西安交通大学

操作系统专题实验报告

班级： 计算机

学号：

姓名：

2022 年 12 月 15 日

实验一 华为云上openEuler操作系统环境

1. 实验目的

加强对操作系统原理的进一步理解，引导学生与操作系统内核交互以达到理解其原理、掌握操作系统的设计技术和设计技巧，提高其动手能力，进一步激发同学们的想象力和创造力。

本专题实验的整体架构基于Linux之上。针对所提出的问题，使用Linux环境下的C语言进行设计实现。通过本专题实验的完成，来学习理解操作系统的基本原理、内部机制和体系结构。

1. 实验内容

1.进程相关编程实验

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

（1）熟悉操作命令、编辑、编译、运行程序。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。

（2）扩展图3-32的程序：a) 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；b) 在return前增加对全局变量的操作并输出结果，观察并解释；c) 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；

2.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。

https://www.cnblogs.com/lakeone/p/3789117.html 获取线程 tid

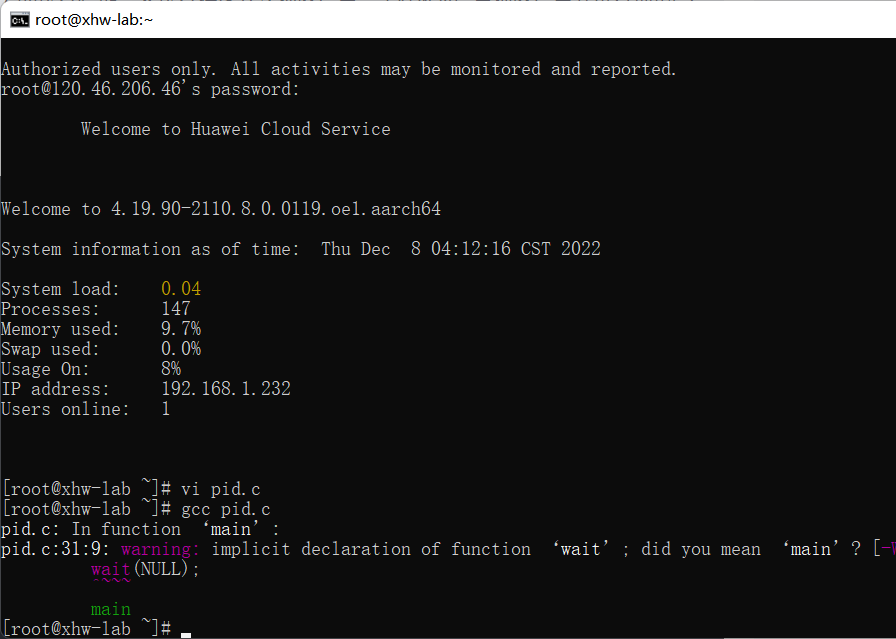
1、在进程中给一变量赋初值并创建两个线程；

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。

1. 实验过程



实验代码：

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

#include<sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

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

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

}

else

{

//parent process

pid1 = getpid();

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

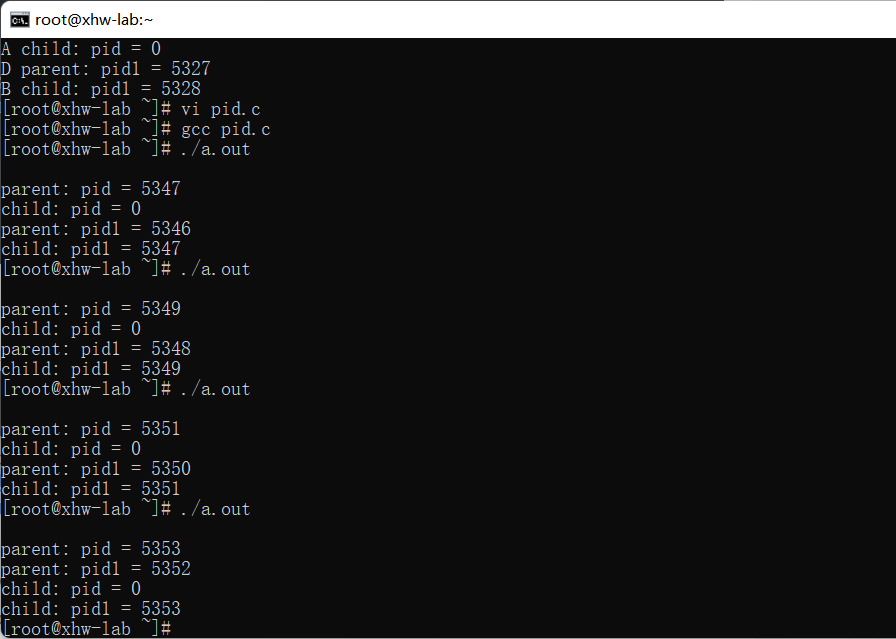
printf("parent: pid1 = %d\n",pid1);//D

wait(NULL);

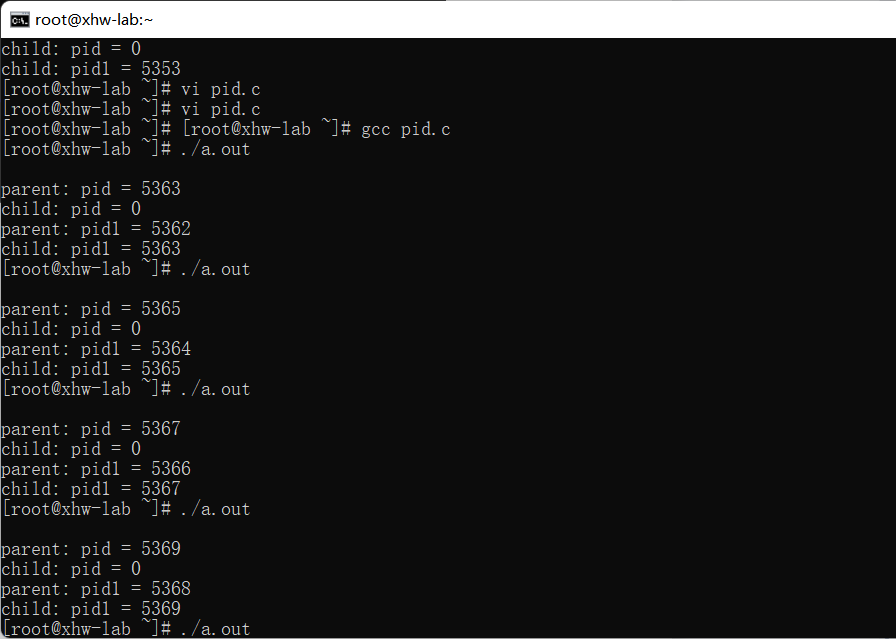
}

return 0;

}



去除掉wait后



同样也是既有可能child先执行，又有可能parent先执行

在去掉wait()前后，都是既有可能parent先输出，又有可能child先输出，这是因为父子进程的执行顺序是不确定的，既有可能父进程先执行，也有可能子进程先执行。wait()的作用是让父进程挂起等待子进程结束。所以只有将wait()放在父进程打印语句之前才能确保子进程先输出。

1. 定义全局变量并观察结果

定义一个a，初始为0，子进程自增1 ，父进程自减1

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int a = 0;

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

a++;

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

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

printf("a: %d\n",a);

}

else

{

//parent process

pid1 = getpid();

a--;

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

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

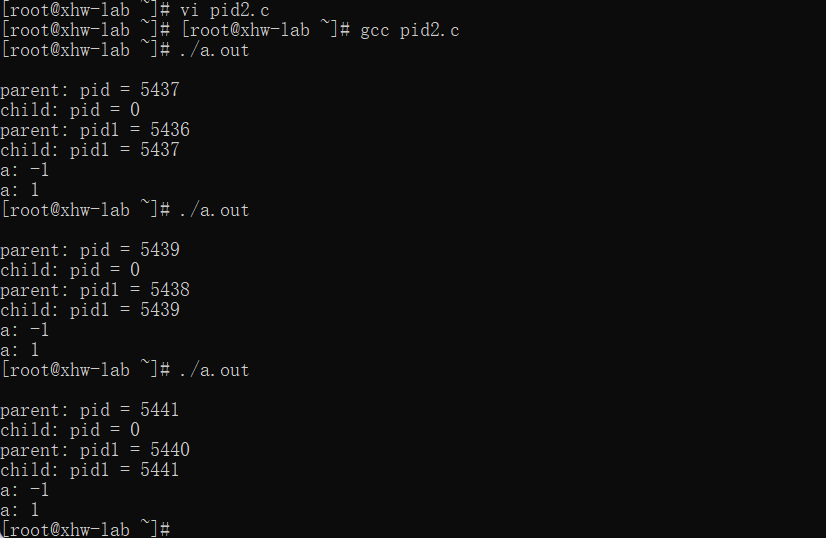
printf("a: %d\n",a);

wait(NULL);

}

return 0;

}



子进程在创建时，继承了全局变量，虽然这两个变量都是a，但是它们其实已经变成了两个，执行时在各自的物理空间内执行。子进程对该全局变量的操作是相互独立的，因此会分别对a进行操作，打印出的a值不同

1. 在return前加一个操作使a自增2

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int a = 0;

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

a++;

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

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

printf("a: %d\n",a);

}

else

{

//parent process

pid1 = getpid();

a--;

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

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

printf("a: %d\n",a);

wait(NULL);

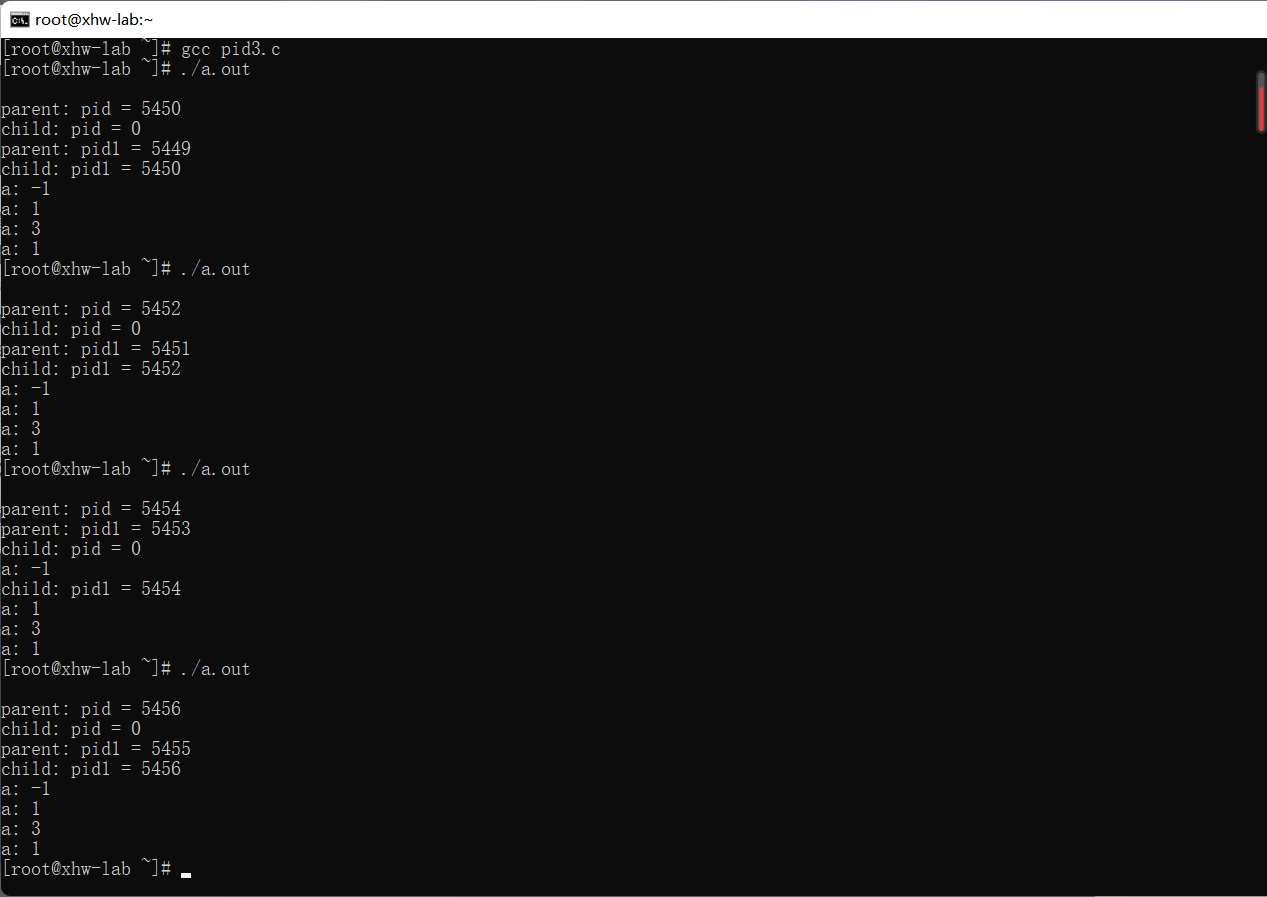
}

a+=2;

printf("a: %d\n",a);

return 0;

}



3.在子进程中调用system函数

在子进程中通过system()函数调用事先编译好的文件

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

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

system("./call");

printf("子进程不输出\n");

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

}

else

{

//parent process

pid1 = getpid();

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

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

wait(NULL);

}

return 0;

}

Call.c:

#include<unistd.h>

# include<stdio.h>

# include <sys/types.h>

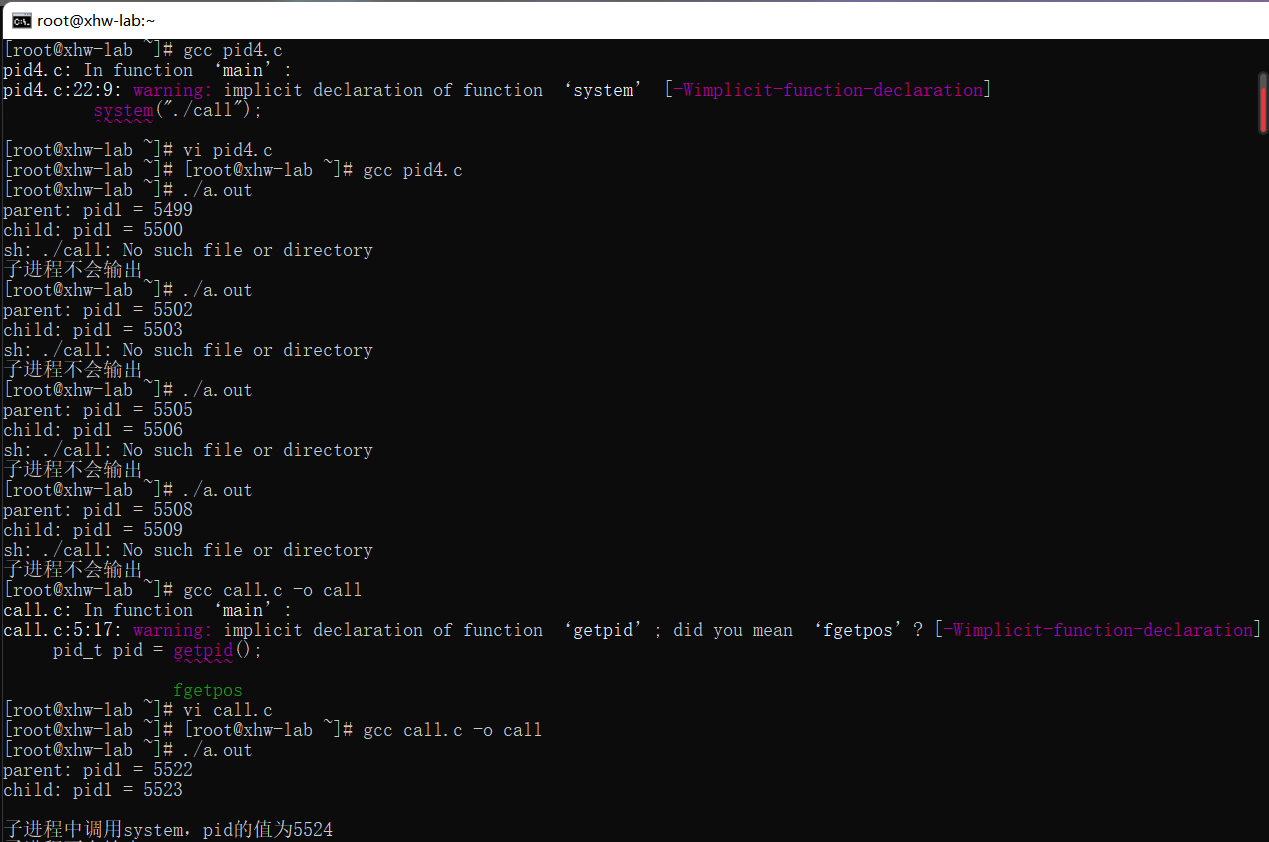
int main()

{

pid\_t pid = getpid();

printf("\n子进程中调用system，pid的值为%d\n",pid);

}



调用system()之后pid和之前有所不同，因为system()函数先执行了fork()函数，新产生的子进程立刻执行了exec()函数，产生了一个新进程，新进程的pid等与原进程不同。

4.在子进程中调用exec函数

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

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

pid1 = getpid();

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

execl("./call","");

printf("子进程不输出\n");

}

else

{

//parent process

pid1 = getpid();

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

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

wait(NULL);

}

return 0;

}

1. 使用pthread实现多线程

使用pthread\_t创建线程号，pthread\_create()创建线程，pthread\_join()来等待线程执行完成

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void\* threadfunc(void\* tid);

int main(void){

pthread\_t threads[2];

int status,i;

for(i=0;i<2;i++){//循环创建10个现场

printf("主函数中创建第%d个线程\n",i+1);

status=pthread\_create(&threads[i],NULL,threadfunc,(void\*)i);

if(status!=0){

printf("线程创建失败 %d\n",status);

exit(-1);

}

}

for(i=0;i<2;i++){

pthread\_join(threads[i],NULL);

}

exit(0);

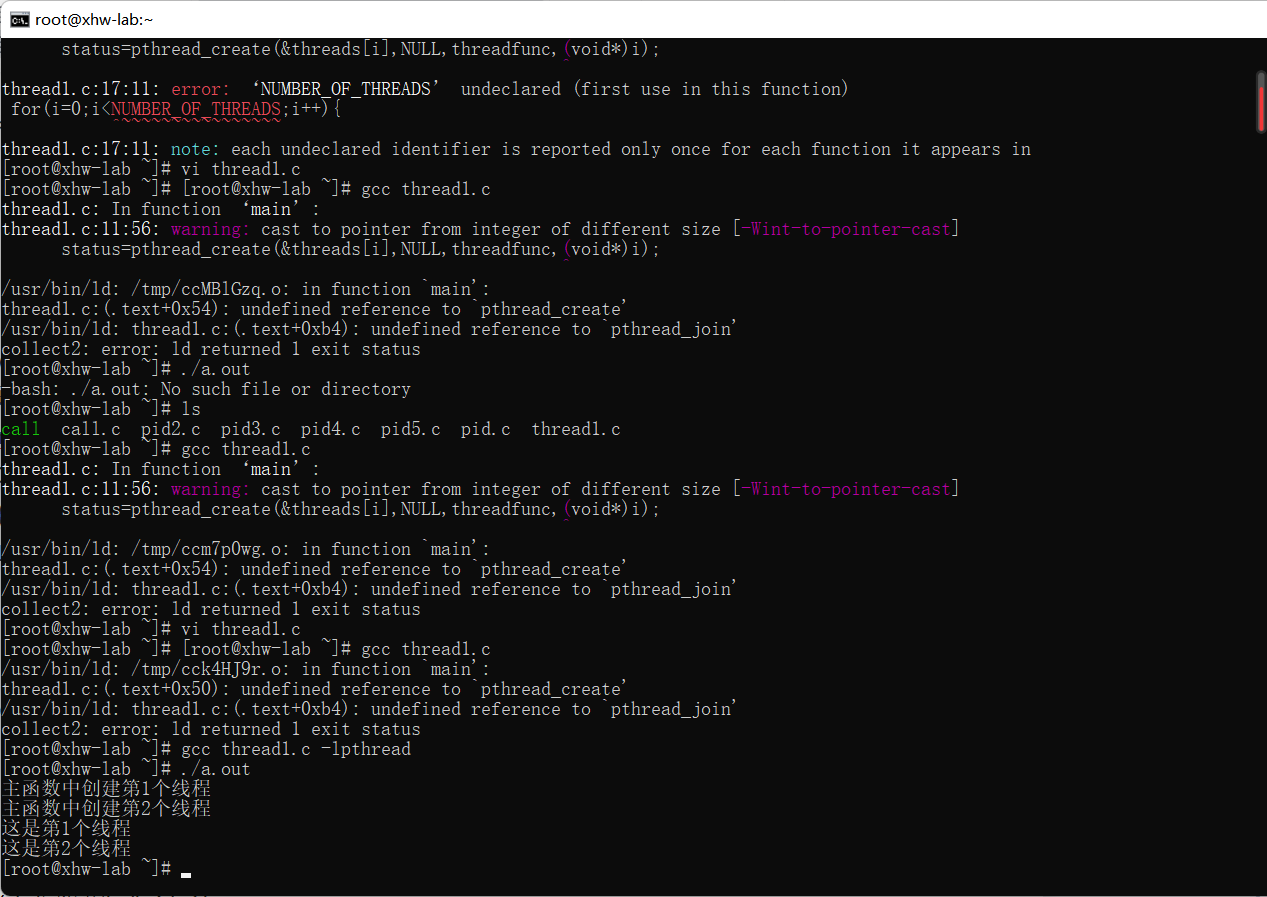
}

void\* threadfunc(void\* tid){

printf("这是第%d个线程\n",tid+1);

pthread\_exit(0);

}



#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int a = 1; // 全局变量

void run() { int i=0; while(i<5002){ a=a+1;i++;printf("%d\n",a);} }

void run2() { int i=0; while(i<5002){ a=a-1;i++;printf("%d\n",a);} }

int main() {

int tmp1, tmp2;

pthread\_t thread1, thread2;

int ret\_thrd1, ret\_thrd2;

ret\_thrd1 = pthread\_create(&thread1, NULL, (void \*)&run, NULL);

if (ret\_thrd1 != 0) {

printf("thread1 create error\n");

}

else {

printf("thread1 create success\n");

}

pthread\_join(thread1,NULL);

ret\_thrd2 = pthread\_create(&thread2, NULL, (void \*)&run2, NULL);

if (ret\_thrd2 != 0) {

printf("thread2 create error\n");

} else {

printf("thread2 create success\n");

}

pthread\_join(thread2,NULL);

}

去除pthread\_join()函数,线程之间的同步关系被打破了，线程thread1还未执行完，就被thread2强占了资源

代码中如果没有pthread\_join主线程会很快结束从而使整个进程结束，从而使创建的线程没有机会开始执行就结束了

加入pthread\_join后，主线程会一直等待直到等待的线程结束自己才结束，而在今天的例子中，主线程会等待线程的结束才继续执行下面的语句（创建thread2），使创建的线程有机会执行。

实验二、进程通信与内存管理

1. 实验目的

本实验在用户态下，根据教材所学习的操作系统原理，完成Linux下进程通信与内存管理算法的实现，通过实验，进一步理解所学理论知识。

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。

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。

模拟实现FIFO算法，LRU算法

1. 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。

1. 实验过程

1、根据流程图编写程序，猜想一下这个程序的运行结果，然后多次运行，观察Delete/quit键前后，会出现什么结果？分析原因。

2、如果程序运行界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”，怎样修改程序使得只有接收到相应的中断信号后再发生跳转，执行输出？

3、将本实验中通信产生的中断通过14 号信号值进行闹钟中断，将signal(3,stop)当中数字信号变为2，体会不同中断的执行样式，从而对软中断机制有一个更好的理解。

4.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；

5.然后按照注释里的要求把代码补充完整，运行程序；

6.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。

1. 实验记录及分析

**#include <stdio.h>**

**#include <signal.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**int wait\_flag;**

**void stop( );**

**main( ) {**

**int pid1, pid2; // 定义两个进程号变量**

**signal(3,stop); // 或者 signal(14,stop);**

**while((pid1 = fork( )) == -1); // 若创建子进程1不成功,则空循环**

**if(pid1 > 0) { // 子进程创建成功,pid1为进程号**

**while((pid2 = fork( )) == -1); // 创建子进程2**

**if(pid2 > 0) {**

**wait\_flag = 1;**

**sleep(5); // 父进程等待5秒**

**kill(pid1,16); // 杀死进程1**

**kill(pid2,17); // 杀死进程2**

**wait(0); // 等待第1个子进程1结束的信号**

**wait(0); // 等待第2个子进程2结束的信号**

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

**exit(0); // 父进程结束**

**}**

**else {**

**wait\_flag = 1;**

**signal(17,stop); // 等待进程2被杀死的中断号17**

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

**exit(0);**

**}**

**}**

**else {**

**wait\_flag = 1;**

**signal(16,stop); // 等待进程1被杀死的中断号16**

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

**exit(0);**

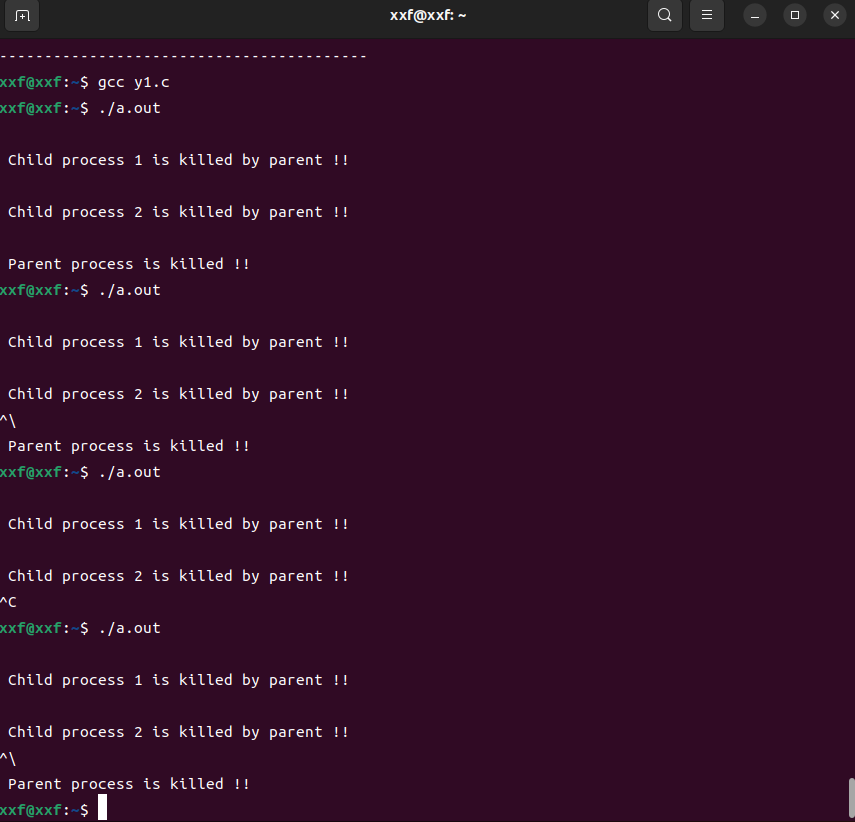
**}**

**}**

**void stop( ) {**

**wait\_flag = 0;**

**}**



猜想：本程序应该开始不显示任何文字，等待键盘按下Delete/Quit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 !!”。

按下Delete/Quit键后程序直接输出“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”。键盘的输入不起作用，这是因为代码里signal函数并没有对输入的中断进行判断。

多次运行程序，发现子进程被杀死的顺序不总是一样的。这是因为两个子进程同时运行，并没有先后的区别。

在stop函数中将wait\_flag置为0，在kill函数前面加上while(wait\_flag==1)，这样只有在接收到中断信号后才会跳转。

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

int wait\_flag;

void stop(int signum);

int main(){

//定义两个进程号变量

int pid1, pid2;

alarm(1);

//signal(SIGINT,stop);

//signal(SIGQUIT,stop);

signal(SIGALRM,stop);

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

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

//父进程

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

wait(0);

// 等待第2个子进程2结束的信号

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;

// 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);

}

}

//pid1

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);

}

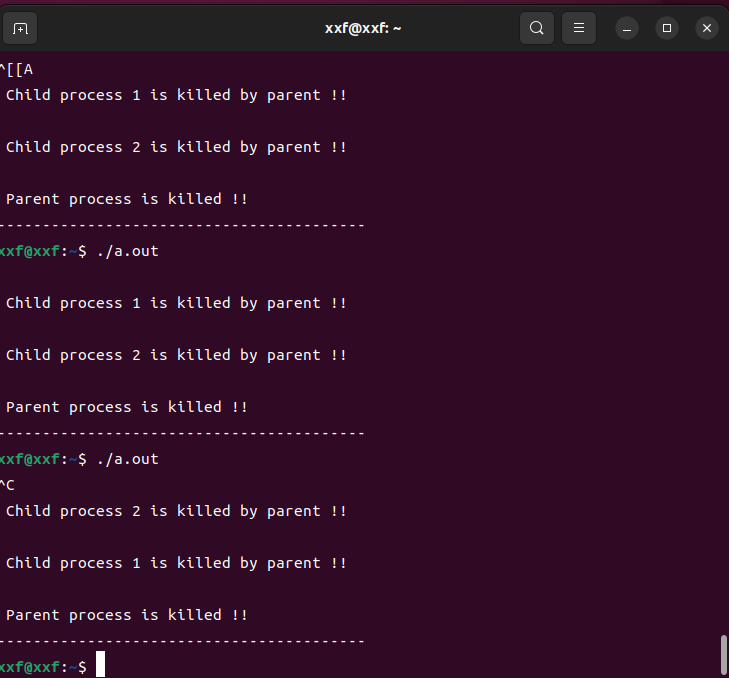
}

void stop(int signum) {

wait\_flag = 0;

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

}



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，分别给子进程1和子进程2发送16，17信号，顺序不一定相同，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。

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。

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函数都可以主动退出，而kill是强制退出。主动退出比较好，如果在某个子进程退出前父进程被kill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。

1. 管道通信：

**#include <unistd.h>**

**#include <signal.h>**

**#include <sys/wait.h>**

**#include <stdlib.h>**

**#include <stdio.h>**

**int pid1,pid2; // 定义两个进程变量**

**Int main( ) {**

**int fd[2];**

**char OutPipe[100],InPipe[100]; // 定义两个字符数组**

**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**

**lockf(fd[1],1,0); // 锁定管道**

**sprintf(OutPipe,"\n Child process 1 is sending message!\n"); // 给Outpipe赋值**

**write(fd[1],OutPipe,50); // 向管道写入数据**

**sleep(5); // 等待读进程读出数据**

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

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

**sprintf(OutPipe,"\n Child process 2 is sending message!\n");**

**write(fd[1],OutPipe,50);**

**sleep(5);**

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

**exit(0);**

**}**

**else {**

**wait(0); // 等待子进程1 结束**

**read(fd[0],InPipe,50); // 从管道中读出数据**

**printf("%s\n",InPipe); // 显示读出的数据**

**wait(0); // 等待子进程2 结束**

**read(fd[0],InPipe,50);**

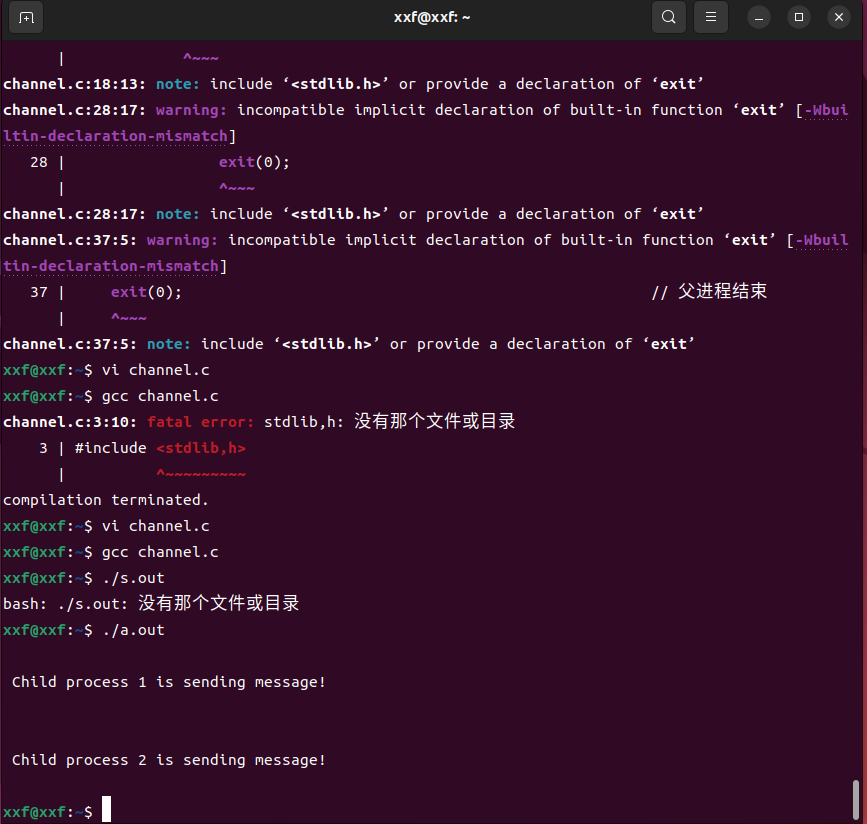
**printf("%s\n",InPipe);**

**exit(0); // 父进程结束**

**}**

**}**

**}**



先猜想一下这个程序的运行结果。

屏幕会输出200个1和200个2.

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。

多次实验发现有时先输出200个1，有时输出200个2.这是因为子进程1和子进程2执行的先后顺序是不一定的，如果进程1先执行则先输入1；如果进程2先执行则先输入2。但是并没有交错输出1或2，这是因为子进程在写入前线上了锁，保证只有一个进程在向管道输入数据。

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？

通过使用函数lockf来锁住管道的写端口。如果不加锁会交错输出1和2

#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[1000]; // 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200次每次向管道写入字符’1’

for(int i = 0;i<200;i++)

{

//sprintf(InPipe,1);

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

}

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<200;i++)

{

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

}

//补充; // 分200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400个字符

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

read(fd[0], InPipe, 400);

InPipe[400] = '\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

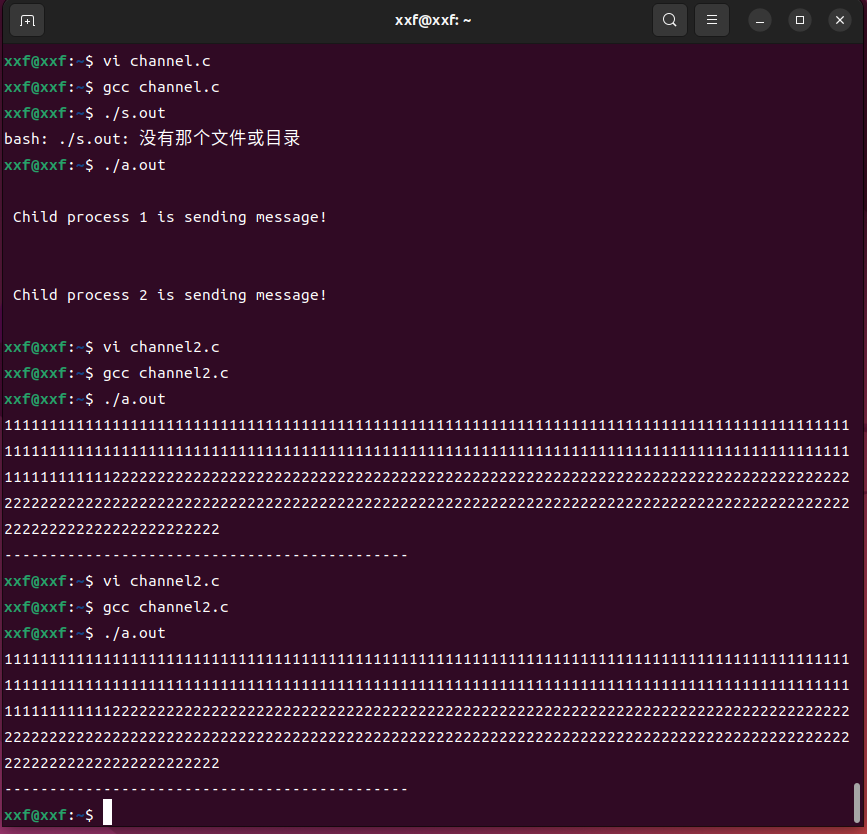
printf("---------------------------------------------\n");

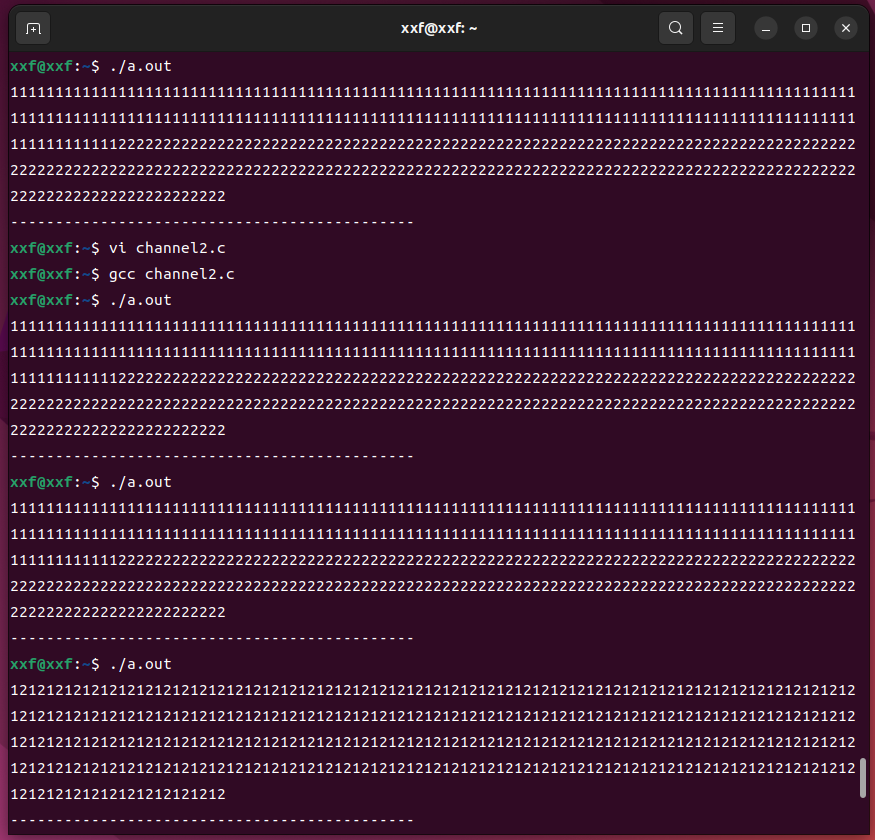
exit(0); // 父进程结束

}

}

}





3.

#include<stdio.h>

#include<stdlib.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;

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("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] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};

//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("Error！\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：\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("end：\n共访问页面次数：%d\n",TOTAL\_INSTRUCTION);

printf("未命中次数：%d\n",diseffect);

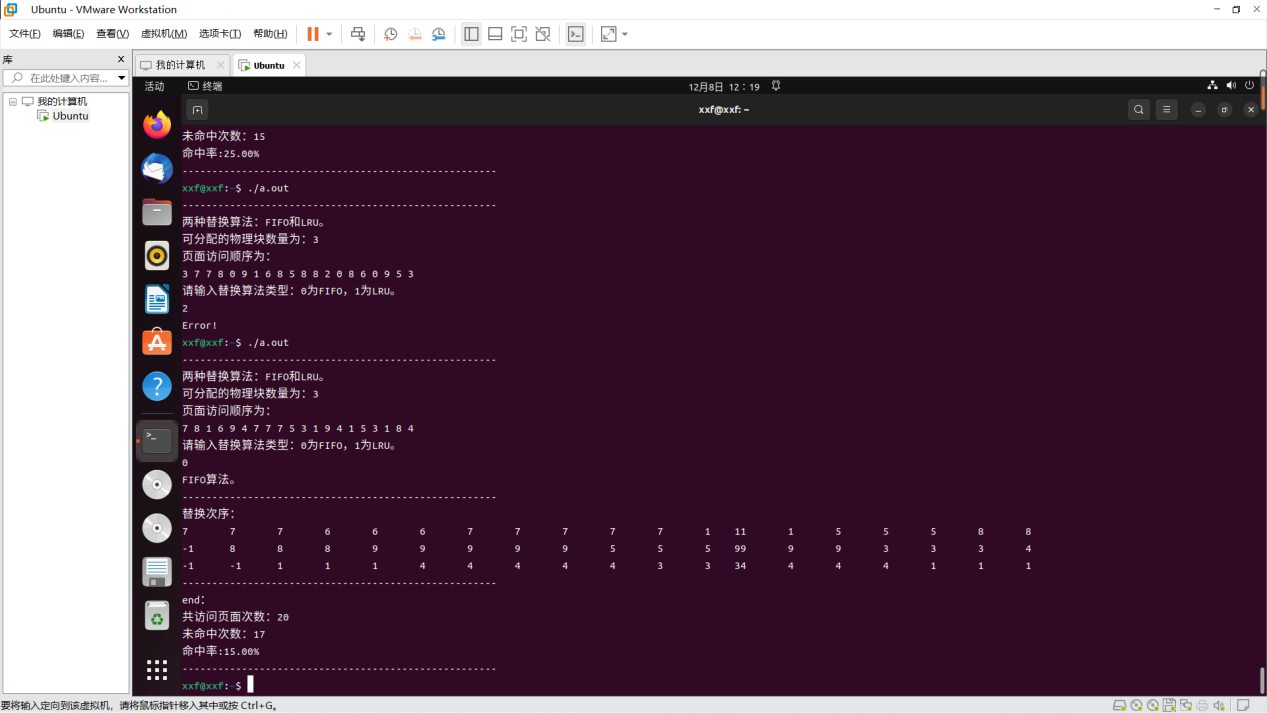
printf("命中率为：%.2f%%\n",hit\_rate);

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

return 0;

}

本程序随机生成页面访问序列，用键盘输入使用的页面置换算法。在主函数中先将各个数据结构和变量初始化，再根据选择的算法类型和当前要使用的页面是否在pagecontrol内使用不同的策略。如果当前需要的页面在pagecontrol里面，则FIFO算法什么也不做，而LRU要更新Stack中的顺序；若当前需要的页面不在pagecontrol里，则需要调入页面。对于FIFO是调入队尾处；对于LRU是调入栈底处，并将栈底值取出置于栈顶，表示最近使用过。将每一次的替换记录在数组temp中，将未命中的次数记录在diseffect中，最后输出temp数组和相关计算数据



实验三、动态模块系统编程与字符设备驱动

1. 实验目的

Linux提供了一种机制，可以随意动态的加载与卸载操作系统部件。Linux动态模块就是这样一种可在系统启动后的任何时候动态连入内核的代码块。当不再需要它时又可以将它从内核中卸载并删除。

理解LINUX字符设备驱动程序的基本原理；

掌握字符设备的驱动运作机制；

学会编写字符设备驱动程序

1. 实验内容

1.创建动态模块源码

2.修改Makefile文件生成编译规则

3.编译创建的模块源码，生成驱动模块

4.安装驱动模块

5.查看是否安装成功

6.使用驱动模块

7.卸载驱动模块

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。

1. 实验过程及记录分析

1.模块的编译

V2.4

#gcc −O2 −g −Wall −DMODULE −D \_\_KERNEL\_\_ −c filename.c

// filename.c为自己编写的模块程序源代码文件

V2.6

2.当前目录建立Makefile文件

执行make即可按照Makefile的规定进行编译，形成.ko模块文件

3.模块的加载

insmod命令

如： insmod filename.ko 或insmod filename.o

4.模块的查看

lsmod

more /proc/modules

dmesg ——查看日志（printk）

5.模块的卸载

rmmod命令

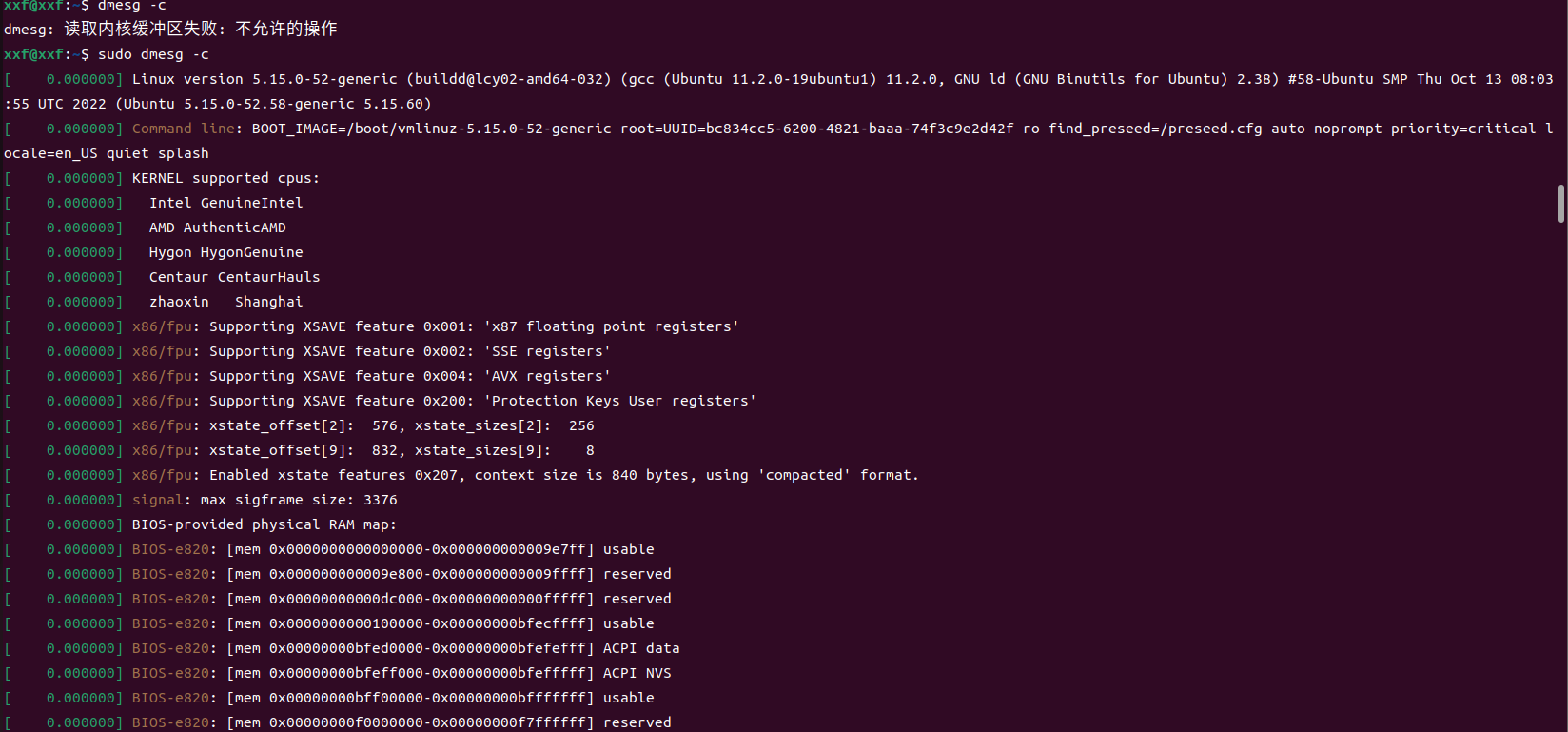
如：rmmod filename

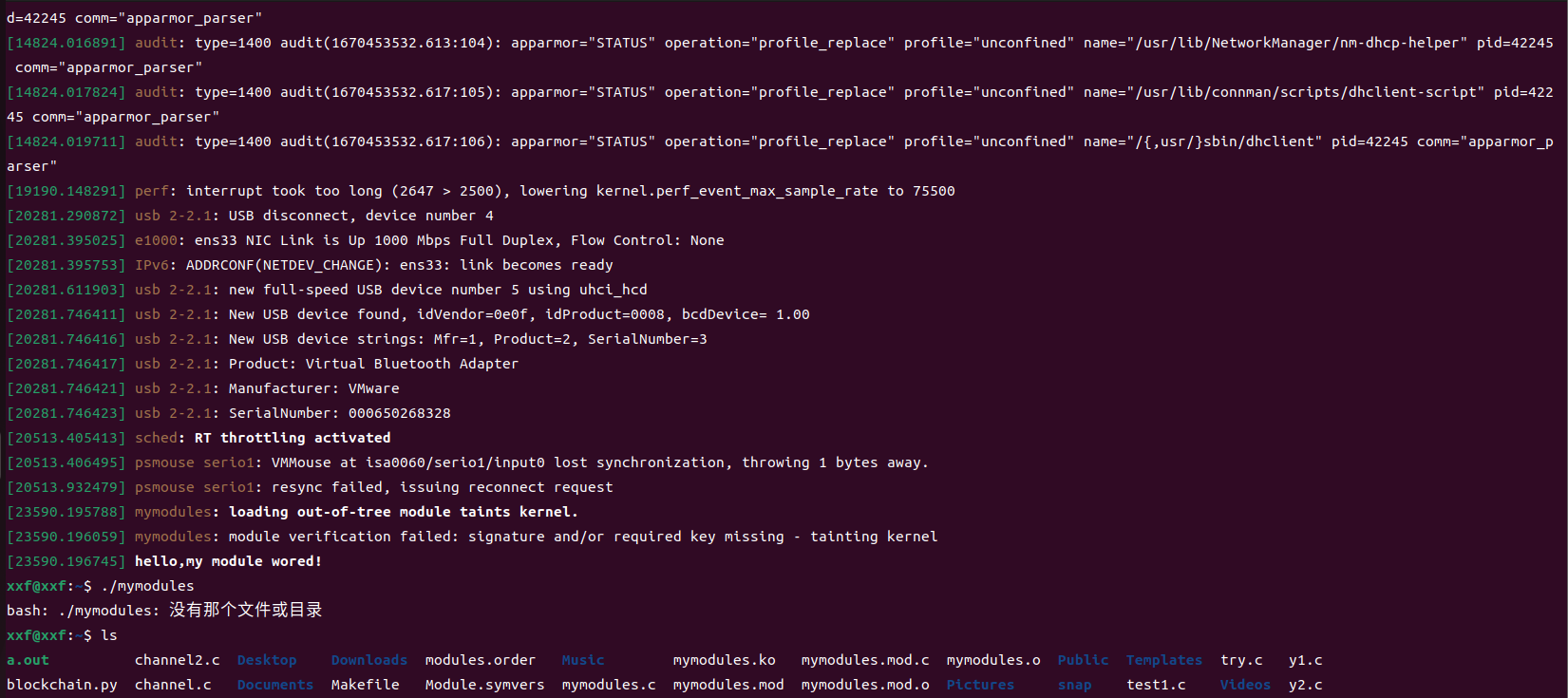


Make时报错

成功make并加载模板insmod

读取内核缓冲区：刚开始没用sudo，导致权限不够

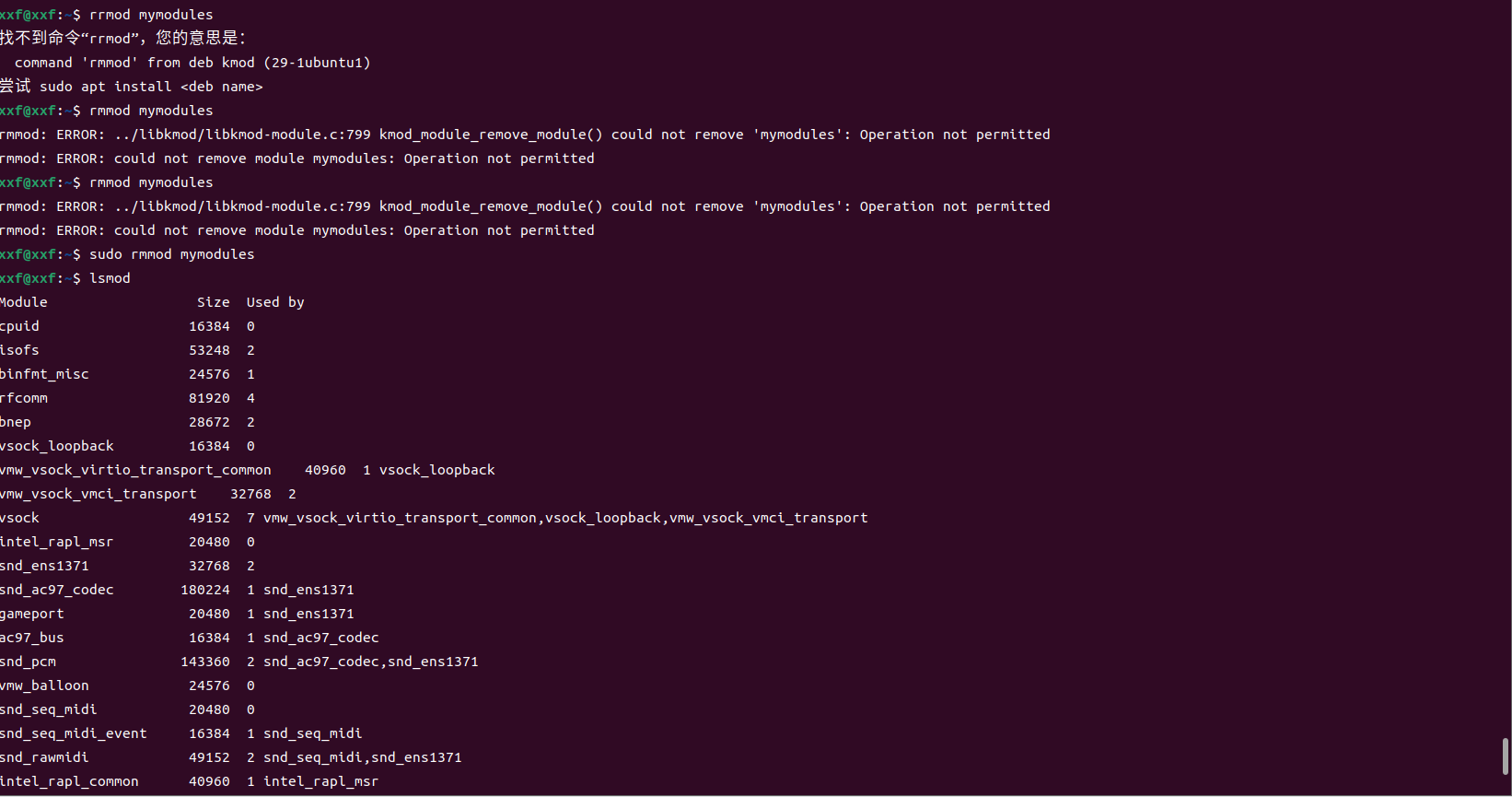


可以看到内核日志信息

Lsmod观察



Rrmod卸载模板并lsmod观察：已经没有了



内核日志信息：已卸载



#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/unistd.h>

#include <linux/time.h>

#include <asm/uaccess.h>

#include <linux/sched.h>

#include <linux/kallsyms.h>

//using syscall 335

#define \_\_NR\_syscall 335

unsigned long \*sys\_call\_table;

unsigned int clear\_cr0(void);

void setback\_cr0(unsigned int val);

static int sys\_mycall(int a, int b, int c);

//to save the original value of the register cr0

unsigned long orig\_cr0;

unsigned long \*sys\_call\_table = 0;

//to save the original syscall func

unsigned long old\_sys\_call\_func;

//set 0 to the 17th bit (WP) in reg cr0

unsigned int clear\_cr0(void)

{

unsigned int cr0 = 0;

unsigned int ret;

//move the value in reg cr0 to reg rax

//movl moves a 32-bits operand

//movq moves a 64-bits operand

//rax is a 64-bits register

//an assembly language code

//asm volatile ("movl %%cr0, %%eax" : "=a"(cr0));//32-bits

asm volatile ("movq %%cr0, %%rax" : "=a"(cr0)); //64-bits

ret = cr0;

//var cr0 is rax

cr0 &= 0xfffeffff; //set 0 to the 17th bit

//asm volatile ("movl %%eax, %%cr0" :: "a"(cr0));//32-bits

//note that cr0 above is a variable while cr0 below is a reg.

asm volatile ("movq %%rax, %%cr0" :: "a"(cr0));

return ret;

}

//recover the value of WP

void setback\_cr0(unsigned int val)

{

//asm volatile ("movl %%eax, %%cr0" :: "a"(val));//32-bits

asm volatile ("movq %%rax, %%cr0" :: "a"(val));//64-bits

}

//my syscall function

static int sys\_mycall(int a,int b,int c)

{

printk("Change syscall successfully!\nReturn a+b+c\nBy xhw-class 92-No 2160506148\n");

return a+b+c;

}

static int \_\_init init\_addsyscall(void)

{

printk("Begin changing syscall...\n");

//Automatically get sys\_call\_table address

sys\_call\_table = (unsigned long \*)kallsyms\_lookup\_name("sys\_call\_table");

//print sys\_call\_table address

printk("sys\_call\_table: 0x%p\n", sys\_call\_table);

//save original syscall func

old\_sys\_call\_func = (int(\*)(void))(sys\_call\_table[\_\_NR\_syscall]);

//modify the value of WP in CR0

orig\_cr0 = clear\_cr0();

//change the syscall address

sys\_call\_table[\_\_NR\_syscall] = (unsigned long)&sys\_mycall;

//setback the value of WP in CR0

//to read only

setback\_cr0(orig\_cr0);

return 0;

}

static void \_\_exit exit\_addsyscall(void)

{

//modify the value of WP in CR0

orig\_cr0 = clear\_cr0();

//change the syscall address

sys\_call\_table[\_\_NR\_syscall] =old\_sys\_call\_func;

//setback the value of WP in CR0

//to read only

setback\_cr0(orig\_cr0);

printk("Recovering syscall...\n");

}

module\_init(init\_addsyscall);

module\_exit(exit\_addsyscall);

MODULE\_LICENSE("GPL");

建立makefile：

ifneq ($(KERNELRELEASE),)

obj-m := modify\_syscall.o

else

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

PWD := $(shell pwd)

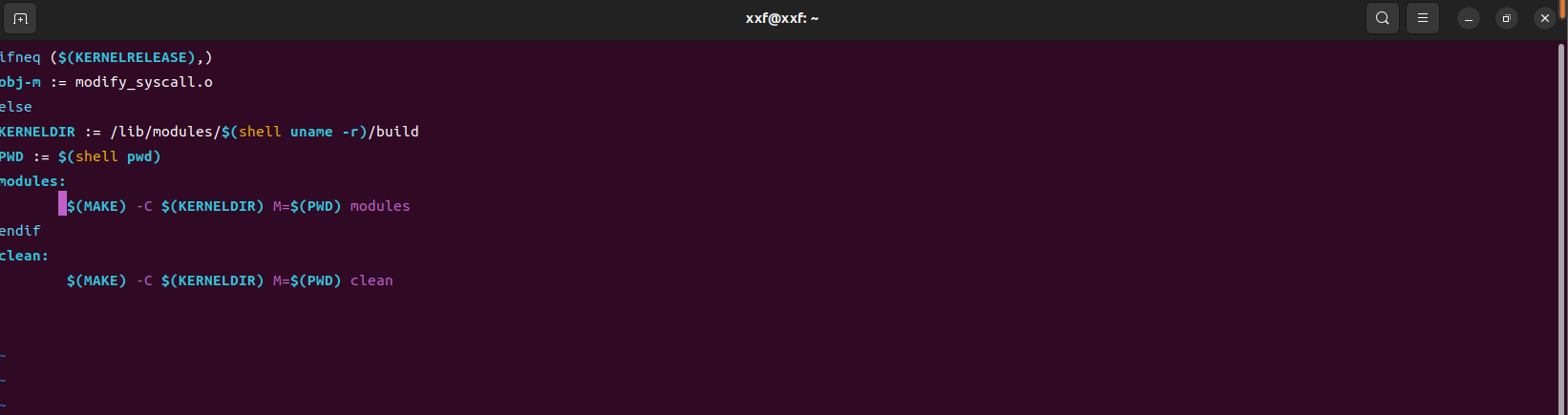
modules:

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

endif

clean:

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



编写test.c测试：

#include<stdio.h>

#include<unistd.h>

#include<sys/syscall.h>

#include<time.h>

int main()

{

printf("Set parameters 78,10,20\nReturn:");

int ret=syscall(321,78,10,20);

if(ret == -1)

printf("syscall failed!\n");

else

printf("%d\n",ret);

return 0;

}

