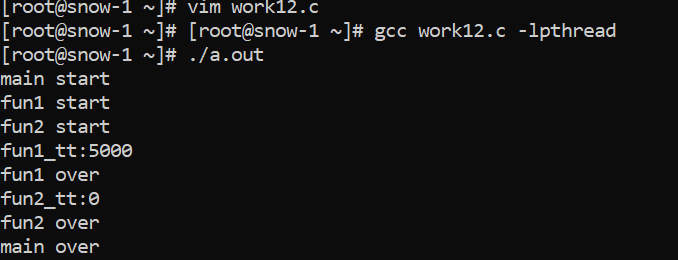
调用exec族函数可以把当前进程替换为一个拥有相同pid的新进程。在本例中调用execve()函数作用为调用可执行文件Hello\_execve替换当前进程，根据输出可以发现替换前后的pid并未发生改变，但execve()函数后的语句由于进程被替换不再运行。在子进程中打开的程序的进程识别码与子进程相同，同样也会执行系统命令。

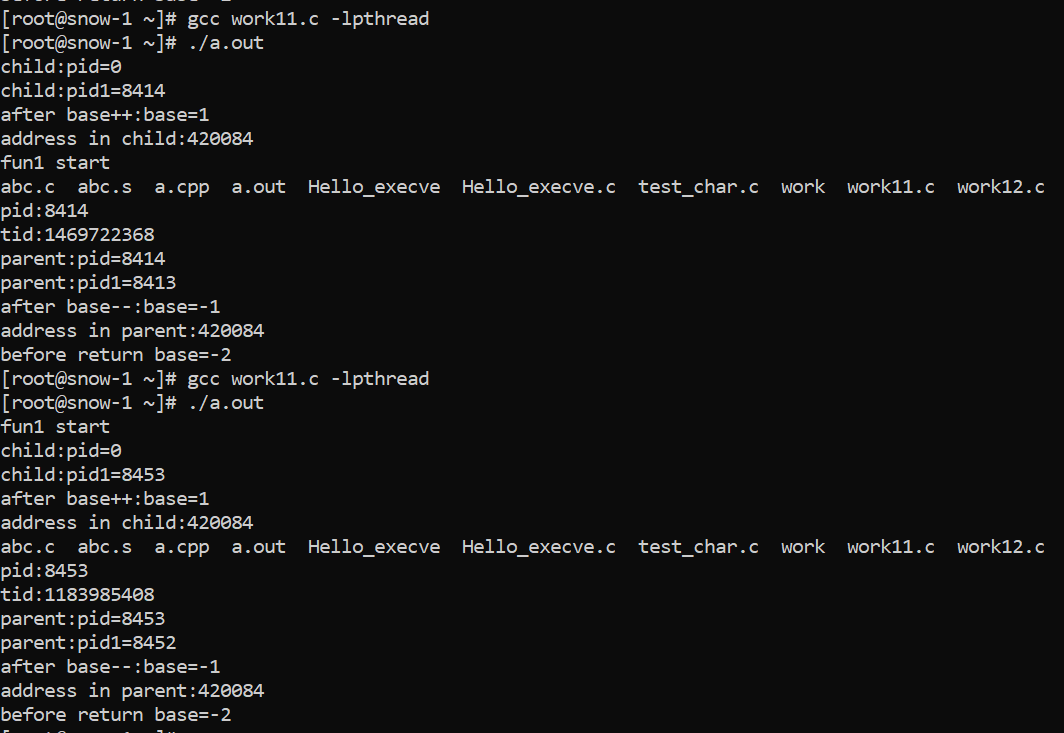
在进程中创建两个线程并初始化一个变量，两个线程分别操作5000次并输出结果：



发现每次输出结果都不同，说明由于两个操作并不互斥，所以产生了混乱的修改结果。

加锁后实现互斥：





分析：

调用getpid()函数得到的是进程的pid，而在内核中，每个线程都拥有自己单独的pid，要得到线程的pid，必须用syscall(SYS\_gettid)函数，而调用pthread\_self()获取的是线程ID，其仅在同一个进程中保证唯一。

在main()函数中，syscall(SYS\_gettid)的值和getpid()得到的值相同，在子线程中却不相同。关于的system()函数和exec族函数的分析与进程中相同，调用exec族函数可以把当前线程替换为一个拥有相同pid的新线程。

进程pid：

进程pid（进程ID）,每个进程在系统中都有一个唯一·的非负整数表示的进程ID，用getpid() 获取进程ID。

线程tid：

线程tid（线程ID），每个线程在所属进程中都有一个唯一的线程ID，用pthread\_self() 获取自身现成ID。有多个进程时，可能会出现多个线程ID相同的线程，故线程tid只在其所属的进程上下文中有意义，不能作为系统中某个线程的唯一标识符。

线程pid：

线程pid，每个线程在系统中都有一个唯一的pid标识符，用系统调用sys\_call(SYS\_gettid()) 获取自身线程pid。主线程pid与所在进程pid相同。

## 实验总结

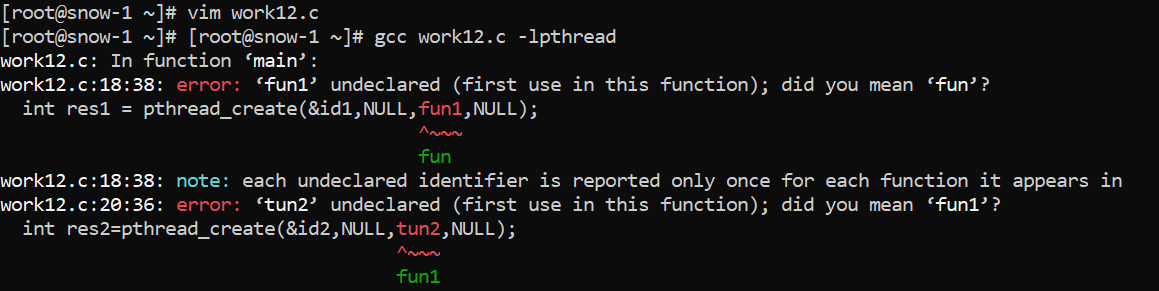
### 实验中的问题与解决过程

1. 问题：在linux中，gcc含有线程创建的文件时报错

解决：再使用gcc命令时，加上”-lpthread”

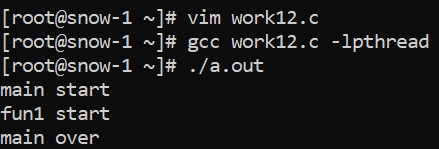
方式：Bing

1. 问题：在work12.c文件编译时报错：



解决：要在主函数之前声明主函数内用到的函数

1. 问题：主程序中运行完第一个线程后自动结束，不再运行第二个线程，如图：



解决：在主线程中运行sleep(2)，防止主线程运行后死亡，不给子线程运行机会

### 实验收获

通过本次实验，我熟悉了云服务器的操作，通过自己的实验熟悉了fork()函数创建新的进程以及在线程中调用system()和exec()族函数的方法和作用。也自己动手写出了一个加锁的算法。培养了对于操作系统这门课程的兴趣。

* + 1. 意见与建议

对自己的建议：提高项目总结能力，对自己做出的实验要有更好的调理，也应该更加仔细的对实验过程进行记录，感觉上实验中做了很多，也遇到了很多问题，投入了很多时间却没记录下很多东西。

## 1.8 附件

### 1.8.1 附件 1 程序

代码如下：

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int base=0;

int main(int argc,char \*argv[],char\*\* environ)

{

pid\_t pid,pid1;

pid=fork();

if(pid<0){

fprintf(stderr,"Fork Failed");return 1;

}

else if (pid==0){

pid1=getpid();

printf("child:pid=%d\n",pid);/\*A\*/

printf("child:pid1=%d\n",pid1);/\*B\*/

base++;

printf("after base++:base=%d\n",base);

printf("address in child:%x\n",&base);

system("ls");

execve("./Hello\_execve",argv,environ);

printf("not!!!\n");

}

else{

wait(NULL);

pid1=getpid();

printf("parent:pid=%d\n",pid);/\*C\*/

printf("parent:pid1=%d\n",pid1);/\*D\*/

base--;

printf("after base--:base=%d\n",base);

printf("address in parent:%x\n",&base);

}

base=base\*2;

printf("before return base=%d\n",base);

return 0;

}

第一次修改，给变量tt加上锁，源码如下：

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

int tt;

//声明一个线程函数

void \*fun1(void \*);

void \*fun2(void \*);

pthread\_mutex\_t mutex;//定义互斥变量

int main()

{

printf("main start\n");

tt=0;

pthread\_mutex\_init(&mutex, NULL);//互斥变量初始化

pthread\_t id1;

pthread\_t id2;

//创建函数线程，并且指定函数线程要执行的函数

int res1 = pthread\_create(&id1,NULL,fun1,NULL);

assert(res1 == 0);

int res2=pthread\_create(&id2,NULL,fun2,NULL);

assert(res2==0);

sleep(2);

printf("main over\n");

pthread\_mutex\_destroy(&mutex);//销毁锁

exit(0);

}

//定义线程函数

void\* fun1(void \*arg)

{

printf("fun1 start\n");

pthread\_mutex\_lock(&mutex);//加锁

for(int i=0; i < 5000;i++)

{

tt++;

}

pthread\_mutex\_unlock(&mutex);//解锁

printf("fun1\_tt:%d\n",tt);

printf("fun1 over\n");

}

void\* fun2(void \*arg)

{

printf("fun2 start\n");

pthread\_mutex\_lock(&mutex);//加锁

for(int i=0; i < 5000;i++)

{

tt--;

}

pthread\_mutex\_unlock(&mutex);//解锁

printf("fun2\_tt:%d\n",tt);

printf("fun2 over\n");

}

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

int tt;

//声明一个线程函数

void \*fun1(void \*);

void \*fun2(void \*);

pthread\_mutex\_t mutex;//定义互斥变量

int main()

{

printf("main start\n");

tt=0;

pthread\_mutex\_init(&mutex, NULL);//互斥变量初始化

pthread\_t id1;

pthread\_t id2;

//创建函数线程，并且指定函数线程要执行的函数

int res1 = pthread\_create(&id1,NULL,fun1,NULL);

int res2=pthread\_create(&id2,NULL,fun2,NULL);

//assert(res2==0);

sleep(2);

printf("main over\n");

pthread\_mutex\_destroy(&mutex);//销毁锁

exit(0);

}

//定义线程函数

void\* fun1(void \*arg)

{

printf("fun1 start\n");

//pthread\_mutex\_lock(&mutex);//加锁

for(int i=0; i < 5000;i++)

{

pthread\_mutex\_lock(&mutex);//加锁

tt++;

//printf("fun1running\n");

pthread\_mutex\_unlock(&mutex);//解锁

}

//pthread\_mutex\_unlock(&mutex);//解锁

printf("fun1\_tt:%d\n",tt);

printf("fun1 over\n");

}

void\* fun2(void \*arg)

{

printf("fun2 start\n");

//pthread\_mutex\_lock(&mutex);//加锁

for(int i=0; i < 5000;i++)

{

pthread\_mutex\_lock(&mutex);//加锁

tt--;

//printf("fun2running\n");

pthread\_mutex\_unlock(&mutex);//解锁

}

//pthread\_mutex\_unlock(&mutex);//解锁

printf("fun2\_tt:%d\n",tt);

printf("fun2 over\n");

}

### 1.8.2 附件 2 Readme

# 2 进程通信与内存管理

## 2-1 进程的软中断通信

### 2.1.1 实验目的

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在POSIX 规范中系统调用的功能和使用。同时加深对进程通信与中断的理解。

### 2.1.2 实验内容

编制实现软中断通信的程序：

使用系统调用fork()创建两个子进程，再用系统调用signal()让父进程捕捉键盘上发出的中断信号（即按delete键），当父进程接收到这两个软中断的某一个后，父进程用系统调用kill()向两个子进程分别发出整数值为16和17软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

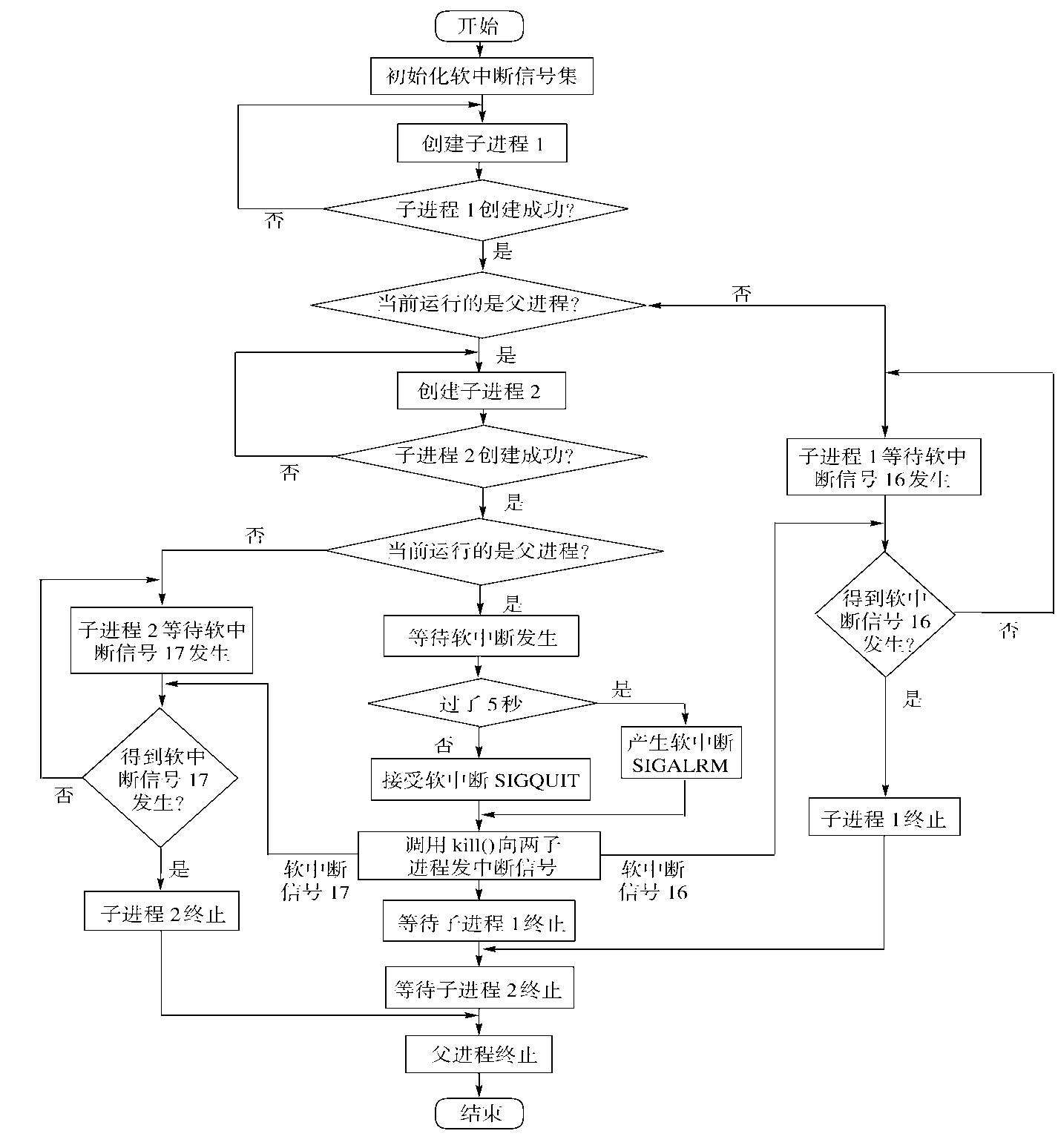
父进程调用wait()函数等待两个子进程终止后，输入以下信息，结束进程执行：

Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

### 2.1.3 实验思想

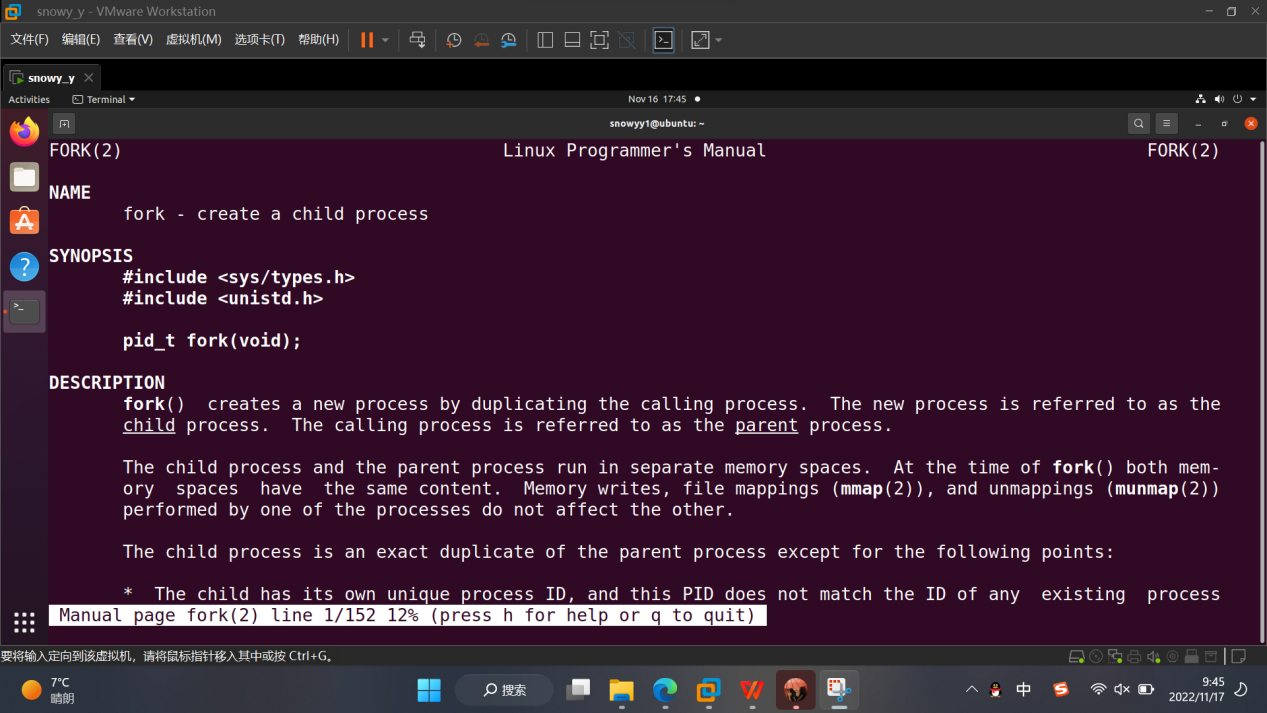
通过如下的流程图，实现进程的软中断的判断：

利用不同的signal信号和其对应的系统赋值，实现不同的中断信号对进程的软中断。

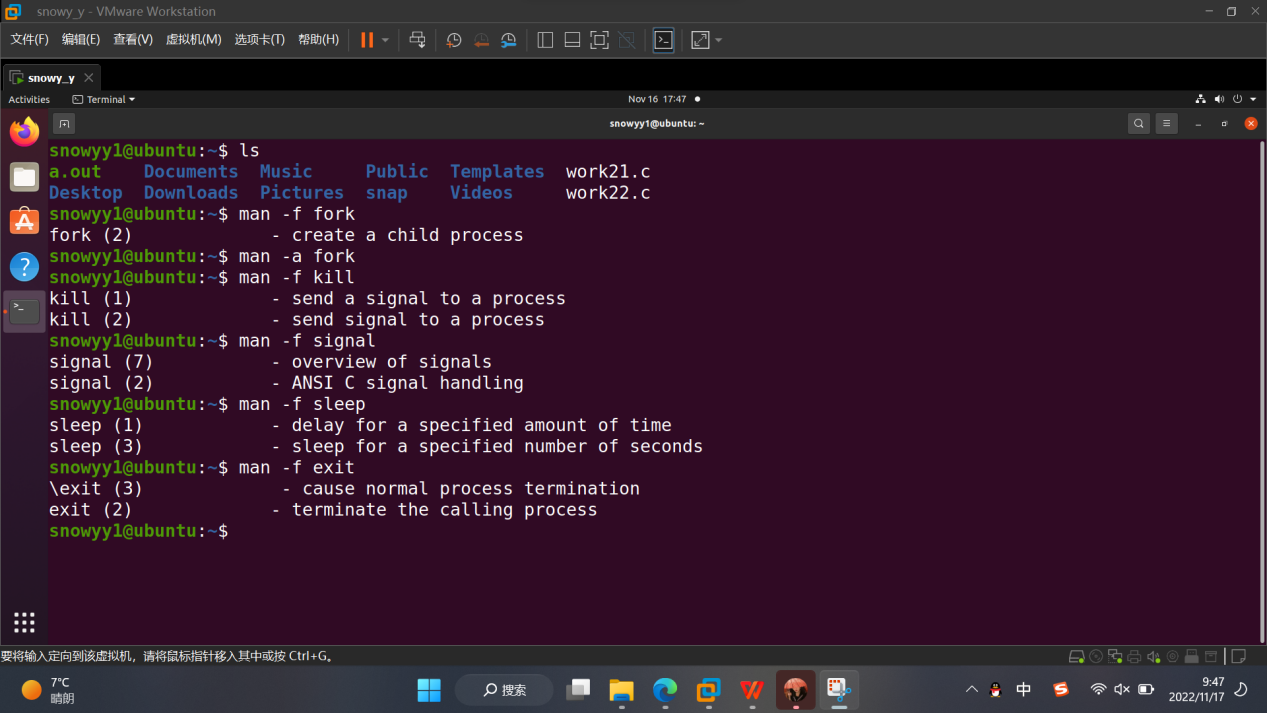
### 2.1.4 实验步骤

#### 1实验前准备：学习MAN命令的用法

man -a fork



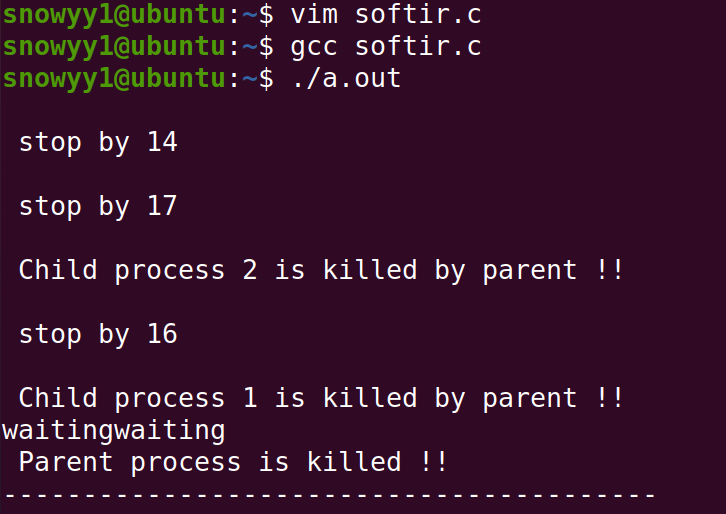
man -f 各种命令

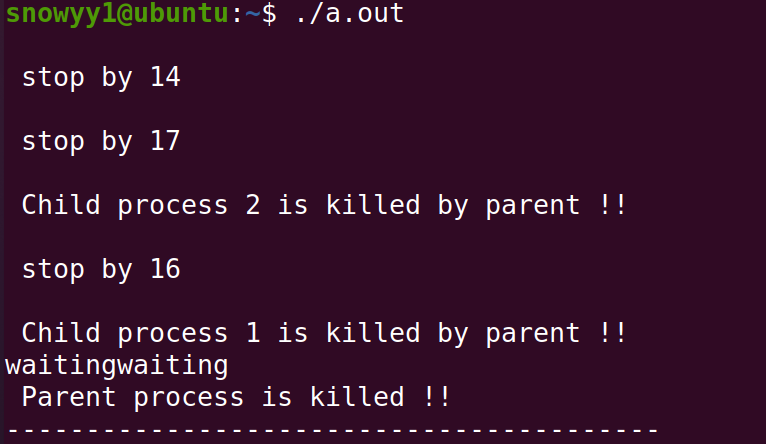


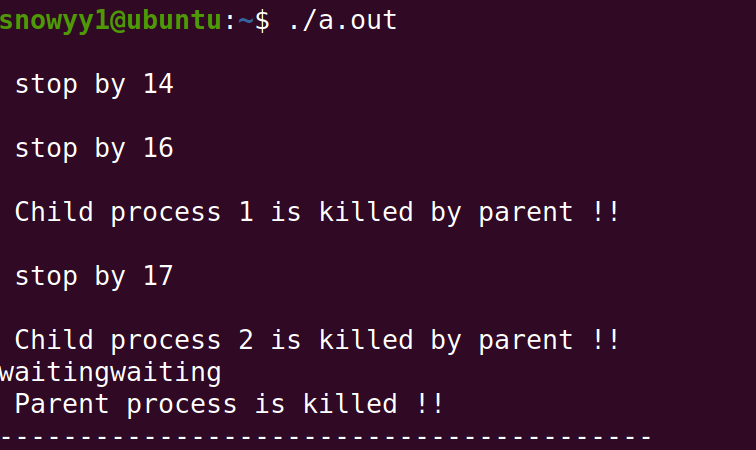
#### 2实验内容：编程实现软中断进程

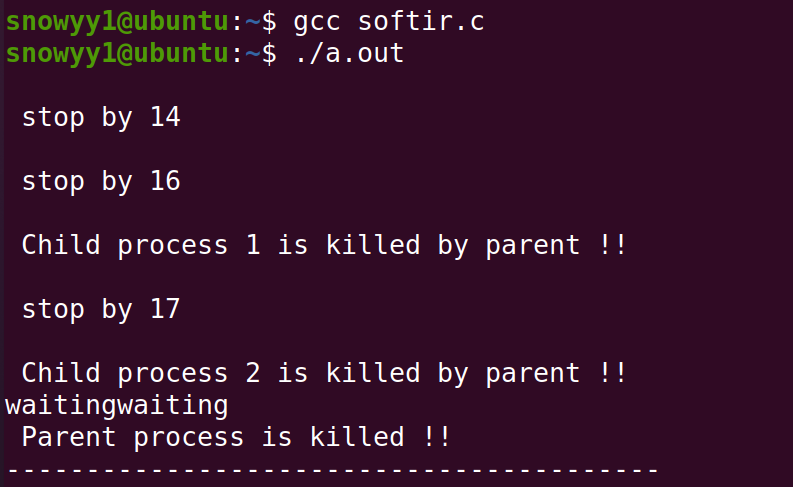
##### 使用alarm进行中断：

代码见附录文件softir.c





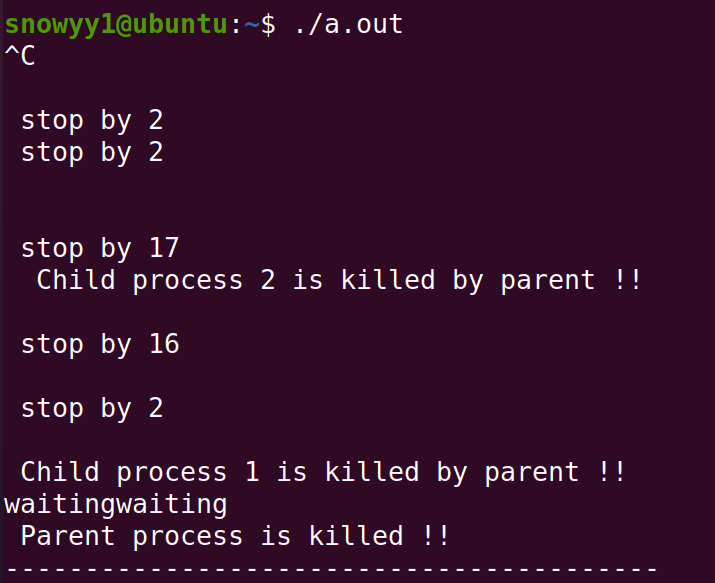
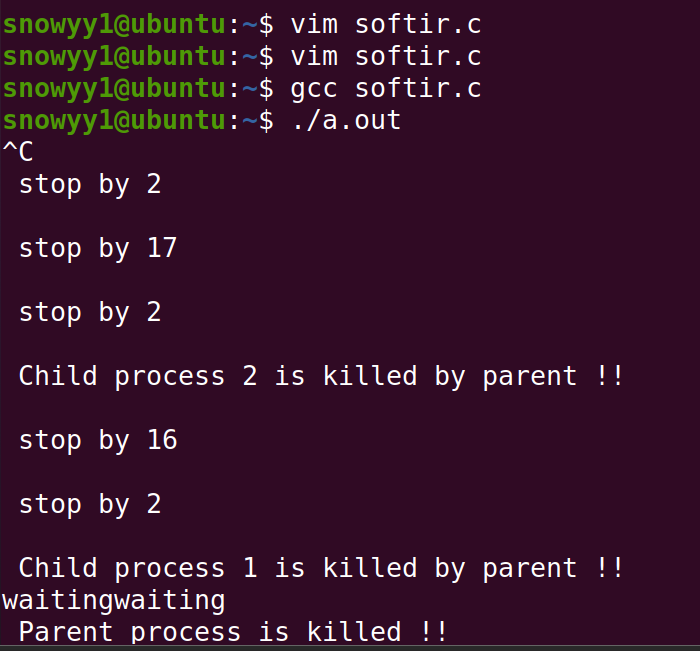




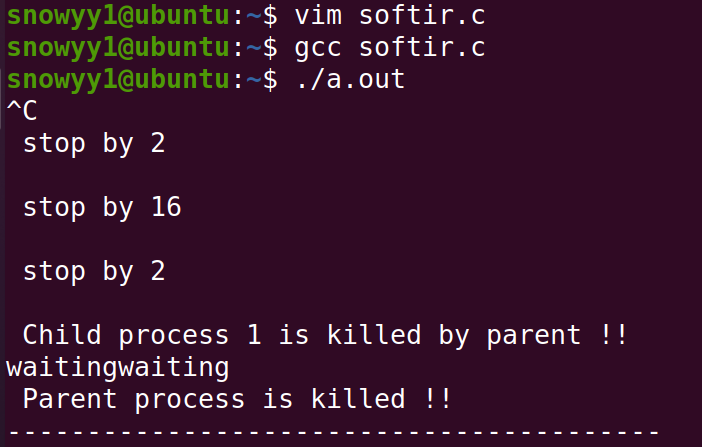
Ps:过程中出现的问题：无法解释’stop by’中间出现的换行

解决：通过在子进程“signal”语句前加入输出标识符，发现该输出标识符会代替回车，猜测该回车是自带机制。

##### 使用delete(ctrl+c)中断：



##### 讨论sleep(5)对结果的影响：



分析：三次参数为2的stop是因为signal(SIGINT,stop)；被继承了，三个进程都收到了信号。五次stop是因为有时结束得太快，在创建子进程之前就使得wait\_flag=1，这样进入子进程后继承的wait\_flag也是1，不用等到kill信号就自己停止了。同理也会出现执行四次的情况，与子进程创建的时机和按下delete的时机有关。

## 2-2进程的管道通信

### 2.2.1 实验目的

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。

### 2.2.2 实验内容

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

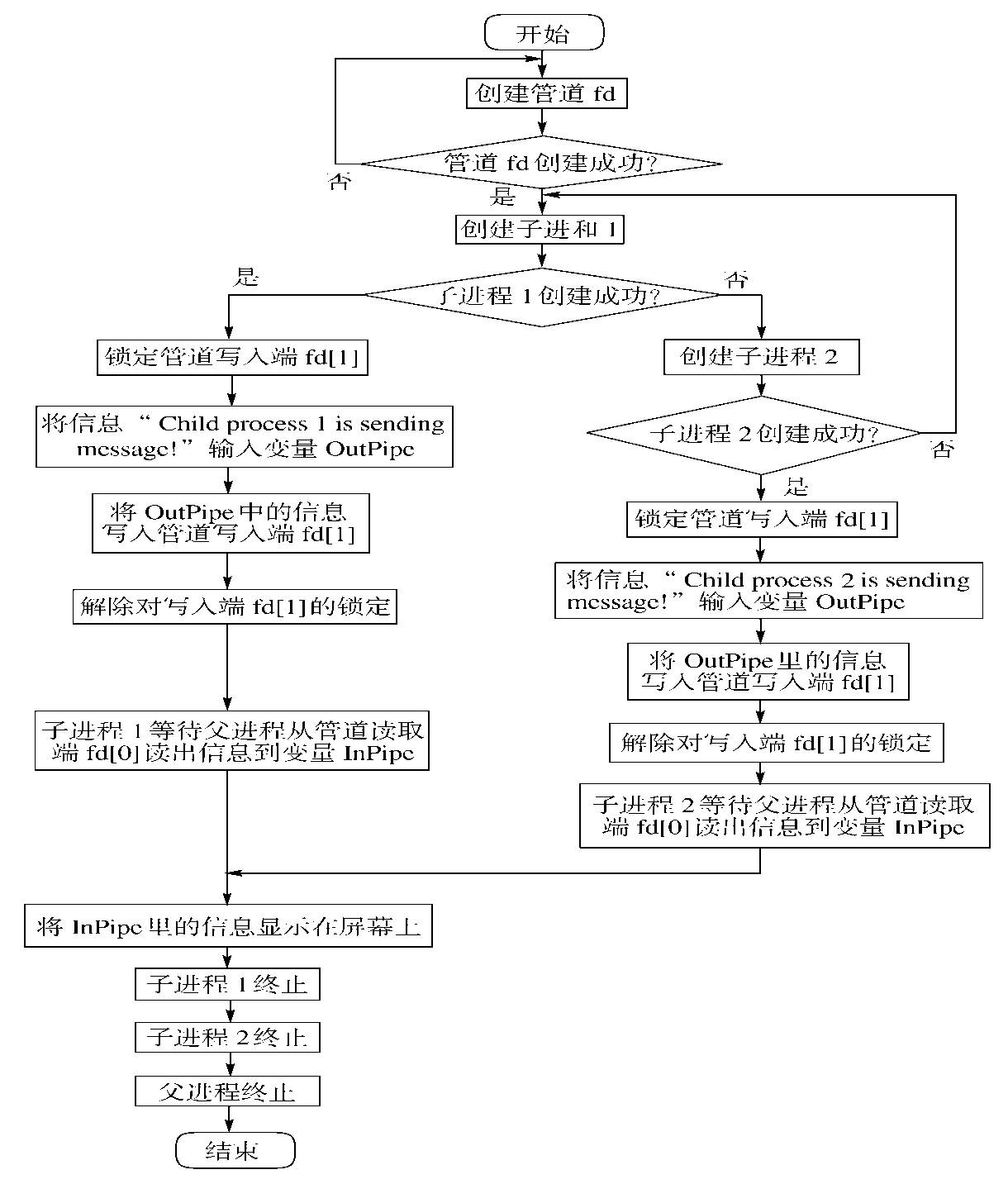
为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

①互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。

②同步，指当写(输入)进程把一定数量(如4KB)的数据写入pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

③确定对方是否存在，只有确定了对方已存在时，才能进行通信。

### 2.1.3 实验思想

下图为管道通信程序流程图，本实验所使用的管道通信思路主要遵循本图所示的判断标准：

### 2.2.4 实验步骤

#### 1实验前准备：

利用博客了解write(),lockf()等函数

#### 2 实验过程

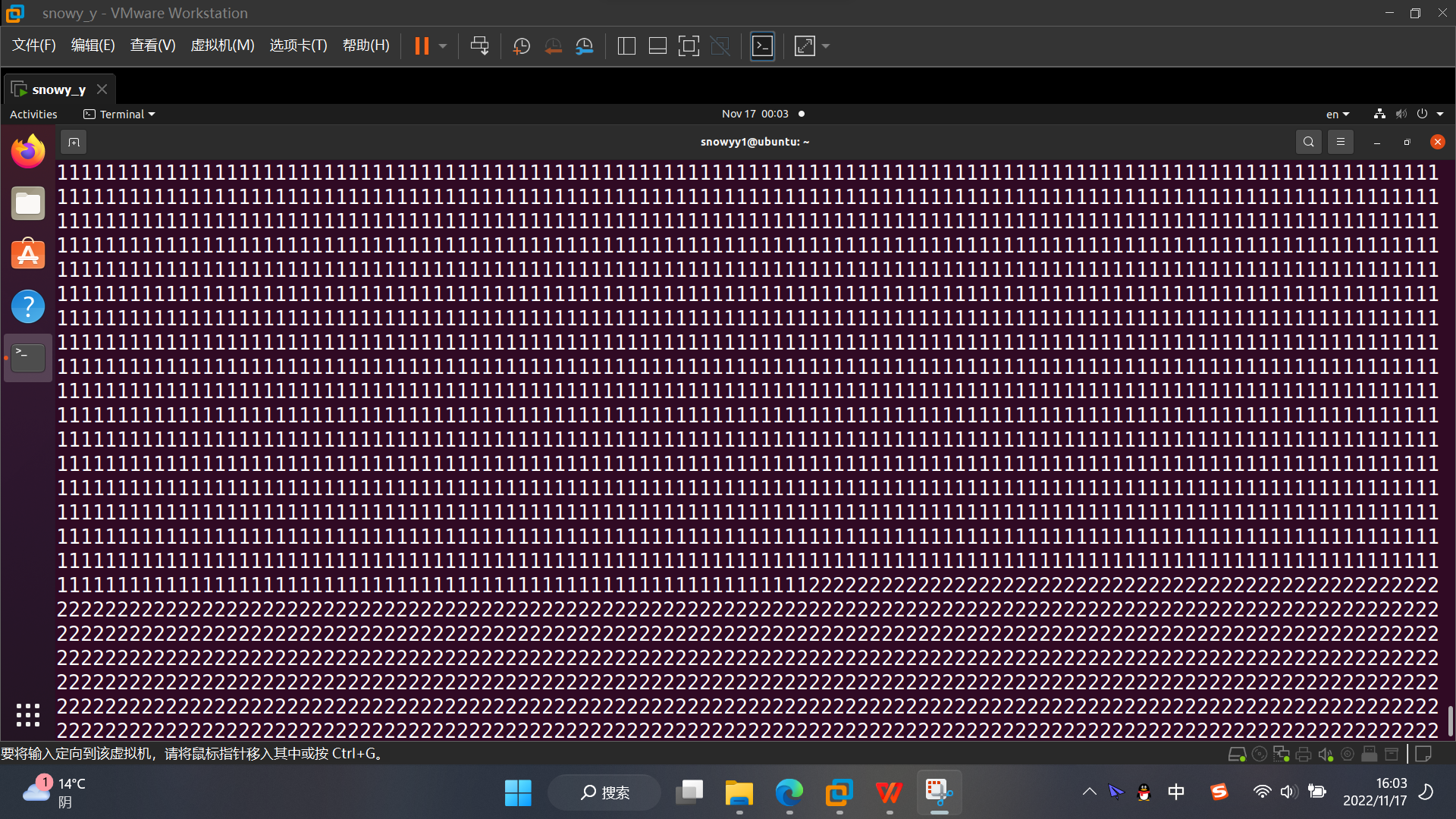
##### 先猜想一下这个程序的运行结果。分析管道通信是怎样实现同步与互斥的

终端会输出2000个1和2000个2但顺序不确定。

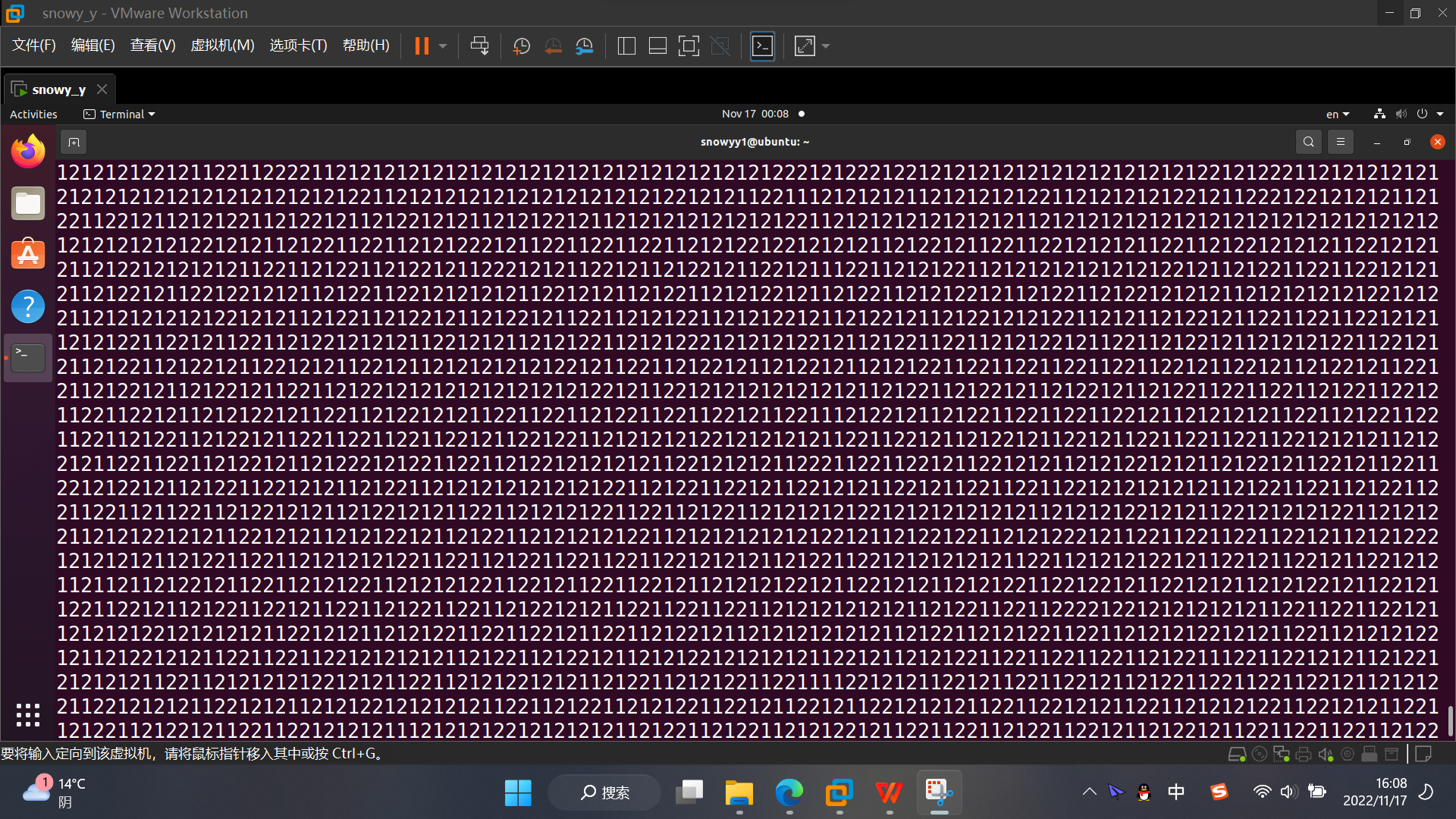
##### 然后按照注释里的要求把代码补充完整，运行程序；

##### 修改程序并运行，体会互斥锁的作用，比较有锁和无锁程序的运行结果，并解释之。

有锁运行结果如下：



去掉锁后结果如下：



Ps:刚开始并没有明显的12混合出现，在写入步骤后加上sleep(0.1)使结果更明显。

## 2-3实验回答问题

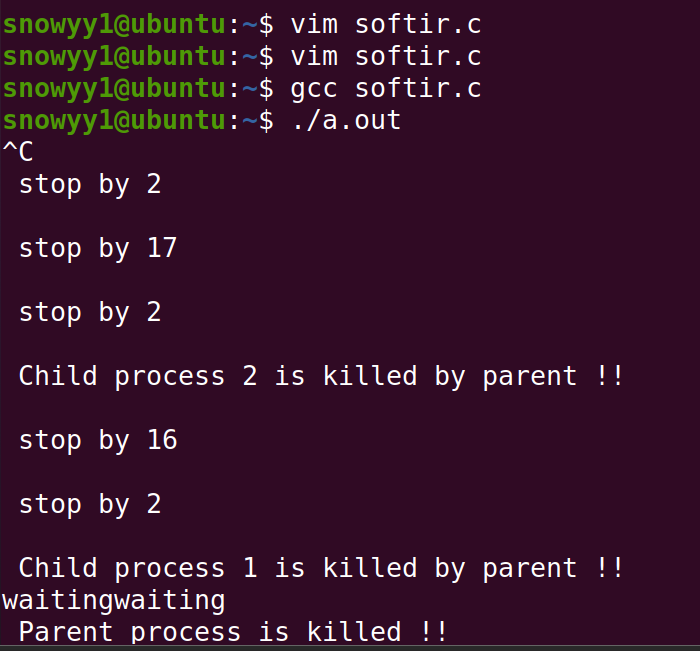
### 2.3.1软中断通信

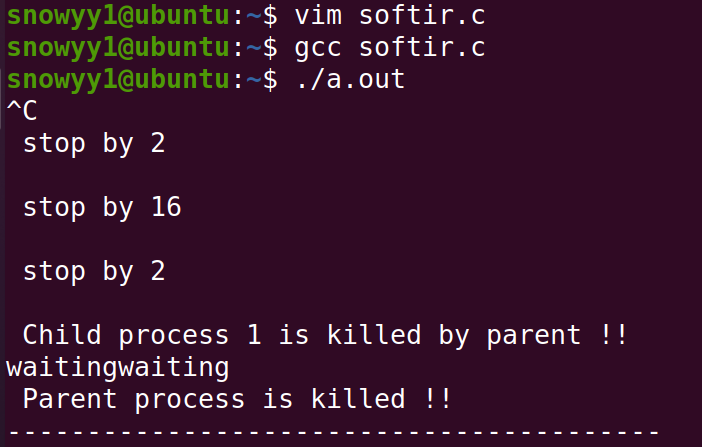
1. 你最初认为运行结果会怎么样？写出你猜测的结果

开始运行后应该开始无任何显示，等待键盘按下Delete键后显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”。

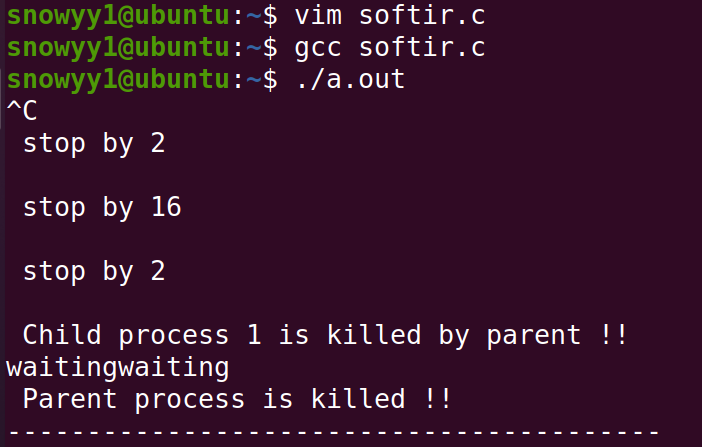
1. 实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将5秒内中断和5秒后中断的运行结果截图，试对产生该现象的原因进行分析。

实际运行代码，发现与使用的中断方法无关，两个子进程结束的先后顺序不固定，因为两者的执行顺序由系统随机决定。同时，我在stop函数中添加了确认调用者的输出，进一步分析了整个过程的内部细节。

在5秒内中断时，结果如下：



而对于五秒后中断，结果如下：



3 . 针对实验过程2，怎样修改的程序？修改前后程序的运行结果是什么？请截图说明。

截图见2，程序中通过加减sleep(5)操作实现结果变化

4. 针对实验过程3，程序运行的结果是什么样子？时钟中断有什么不同？

分析：三次参数为2的stop是因为signal(SIGINT,stop)在子进程中也在被运行，一旦输入delete，三个进程会分别运行一次stop，三个进程都收到了信号。两次参数为2的stop是因为有时父进程结束得太快，子进程还没有运行复制来的signal(SIGINT,stop)就结束了。

5.kill 命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？

kill命令使用了两次，分别给两个子进程以随机顺序发送16，17信号，但是子进程1先输出的概率大。通过将kill命令插入wait命令中间可以控制子进程执行顺序，因为父进程在第一个等待结束后才会发送下一个kill信号。（代码间softir.c有些地方被注释了）使用kill命令后，子进程接收到kill命令并调用stop函数，stop函数将wait\_flag置为0，输出被杀死的信号并结束。

6.使用kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

进程调用return函数和exit函数可以主动退出。在两种退出方式中，主动退出比较好，如果在某个子进程退出前父进程被强制退出，则子进程会被init进程接管；如果用kill命令杀死某个子进程而其父进程没有调用wait函数等待，则该子进程为处于僵死状态占用资源。

### 2.3.2管道通信

1. 你最初认为运行结果会怎么样？

终端会输出2000个1和2000个2但顺序不确定。

1. 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

输出了连续的2000个1和2000个2，两者先后不确定。

1. 实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？

通过flock函数实现便捷的上锁和解锁，如果没有互斥操作，1、2会混乱的输出。

1. 把README和程序源代码附到实验报告后

源码见附件pipecom.c

## 2-4实验总结

本次实验成功实现了进程的软中断和管道通信，通过解决大量实验过程中的问题，提高了对进程软中断和管道通信的理解，对Linux操作系统也更加的熟悉了。

### 2.4.1 实验中的问题与解决过程

1.问题：无法解释’stop by’中间出现的换行

解决：通过在子进程“signal”语句前加入输出标识符，发现该输出标识符会代替回车，猜测该回车是自带机制。

2.问题：最初进行进程管道通信时，即使不加锁也看不出来乱序输出

解决：在写入步骤后加上sleep(0.1)使结果更明显。

### 2.4.2 实验收获

本次实验成功实现了进程的软中断和管道通信，通过解决大量实验过程中的问题，提高了对进程软中断和管道通信的理解，对Linux操作系统也更加的熟悉了。

## 2-5 源码

Paperchange.c:

#include<stdio.h>

#include<stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#define AP 10

#define PP 3

#define TOTAL\_INSTRUCTION 20

//队列数据结构

int Queue[PP+1]={};

int head = 0;

int tail = 0;

//特殊的栈

int Stack[PP] = {};

int top = PP-1;

int bottom = PP-1;

//用于存储中间结果

int temp[PP][TOTAL\_INSTRUCTION];

//检查pagecontrol是否还有空位

int isConEmpty(int\* first\_empty,int pagecontrol[],int control\_num)

{

int flag = 0;

for(int iter = 0;iter<control\_num;iter++)

{

if(pagecontrol[iter]==-1)

{

flag = 1;

\*first\_empty = iter;

break;

}

}

return flag;

}

/\*

\* FIFO算法

\* page[cur]是当前需要调入的页面

\*/

void FIFO(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//队列记录的是最先放入数据的位置而不是页面号

Queue[tail]=first\_empty;

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

//如果没有空位，则替换最先

else

{

page[pagecontrol[Queue[head]]]=0;

pagecontrol[Queue[head]]=curpage;

page[curpage] = 1;

Queue[tail]=Queue[head];

if((head+1)%(PP+1)!=tail)

{

head = (head+1)%(PP+1);

}

else

{

//按照队列的设置不会出现队列空

printf("队列空！\n");

exit(1);

}

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

}

//LRU算法

void LRU(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//先检查在不在pagecontrol里面

for(int iter = 0; iter<PP;iter++)

{

if(pagecontrol[iter]==curpage)

{

int iter2;//renew the stack to ensure the newest list

for(iter2 = 0;iter2<PP;iter2++)

{

if(Stack[iter2]==iter)

break;

}

for(;iter2<PP-1;iter2++)

{

Stack[iter2]=Stack[iter2+1];

}

Stack[top]=iter;

return;

}

}

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//链表记录的是最先放入数据的位置而不是页面号

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

bottom--;

Stack[top]=first\_empty;

}

//如果没有空位，替换最近使用最少

else

{

page[pagecontrol[Stack[bottom+1]]]=0;

pagecontrol[Stack[bottom+1]]=curpage;

page[curpage] = 1;

int tmp = Stack[bottom+1];

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

Stack[top]=tmp;

}

}

int main()

{

//队列初始化为-1

for(int iter=0;iter<PP+1;iter++)

{

Queue[iter]=-1;

}

//栈初始化为-1

for(int iter=0;iter<PP;iter++)

{

Stack[iter]=-1;

}

/\*定义变量

\* page是页面

\* pagecontrol是内存分配的页面，初始化为-1

\* pageseq是页面使用的顺序

\*/

printf("-----------------------------------------------------\n");

printf("本程序模拟两种替换算法：FIFO和LRU。\n");

int page[AP];

int pagecontrol[PP];

for(int iter = 0;iter<PP;iter++)

{

pagecontrol[iter] = -1;

}

for(int iter = 0;iter<AP;iter++)

{

page[iter] = 0;

}

int pageseq[TOTAL\_INSTRUCTION] = {};

//为pageseq的每一个元素赋随机值

srand((unsigned int)getpid());

for(int iter = 0;iter<TOTAL\_INSTRUCTION;iter++)

{

pageseq[iter]=rand()%AP;

}

//测试

//int pageseq[TOTAL\_INSTRUCTION] = {7,0,1,2,0,3,1,4,2,3,0,2,5,1,3,0,1,7,0,1};

//替换算法类型，0为FIFO，1为LRU

printf("可分配的物理块数量为：%d\n",PP);

printf("页面访问顺序为：\n%d",pageseq[0]);

for(int iter = 1;iter<TOTAL\_INSTRUCTION;iter++)

{

printf(" %d",pageseq[iter]);

}

printf("\n");

int algorithm\_type = 0;

printf("请输入替换算法类型：0为FIFO，1为LRU。\n");

scanf("%d",&algorithm\_type);

switch(algorithm\_type)

{

case 0: {printf("FIFO算法。\n");break;}

case 1: {printf("LRU算法。\n");break;}

default: {printf("不支持该种替换算法！\n");exit(1);}

}

//cur是当前页面序列下标

int cur = 0;

//diseffect是未命中次数

int diseffect = 0;

while(cur!=TOTAL\_INSTRUCTION)

{

//如果序列当前页面在已经分配的页面中

if(page[pageseq[cur]]==1)

//FIFO什么都不用做

//LRU要更新记录

{

if(algorithm\_type==1)

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//如果不在当前已经分配的页面中

else

{

diseffect++;

//使用FIFO算法

if(algorithm\_type == 0)

{

FIFO(pageseq[cur],page,pagecontrol,PP);

}

//使用LRU算法

else

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//记录pagecontrol的值便于输出

for(int iter = 0;iter<PP;iter++)

{

temp[iter][cur]=pagecontrol[iter];

}

//查看下一个待调用的页面

cur++;

}

//计算命中率

double hit\_rate = 100\*(1-((double)diseffect/TOTAL\_INSTRUCTION));

//输出相关信息

printf("-----------------------------------------------------\n");

printf("替换次序(-1表示空)：\n");

for(int iter = 0; iter<PP;iter++)

{

printf("%d",temp[iter][0]);

for(int iter2 = 1;iter2<TOTAL\_INSTRUCTION;iter2++)

{

printf("\t%d",temp[iter][iter2]);

}

printf("\n");

}

printf("-----------------------------------------------------\n");

printf("共访问页面次数，即TOTAL\_INSTRUCTION的大小：%d\n",TOTAL\_INSTRUCTION);

printf("未命中次数，即diseffect为：%d\n",diseffect);

printf("命中率，即1-diseffect / total\_instruction\*100%%为：%.2f%%\n",hit\_rate);

printf("-----------------------------------------------------\n");

return 0;

}

Pipecom.c:

#include<stdio.h>

#include<sys/types.h>

#include<signal.h>

#include<unistd.h>

#include<stdlib.h>

#include<wait.h>

int pid1,pid2; // 定义两个进程变量

int main( ) {

int fd[2];

char InPipe[3000]; // 定义读缓冲区

char c1='1',c2='2';

pipe(fd); // 创建管道

while((pid1 = fork( )) == -1); // 如果进程1创建不成功,则空循环

if(pid1 == 0) { // 如果子进程1创建成功,pid1为进程号

//在子进程1中

//补充； // 锁定管道

//lockf(fd[1],1,0);

//补充; // 分2000次每次向管道写入字符’1’

for(int i = 0;i<2000;i++)

{

//sprintf(InPipe,1);

write(fd[1],&c1,1);

sleep(0.1);

}

sleep(5); // 等待读进程读出数据

//lockf(fd[1],0,0);

//补充; // 解除管道的锁定

exit(0); // 结束进程1

}

else {

while((pid2 = fork()) == -1); // 若进程2创建不成功,则空循环

if(pid2 == 0) {

//在子进程2中

//lockf(fd[1],1,0);

for(int i = 0;i<2000;i++)

{

write(fd[1],&c2,1);

sleep(0.1);

}

//补充; // 分2000次每次向管道写入字符’2’

sleep(5);

//lockf(fd[1],0,0);

exit(0);

}

else {

//在父进程中

wait(0);

//补充; // 等待子进程1 结束

wait(0); // 等待子进程2 结束

//补充; // 从管道中读出4000个字符

lockf(fd[0],1,0);

read(fd[0], InPipe, 4000);

InPipe[4000] = '\0';

lockf(fd[0],1,0);

//补充; // 加字符串结束符

printf("%s\n",InPipe); // 显示读出的数据

printf("---------------------------------------------\n");

exit(0); // 父进程结束

}

}

}

Softir.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include<stdlib.h>

#include<sys/wait.h>

int wait\_flag;

void stop(int signum);

int main(){

//定义两个进程号变量

int pid1, pid2;

//alarm(1);

//捕捉中断信号sig后执行function规定的操作。

signal(SIGINT,stop)//

//signal(SIGQUIT,stop);//3,ctrl \

//signal(SIGALRM,stop);//alarm函数

//若创建子进程1不成功,则空循环

while((pid1 = fork( )) == -1);

//子进程创建成功,pid1为进程号

//在父进程里

if(pid1 > 0) {

//创建子进程2 (与子进程1并列)

while((pid2 = fork( )) == -1);

//在父进程里

if(pid2 > 0) {

wait\_flag = 1;

// 父进程等待5秒

// sleep(5);

//收到中断信号后stop函数中将wait\_flag置为0，再杀死子进程

//signal(SIGALRM,stop);

while(wait\_flag == 1);

// 杀死进程1发中断号16

//kill的作用不是使进程结束，而是发送信号让子进程自己结束。

kill(pid1,16);

//--------------------------

// 杀死进程2发中断号17

kill(pid2,17);

// 等待第1个子进程1结束的信号

printf("waiting");

wait(0);

// 等待第2个子进程2结束的信号

printf("waiting");

wait(0);

printf("\n Parent process is killed !!\n");

printf("-----------------------------------------\n");

// 父进程结束

exit(0);

}

//在pid2里

else {

wait\_flag = 1;

//补充-----------------------

// 等待进程2被杀死的中断号17

signal(17,stop);

//--------------------------

while(wait\_flag==1);

printf("\n Child process 2 is killed by parent !!\n");

exit(0);

}

}

//在pid1里

else {

wait\_flag = 1;

// 等待进程1被杀死的中断号16

signal(16,stop);

while(wait\_flag==1);

printf("\n Child process 1 is killed by parent !!\n");

exit(0);

}

}

void stop(int signum) {

//补充-----------------------

wait\_flag = 0;

printf("\n stop by %d \n",signum);

//--------------------------

}

# 3 页面置换算法

## 实验目的

模拟实现页面置换算法中的模拟实现FIFO算法、LRU算法，从而加深对页面置换算法的理解。

## 实验内容

实现FIFO算法、LRU算法，输出两种算法的缺页率

## 实验思想

### FIFO算法

在分配内存页面数（AP）小于进程页面数（PP）时，当然是最先运行的AP个页面放入内存；

这时又需要处理新的页面，则将原来放的内存中的AP个页中最先进入的调出（FIFO），再将新页面放入；

以后如果再有新页面需要调入，则都按上述规则进行。

算法特点：所使用的内存页面构成一个队列。

### LRU算法

当内存分配页面数（AP）小于进程页面数（PP）时，把最先执行的AP个页面放入内存。

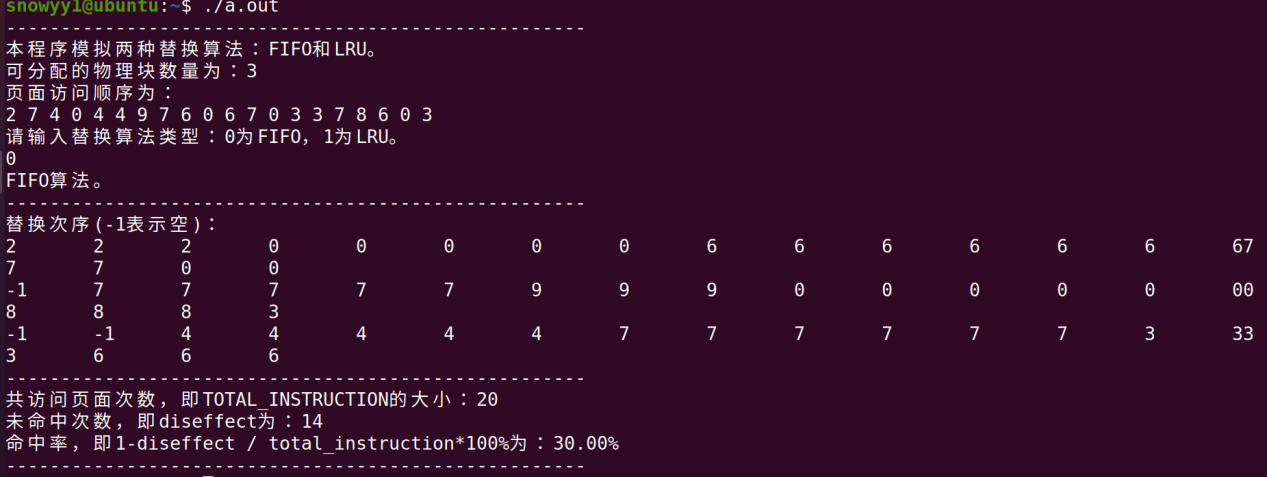
当需调页面进入内存，而当前分配的内存页面全部不空闲时，选择将其中最长时间没有用到的那一页调出，以空出内存来放置新调入的页面（LRU）。

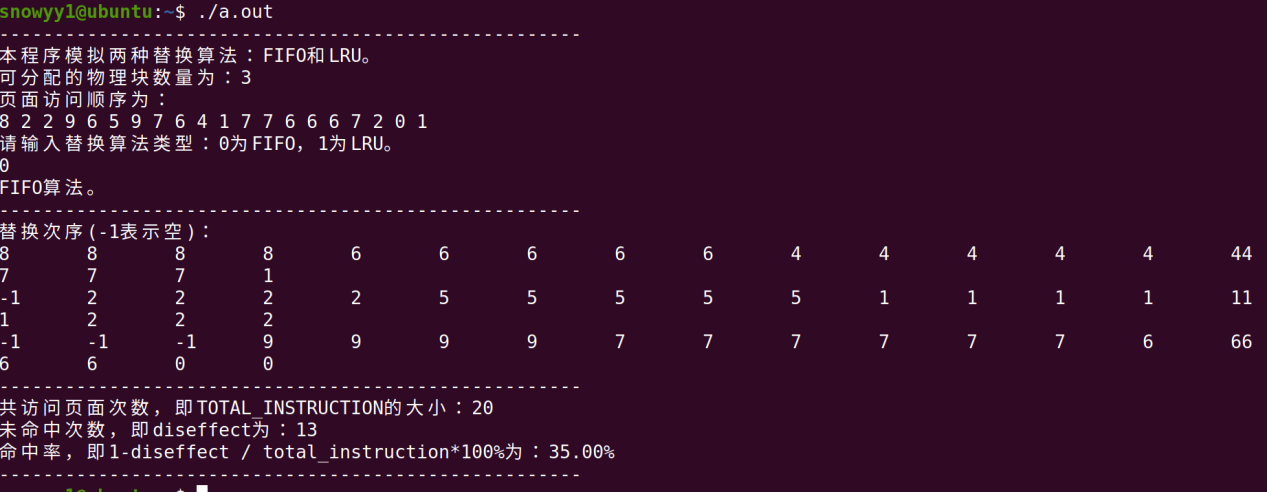
算法特点：每个页面都有属性来表示有多长时间未被CPU使用的信息。

## 3.4 实验步骤

### 1 FIFO算法

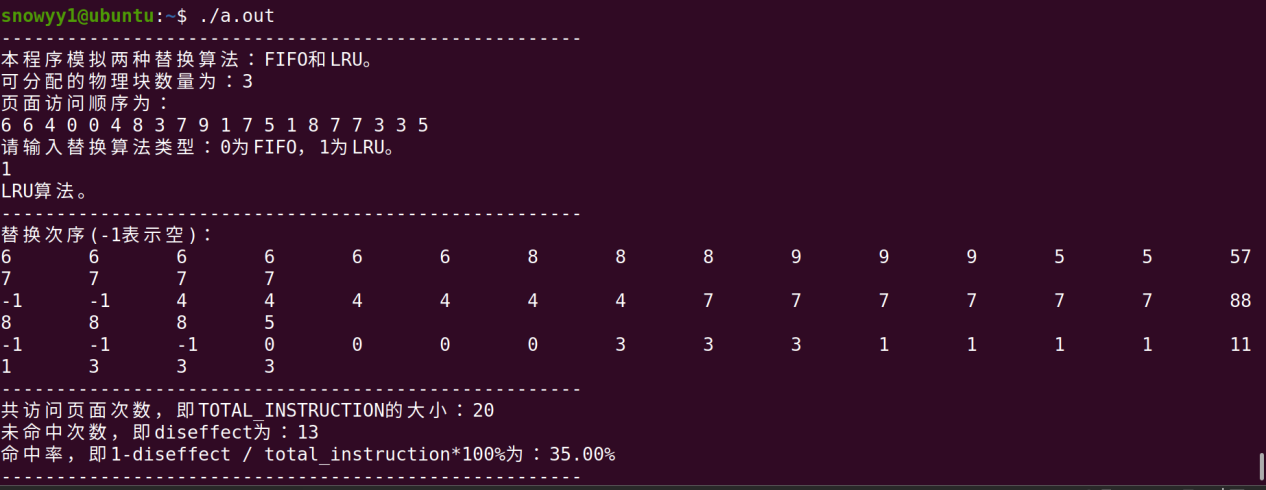
实现了基本的FIFO算法，同时观察了belady现象

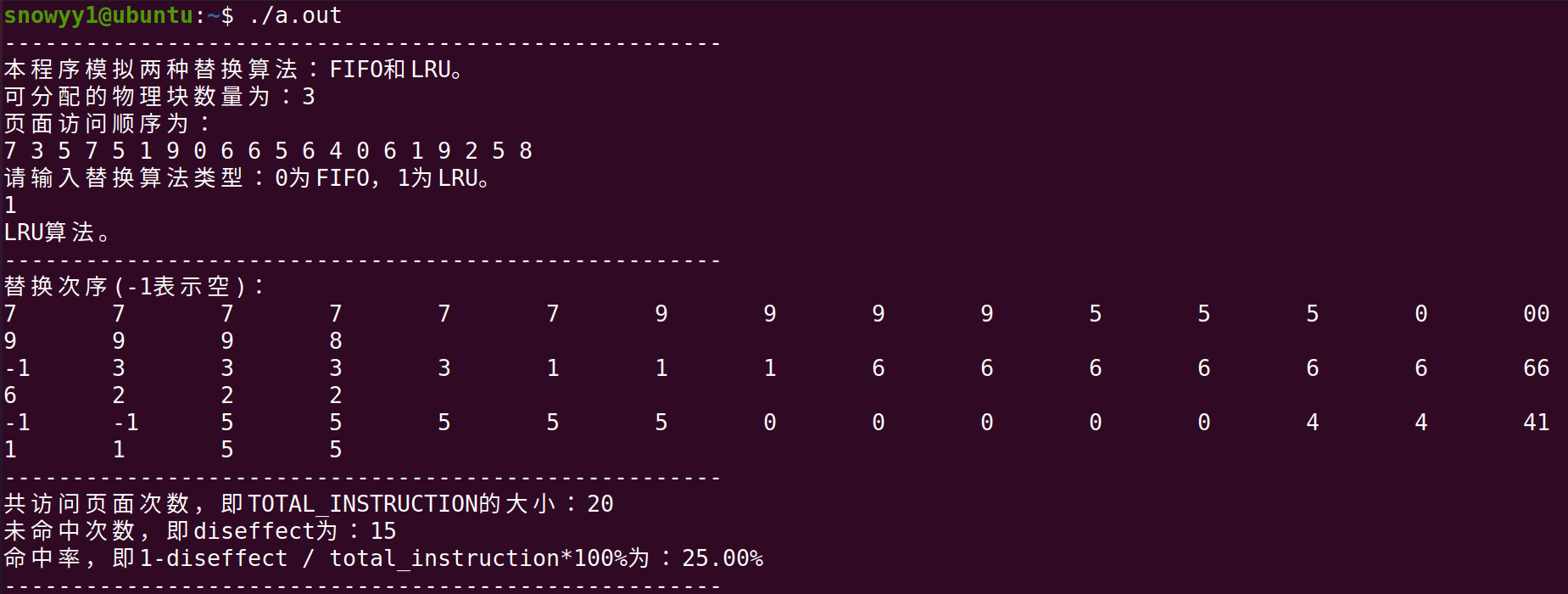




发现FIFO算法两次命中率分别为30%和35%

### 2 LRU算法





LRU算法两次命中率分别为35%和25%

## 源码

源码请见附件pagechange.c:

#include<stdio.h>

#include<stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#define AP 10

#define PP 3

#define TOTAL\_INSTRUCTION 20

//队列数据结构

int Queue[PP+1]={};

int head = 0;

int tail = 0;

//特殊的栈

int Stack[PP] = {};

int top = PP-1;

int bottom = PP-1;

//用于存储中间结果

int temp[PP][TOTAL\_INSTRUCTION];

//检查pagecontrol是否还有空位

int isConEmpty(int\* first\_empty,int pagecontrol[],int control\_num)

{

int flag = 0;

for(int iter = 0;iter<control\_num;iter++)

{

if(pagecontrol[iter]==-1)

{

flag = 1;

\*first\_empty = iter;

break;

}

}

return flag;

}

/\*

\* FIFO算法

\* page[cur]是当前需要调入的页面

\*/

void FIFO(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//队列记录的是最先放入数据的位置而不是页面号

Queue[tail]=first\_empty;

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

//如果没有空位，则替换最先

else

{

page[pagecontrol[Queue[head]]]=0;

pagecontrol[Queue[head]]=curpage;

page[curpage] = 1;

Queue[tail]=Queue[head];

if((head+1)%(PP+1)!=tail)

{

head = (head+1)%(PP+1);

}

else

{

//按照队列的设置不会出现队列空

printf("队列空！\n");

exit(1);

}

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

}

//LRU算法

void LRU(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//先检查在不在pagecontrol里面

for(int iter = 0; iter<PP;iter++)

{

if(pagecontrol[iter]==curpage)

{

int iter2;//renew the stack to ensure the newest list

for(iter2 = 0;iter2<PP;iter2++)

{

if(Stack[iter2]==iter)

break;

}

for(;iter2<PP-1;iter2++)

{

Stack[iter2]=Stack[iter2+1];

}

Stack[top]=iter;

return;

}

}

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//链表记录的是最先放入数据的位置而不是页面号

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

bottom--;

Stack[top]=first\_empty;

}

//如果没有空位，替换最近使用最少

else

{

page[pagecontrol[Stack[bottom+1]]]=0;

pagecontrol[Stack[bottom+1]]=curpage;

page[curpage] = 1;

int tmp = Stack[bottom+1];

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

Stack[top]=tmp;

}

}

int main()

{

//队列初始化为-1

for(int iter=0;iter<PP+1;iter++)

{

Queue[iter]=-1;

}

//栈初始化为-1

for(int iter=0;iter<PP;iter++)

{

Stack[iter]=-1;

}

/\*定义变量

\* page是页面

\* pagecontrol是内存分配的页面，初始化为-1

\* pageseq是页面使用的顺序

\*/

printf("-----------------------------------------------------\n");

printf("本程序模拟两种替换算法：FIFO和LRU。\n");

int page[AP];

int pagecontrol[PP];

for(int iter = 0;iter<PP;iter++)

{

pagecontrol[iter] = -1;

}

for(int iter = 0;iter<AP;iter++)

{

page[iter] = 0;

}

int pageseq[TOTAL\_INSTRUCTION] = {};

//为pageseq的每一个元素赋随机值

srand((unsigned int)getpid());

for(int iter = 0;iter<TOTAL\_INSTRUCTION;iter++)

{

pageseq[iter]=rand()%AP;

}

//测试

//int pageseq[TOTAL\_INSTRUCTION] = {7,0,1,2,0,3,1,4,2,3,0,2,5,1,3,0,1,7,0,1};

//替换算法类型，0为FIFO，1为LRU

printf("可分配的物理块数量为：%d\n",PP);

printf("页面访问顺序为：\n%d",pageseq[0]);

for(int iter = 1;iter<TOTAL\_INSTRUCTION;iter++)

{

printf(" %d",pageseq[iter]);

}

printf("\n");

int algorithm\_type = 0;

printf("请输入替换算法类型：0为FIFO，1为LRU。\n");

scanf("%d",&algorithm\_type);

switch(algorithm\_type)

{

case 0: {printf("FIFO算法。\n");break;}

case 1: {printf("LRU算法。\n");break;}

default: {printf("不支持该种替换算法！\n");exit(1);}

}

//cur是当前页面序列下标

int cur = 0;

//diseffect是未命中次数

int diseffect = 0;

while(cur!=TOTAL\_INSTRUCTION)

{

//如果序列当前页面在已经分配的页面中

if(page[pageseq[cur]]==1)

//FIFO什么都不用做

//LRU要更新记录

{

if(algorithm\_type==1)

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//如果不在当前已经分配的页面中

else

{

diseffect++;

//使用FIFO算法

if(algorithm\_type == 0)

{

FIFO(pageseq[cur],page,pagecontrol,PP);

}

//使用LRU算法

else

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//记录pagecontrol的值便于输出

for(int iter = 0;iter<PP;iter++)

{

temp[iter][cur]=pagecontrol[iter];

}

//查看下一个待调用的页面

cur++;

}

//计算命中率

double hit\_rate = 100\*(1-((double)diseffect/TOTAL\_INSTRUCTION));

//输出相关信息

printf("-----------------------------------------------------\n");

printf("替换次序(-1表示空)：\n");

for(int iter = 0; iter<PP;iter++)

{

printf("%d",temp[iter][0]);

for(int iter2 = 1;iter2<TOTAL\_INSTRUCTION;iter2++)

{

printf("\t%d",temp[iter][iter2]);

}

printf("\n");

}

printf("-----------------------------------------------------\n");

printf("共访问页面次数，即TOTAL\_INSTRUCTION的大小：%d\n",TOTAL\_INSTRUCTION);

printf("未命中次数，即diseffect为：%d\n",diseffect);

printf("命中率，即1-diseffect / total\_instruction\*100%%为：%.2f%%\n",hit\_rate);

printf("-----------------------------------------------------\n");

return 0;

}

## 扩展要求

### kill 和 signal 怎样实现进程之间的通信？

当父进程调用Kill函数，给子进程释放信号，子进程使用signal函数接受信号，并且结束自己的进程

### 在管道通信实验中怎样实现同步与互斥？

父子进程的同步主要表现在两个方面：1，父进程读出之前确定管道中有数据，否则阻塞自己，这一点通过系统调用wait()函数既可以实现，当子进程结束时父进程才执行，那么此时管道中肯定有子进程写入的数据了；2，子进程在写入之前要确定管道中的数据已被父进程读出，否则不能写入或者阻塞自己。这可以通过进程见的互斥来间接办到，因为子进程间的互斥，所以每个子进程在执行开始都对管道pipe加锁，并且子进程在向管道中写入数据后还有调用sleep()系统用调用睡眠若干时间，那么就可保证父进程能够从管道中读出数据，然后下一子进程才能写入。

### 如果是FIFO，观察BLEADY现象。

实验中通过增加内存中可存储的页数，发现缺页率有概率增加，这证明了Belady现象的存在。

# 4 动态模块与设备驱动

## 实验目的

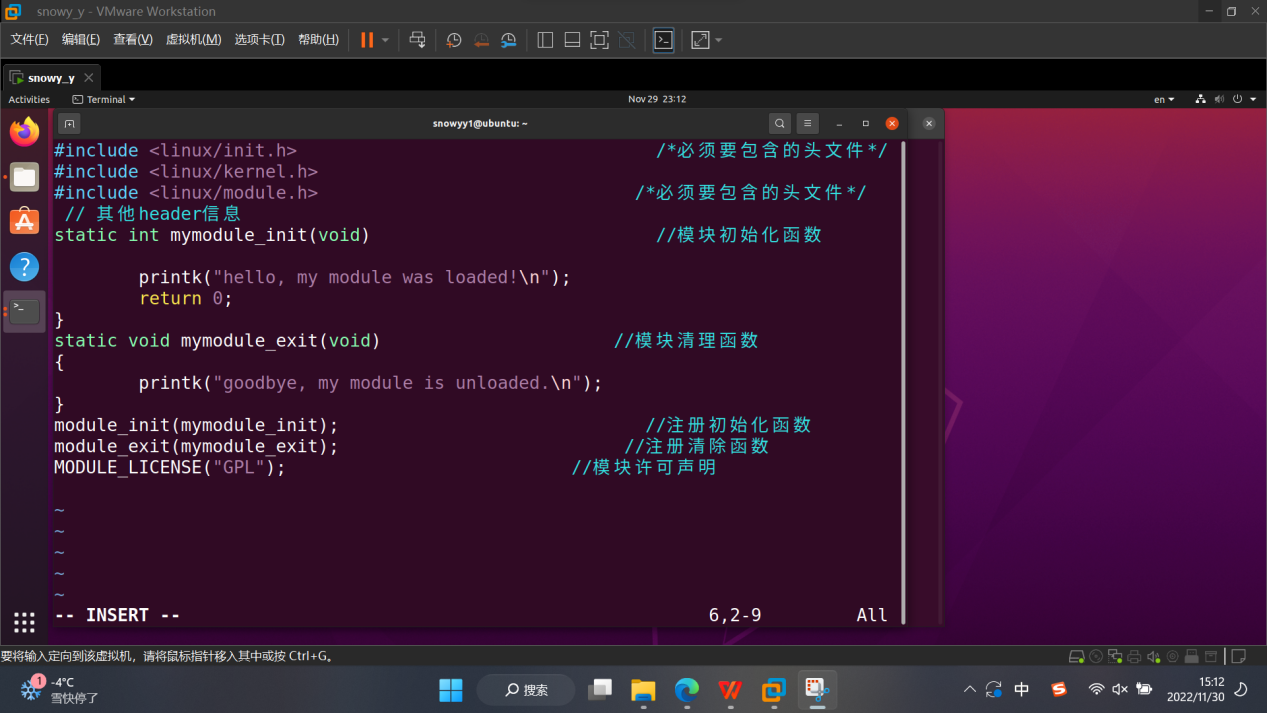
熟悉Linux操作系统的模块安装与编译，为后续字符驱动设备实验配制相关环境。

## 4.2 实验内容

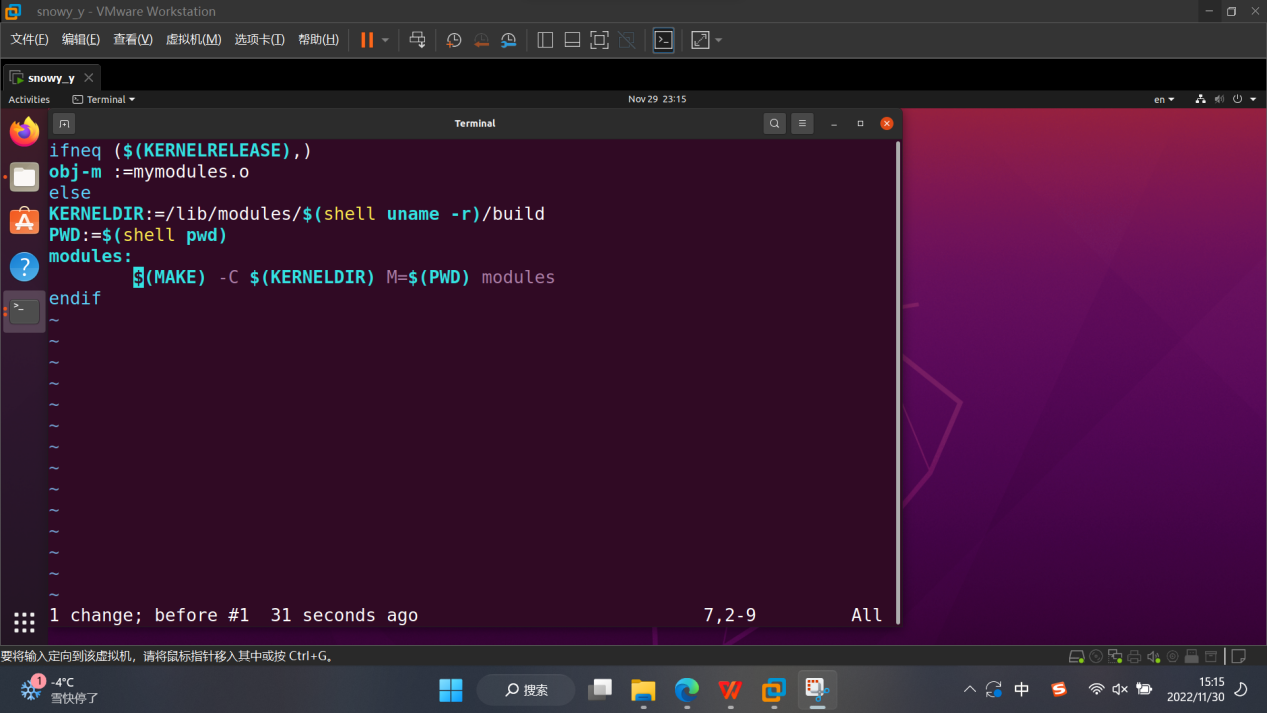
编译、安装、卸载内核模块，并通过模块日志查看模块的输出状况。

## 实验步骤（与问题解决）

#### 步骤一：设计内核模块程序mymodules.c



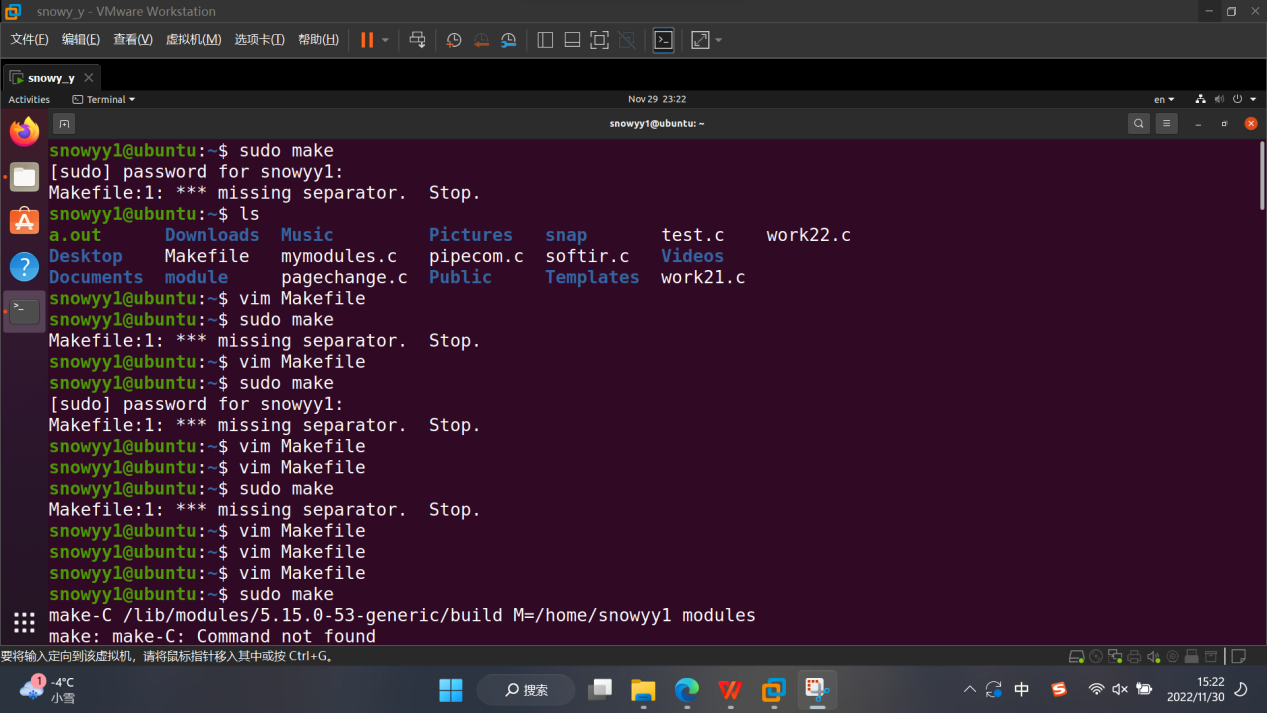
#### 步骤二：在相同文件夹下创建Makefile文件用来编译



Ps:问题：1.没有安装make，尝试在root权限下用apt-get install make失败；换源也失败；查找博客有人说是虚拟机没有联网的问题，尝试手动配置虚拟机网络，发现该电脑禁止共享网络；尝试修复注册表问题，下载了ccleaner，问题未解决。

解决：发现只给apt命令赋予权限，即（sudo apt-get install make）即可避开注册表等问题

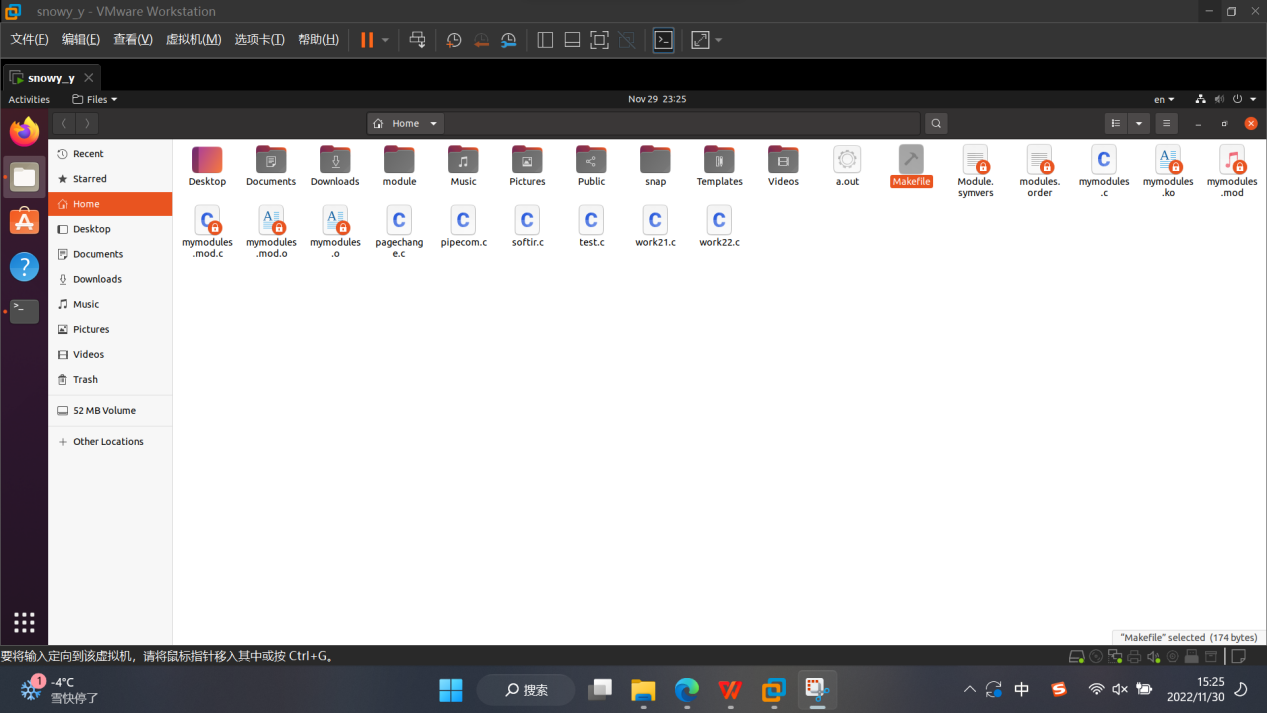
1. make命令执行失败，Makefile出现如下问题：



解决：通过博客了解到Makefile文件对格式敏感，只有命令行之前可以使用tab，且-m之前要空格（像终端中一样）

#### 步骤三：编译内核模块程序

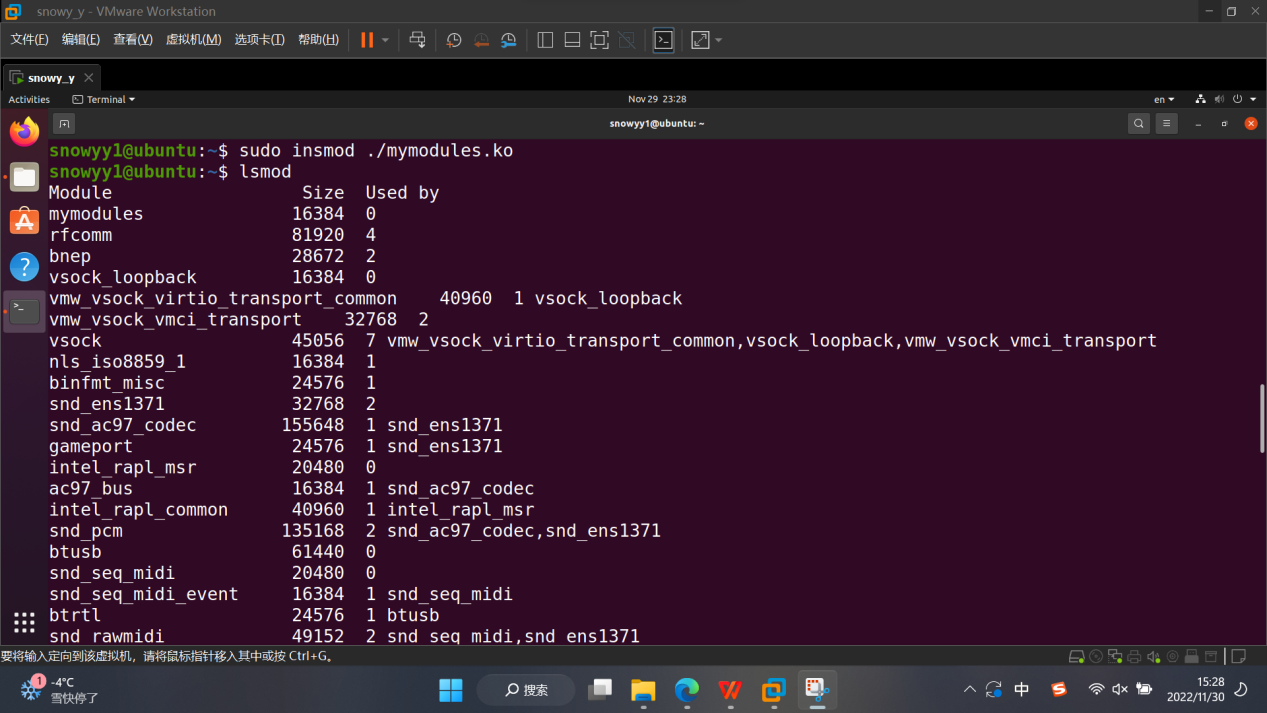
编译后结果如下：



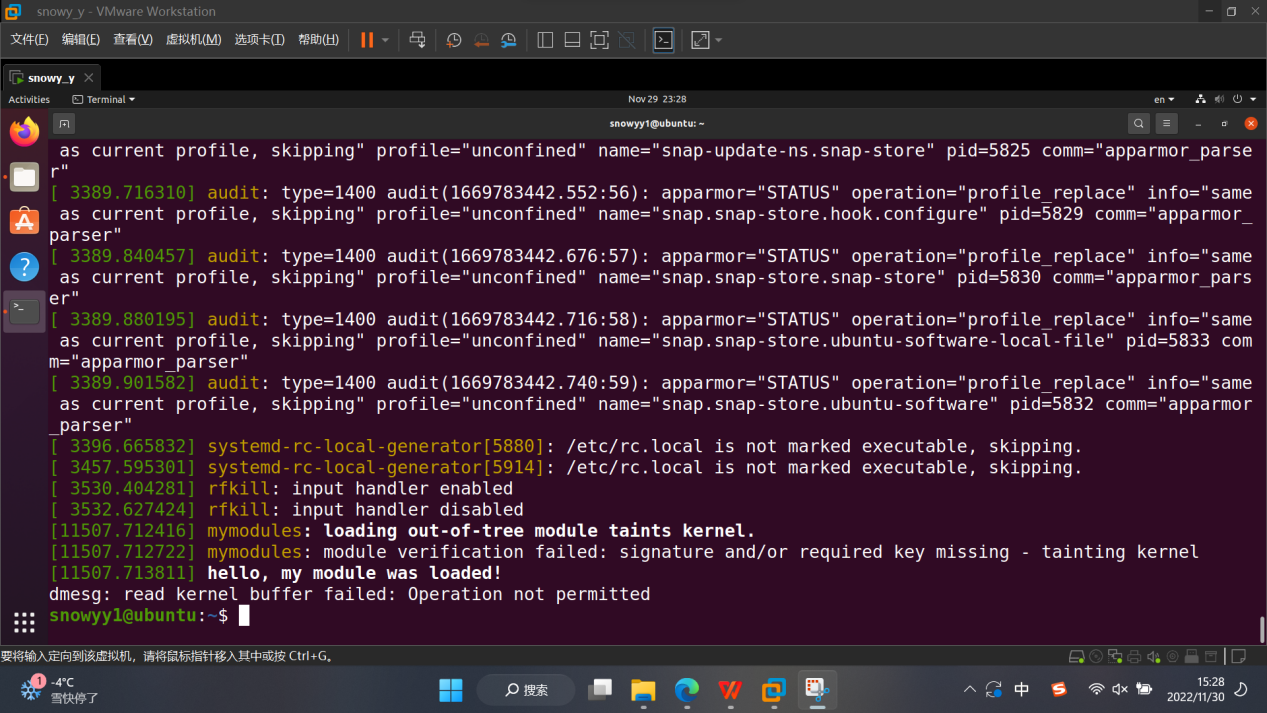
Ps:问题：出现编译错误

解决：函数返回值应是整形的没给返回值，另一处未加分号

用lsmod查看模块：



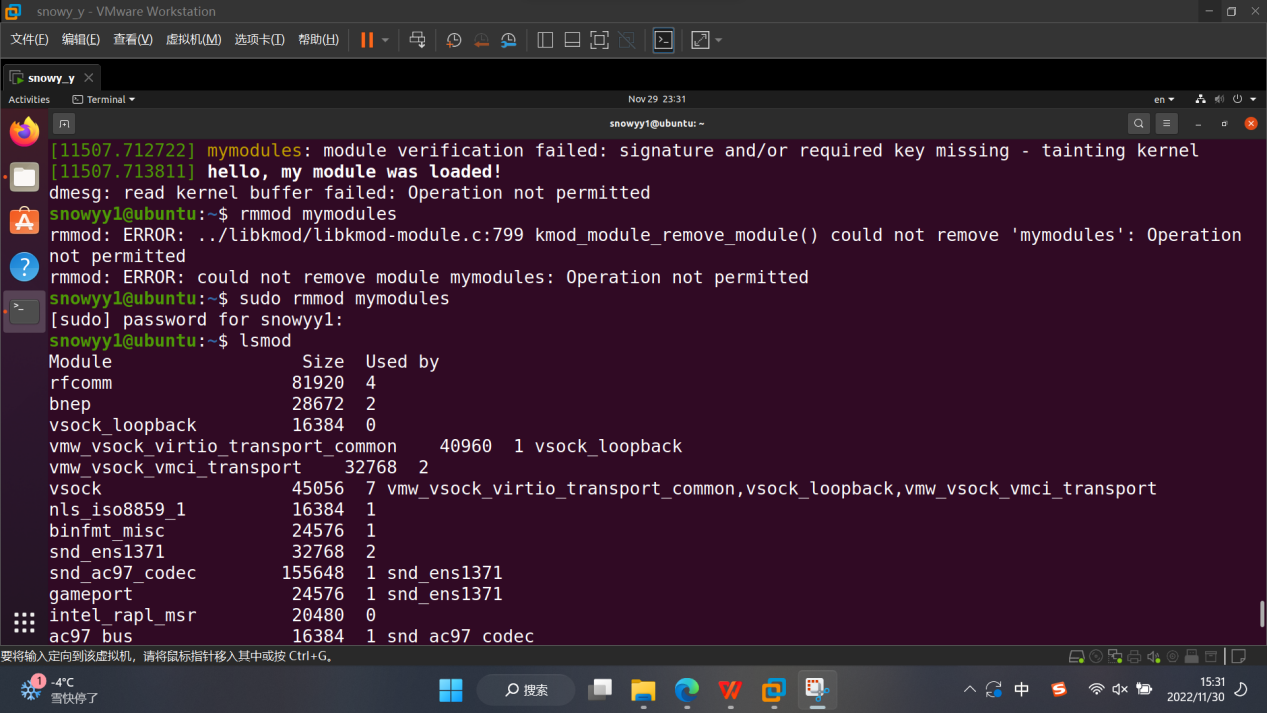
用dmesg -c查看日志：



#### 步骤五：卸载内核动态模块

使用命令：rmmod mymodules

可以看到mymodules已经被卸载



Ps:问题：出现operation not permitted

解决：使用sudo给管理员权限

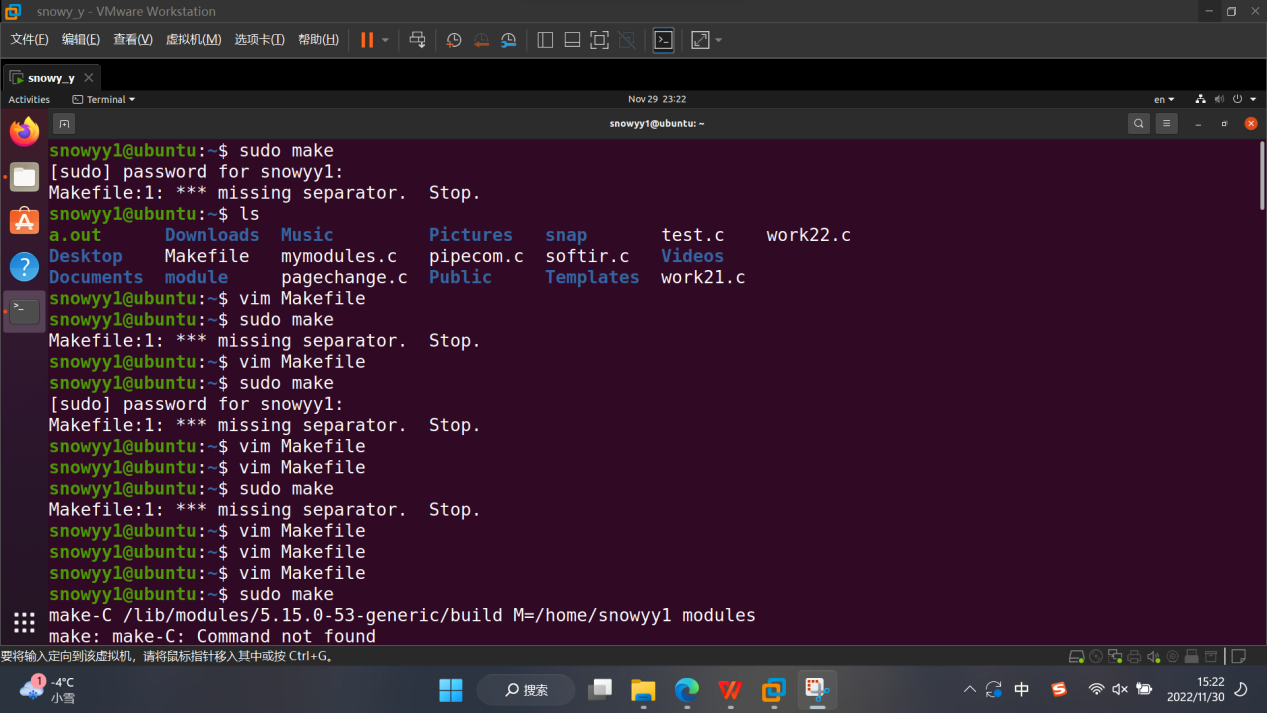
## 4.4实验总结

### 4.4.1 实验中的问题与解决过程

问题：1.没有安装make，尝试在root权限下用apt-get install make失败；换源也失败；查找博客有人说是虚拟机没有联网的问题，尝试手动配置虚拟机网络，发现该电脑禁止共享网络；尝试修复注册表问题，下载了ccleaner，问题未解决。

解决：发现只给apt命令赋予权限，即（sudo apt-get install make）即可避开注册表等问题

1. make命令执行失败，Makefile出现如下问题：



解决：通过博客了解到Makefile文件对格式敏感，只有命令行之前可以使用tab，且-m之前要空格（像终端中一样）

3.问题：出现编译错误

解决：函数返回值应是整形的没给返回值，另一处未加分号

4.问题：出现operation not permitted

解决：使用sudo给管理员权限

### 4.4.2 实验收获

熟悉了动态模块的编译、加载与卸载，熟悉了Linux操作系统的管理员权限使用技巧等。

## 4.5 源码

mymodules.c:

#include <linux/init.h> /\*必须要包含的头文件\*/

#include <linux/kernel.h>

#include <linux/module.h> /\*必须要包含的头文件\*/

// 其他header信息

static int mymodule\_init(void) //模块初始化函数

{

printk("hello, my module was loaded!\n");

return 0;

}

static void mymodule\_exit(void) //模块清理函数

{

printk("goodbye, my module is unloaded.\n");

}

module\_init(mymodule\_init); //注册初始化函数

module\_exit(mymodule\_exit); //注册清除函数

MODULE\_LICENSE("GPL"); //模块许可声明

Makefile:

ifneq ($(KERNELRELEASE),)

obj-m :=mymodules.o

else

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

PWD:=$(shell pwd)

modules:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

# 5 字符设备驱动

## 5.1 实验目的

1.理解LINUX字符设备驱动程序的基本原理

2.掌握字符设备的驱动运作机制

3.学会编写字符设备驱动程序

## 5.2实验内容

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作，并编写聊天程序实现对该设备的同步和互斥操作。同时实现私聊功能。

## 5.3 实验步骤

#### 文件准备

文件结构：chardev文件夹下有chardev.c和chat.c和do.sh和del.sh和Makefile文件，编写细节见下：

步骤一：编写字符驱动文件chardev.c

以模板为基础

包含一个\_device\_data结构体用来设定设备数据内容

包含一个device\_read函数用来将键盘输入的字符存入buffer

包含一个device\_write函数用来将buffer中的内容输出到屏幕

包含一个device\_open函数用来打开设备

包含一个device\_release函数用来释放资源

写一个file\_operation来将函数与文件对应

包含一个设备初始化函数和设备结束函数

步骤二：编写编译文件Makefile

编写过程要注意语法和tab的使用，具体内容见附件



步骤三：安装和卸载字符设备程序

安装模块，并且创建是个设备文件，最后开启这些文件的读写权限

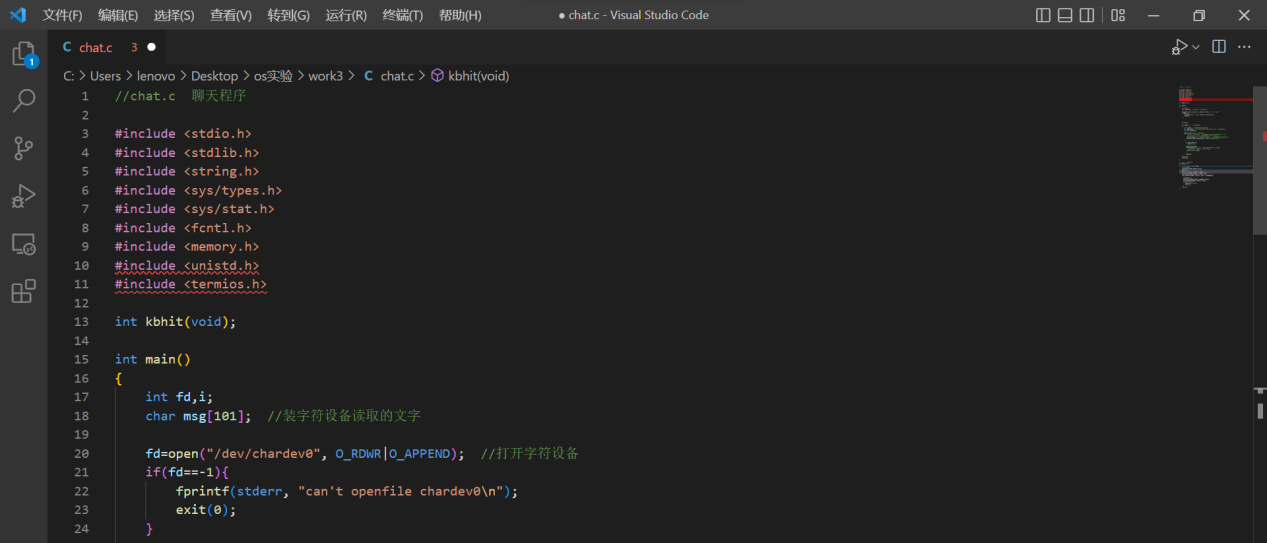


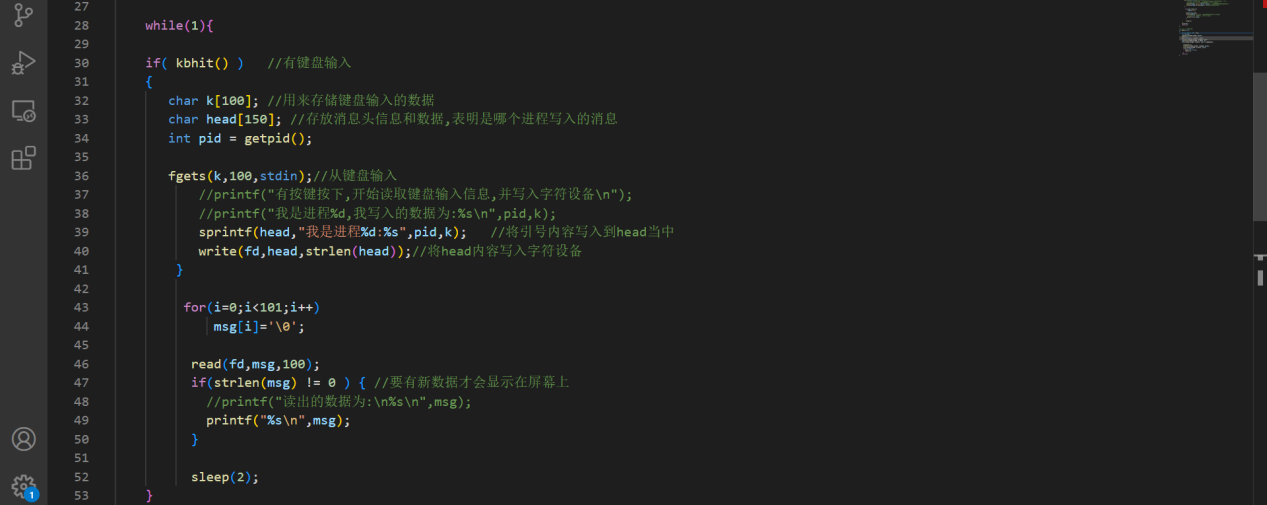
步骤四：编写聊天程序chat.c

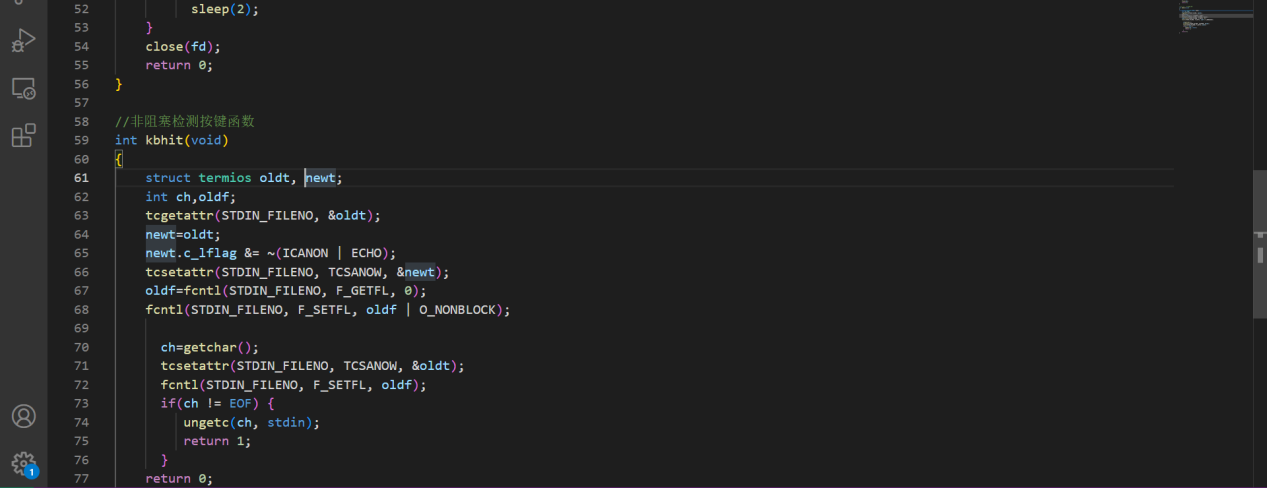
使用非阻塞控制方法，如果某用户的键盘有输入，其他用户就可以看到该用户的输入，并且在屏幕上输出是哪个进程输出了内容方便沟通。Ps:文件中从键盘读取内容使用了fgets()函数，对head内容输如使用了sprintf()函数

错误：刚开始会出现其他用户输出总是附带部分以前用户的输出的问题

解决：每次一个用户开始输出之前，总是将中间数组msg[101]清零







#### 2.编译安装过程

将makefile和要编译的文件放在同一个目录下，在终端输入“make”命令，编译字符驱动，生成编译文件。

给do.sh和del.sh特权。命令是：chmod +x do.sh 和 chmod +x del.sh

运行do.sh安装字符驱动 方法：./do.sh

编译chat.c文件，方法：gcc -o chat chat.c

运行三次chat程序，创建三个用户窗口来做test。方法：./chat

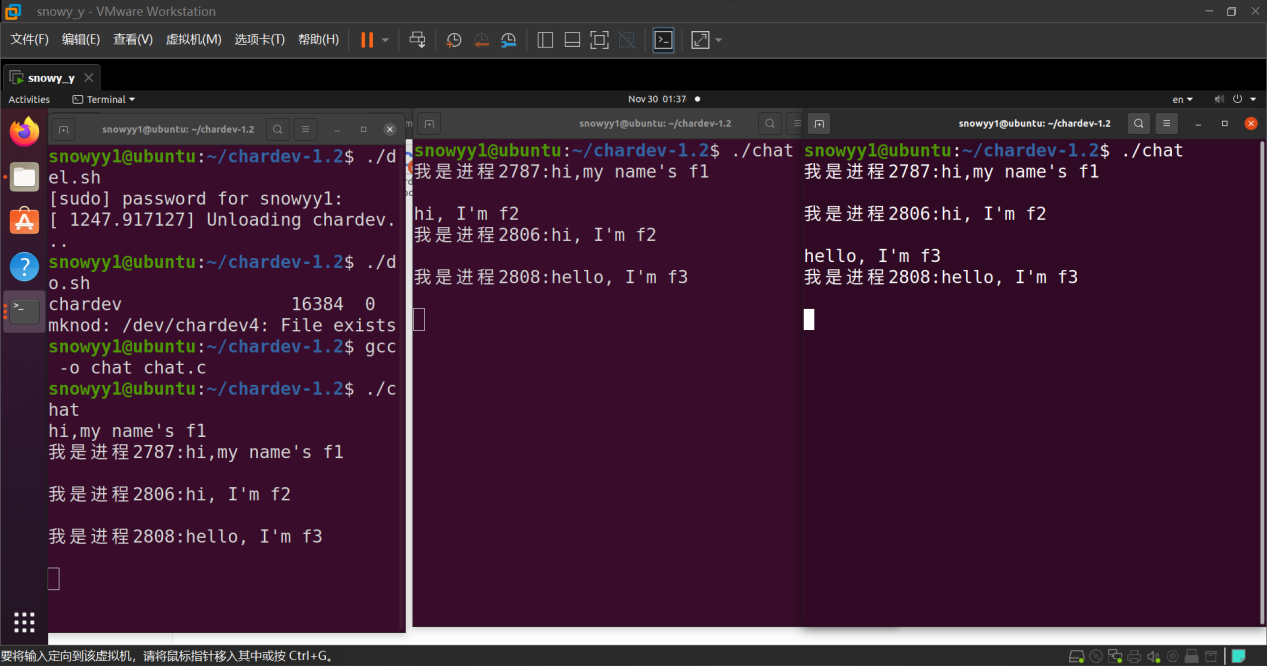
最后./del.sh卸载字符驱动

在代码中用符号位实现了读写进程的互斥

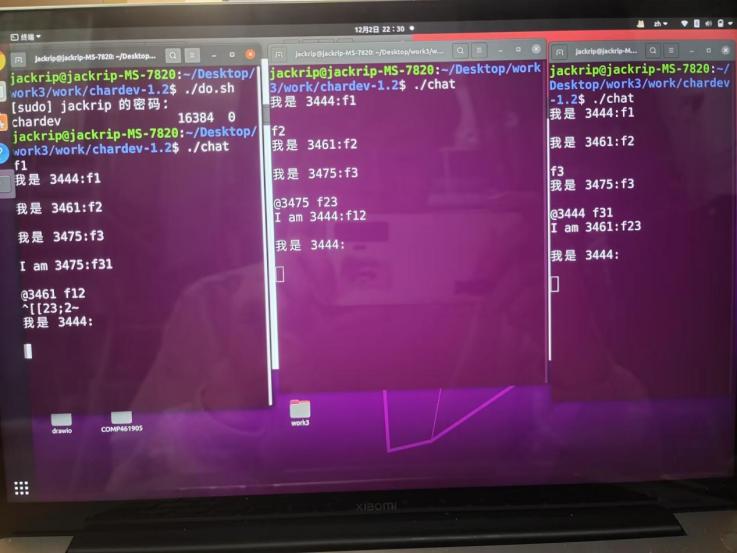


#### 聊天功能展示

（私聊功能在后面展示）创建三个user广播通信如下：



进一步实现私聊功能：实现思路是在判断键盘输入函数将键盘输入读取后，判断第一个字符是否是@，如果是就将后面直到’ ’前的数字用atoi()函数转变为整形，再与要读取的进程号进行对比，如果不是，就不输出在屏幕上，如果是就输出在屏幕上，从而实现私聊。



## 5.4 实验总结

### 5.4.1 实验中的问题与解决过程

问题：虚拟机遇到了网络问题，无法进行实验

解决：重新配制虚拟机环境

## 5.4.2 实验收获

通过自己写一个字符驱动设备，更加深入地了解了动态模块的运用字符驱动设备的实现，以及各种接口的编程实现。

## 5.5 源码

Chardev.c:

//chardev.c 字符驱动程序

#include <linux/init.h>

#include <linux/module.h>

#include <linux/cdev.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/vmalloc.h>

#include <linux/slab.h>

#define MAX\_DEVICES 10

#define MAJOR\_NUM 101

#define MINOR\_NUM 0

#define BUFFER\_SIZE 1048576

struct \_device\_data{

struct cdev chardev;

unsigned char \*buffer;

int npos;

}\*mydata[MAX\_DEVICES];

static ssize\_t device\_read(struct file \*filp, char \* buff, size\_t len, loff\_t\* offset)

{

int nlen=len;

struct \_device\_data\* pdb=filp->private\_data;

if(nlen>pdb->npos-\*offset)

nlen=pdb->npos-\*offset;

if(copy\_to\_user(buff, (pdb->buffer)+\*offset, nlen))

return -EFAULT;

\*offset += nlen;

return nlen;

}

static ssize\_t device\_write(struct file \*filp, const char \* buff, size\_t len, loff\_t\* offset)

{

int nlen=len;

struct \_device\_data\* pdb=filp->private\_data;

if(nlen>BUFFER\_SIZE-pdb->npos)

nlen=BUFFER\_SIZE-pdb->npos;

if(nlen==0)

return -ENOMEM;

if(copy\_from\_user(&pdb->buffer[pdb->npos], buff, nlen))

return -EFAULT;

pdb->npos += nlen;

return nlen;

}

static int device\_open(struct inode \*inode, struct file \* filp)

{

int nminor=iminor(inode);

if(!mydata[nminor]->buffer)

mydata[nminor]->buffer=(unsigned char \*)vmalloc(BUFFER\_SIZE);

if(!mydata[nminor]->buffer)

return -ENOMEM;

filp->private\_data=mydata[nminor];

if((filp->f\_flags&O\_ACCMODE)==O\_WRONLY)

mydata[nminor]->npos=0;

return 0;

}

static int device\_release(struct inode\* inode, struct file\* filp)

{

/\*

int i;

for(i=0; i<MAX\_DEVICES; ++i){

cdev\_del(&mydata[i]->chardev);

if(mydata[i]->buffer)

vfree(mydata[i]->buffer);

kfree(mydata[i]);

}

\*/

return 0;

}

struct file\_operations fops = {

.owner=THIS\_MODULE,

.read=device\_read,

.write=device\_write,

.open=device\_open,

.release=device\_release,

};

static int device\_init(void)

{

int i, ndev, ret;

printk(KERN\_INFO "Loading " KBUILD\_MODNAME "...\n");

for(i=0; i<MAX\_DEVICES; ++i) {

mydata[i] =(struct \_device\_data\*)kmalloc(sizeof(\*mydata[0]), GFP\_KERNEL);

if(!mydata[i]){

printk(KERN\_EMERG "Can't allocate memory to mydata\n");

return -ENOMEM;

}

mydata[i]->buffer=NULL;

mydata[i]->npos=0;

cdev\_init(&mydata[i]->chardev, &fops);

mydata[i]->chardev.owner=THIS\_MODULE;

ndev=MKDEV(MAJOR\_NUM, MINOR\_NUM+i);

ret=cdev\_add(&mydata[i]->chardev, ndev,1);

if(ret){

printk(KERN\_EMERG "Can't register device[%d]!\n", i);

return -1;

}

}

return 0;

}

static void device\_exit(void)

{

int i;

printk(KERN\_INFO "Unloading " KBUILD\_MODNAME "...\n");

for(i=0; i<MAX\_DEVICES; ++i){

cdev\_del(&mydata[i]->chardev);

if(mydata[i]->buffer)

vfree(mydata[i]->buffer);

kfree(mydata[i]);

}

}

module\_exit(device\_exit);

module\_init(device\_init);

MODULE\_LICENSE("GPL");

Chat.c:

//chat.c 聊天程序

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <memory.h>

#include <unistd.h>

#include <termios.h>

int kbhit(void);

int main()

{

int fd,i;

char msg[101];

fd=open("/dev/chardev0", O\_RDWR|O\_APPEND);

if(fd==-1){

fprintf(stderr, "can't openfile chardev0\n");

exit(0);

}

while(1){

if( kbhit() )

{

char k[100]; //用来存储键盘输入的数据

char head[150]; //存放消息头信息和数据,表明是哪个进程写入的消息

int pid = getpid();

fgets(k,100,stdin);

//printf("有按键按下,开始读取键盘输入信息,并写入字符设备\n");

//printf("我是进程%d,我写入的数据为:%s\n",pid,k);

int delta=0;

if(k[0]=='@'){

delta++;

sprintf(head,"%c",k[delta]);

while(k[delta+1]!=' '){

sprintf(head,"%c",k[delta]);

delta++;

}

sprintf(head," "); delta+=2;

}

sprintf(head,"我是进程%d:%s",pid,k+delta);

write(fd,head,strlen(head));//写入字符设备

}

for(i=0;i<101;i++)

msg[i]='\0';

read(fd,msg,100);

/\*

if(msg[0]=='@'){

int namel=0;

int pid[10];

for(int j=0;j++;j<10)

{

pid[j]=0;

}

while(msg[namel]!=':'){

namel++;

pid[namel]=msg[namel];

}

namel--;

}\*/

int delta=0;

if(strlen(msg) != 0 ) { //要有新数据才会显示在屏幕上

//printf("读出的数据为:\n%s\n",msg);

if(msg[0]=='@'){

while(msg[delta]!=' ') delta++;

msg[delta]='\0';

int tmp=atoi(msg+1);

if(getpid()!=tmp) goto end;

delta++;

}

printf("%s\n",msg+delta); end:;

}

sleep(2);

}

close(fd);

return 0;

}

//非阻塞检测按键函数

int kbhit(void)

{

struct termios oldt, newt;

int ch,oldf;

tcgetattr(STDIN\_FILENO, &oldt);

newt=oldt;

newt.c\_lflag &= ~(ICANON | ECHO);

tcsetattr(STDIN\_FILENO, TCSANOW, &newt);

oldf=fcntl(STDIN\_FILENO, F\_GETFL, 0);

fcntl(STDIN\_FILENO, F\_SETFL, oldf | O\_NONBLOCK);

ch=getchar();

tcsetattr(STDIN\_FILENO, TCSANOW, &oldt);

fcntl(STDIN\_FILENO, F\_SETFL, oldf);

if(ch != EOF) {

ungetc(ch, stdin);

return 1;

}

return 0;

}

Makefile:

obj-m := chardev.o

PWD := $(shell pwd)

KVER ?= $(shell uname -r)

KDIR := /lib/modules/$(KVER)/build

all:

$(MAKE) -C $(KDIR) M=$(PWD)

clean:

rm -rf .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.mod \*.symvers \*.order