**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与工程实验室 **年 月 日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与教育软件** | **年级/专业/班** | **软件162** | **姓名** | 吴广城 | **学号** | 1606100138 |
| **实验课程名称** | **操作系统实验** | | | | | **成绩** |  |
| **实验项目名称** | 实验1 进程管理与进程通信 | | | | | **指导老师** | 陈康民 |

(\*\*\*报告只能为文字和图片,老师评语将添加到此处,学生请勿作答\*\*\*)

1. **实验目的**

**进程管理**

（一）进程的创建实验

1、掌握进程的概念，明确进程的含义

1. 认识并了解并发执行的实质

（二）进程的控制实验

1、掌握进程另外的创建方法

1. 熟悉进程的睡眠、同步、撤消等进程控制方法

（三）进程互斥实验

1、进一步认识并发执行的实质

1. 分析进程竞争资源的现象，学习解决进程互斥的方法
2. 守护进程实验

**进程间通信**

（一）信号机制实验

1、了解什么是信号

2、熟悉LINUX系统中进程之间软中断通信的基本原理

（二）进程的管理通信实验

1、了解什么是管道

2、熟悉UNIX/LINUX支持的管道通信方式

（三）消息的传送与接收实验

1、了解什么是消息

2、熟悉消息传送的机理

（四）共享存储区通信

了解和熟悉共享存储机制

1. **实验内容**

**进程管理**

**（一） 进程的创建实验**

1、编写一段程序，使用系统调用fork( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示'a'，子进程分别显示字符'b'和字符'c'。试观察记录屏幕上的显示结果，并分析原因。

2、修改上述程序，每一个进程循环显示一句话。子进程显示'daughter …'及'son ……'，父进程显示 'parent ……'，观察结果，分析原因。

**（二）进程的控制实验**

1、用fork( )创建一个进程，再调用exec( )用新的程序替换该子进程的内容

2、利用wait( )来控制进程执行顺序

**（三）进程互斥实验**

1、修改实验（一）中的程序2，用lockf( )来给每一个进程加锁，以实现进程之间的互斥

2、观察并分析出现的现象

**（四）守护进程实验**

写一个使用守护进程(daemon)的程序，来实现：

1, 创建一个日志文件/var/log/Mydaemon.log

1. 每分钟都向其中写入一个时间戳(使用time\_t的格式)

**进程间通信**

**（一）信号机制实验**

1、编写程序：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来的中断信号（即按^c键）；捕捉到中断信号后，父进程用系统调用kill( )向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child process1 is killed by parent!

Child process2 is killed by parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent process is killed!

2、分析利用软中断通信实现进程同步的机理

**（二）进程的管理通信实验**

编写程序实现进程的管道通信。用系统调用pipe( )建立一管道，二个子进程P1和P2分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。

**（三）消息的传送与接收实验**

消息的创建、发送和接收。使用系统调用msgget( ),msgsnd( ),msgrev( ),及msgctl( )编制一长度为１k的消息发送和接收的程序。

**（四）共享存储区通信**

编制一长度为1k的共享存储区发送和接收的程序。

**三、实验原理**

**进程管理**

1. **进程的创建实验：**

进程:UNIX中，进程既是一个独立拥有资源的基本单位，又是一个独立调度的基本单位。一个进程实体由若干个区（段）组成，包括程序区、数据区、栈区、共享存储区等。每个区又分为若干页，每个进程配置有唯一的进程控制块PCB，用于控制和管理进程。

**fork( ) 的使用**

创建一个新进程。

系统调用格式：

pid=fork( )

参数定义：

int fork( )

fork( )返回值意义如下：

0：在子进程中，pid变量保存的fork( )返回值为0，表示当前进程是子进程。

>0：在父进程中，pid变量保存的fork( )返回值为子进程的id值（进程唯一标识符）。

-1：创建失败。

**（二）进程的控制实验**

exec( )没有建立一个与调用进程并发的子进程，而是用新进程取代了原来进程。用fork( )建立子进程，然后在子进程中使用exec( )，这样就实现了父进程与一个与它完全不同子进程的并发执行。

wait( )将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在wait( )前已有子进程暂停或终止，则调用进程做适当处理后便返回。

**（三）进程互斥实验**

涉及的系统调用**lockf(files,function,size)**

用作锁定文件的某些段或者整个文件。

本函数的头文件为#include "unistd.h"

参数定义：

int lockf(files,function,size)

int files,function;

long size;

其中：files是文件描述符；function是锁定和解锁：1表示锁定，0表示解锁。size是锁定或解锁的字节数，为0，表示从文件的当前位置到文件尾。

**进程间通信**

**（一）信号机制实验**

信号的基本概念：

每个信号都对应一个正整数常量(称为signal number,即信号编号。定义在系统头文件<signal.h>中)，代表同一用户的诸进程之间传送事先约定的信息的类型，用于通知某进程发生了某异常事件。每个进程在运行时，都要通过信号机制来检查是否有信号到达。若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。实质上，信号机制是对中断机制的一种模拟，故在早期的UNIX版本中又把它称为软中断。

**（二）进程的管理通信实验**

所谓管道，是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为pipe文件。由写进程从管道的写入端（句柄1）将数据写入管道，而读进程则从管道的读出端（句柄0）读出数据。

|  |  |  |
| --- | --- | --- |
| 句柄fd[0]  句柄fd[1] |  | 读出端  写入端 |

管道的类型：1、有名管道（长期存在的、具有路径名的文件）2、无名管道（临时文件）

**1、pipe( )**

建立一无名管道。

系统调用格式

pipe(filedes)

参数定义

int pipe(filedes);

int filedes[2];

其中，filedes[1]是写入端，filedes[0]是读出端。

该函数使用头文件如下：

#include <unistd.h>

#inlcude <signal.h>

#include <stdio.h>

**2、read( )**

系统调用格式

read(fd,buf,nbyte)

功能：从fd所指示的文件中读出nbyte个字节的数据，并将它们送至由指针buf所指示的缓冲区中。如该文件被加锁，等待，直到锁打开为止。

参数定义

int read(fd,buf,nbyte);

int fd;

char \*buf;

unsigned nbyte;

**3、write( )**

系统调用格式

read(fd,buf,nbyte)

功能：把nbyte 个字节的数据，从buf所指向的缓冲区写到由fd所指向的文件中。如文件加锁，暂停写入，直至开锁。

**（三）消息的传送与接收实验**

消息（message）是一个格式化的可变长的信息单元。消息机制允许由一个进程给其它任意的进程发送一个消息。当一个进程收到多个消息时，可将它们排成一个消息队列。消息使用二种重要的数据结构：一是消息首部，其中记录了一些与消息有关的信息，如消息数据的字节数；二个消息队列头表，其每一表项是作为一个消息队列的消息头，记录了消息队列的有关信息。

**（四）共享存储区通信**

共享存储区（Share Memory）是UNIX系统中通信速度最高的一种通信机制。该机制可使若干进程共享主存中的某一个区域，且使该区域出现（映射）在多个进程的虚地址空间中。另一方面，一个进程的虚地址空间中又可连接多个共享存储区，每个共享存储区都有自己的名字。当进程间欲利用共享存储区进行通信时，必须先在主存中建立一共享存储区，然后将它附接到自己的虚地址空间上。此后，进程对该区的访问操作，与对其虚地址空间的其它部分的操作完全相同。进程之间便可通过对共享存储区中数据的读、写来进行直接通信。

**四、实验设备**

Linux操作系统计算机一台

**五、实验要求**

完成实验内容中要求的操作，并回答实验操作后的思考题.

**六、实验程序**

**（一）进程的创建实验**

1、程序一：

#include <stdio.h>

main( )

{

int p1,p2;

while((p1=fork( ))==-1); /\*创建子进程p1\*/

if (p1==0) putchar('b');

else

{

while((p2=fork( ))==-1); /\*创建子进程p2\*/

if(p2==0) putchar('c');

else putchar('a');

}

}



运行结果：bca，bac, abc ,……都是有可能的。

分析：第一个while里如果fork( )调用成功，它向父进程返回子进程的PID，并向子进程返回0，即fork( )被调用了一次，但返回了两次。接下来父进程和子进程P1两个分支运行，判断P1==0，子进程符合，输出‘b’，同理else语句里，子进程输出‘c’，父进程输出‘a’，但是程序是并行执行的，所以输出次序带有随机性。

2、程序二：

#include <stdio.h>

main( )

{

int p1,p2,i;

while((p1=fork( ))==-1); /\*创建子进程p1\*/

if (p1==0)

for(i=0;i<10;i++)

printf("daughter %d\n",i);

else

{

while((p2=fork( ))==-1); /\*创建子进程p2\*/

if(p2==0)

for(i=0;i<10;i++)

printf("son %d\n",i);

else

for(i=0;i<10;i++)

printf("parent %d\n",i);

}

}

**思考题：**

1. 系统是怎样创建进程的？

答：系统创建的第一个进程是init进程。系统中所有的进程都是由当前进程使用系统调用fork()创建的。子进程被创建后继承了父进程的资源。 子进程在创建后执行的是父进程的程序代码。

1. 当首次调用新创建进程时，其入口在哪里？

答：fork系统调用创建的子进程继承了原进程的context，也就是说fork调用成功后，子进程与父进程并发执行相同的代码。但由于子进程也继承了父进程的程序指针，所以子进程是从fork（）后的语句开始执行(也就是新进程调用的入口)。另外fork在子进程和父进程中的返回值是不同的。在父进程中返回子进程的PID（正整数），而在子进程中返回0。所以可以在程序中检查PID的值，使父进程和子进程执行不同的分支。

1. **进程的控制实验**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

main( )

{

int pid;

pid=fork( ); /\*创建子进程\*/

switch(pid)

{

case -1: /\*创建失败\*/

printf("fork fail!\n");

exit(1);

case 0: /\*子进程\*/

execl("/bin/ls","ls","-1","-color",NULL);

printf("exec fail!\n");

exit(1);

default: /\*父进程\*/

wait(NULL); /\*同步\*/

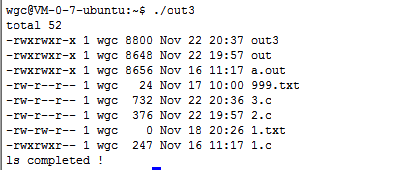
printf("ls completed !\n");

exit(0);

}

}

**运行结果：**



分析：程序在调用fork( )建立一个子进程后，马上调用wait( )，使父进程在子进程结束 之前，一直处于睡眠状态。子进程用exec( )装入命令ls ，exec( )后，子进程的代码被 ls的代码取代，这时子进程的PC指向ls的第1条语句，开始执行ls的命令代码。

可以看出在这里wait( )给我们实现进程的同步。

**思考题：**

（1）可执行文件加载时进行了哪些处理？

答：可执行文件加载时首先是创建一个新进程的fork系统调用，然后用于实现进程自我终止的exit系统调用；改变进程原有代码的exec系统调用；用于将调用进程挂起并等待子进程终止的wait系统调用；获得进程标识符的getpid系统调用等处理过程。

（2）什么是进程同步？wait( )是如何实现进程同步的？

答：进程同步是指对多个相关进程在执行次序上进行协调，以使并发执行的主进程之间有效地共享资源和相互合作，从而使程序的执行具有可在现行。

**（三）进程互斥实验**

#include <stdio.h>

#include <unistd.h>

main()

{

int p1,p2,i;

while((p1=fork( ))== -1); /\*创建子进程p1\*/

if (p1==0)

{

lockf(1,1,0); /\*加锁，这里第一个参数为stdout（标准输出设备的描述符）\*/

for(i=0;i<10;i++)

printf("daughter %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

else

{

while((p2=fork( ))==-1); /\*创建子进程p2\*/

if (p2==0)

{

lockf(1,1,0); /\*加锁\*/

for(i=0;i<10;i++)

printf("son %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

else

{

lockf(1,1,0); /\*加锁\*/

for(i=0;i<10;i++)

printf(" parent %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

}

}

运行结果：

parent…

son…

parent…

daughter…

其实大致上与未上锁的输出结果相同，也是随着执行时间不同，输出结果的顺序有所不同。

分析：上述程序执行时，不同进程之间不存在共享临界资源问题，所以加锁与不加锁效果相同。

**（四）守护进程实验**

main(){

time\_t t; //建立time\_t格式变量

FILE \*fp; //建立文件

fp=fopen("/var/log/Mydaemon.log","a");//打开文件

pid\_t pid; //守护神

pid=fork();

if(pid>0){

printf("Daemon on duty!\n");

exit(0);

}

else if(pid<0){

printf("Can't fork!\n");

exit(-1);

}

while(1){

if(fp>=0){

sleep(60); //等待一分钟再往文件中写入时间戳

printf("Daemon on duty!\n");

t=time(0);

fprintf(fp,"The current time is %s\n",asctime(localtime(&t)));

}

}

fclose(fp);//关闭文件

}



**进程间通信**

**（一）信号机制实验**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void waiting( ),stop( );

int wait\_mark;

main( )

{

int p1,p2,stdout;

while((p1=fork( ))==-1); /\*创建子进程p1\*/

if (p1>0)

{

while((p2=fork( ))==-1); /\*创建子进程p2\*/

if(p2>0)

{

wait\_mark=1;

signal(SIGINT,stop); /\*接收到^c信号，转stop\*/

waiting( );

kill(p1,16); /\*向p1发软中断信号16\*/

kill(p2,17); /\*向p2发软中断信号17\*/

wait(0); /\*同步\*/

wait(0);

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

exit(0);

}

else

{

wait\_mark=1;

signal(SIGINT,SIG\_IGN);// 子信号屏蔽^C中断信号

signal(17,stop); /\*接收到软中断信号17，转stop\*/

waiting( );

lockf(stdout,1,0);

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

lockf(stdout,0,0);

exit(0);

}

}

else

{

wait\_mark=1;

signal(SIGINT,SIG\_IGN);// 子信号屏蔽^C中断信号

signal(16,stop); /\*接收到软中断信号16，转stop\*/

waiting( );

lockf(stdout,1,0);

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

lockf(stdout,0,0);

exit(0);

}  
 }

void waiting( )

{

while(wait\_mark!=0);

}

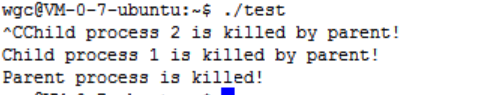
void stop( )

{

wait\_mark=0;

}

运行结果：



分析：上述程序中，由于子进程都会继承父进程的^C的中断信号机制，所以子进程需要屏蔽掉^C的中断的干扰。signal( )都放在一段程序的前面部位，而不是在其他接收信号处。这是因为signal( )的执行只是为进程指定信号值16或17的作用，以及分配相应的与stop( )过程链接的指针。因而，signal( )函数必须在程序前面部分执行。

**思考题：**

1、该程序段前面部分用了两个wait(0)，它们起什么作用？

答：用了两个wait(0)的作用是同时使两个子进程P1和P2发出软中断信号，而不用等待。

2、该程序段中每个进程退出时都用了语句exit(0)，为什么？

答：用exit（0）的作用是使子进程实现自我终止，正常退出此次操作，返回操作系统。

3、为何预期的结果并未显示出？

由于子进程都会继承父进程的^C的中断信号机制，所以子进程需要屏蔽掉^C的中断的干扰。

4、程序该如何修改才能得到正确结果？

为了使程序运行得到正确的结果，可以在每个子进程程序段开头加上忽略“^C”中断信号的语句，即：signal(SIGINT,SIG\_IGN) 。

**（二）进程的管理通信实验**

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

int pid1,pid2;

main( )

{

int fd[2];

char outpipe[100],inpipe[100];

pipe(fd); /\*创建一个管道\*/

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

if(pid1==0)

{

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

sprintf(outpipe,"child 1 process is sending message!");

/\*把串放入数组outpipe中\*/

write(fd[1],outpipe,50); /\*向管道写长为50字节的串\*/

sleep(5); /\*自我阻塞5秒\*/

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

exit(0);

}

else

{

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

if(pid2==0)

{ lockf(fd[1],1,0); /\*互斥\*/

sprintf(outpipe,"child 2 process is sending message!");

write(fd[1],outpipe,50);

sleep(5);

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

exit(0);

}

else

{ wait(0); /\*同步\*/

read(fd[0],inpipe,50); /\*从管道中读长为50字节的串\*/

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

wait(0);

read(fd[0],inpipe,50);

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

exit(0);

}

}

}

延迟5秒后显示：child 2 process is sending message!

再延迟5秒：child 1 process is sending message!



分析：管道应是按照先进先出的规则处理信息。

**思考题：**

（1）程序中的sleep(5)起什么作用？

答：sleep（5）是使管道自我阻塞5秒。

（2）子进程1和2为什么也能对管道进行操作？

答：无名管道：只用该系统调用所返回的文件描述符来标识该文件，故只有调用pipe( )的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。

因为父进程用pipe()创建了一个无名管道，子进程1和2都知道该管道文件描述符，所以进程1和2都能对管道进行操作。

**（三）消息的传送与接收实验**

1、client.c

#include <sys/types.h>

#include <sys/msg.h>

#include <sys/ipc.h>

#define MSGKEY 75

struct msgform

{ long mtype;

char mtext[1000];

}msg;

int msgqid;

void client()

{

int i;

msgqid=msgget(MSGKEY,0777); /\*打开75#消息队列\*/

for(i=10;i>=1;i--)

{

msg.mtype=i;

printf(“(client)sent\n”);

msgsnd(msgqid,&msg,1024,0); /\*发送消息\*/

}

exit(0);

}

main( )

{

client( );

}

2、server.c

#include <sys/types.h>

#include <sys/msg.h>

#include <sys/ipc.h>

#define MSGKEY 75

struct msgform

{ long mtype;

char mtext[1000];

}msg;

int msgqid;

void server( )

{

msgqid=msgget(MSGKEY,0777|IPC\_CREAT); /\*创建75#消息队列\*/

do

{

msgrcv(msgqid,&msg,1030,0,0); /\*接收消息\*/

printf(“(server)received\n”);

}while(msg.mtype!=1);

msgctl(msgqid,IPC\_RMID,0); /\*删除消息队列，归还资源\*/

exit(0);

}

main( )

{

server( );

}

**运行结果：**



分析：两个程序分别编辑，但是server的程序需要后台运行。

**（四）共享存储区通信**

#include <sys/types.h>

#include <sys/shm.h>

#include <sys/ipc.h>

#define SHMKEY 75

int shmid,i; int \*addr;

void client( )

{ int i;

shmid=shmget(SHMKEY,1024,0777); /\*打开共享存储区\*/

addr=shmat(shmid,0,0); /\*获得共享存储区首地址\*/

for (i=9;i>=0;i--)

{ while (\*addr!=-1);

printf("(client) sent\n");

\*addr=i;

}

exit(0);

}

void server( )

{

shmid=shmget(SHMKEY,1024,0777|IPC\_CREAT); /\*创建共享存储区\*/

addr=shmat(shmid,0,0); /\*获取首地址\*/

do

{

\*addr=-1;

while (\*addr==-1);

printf("(server) received\n");

}while (\*addr);

shmctl(shmid,IPC\_RMID,0); /\*撤消共享存储区，归还资源\*/

exit(0);

}

main( )

{

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

if (!i) server( );

system("ipcs -m");

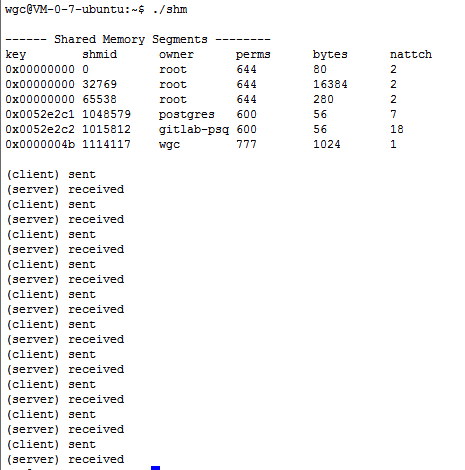
while ((i=fork( ))==-1);

if (!i) client( );

wait(0);

wait(0);

}



1、为了便于操作和观察结果，用一个程序作为“引子“，先后fork()两个子进程，server和client，进行通信。

2、server端建立一个key为75的共享区，并将第一个字节置为-1，作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时，表示收到了信息，进行处理。然后再次把它的值设为-1，如果遇到的值为0，则视为为结束信号，取消该队列，并退出server。server每接收到一次数据后显示“(server)received”。

3、client端建立一个key为75的共享区，当共享取得第一个字节为-1时，server端空闲，可发送请求。client随即填入9到0。期间等待 server 端的再次空闲。进行完这些操作后，client退出。client每发送一次数据后显示“(client)sent”。

4、父进程在server和client均退出后结束。

思考题：

1、比较两种消息通信机制中数据传输的时间

由于两种机制实现的机理和用处都不一样，难以直接进行时间上的比较。如果比较其性能，应更加全面的分析。

（1）消息队列的建立比共享区的设立消耗的资源少。前者只是一个软件上设定的问题，后者需要对硬件的操作，实现内存的映像，当然控制起来比前者复杂。如果每次都重新进行队列或共享的建立，共享区的设立没有什么优势。

（2）当消息队列和共享区建立好后，共享区的数据传输，受到了系统硬件的支持，不耗费多余的资源；而消息传递，由软件进行控制和实现，需要消耗一定的cpu的资源。从这个意义上讲，共享区更适合频繁和大量的数据传输。

（3）消息的传递，自身就带有同步的控制。当等到消息的时候，进程进入睡眠状态，不再消耗cpu资源。而共享队列如果不借助其他机制进行同步，接收数据的一方必须进行不断的查询，白白浪费了大量的cpu资源。可见，消息方式的使用更加灵活。

1. **总结心得**

通过该实验，对 UNIX/LINUX 下进程创建的实现原理与机制有了一个初步的认识，可以用fork() 函数来创建进程和用wait() 函数使进程进入休眠，从而实现进程的同步。进程间的通信中的实验可以用kill() 函数和signal() 函数实现简单的软中断程序，并且领会到了程序中wait\_mark 所起的作用。通过此次实验，我对进程间的管理与通信了有很 较多的认识。