**操作系统实训报告**

**学号: 201823010205**

**姓名: 段明宇**

**班级: 电科18-2**

**2019年11月**

**山东科技大学**

|  |  |  |
| --- | --- | --- |
| 实习成绩 | |  |
| 指导教师评语 | 指导教师签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_年 \_\_\_\_ 月\_\_\_\_日 | |

**实验一 进程控制**

1. **实训目的**

加深对进程概念的理解，明确进程和程序的区别；掌握Linux操作系统的进程创建和终止操作，体会父进程和子进程的关系及进程状态的变化，及子进程共享父进程资源；进一步认识并发执行的实质，编写并发程序。

1. **实训原理**

fork（）函数：仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

* + 在父进程中，fork返回新创建子进程的进程ID；
  + 在子进程中，fork返回0；
  + 如果出现错误，fork返回一个负值。

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

1. **代码及运行结果**

**（1）一个父进程，两个子进程**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main(void)

{

pid\_t pid1,pid2;

int count=0;

if((pid1=fork())<0){

printf("创建进程1失败");

}else{

if(pid1==0){

//子进程1执行

printf("Child1 process: ");

printf("PID=%d PPID=%d \n",getpid(),getppid());

count++;

printf("count=%d",count);

sleep(2);

}else{

if((pid2=fork())<0){

printf("创建进程2失败");

}else{

if(pid2==0){ //子进程2执行

printf("Child2 process: ");

printf("PID=%d PPID=%d \n",getpid(),getppid());

count++;

printf("count=%d",count);

}

else{

//父进程执行

wait();

wait();

printf("Parent process: ");

printf("PID=%d PPID=%d \n",getpid(),getppid());

count++;

printf("count=%d",count);

exit(0);

}

}

}

}

}

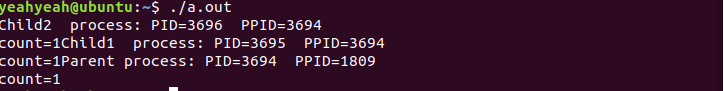
结果分析

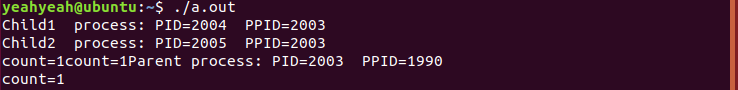
A

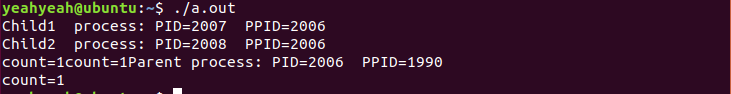
C

**（2）作业一的情况**

源代码运行结果如下图：



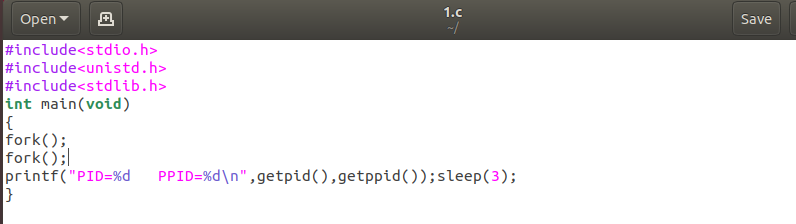




**（3）作业二的情况**

编写程序。编写两个程序：

第一个：代码及运行结果如下：





画图即下图所示：

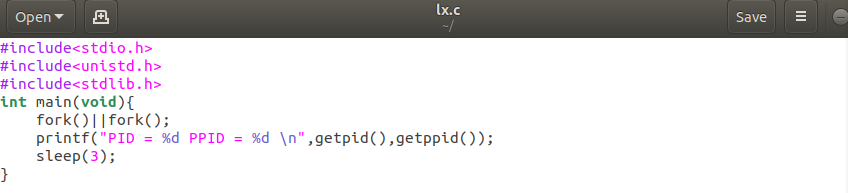
A

C

B

D

第二个：代码及运行结果如下：





画图如下图所示：

A

B

C

**（4）作业三的情况**

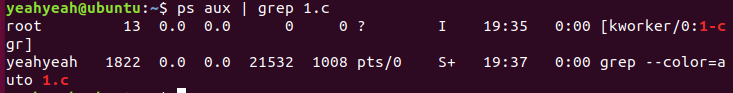
作业运行结果如下图：



1. **查看进程的PCB**

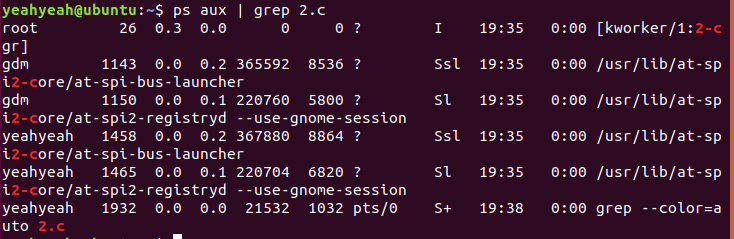
进程控制块PCB是进程组成中最关键的部分。每个进程有唯一的进程控制块;操作系统根据PCB对进程实施控制和管理,进程的动态、并发等特征是利用PCB表现出来的;PCB是进程存在的唯一标志。

PCB中有表明进程状态的信息:该进程的状态是运行态、就绪态还是阻塞态,利用状态信息来描述进程的动态性质。而在Ubuntu中查看进程的PCB所用到的命令行ps aux来进行ps aux是用来查看所有进程的，查看单个进程则是ps aux | grep + 文件名即可。查找资料得知，USER：说明该程序是属于哪一个人的；PID：该程序的代号；%CPU：代表该程序使用了多少CPU资源；%MEM：代表该程序使用了多少的RAM；VSZ,RSS：占去的ram的大小（bytes）；TTY：是否为登入者执行的序，若为tty1-tty6则为本机登入者，若为 pts/?? 则为远程登入者执行的程序；STAT：该程序的状态；START：该程序开始的日期；TIME：该程序运行的时间；COMMAND：该程序的内容。查看内容为实验一所运行的两个程序1.c和2.c且查看结果如下：



分析1.c的PCB：

在1.c中，可直接出来其PCB内容，对于第一个root即为每次执行访问的信息。下边一行则是1.c文件的状态，表达信息为：该程序属于yeahyeah用户，1.c的程序代号为1822，占用CPU资源为0.0，使用RAM0.0，该程序占用ram为21532、1008 bytes。且此程序为远程登录者执行的程序，该程序目前处于一个信息头的前台进程组，执行时间为19：37，执行时间为0：00，最后则是该程序的内容。



而在查看2.c时，出来部分无关内容，只是字符中含有2和c的内容，排除之后，即最后一项为查询内容。表达信息为：该程序属于yeahyeah用户，2.c的程序代号为1932，占用CPU资源为0.0，使用的RAM是0.0，该程序占用ram为21532、1032 bytes。且此程序为远程登录者执行的程序，该程序目前处于一个信息头的前台进程组，执行时间为19：38，执行时间为0：00，最后则是该程序的内容。

**四、实验分析**

实验合理利用了fork()函数创建所需要的进程。通过fork（）函数的数量及应用来得出改变子进程以及孙子进程等。

**实验二 进程的管道通信**

**一、实训目的**

（1）加深对进程概念的理解，明确进程和程序的区别；

（2）学习进程创建的过程，进一步认识并发执行的实质；

（3）分析进程争用资源的现象，学习解决进程互斥的方法；

（4）学习解决进程同步的方法；

（5）掌握Linux系统进程间通过管道通信的具体实现方法。

**二、实训原理**

格式：int fork();

返回值：在子进程中返回0；在父进程中返回所创建的子进程的ID值；当返回-1时，创建失败。

wait() 常用来控制父进程与子进程的同步。

在父进程中调用wait()，则父进程被阻塞，进入等待队列，等待子进程结束。当子进程结束时，父进程从wait()返回继续执行原来的程序。

返回值：大于0时，为子进程的ID值；等于-1时，调用失败。

exit() 是进程结束时最常调用的。

格式：void exit( int status); 其中，status为进程结束状态。

pipe() 用于创建一个管道

格式：pipe(int fd);

其中fd是一个由两个数组元素fd[0]和fd[1]组成的整型

数组，fd[0]是管道的读端口，用于从管道读出数据,fd[1]是管道的写端口，用于向管道写入数据。

返回值：0 调用成功；-1 调用失败。

sleep() 使调用进程睡眠若干时间，之后唤醒。

格式：sleep(int t); 其中t为睡眠时间。

lockf() 用于对互斥资源加锁和解锁。在本实验中该调用的格式为：

lockf(fd[1],1,0)；/\* 表示对管道的写入端口加锁。

lockf(fd[1],0,0)；/\* 表示对管道的写入端口解锁。

write(fd[1],String,Length) 将字符串String的内容写入 管道的写入口。

read(fd[0],String,Length) 从管道的读入口读出信息放入字符串String中。

1. **运行结果**

编译代码三个子进程发送消息，父进程分别读，如下：

#include <stdio.h>

#include <sys/types.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <error.h>

#include <wait.h>

#include <unistd.h>

int main( ){

int pid1,pid2,pid3;

int fd[2];

char outpipe[60],inpipe[60];

pipe(fd);//创建一个管道

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

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

if(pid1==0){

printf("The Child process 1 is sending message!\n");

lockf(fd[1],1,0);//互斥

sprintf(outpipe,"This is the child 1 process's message!\n");

write(fd[1],outpipe,60);

sleep(1);//自我阻塞1秒,让出机会执行下一个进程，增加并发度

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

exit(0);

}

else{

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

printf("pid2=%d\n",pid2);

if(pid2==0){

printf("The Child process 2 is sending message!\n");

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

sprintf(outpipe,"This is the child 2 process's message!\n");

write(fd[1],outpipe,60);

sleep(1);

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

exit(0);

}

else{

while((pid3=fork( ))==-1);

printf("pid3=%d\n",pid3);

if(pid3==0){

printf("The Child process 3 is sending message!\n");

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

sprintf(outpipe,"This is the child 3 process's message!\n");

write(fd[1],outpipe,60);

sleep(1);

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

exit(0);

}

else{

wait(0);//同步

read(fd[0],inpipe,60);

printf("\n%s",inpipe);

wait(0);

read(fd[0],inpipe,60);

printf("%s\n",inpipe);

wait(0);

read(fd[0],inpipe,60);

printf("%s\n",inpipe);

exit(0);

}

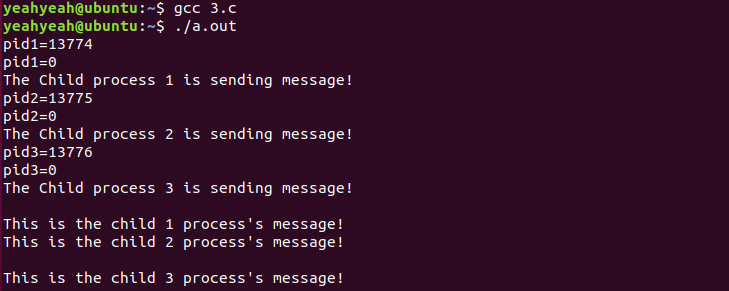
}

}

return 0;

}

运行结果如下图所示：



1. **作业回答**

**4.1指出父进程与两个子进程并发执行的顺序，并说明原因。**

父进程与子进程执行顺序：子进程先执行，然后父进程才执行。这是由进程的同步机制决定的，因为只有子进程向管道中写入信息后，父进程才能读取；否则父进程自己调用wait()系统调用将自己阻塞,将处理机交由子进程。

父进程监控所有进程：子进程此时已经执行了，父进程为了监控子进程以及父进程的运行情况，调用waitpid函数进行监控，在子进程执行过程中，父进程不会消亡，一直处于运行状态，子线程运行完成后，执行waitpid查看当前进程情况。

主进程一定是先执行，一旦启动子进程后续的代码就并发，没有先后顺序。所以每输入一次fock() 指令运行一次结果，就开启了一个子进程。

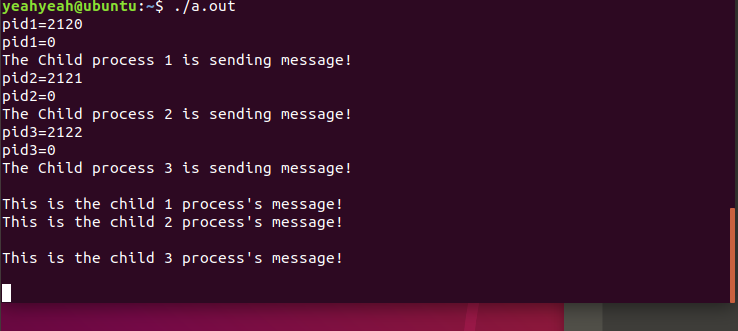
**4.2若不对管道加以互斥控制，会有什么后果？**

管道进行互斥控制，是为防止两个子进程对管道资源进行争夺而产生信息丢失或覆盖。如果不加控制，那么可能一个 子进程写入的信息还没来得及被父进程读出，另一个子进程又先写入信息，那么之前的进程写入的信息将被覆盖，父进程也就读不到之前进程传递来  
的信息了。故而这种资源是临界资源。因此，可能发生死锁。

**4.3说明你是如何实现父子进程之间的同步的。**

利用Wait()函数,使父进程等待子进程运行结束。父进程读出之前确定管道中有数据，否则阻塞自己。当子进程结束时，父进程才执行，此时管道中肯定已经有子进程写入的数据了。子进程在写入之前要确定管道中的数据已经被父进程读出，否则不能写入或者阻塞自己。这可以通过进程间的互斥来间接的办到。因为子进程间的互斥的，所以每个子进程在执行开始都对管道pipe加锁，因此同时就只能有一个子进程向管道写入数据，并且子进程在向管道中写入数据后还要调用sleep()系统调用睡眠若干时间，所以这样就可以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。那么这样就保证了开头所说的子进程在写入之前要确定管道中的数据已经被父进程读出，否则不能写入或者阻塞自己。

**4.4源代码中，如果父进程执行读消息四次什么结果？**

第四次没有结果，一直占用CPU。运行结果及状态如下图所示：

**4.5父进程发送消息，三个子进程分别读。**

#include <stdio.h>

#include <sys/types.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <error.h>

#include <wait.h>

#include <unistd.h>

#include <strings.h>

int main( ){

int pid1,pid2,pid3;

int fd[2];

char string1[3][60];

char string2[3][60];

pipe(fd);//创建一个管道

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

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

if(pid1==0){

printf("The Child process 1 is reviving message!\n");

read(fd[0],string2[1],60);

printf("\n%s",string2[1]);

//自我阻塞1秒,让出机会执行下一个进程，增加并发度

exit(0);

}

else{

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

printf("pid2=%d\n",pid2);

if(pid2==0){

printf("The Child process 2 is reviving message!\n");

read(fd[0],string2[2],60);

printf("\n%s",string2[2]);

//自我阻塞1秒,让出机会执行下一个进程，增加并发度

}

else{

while((pid3=fork( ))==-1);

printf("pid3=%d\n",pid3);

if(pid3==0){

printf("The Child process 3 is reviving message!\n");

read(fd[0],string2[3],60);

printf("\n%s",string2[3]);

//自我阻塞1秒,让出机会执行下一个进程，增加并发度

exit(0);

}

else{

printf("The parent process is sending message!\n");

lockf(fd[1],1,0);//互斥

sprintf(string1[1],"This is the children1 process's message!\n");

write(fd[1],string1[1],60);

//自我阻塞1秒,让出机会执行下一个进程，增加并发度

wait(0);

sprintf(string1[2],"This is the children2 process's message!\n");

write(fd[1],string1[2],60);

wait(0);

sprintf(string1[3],"This is the children3 process's message!\n");

write(fd[1],string1[3],60);

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

//wait(0);

exit(0);

}

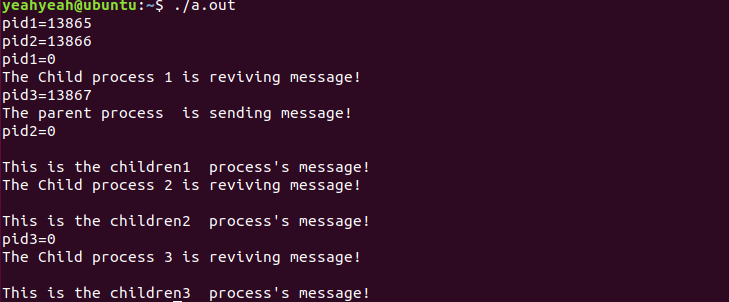
}

}

return 0;

}

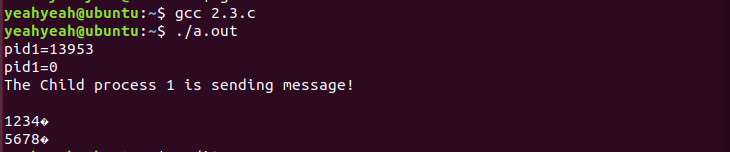
实验编译运行结果如下：



**4.6 新代码，进程A创建一个子进程B, B发送消息“12345678”,A分两次接收，一次接收4个字符。**



运行结果如下：



**实验三 局部性原理**

**一、实训目的**

加深对操作系统存储管理的理解

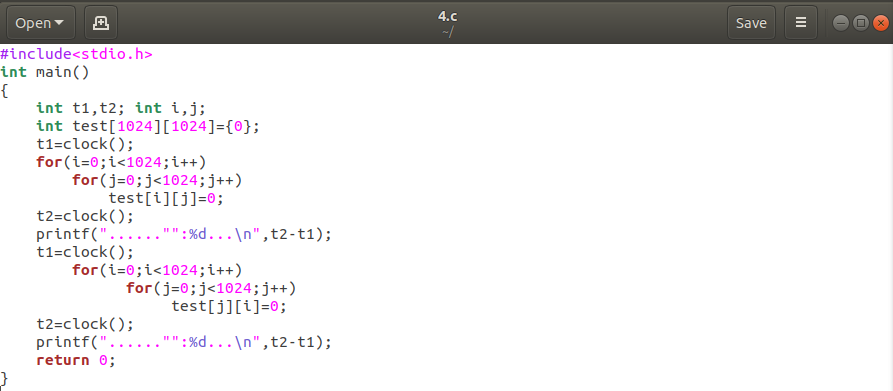
**二、实训原理**

程序的局部性原理：指程序在执行时呈现出局部性规律，即在一段时间内，整个程序的执行仅限于程序中的某一部分。相应地，执行所访间的存储空间也局限于某个内存区域。局部性原理又表现为:时间局部性和空间局部性。时间局部性是指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行:如果某数据被访问，则不久之后该数据可能再次被访问。空间局部性是指一旦程序访问了某个存储单元，则不久之后其附近的存储单元也将被访问..

程序对比分析 for(i=0;i<1024;i++) for(j=0;j<1024;j++) test[j][i]=0; 程序是按列把数组中的元素清“0”的，所以，每执行一次test[j][i]=0就会产生一次缺页中断。因为开始时第一页已经在主存了，因此程序执行时就可以对元素test[1][1]清零，但下一个元素test[2][1]不在该页，就产生缺页中断。按程序上述的编制方法，每装入一页只对一个元素清零后就要产生缺页中断于是总共要产生1024\*1024-1次缺页中断。 for(i=0;i<1024;i++) for(j=0;j<1024;j++) test[i][j]=0;按行清零，每转入一页后就对一行的元素全部清零后才产生缺页中断，故总共产生1024-1次缺页中断。

**三、运行结果**

实验源代码运行：

****

实验多次运行结果：



将行和列的数统一变动，即将中间部分代码改为：

for(i=0;i<2048;i++)

for(j=0;j<2048;j++)

test[i][j]=0;

t2=clock();

printf("......"":%d...\n",t2-t1);

t1=clock();

for(i=0;i<2048;i++)

for(j=0;j<2048;j++)

test[j][i]=0;

出现结果：



结果即为越界，因此将数改小，来分析其结果与内容。

将代码改为：

for(i=0;i<888;i++)

for(j=0;j<888;j++)

test[i][j]=0;

t2=clock();

printf("......"":%d...\n",t2-t1);

t1=clock();

for(i=0;i<888;i++)

for(j=0;j<888;j++)

test[j][i]=0;

输出内容为：



**四、实验分析**

清零分按行清零和按列清零

按行来清零，第零行已在内存中，所以发生1024-1次缺页中断。

按列把数组中的元素清“0”，每执行一次test[j][i]=0就会产生一次缺页中断。因为开始时第一页已经在主存了，因此程序执行时就可以对元素test[1][1]清零，但下一个元素test[2][1]不在该页，就产生缺页中断。按程序上述的编制方法，每装入一页只对一个元素清零后就要产生缺页中断于是总共要产生1024\*1024-1次缺页中断。

而且，根据实验结果来分析，按行清零的次数远小于按列清零的次数，而且，当行和列的界值变小时行的变化远小于列的变化，

**实验四 测试proc文件系统功能**

**一、实验目的**

理解proc伪文件系统的基本概念和功能，掌握常见操作命令。通过实验深入理解操作系统为用户提供服务的方式、方法。

**二、实训原理**

1.Proc是什么

首先，proc是Linux系统根目录下名为“proc”的文件目录。

其次，它是一种伪文件系统。它只存在内存当中，不像普通文件一样占用外存空间。它以文件系统的方式提供应用程序访问系统内核数据的操作接口。proc伪文件系统动态地从系统内核读出所需信息并提交给应用程序。

1. Proc文件系统提供哪些信息？

* 与进程相关的目录
* 通用系统信息
* 网络信息
* 系统控制信息

1. Proc可以作为动态添加的LKM模块的输出接口。
2. 应用程序可以通过读取proc中相应的模块信息来获取动态添加的LKM模块输出。

**三、运行结果**

请运行以下命令并仔细观察实验结果，理解proc文件系统的功能、作用

$ cat /proc/cpuinfo

查看CPU信息

$ cat /proc/modules

保存了当前系统中被加载模块的相关信息。配合使用cat命令可以查看当前系统加载模块的情况。

$ cat /proc/meminfo

内存分析

$ cat /proc/iomem

IO memory空间的地址资源分配情况，以树状结构显示。

$ cat /proc/devices

查看硬件设备

$ cat /proc/self/maps

为一个目录，不同的进程访问该目录时获得的信息是不同的，内容等价于/proc/本进程pid/。

$ cat /proc/filesystems

/proc/filesystems里存放着本台机器linux系统支持的文件挂载系统。查看内核支持的文件系统

$ cat /proc/version

查询出来的结果是：Linux的的内核版本号，gcc编译器版本号，和Ubuntu的版本号。

