# 操作系统实验报告

姓名:陈扬

学号:17150011001

[操作系统实验报告](#header-n0)  
 [实验一 vi编辑器的使用](#header-n5)  
 [1. 实验目的](#header-n6)  
 [实验2 Linux进程控制](#header-n13)  
 [一、**实验目的**](#header-n14)  
 [**二、预备知识**](#header-n20)  
 [**三、实验内容**](#header-n25)  
 [**实验3 Linux进程间的通信**](#header-n65)  
 [**一．实验目的**](#header-n66)  
 [**二．实验内容**](#header-n68)

### 实验一 vi编辑器的使用

#### 1. 实验目的

（1） 理解vi的三种运行模式及其切换方法。

（2） 学会使用vi的各种操作命令进行文本文件的编辑。

（3） 用vi编写Linux下C程序，会用gcc编译。

给学生一个使用vi创建与编辑文件的机会。vi是UNIX中最常用的编辑器。对vi编辑器的全面而清晰的理解，对于后续内容的学习至关重要。

实验总结:本来就会用

### 实验2 Linux进程控制

#### 一、**实验目的**

1.了解进程与程序的区别，加深对进程概念的理解；

2.进一步认识进程并发执行的原理，理解进程并发执行的特点，区别进程并发执行与顺序执行；

3.分析进程争用临界资源的现象，学习解决进程互斥的方法。

4.了解fork( )系统调用的返回值，掌握用fork()创建进程的方法；

5.熟悉wait、exit等系统调用。

#### **二、预备知识**

（1）fork系统调用：创建一个新进程。即完成创建子进程，也返回一个值。

**<0,创建失败；=0，子进程执行中；〉0，主进程进行中。**

（2）getid：获得一个进程的id

（3）lockf：在进程同步控制中为进程加锁

#### **三、实验内容**

**实验内容一**

1．编写一C语言程序（程序名为fork.c），使用系统调用fork( )创建两个子进程。当程序运行时，系统中有一个父进程和两个子进程在并发执行。父亲进程执行时屏幕显示“I am father”，儿子进程执行时屏幕显示“I am son”，女儿进程执行时屏幕显示“I am daughter”。

2．多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

# include<stdio.h>  
# include<sys/types.h>  
# include<unistd.h>  
  
int main()  
{  
 int pid1,pid2;  
 printf("I am father!\n");  
   
 if ((pid1 = fork())<0)  
 {  
 printf("Child1 fail create!\n");  
 return 1;  
 }  
 else if (pid1 == 0)  
 {  
 for (int i=1;i<=11000;i++)printf("");  
 printf("I am son!\n");  
 return 1;  
 }  
 if ((pid2 = fork())<0)  
 {  
 printf("Child2 fail create!\n");  
 return 1;  
 }  
 else if (pid2 == 0)  
 {   
 for (int i=1;i<=10000;i++)printf("");  
 printf("I am daughter!\n");  
 return 1;  
 }  
 for (int i=1;i<=100000;i++)printf("");  
 return 0;  
}

➜ ex02 git:(master) ✗ gcc fock.c -o fock  
➜ ex02 git:(master) ✗ ./fock  
I am father!  
I am son!  
I am daughter!  
➜ ex02 git:(master) ✗ ./fock  
I am father!  
I am daughter!  
I am son!

实验总结:因为一个时间片太快了,一个子进程直接就执行完了,所以我在每个子进程中加了空的print

**实验内容二**

1．编写一C语言程序（程序名为fork.c），使用系统调用fork( )创建一个子进程，然后在子进程中再创建子子进程。当程序运行时，系统中有一个父进程、一个子进程和一个孙子进程在并发执行。父亲进程执行时屏幕显示“I am grandfather”，儿子进程执行时屏幕显示“I am father”，**孙子进程执行时**屏幕显示“grandson”。

2．多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

# include<stdio.h>  
# include<sys/types.h>  
# include<unistd.h>  
  
int main()  
{  
 int pid1,pid2;  
 printf("I am grandfather!PID1=%d,PID2=%d\n",pid1,pid2);  
 if ((pid1 = fork())<0)  
 {   
 printf("father fail create!PID1=%d,PID2=%d\n",pid1,pid2);  
 return 0;  
 }  
 else if (pid1 == 0)  
 {   
   
 if ((pid2 = fork())<0)  
 {  
 printf("son fail create!PID1=%d,PID2=%d\n",pid1,pid2);  
 return 0;  
 }  
 else if (pid2 == 0)  
 {  
 printf("I am grandson!PID1=%d,PID2=%d\n",pid1,pid2);  
   
 return 0;  
 }  
 else  
 {  
 printf("I am father!PID1=%d,PID2=%d\n",pid1,pid2);  
 return 0;  
 }  
 }  
 for (int i=1;i<=10000;i++)printf("");  
// printf("nothing to do:%d\n",i);  
  
 return 0;  
}

➜ ex02 git:(master) ✗ gcc fock\_2.c -o fock\_2  
➜ ex02 git:(master) ✗ ./fock\_2  
I am grandfather!PID1=256892982,PID2=32766  
I am father!PID1=0,PID2=28809  
I am grandson!PID1=0,PID2=0

实验总结:要在mian函数结尾加for (int i=1;i<=10000;i++)printf("");,不然程序执行太快,只能看到I am grandfather!PID1=301092918,PID2=32766

**实验内容三**

修改程序，在父、子进程中分别使用wait、exit、lockf等系统调用“实现”其同步推进，多次反复运行改进后的程序，观察并记录运行结果。

**Linux\***\*进程控制wait()函数解析\*\*

wait（）会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait（）时子进程已经结束，则wait（）会立即返回子进程结束状态值。子进程的结束状态值会由参数status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数status 可以设成NULL。子进程的结束状态值请参考waitpid（）。返回值如果执行成功则返回子进程识别码（PID），如果有错误发生则返回-1。

#include<stdlib.h>  
#include<unistd.h>  
#include<sys/types.h>  
#include<sys/wait.h>  
#include <stdio.h>  
/\*  
WEXITSTATUS(stat\_val) is a macro (so in fact it does not "return" something, but "evaluates" to something).  
  
For how it works you might like to look it up in the headers (which should be #included via <sys/wait.h>) that come with the C-compiler you use.  
  
The implementation of this macro might differ from one C-implementation to the other.  
  
Please note, that this macro only gives a sane value, if the macro WIFEXITED(stat\_val) gave you a value unequal to 0.  
  
Verbatim from waitpid()'s POSIX specification:  
  
 WEXITSTATUS(stat\_val)  
  
 If the value of WIFEXITED(stat\_val) is non-zero, this macro evaluates to the low-order 8 bits of the status argument that the child process passed to \_exit() or exit(), or the value the child process returned from main().  
 The motivation behind adding up the return code(s?) of a particular program is only known to the code's author and the hopefully existing documentation.  
\*/  
int main()  
{  
 pid\_t pid;  
 int status,i;  
 if(fork()==0){  
 printf("This is the child process .pid =%d\n",getpid());  
 exit(5);  
 }  
 else{  
 sleep(1);  
 printf("This is the parent process ,wait for child...\n");  
 pid=wait(&status);  
 i=WEXITSTATUS(status);  
 printf("child’s pid =%d .exit status=%d\n",pid,i);  
 }  
 return 0;   
}

➜ ex02 git:(master) ✗ gcc wait.c -o wait  
➜ ex02 git:(master) ✗ ./wait  
This is the child process .pid =29059  
This is the parent process ,wait for child...  
child’s pid =29059 .exit status=5

子进程的PID是29059,exit status是5

Linux进程控制 exix()

#include<stdlib.h>

void exit(int status);

不像fork那么难理解，从exit的名字就能看出，这个系统调用是用来终止一个进程的。无论在程序中的什么位置，只要执行到exit系统调用，进程就会停止剩下的所有操作，清除包括PCB在内的各种数据结构，并终止本进程的运行。请看下面的程序：

#include<stdlib.h>  
#include<stdio.h>  
int main()  
{  
 printf("this process will exit!\n");  
 exit(0);  
 printf("never be displayed!\n");  
}

➜ ex02 git:(master) ✗ gcc exit.c -o exit  
➜ ex02 git:(master) ✗ ./exit  
this process will exit!

我们可以看到，程序并没有打印后面的"never be displayed!\n"，因为在此之前，在执行到exit(0)时，进程就已经终止了。

exit 系统调用带有一个整数类型的参数status，我们可以利用这个参数传递进程结束时的状态，比如说，该进程是正常结束的，还是出现某种意外而结束的，一般 来说，0表示没有意外的正常结束；其他的数值表示出现了错误，进程非正常结束。

Linux进程控制lockf()

利用系统调用lockf（fd，mode，size），对指定区域（有size指示）进行加锁或解锁，以实现进程的同步或互斥。其中，fd是文件描述字；mode是锁定方式，mode=1表示加锁，mode=0表示解锁；size是指定文件fd的指定区域，用0表示从当前位置到文件结尾（注：有些Linux系统是locking（fd，mode，size））

#include<stdio.h>  
#include<sys/types.h>  
#include<unistd.h>  
int main(void)  
{  
 int pid1,pid2;  
 lockf(1,1,0);  
 printf("Parent process:a\n");  
 if((pid1=fork())<0)  
 {  
 printf("child1 fail create\n");  
 return 0;  
 }  
 else if(pid1==0)  
 {  
 lockf(1,1,0);  
 printf("This is child1(pid=%d) process:b\n",getpid());  
 lockf(1,0,0);  
 return 0;  
 }  
 if((pid2=fork())<0)  
 {  
 printf("child2 fail create\n");  
 return 0;  
 }  
 else if(pid2==0)  
 {  
 lockf(1,1,0);  
 printf("This is child2(pid=%d) process:c\n",getpid());  
 lockf(1,0,0);  
 return 0;  
 }  
 return 0;  
}

➜ ex02 git:(master) ✗ gcc lock.c -o lock  
➜ ex02 git:(master) ✗ ./lock  
Parent process:a  
This is child1(pid=29346) process:b  
This is child2(pid=29347) process:c

lockf()的基本功能是实现资源的互斥访问，即同一时刻同一个资源只允许一个进程访问，本次实验中的资源即系统的标准输出stdout，互斥的结果为同一时刻只可以有一个进程执行输出，并且不可以被打断。b进程先用lockf(1,1,0)锁上stdout，执行输出之后，立即用lockf(1,0,0)释放stdout,而此时进程没有立即进入下一次循环，而是休眠了3秒，也就是说没有立即又给stdout加锁，在休眠的这段时间，c可以获得stdout资源，于是c执行输出；同理，c加锁，输出，释放锁之后，和b一样，也执行了休眠，于是b可以获得stdout资源，于是就出现了两个进程交替执行输出的情况。

实验总结:深入理解了wait,exit和lockf机制

### **实验3 Linux进程间的通信**

#### **一．实验目的**

学习如何利用管道机制、消息缓冲队列、共享存储区机制进行进程间的通信，并加深对上述通信机制的理解。

#### **二．实验内容**

1．了解系统调用pipe()的功能和实现过程。 2．编写一C语言程序，使其用管道来实现父子进程间通信。子进程向父进程发送字符串“is sending a message to parent!”；父进程则从管道中读出子进程发来的消息，并将其显示到屏幕上，然后终止。 3．运行该程序，观察、记录并简单分析其运行结果。

# include <stdio.h>  
# include <unistd.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
int main()  
 {  
 int i,j,fd[2];  
 char S[100];  
 pipe(fd);  
 if ((i=fork())==0)  
 {  
 sprintf(S,"Child Process 1 is sending a message!\n");  
 write(fd[1],S,50);  
 sleep(3);  
 return 0;  
 }  
 if ((j=fork())==0)  
 {  
 sprintf(S,"Child Process 2 is sending a message!\n");  
 write(fd[1],S,50);  
 sleep(2);  
 return 0;  
 }  
 else{  
 wait(0);  
 read(fd[0],S,50);  
 printf("%s",S);  
 read(fd[0],S,50);  
 printf("%s",S);  
 return 0;  
 }  
 return 0;  
}

➜ ex03 git:(master) ✗ gcc pipe.c -o pipe  
➜ ex03 git:(master) ✗ ./pipe  
Child Process 1 is sending a message!  
Child Process 2 is sending a message!

**实验3-2：pipe函数使用**

pipe(建立管道)

1. 表头文件 #include<unistd.h>
2. 定义函数 int pipe(int filedes[2]);
3. 函数说明

pipe()会建立管道,并将文件描述词由参数filedes数组返回。

filedes[0]为管道里的读取端，所以pipe用read调用的

filedes[1]则为管道的写入端。

返回值: 若成功则返回零,否则返回-1,错误原因存于error中。

错误代码:

EMFILE进程已用完文件描述词最大量

ENFILE系统已无文件描述词可用。

EFAULT参数 filedes 数组地址不合法。

#include <unistd.h>  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
int main( void )  
{  
 int filedes[2];  
 char buf[80];  
 pid\_t pid;  
   
 pipe( filedes );  
   
 if ( (pid=fork()) > 0 )  
 {  
 printf( "This is in the father process,here write a string to the pipe.\n" );  
 char s[] = "Hello world , this is write by pipe.\n";  
 write( filedes[1], s, sizeof(s) );  
 close( filedes[0] );  
 close( filedes[1] );  
 }  
 else  
 {  
 printf( "This is in the child process,here read a string from the pipe.\n" );  
 read( filedes[0], buf, sizeof(buf) );  
 printf( "%s\n", buf );  
 close( filedes[0] );  
 close( filedes[1] );  
 }  
   
 waitpid( pid, NULL, 0 );   
 return 0;  
}

➜ ex03 git:(master) ✗ gcc pipe2.c -o pipe2  
➜ ex03 git:(master) ✗ ./pipe2  
This is in the father process,here write a string to the pipe.  
This is in the child process,here read a string from the pipe.  
Hello world , this is write by pipe.

当管道中的数据被读取后，管道为空。一个随后的read()调用将默认的被阻塞，等待某些数据写入。

如果一个管道的写端一直在写，而读端的引⽤计数是否⼤于0决定管道是否会堵塞，引用计数大于0，只写不读再次调用write会导致管道堵塞；

如果一个管道的读端一直在读，而写端的引⽤计数是否⼤于0决定管道是否会堵塞，引用计数大于0，只读不写再次调用read会导致管道堵塞；

而当他们的引用计数等于0时，只写不读会导致写端的进程收到一个SIGPIPE信号，导致进程终止，只写不读会导致read返回0,就像读到⽂件末尾⼀样。