#### 进程间的通信

##### 一 绪论

###### 1 进程间通信

在用户空间实现进程间通信是不可能的，需要通过Linux内核通信。

1)进程 A 和 进程 B 交流的过程 → 进程之间的通信

用户空间内的进程A和B，是独立的小格子，封闭的，在用户空间是不可能的

2)进程之间的交流 a和b a产生数据给b用，则这两个进程之间就需要 一个先运行，一个后运行。

思考：在用户空间编写一个全局变量，设定a运行完后，使得flag=1，b在flag=1的时候才能运行，这种方法测试出来不可行。a和b是独立封闭的进程，只会有a在运行，b一直处于查询状态. 因为进程是独立的,进程和进程之间无法进行交流操作的。

3)解决思路?

找到进程a和b共同可以找到的一个东西 → linux内核

在内核中设置一个对象，a和b对同一个对象操作，a先把产生的数据给对象，b再从对象中去拿.

4) 此处的对象就是进程间通信方式，对象不一样，进程间的通信方式就不一样。

#include <stdlib.h>  
#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main()  
{  
 pid\_t pid;  
 pid = fork();  
  
 if(pid == 0){  
 int i =0;  
 for(i=0;i<5;i++){  
 printf("this is child process i=%d\n",i);  
 usleep(100);  
 }  
 }  
 if(pid>0){  
 int i =0;  
 for(i=0;i<5;i++){  
 printf("this is parent process i=%d\n",i);  
 usleep(100);  
 }  
 }  
  
 while (1);  
 return 0;  
}  
// 进程间直接通信 测试实例

(1) 在vm虚拟机上安装ubuntu https://www.cnblogs.com/Uni-Hoang/p/12871938.html  
(2) 在虚拟机上编写c语言 sudo apt install gcc =>c的编译环境  
(3) 建议参照以下教程实现window10宿主机和ubuntu虚拟机器之间的复制粘贴,基于VMware 的快捷工具https://blog.csdn.net/weixin\_45337440/article/details/103403870  
(4) 使用linux自带的C语言编辑器 https://jingyan.baidu.com/article/f0e83a259ef1c622e5910121.html  
touch demo.c  
gedit demo.c  
gcc demo.c  
./a.out => hello world

###### 2 线程间通信

可以在用户空间就可以实现，可以通过全局变量通信。

线程是对进程进一步的细分。进程是资源分配的最小单位，线程是CPU调度的最小单位。

线程间直接通信实例  
#include "stdio.h"  
#include "stdlib.h"  
#include "pthread.h"  
#include <unistd.h>  
  
int thread\_inter =0;  
void \*fun(void \*var){ //child thread code  
 int j;  
 while (thread\_inter == 0);// child thread won't code until thread\_inter!=0  
 for (j = 0; j < 10; j++) {  
 usleep(100);  
 printf("this is fun j =%d\n",j);  
 }  
}  
int main(){ // main thread  
 int i;  
 char str[] = " hello linux\n";  
 pthread\_t tid;  
 int ret;  
 ret = pthread\_create(&tid,NULL,fun,(void \*)str);  
 if(ret<0){  
 printf(" create thread failure\n");  
 return -1;  
 }  
 for (i = 0; i < 10; i++) {  
 usleep(100);  
 printf("this is main fun i=%d\n",i);  
 }  
 thread\_inter = 1;  
 while (1);  
 return 0;  
}  
  
注意此处编译的时候,应该使用 gcc thread.c -lpthread -o createThread命令行  
原因:pthread库不是Linux系统默认的库，连接时需要使用库libpthread.a

###### 3 通信方式

(6种，拥有同一个内核)

* 管道通信：无名管道、有名管道(文件系统中有名字)
* 信号通道：信号(通知)通信包括信号的发送、信号的接收、信号的处理
* IPC (inter-process Communication)通信：共享内存、消息队列和信号灯

以上都是单机

* socket通信：存在于一个网络中的两个进程之间的通信(两个Linux内核)==>网络编程

###### 4 学习思路

每一种通信方式都是 → 基于文件io的思想。(不用的通信方式,函数名可能不同，但是逻辑思想相同)

open:开辟内存空间  
write:像内存空间中写入数据  
read:从内存空间中读取数据  
close:释放掉内存空间

##### 二 IPC通信框架

1、管道通信框架=> 缓冲区=>队列形式的缓冲区

1) 无名管道 pipe创建,亲缘关系间的进程可以使用 对象是队列

2) 有名管道 mkfifo创建，open开辟。非亲缘关系进程可以使用，管道文件，成为进程间交互的地图指引，不同进程通过这个地图指引找到同一片数据队列缓冲区。即有名管道 对象是队列

2、信号通信框架==>信号通道==>使用函数,通知内核来发信号. 对象是信号

1) 信号的发送(发送信号进程) : kill,raise,alarm   
2) 信号的接收(接受信号进程) : pause,sleep,while(1)  
3) 信号的处理(接受信号进程) : signal

3、IPC通信框架(和文件io类似的)==>以文件i/o的思想来学习

1) 共享内存(数据读取后，数据还会存在。需要对内核缓冲区在进程中映射)

2) 消息队列(链式队列，和管道类似，读完之后就会被删。可以利用数据类型不同，实现基于一个缓冲区的双工通信)

3) 信号灯(多个信号量的集合)

|  |  |
| --- | --- |
| 文件I/O | IPC |
| open | Msg*get,Shm*get,Sem\_get |
| read/write | msgsnd msgrecv shmat shmdt semop |
| close | msgctrl shmctrl semctrl |

##### 三 无名管道通信

###### 1 概述

管道是一个队列（无法形成文件）

内核管道，是由队列实现。因为是队列，所以是特殊的，无法使用常用的文件描述符来创建管道。

open函数只能创建文件(实实在在的).

因此，此处实现的函数是pipe，linux系统自包含函数

pipe(1,2,3)   
1 文件名称  
2 打开模式  
3 权限  
返回值 => 文件描述符  
  
因为是无名管道 ，所以没有文件名称，相应的也就没有对文件的相关权限。同时默认就是队列，所以打开模式也没有可选的参数。

返回值 ：成功是0，出错是-1。 但是又要使用返回的文件描述符来操作读写===>所以文件描述符是通过参数的形式返回过来了。即pipe自带的参数没有可使用的，写在括号里的引用参数是返回的文件描述符。

总结:

pipe函数返回值是0，1代表创建成功与否。返回的文件描述符是在参数中。读完队列出队了，数据就没了。

open函数返回的直接就是文件描述符，读完数据还在文件中。

返回的文件描述符是 size两个大小的数组，单工工作的模式 一个入，一个出。

fd[]==>filedescription  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_> 队列方向  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 fd[1] write fd[0] read   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

注意:在创建pipe成功后,printf打印出fd[0],fd[1]的值是3,4.==>linux系统在创建的时候,会自动创建0,1,2来自己使用.记住即可.fd[0]打出出来是3。

###### 2 pipe函数

所需头文件  
#include<unistd.h>  
函数原型  
int pipe(int fd[2]);  
函数参数  
fd[2]: 文件描述符数组  
函数返回  
成功:信号灯集ID  
失败: -1

测试代码

pipe\_1.c  
  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
  
int main(){  
 int fd[2];  
 int ret = pipe(fd);  
 if(ret<0){  
 printf("create pipe failure\n");  
 return -1;  
 }  
 printf("create pipe success fd[0]=%d,fd[1]=%d\n",fd[0],fd[1]);  
 return 0;  
}

###### 3 单进程的内部的读写

即一个进程自己写了之后,在自己读出来.看效果pipe\_2.c  
write(写到什么地方去,写什么(要写的内容是什么),写多少个) //三个参数  
read(从什么地方读,读/存到那里去,读多少个) // 参数含义  
  
  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
  
int main(){  
 int fd[2];  
 char writebuf[] = "hello linux";  
 char readbuf[128]={0};  
 int ret = pipe(fd);  
 if(ret<0){  
 printf("create pipe failure\n");  
 return -1;  
 }  
 printf("create pipe success fd[0]=%d,fd[1]=%d\n",fd[0],fd[1]);  
  
 // wirte data into pipe   
 write(fd[1],writebuf,sizeof(writebuf));  
 //read from pipe  
 read(fd[0],readbuf,sizeof(readbuf));  
  
 // emplify   
 printf("readbuf = %s\n",readbuf);  
  
 close(fd[0]);  
 close(fd[1]);  
  
 return 0;  
}

###### 4 管道特殊性

无名管道是创建在内存中的，而不是具体的文件，进程结束，空间释放，管道就不存在了。

管道中的东西，读完了就删除了。队列出队，队列内自然就没有数据了。

如果管道中没有可读的数据，就会造成 ===>读阻塞，一直处于读的状态。

###### 5 读阻塞

代码： pipe\_3.c  
逻辑思路 写一次,读两次 第二次自然就会阻塞  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
#include "string.h"  
int main(){  
 int fd[2];  
 char writebuf[] = "hello linux";  
 char readbuf[128]={0};  
 int ret = pipe(fd);  
 if(ret<0){  
 printf("create pipe failure\n");  
 return -1;  
 }  
 printf("create pipe success fd[0]=%d,fd[1]=%d\n",fd[0],fd[1]);  
  
 // wirte data into pipe   
 write(fd[1],writebuf,sizeof(writebuf));  
 //read from pipie  
 read(fd[0],readbuf,sizeof(readbuf));  
  
 // emplify  
 printf("readbuf = %s\n",readbuf);  
  
 //clear data.prepar for the second time to read.  
 memset(readbuf,0,128);  
  
 // read form the pipe queue again // blocked with this code line  
 read(fd[0],readbuf,sizeof(readbuf));  
  
 // second time to emplify   
 printf("second readbuf = %s\n",readbuf);  
  
 close(fd[0]);  
 close(fd[1]);  
  
 return 0;  
}

查看进程的状态  
新打开一个终端  
ps -ajx|grep ./a.out  
状态处于 S+ 睡眠阻塞状态

###### 6 写阻塞

代码 pipe\_4.c  
逻辑思路: fd[1]写入端,不停的写入,超过了管道所能容纳最大内存值.  
代码思路:去掉读代码,同时循环不停的写入.  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
#include "string.h"  
int main(){  
 int fd[2];  
 int i = 0;  
 char writebuf[] = "hello linux";  
 char readbuf[128]={0};  
 int ret = pipe(fd);  
 if(ret<0){  
 printf("create pipe failure\n");  
 return -1;  
 }  
 printf("create pipe success fd[0]=%d,fd[1]=%d\n",fd[0],fd[1]);  
  
 //while(i<5500){  
 while(i<5400){  
 // wirte data into pipe   
 write(fd[1],writebuf,sizeof(writebuf));  
 i++;  
 }  
 // if the pipe storage is full,this sentence won't be printfed;  
 printf("write pipe end\n");  
  
 close(fd[0]);  
 close(fd[1]);  
  
 return 0;  
}  
  
大小是5456~5457个i\* sizeof("hello linux")之间

###### 7 综合实例1

无名管道实现两个进程间通信  
代码 pipe\_5.c  
  
#include <stdlib.h>  
#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main() {  
 pid\_t pid;  
 int fd[2];  
 int ret;  
 char process\_inter = 0;  
 ret = pipe(fd);  
 if (ret < 0) {  
 printf("create pipe failure\n");  
 return -1;  
 }  
 printf("create pipe success\n");  
 pid = fork();  
 if (pid == 0) { // child process code  
 int i = 0;  
 read(fd[0], &process\_inter, 1); // if pipe empty sleep  
 while (process\_inter == 0); // changed by parent process  
 for (i = 0; i < 5; i++) {  
 printf("this is child process i=%d\n", i);  
 usleep(100);  
 }  
 }  
 if (pid > 0) { // parent process code  
 int i = 0;  
 for (i = 0; i < 5; i++) {  
 printf("this is parent process i=%d\n", i);  
 usleep(100);  
 }  
 process\_inter = 1;  
 sleep(5);  
 // change process\_inter and write into pipe  
 write(fd[1], &process\_inter, 1);  
 }  
  
 while (1);  
 return 0;  
}

缺点:无名管道不能实现不是父子进程(亲缘关系)之间的通信.

上述进程有先后条件制约关系,且在同一个main方法内,公用一个管道.但是两个进程之间完全没有关系的时候==>两个不同的管道,就无法实现父子进程之间的通信

##### 四 有名管道

###### 1 文件分类

linux七种文件分类   
普通文件(-)   
目录文件(d)  
连接文件(特指软链接文件)(l)  
命名管道文件(p)  
套接字文件(S),用于实现网络两个进程进行通信  
字符设备文件(C),如键盘支持以character为单位进行线性访问  
块设备文件(b),如硬盘  
  
  
创建文件   
open函数只能创建普通文件  
mkdir 目录文件  
ln -s [option] 软连接文件  
mkfifo 创建管道文件 ====> make fist in fisrt out  
  
文件系统中，实实在在存在一个文件节点===>管道文件

###### 2 mkfifo函数

函数形式 int mkfifo(const char \* filename,mode\_t mode)  
功能: 创建管道文件  
  
参数:管道文件名、权限，创建的文件权限仍然和umask(掩码)有关系。  
(Linux umask命令指定在建立文件时预设的权限掩码。  
umask可用来设定[权限掩码]。[权限掩码]是由3个八进制的数字所组成，将现有的存取权限减掉权限掩码后，即可产生建立文件时预设的权限。)

返回值: 成功0,失败-1

测试代码

#include "stdio.h"  
#include "unistd.h"  
#include "stdlib.h"  
#include <unistd.h> //for read close  
#include <sys/stat.h> //for mkfifo  
#include<fcntl.h> //for open  
  
int main(){  
 int ret; // result  
 ret = mkfifo("./myfifo",0777);  
 if(ret<0){  
 printf("creat myfifo failure\n");  
 return -1;  
 }  
 printf("create myfifo success\n");  
 return 0;  
}  
  
ls -al 查看  
prwxr-xr-x 1 syj syj 0 10月 10 14:42 myfifo  
  
7种系统文件中 => 有名管道,字符设备,块设备,套接字 只会有文件节点(inode),不占磁盘空间.  
普通文件(-)目录文件(d)连接文件(特指软链接文件)(l)既有文件节点(inode),也会占磁盘空间.

###### 3 umask权限掩码

mkfifo 生成的文件权限和umask之间的关系  
无名管道 只能实现有亲缘关系之间的通信  
有名管道 毫无关系的任务/进程之间的通信 mkfifo ==>open函数只能新建普通文件

预备知识 请先查看https://blog.csdn.net/huxinguang\_ios/article/details/81026811  
直接umask查看的情况  
umask  
0022 ==>此处可能不同,  
根据拆分0 0 2 2  
rwx =>2 值 则r-x 没有写的权限  
  
ls -l  
prwxr-xr-x 1 syj syj 0 10月 10 14:42 mkfifo  
rwx r-x r-x  
111 101 101 =>四位数字=> 0 755 原本设定的是0777   
  
修改权限掩码  
umask 001 组用户只有rw 没有x权限  
因为代码中设定了为0777除去组用户的执行权限  
则可以推导出ls-l的权限是0776  
p rwx rwx rw- 1 syj syj 0 10月 10 15:09 mkfifo  
  
得出结论 代码中设定的权限依然和umask有关系  
  
注意:  
mkfifo目前还只是生成了一个文件节点,并没有在内核中开辟出一块内存/管道,真正开辟的是在使用open函数打开的时候.  
mkfifo 创建管道  
open 打开管道===> 字符设备和块设备 也是一样的,创建和打开函数不一样

###### 4 综合实例 1

管道实现无亲缘关系进程之间的通信  
逻辑思路 first进程先运行完,通信告知second进程开始运行.两个进程之间没有亲缘关系.  
代码逻辑: first.c运行代码之后,修改存入管道的值,second进程从管道中取出这个值,这个值改变second中代码的运行逻辑,使得second代码继续运行完.否则就一直等待.

first.c  
  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
#include "sys/types.h"  
#include "fcntl.h"  
  
int main(){  
 int fd;  
 int i;  
 char process\_inter =0;  
 // open named\_pipe file,make sure you have kept mkfifo rightly  
 fd=open("./myfifo",O\_WRONLY);  
 if(fd<0){  
 printf("open myfifo failure\n");  
 return -1;  
 }  
 printf("first open myfifo success\n");  
  
 for (i = 0; i < 5; ++i) {  
 printf("this is first process i = %d\n",i);  
 usleep(100);  
 }  
 process\_inter=1;  
 sleep(5);  
 write(fd,&process\_inter,1);  
 while (1);  
 return 0;  
}

second.c  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
#include "sys/types.h"  
#include "fcntl.h"  
  
int main(){  
 int fd;  
 int i;  
 char process\_inter =0;  
 // open named\_pipe file,make sure you have kept mkfifo correctly  
 fd=open("./myfifo",O\_RDONLY);  
 if(fd<0){  
 printf("open myfifo failure\n");  
 return -1;  
 }  
 printf("second open myfifo success\n");  
 read(fd,&process\_inter,1);  
 // process\_inter value will changed if named\_pipe run correctly  
 while (process\_inter==0);  
 for (i = 0; i < 5; ++i) {  
 printf("this is second process i = %d\n",i);  
 usleep(100);  
 }  
 while (1);  
 return 0;  
}

gcc -o first first.c   
gcc -o second second.c  
然后在两个不同的terminal中先运行./first,再运行./second.  
正确结果: 运行first后,进程阻塞.这时候运行./second,阻塞消除.first打印出数据,second之后接着也打印出数据.  
  
  
总结: mkfifo 为两个不相关联的进程之间,创建了一个共同点myfifo管道文件.  
不同进程之间open同一个管道文件,相互就可以通过管道来通信交流.  
注意:  
mkfifo 创建管道  
open 打开管道===> 字符设备和块设备 也是一样的,创建和打开函数不一样

##### 五 信号通信

###### 1 概述

管道通信框架=> 缓冲区=>队列形式的缓冲区  
无名管道 pipe创建,亲缘关系间的进程可以使用  
有名管道 mkfifo创建,open开辟. 非亲缘关系进程可以使用,管道文件,成为进程间交互的地图指引,不同进程通过这个地图指引找到同一片数据队列缓冲区.=>即有名管道

信号通信  
缓冲数据的这个对象,linux已经存在了,不用自己去创建管道对象.直接在用户空间(内核外可操作的空间按)  
不能传送信号,用户空间a和b进程均可以和内核交流,则使用内核中的信号对象作为媒介来实现通信.  
  
查看Linux内核可以发送多少中信号 ==> 64种  
kill -l  
 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP  
 6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1  
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM  
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP  
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ  
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR  
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3  
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13  
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12  
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7  
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  
63) SIGRTMAX-1 64) SIGRTMAX   
  
问题:  
1 选取哪一种信号?  
2 内核发给哪一个进程?===>具体哪一个pid(进程号)  
3 kill -9 pid => 杀死一个进程  
kill进程 9号信号 pid(某一个进程)  
这就是一个进程 发送信号 给另一个进程的实例

###### 2 kill 函数

信号的发送  
头文件  
#include<signal.h>  
#include<sys/types.h>  
函数原型  
Int kill(pid\_t pid,int signal)  
  
函数参数值  
pid\_t pid   
正数 要接受信号的进程的进程号  
0 :信号要被发送到所有和pid进程在同一个进程组的进程  
-1 :信号发给所有的进程表中的进程(注意:除了进程号最大的进程外)  
  
signal: 信号  
  
函数返回值  
成功0,失败-1

###### 3 自定义kill函数

字符串转换为整型函数==>atoi==>小技巧 man 函数名 可以查看linuxlib中的一些函数

函数思路:传入三个参数 a进程 信号值 b进程  
1 测试传参转换为整型正确与否??  
#include "sys/types.h"  
#include "signal.h"  
#include "unistd.h"  
#include "stdio.h"  
#include "stdlib.h"  
  
int main(int argc,char \*argv[]){  
 int sig;  
 int pid;  
 if(argc < 3){  
 printf("please input param\n");  
 return -1;  
 }  
 sig = atoi(argv[1]);  
 pid = atoi(argv[2]);  
 printf("sig=%d,pid=%d\n",sig,pid);  
 return 0;  
}  
  
  
编译===>gcc -o mykill\_1 mykill\_1.c  
运行===>./mykill\_1 1 2  
输出结果==>sig=1,pid=2

2 测试kill 自定义的进程 (后续的操作gcc -o xxx xxx.c 命令相同,因此省略)  
test.c  
#include "stdio.h"  
  
int main(){  
 while (1);  
 return 0;  
}  
  
  
mykill\_2.c //增加了kill一行代码  
#include "sys/types.h"  
#include "signal.h"  
#include "unistd.h"  
#include "stdio.h"  
#include "stdlib.h"  
  
int main(int argc,char \*argv[]){  
 int sig;  
 int pid;  
 if(argc < 3){  
 printf("please input param\n");  
 return -1;  
 }  
 sig = atoi(argv[1]);  
 pid = atoi(argv[2]);  
 printf("sig=%d,pid=%d\n",sig,pid);  
 kill(pid,sig);// 新增的kill代码  
 return 0;  
}  
  
  
ps -ajx => 找到test对应的pid数值 例如 pid = 36766  
  
运行 ./mykill\_2 9 36766  
输出结果: ./test==> 已杀死/killed  
  
  
注意：其中argc是外部输入的参数个数，argv[ ]是参数的字符串数组

###### 4 信号通信框架

**信号的发送(发送信号进程) :**kill(可以和任意进程发送信号,可以让内核发送不同的信号)   
raise(只能够向当前进程发送信号,但可以让内核发送不同的信号)  
alarm (只能够向当前进程发送信号,只能让内核发送闹钟信号==>终止信号)  
**信号的接收(接受信号进程) :**pause(立即暂停进程,使进程进入睡眠状态)  
sleep(立即暂停,使进程进入睡眠状态一段时间)  
while(1)(原地运行.进程始终是运行状态)  
**信号的处理(接受信号进程) :**signal(接受信号,进行相应的处理)

###### 5 raise函数

发信号给自己 ===>相当于 kill(getpid(),sig)  
所需头文件   
#include<signal.h>  
#include<sys/types.h>  
函数原型  
int raise(int sig);  
函数传入值  
sig: 信号  
  
进程a,调用内核的raise函数,raise函数发一个信号给进程a自己.==>此函数可以用于进程和内核间的信息交流.  
raise\_1.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
  
int main(){  
 // 不加\n的情况 库缓存，不会加载到内核 =》\_exit()不在内核库中，退出时，不会执行前面的代码  
 // 加上了\n的情况，加载到内核 exit()包含在内核库中，推出前执行前面的代码  
 // printf("raise before\n");  
 printf("raise before");  
 // 验证后可得 调用的是\_exit()  
 raise(9);  
 printf("raise after\n");  
 return 0;  
}

注意:此处fork函数,一个进程，包括代码、数据和分配给进程的资源。fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，  
也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。  
其中一个进程（使用新的 pid）里面的 fork () 返回零， 这个进程就是“子进程”；  
而另一个进程（使用原来的 pid）中的 fork () 返回前面那个子进程的 pid，他自己被称为“父进程”   
  
在使用fork()函数创建子进程的时候，我们的头脑内始终要有一个概念：在调用fork()函数前是一个进程在执行这段代码，而调用fork()函数后就变成了两个进程在执行这段代码。两个进程所执行的代码完全相同，都会执行接下来的if-else判断语句块。  
当子进程从父进程内复制后，父进程与子进程内都有一个"pid"变量：在父进程中，fork()函数会将子进程的PID返回给父进程，即父进程的pid变量内存储的是一个大于0的整数；而在子进程中，fork()函数会返回0，即子进程的pid变量内存储的是0；如果创建进程出现错误，则会返回-1，不会创建子进程。  
  
SIGTSTP 该信号用于暂停交互进程,用户可键入SUSP字符(通常是CTRL-Z)发出这个信号.   
==>暂停进程   
  
linux中进程状态 ==>ps -ajx  
R 正在运行或将要运行；  
S 休眠可中断；  
D 休眠不可中断；（通常 IO 的进程）；  
T 停止或被追踪；  
Z 僵尸状态；  
X 死亡状态 ；  
  
raise\_2.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"   
  
int main(){  
 pid\_t pid;  
 pid = fork();  
 if(pid>0){ //parent peocess create process   
 sleep(8); // Sleep=>Run  
 while(1);  
 }  
 if(pid==0){ //child process creat success  
 printf("raise function before\n");  
   
 raise(SIGTSTP);// status change to T  
 printf("raise function after\n");  
 exit(0);  
 }  
 return 0;  
}  
  
  
新启动terminal   
ps -ajx 查看进程状态  
  
运行效果  
前8秒内 父进程睡眠  
 36779 37462 37462 36779 pts/2 37462 S+ 1000 0:00 ./raise\_2  
 37462 37463 37462 36779 pts/2 37462 T+ 1000 0:00 ./raise\_2  
八秒钟后 父进程运行  
 36779 37437 37437 36779 pts/2 37437 R+ 1000 0:25 ./raise\_2  
 37437 37438 37437 36779 pts/2 37437 T+ 1000 0:00 ./raise\_2  
子进程  
运行了 raise(SIGTSTP);// status change to T 一直为T

###### 6 raise函数和kill函数

raise函数和kill函数结合使用的时候,熟悉进程状态  
  
waitpid函数详解  
格式 pid\_t waitpid(pid\_t pid,int \*status,int options)  
  
pid   
从参数的名字pid和类型pid\_t中就可以看出，这里需要的是一个进程ID。但当pid取不同的值时，在这里有不同的意义。  
pid>0时，只等待进程ID等于pid的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。  
pid=-1时，等待任何一个子进程退出，没有任何限制，此时waitpid和wait的作用一模一样。  
pid=0时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬。  
pid<-1时，等待一个指定进程组中的任何子进程，这个进程组的ID等于pid的绝对值。  
  
options  
options提供了一些额外的选项来控制waitpid，目前在Linux中只支持WNOHANG和WUNTRACED两个选项，这是两个常数，可以用"|"运算符把它们连接起来使用，比如：  
  
ret=waitpid(-1,NULL,WNOHANG | WUNTRACED);  
如果我们不想使用它们，也可以把options设为0，如：  
ret=waitpid(-1,NULL,0);  
如果使用了WNOHANG(wait no hung)参数调用waitpid，即使没有子进程退出，它也会立即返回，不会像wait那样永远等下去。  
  
返回值和错误  
waitpid的返回值比wait稍微复杂一些，一共有3种情况：  
当正常返回的时候，waitpid返回收集到的子进程的进程ID；  
如果设置了选项WNOHANG，而调用中waitpid发现没有已退出的子进程可收集，则返回0；  
如果调用中出错，则返回-1，这时errno会被设置成相应的值以指示错误所在；  
当pid所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid就会出错返回，这时errno被设置为ECHILD；  
  
raise\_3.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
 pid\_t pid;  
 pid = fork();  
 if(pid>0){  
 sleep(8);  
 // 当子进程T暂停状态,将子进程kill掉,子进程所占用的空间还没有被回收,所以此时,子进程处于僵尸进程状态 Z  
 // if(waitpid(pid,NULL,0)==0) 可用于测试效果  
 if(waitpid(pid,NULL,WNOHANG)==0) //waitpid发现没有已退出的子进程可收集，则返回0  
 {  
 kill(pid,9);  
 }  
 while(1);  
 }  
 if(pid==0){  
 printf("raise function before\n");  
 raise(SIGTSTP);  
 printf("raise function after\n");  
 exit(0);  
 }  
 return 0;  
}

使用wait函数回收子进程空间  
  
  
raise\_4.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
 pid\_t pid;  
 pid = fork();  
 if(pid>0){  
 sleep(8);  
 // 当子进程T暂停状态,将子进程kill掉,子进程所占用的空间还没有被回收,所以此时,子进程处于僵尸进程状态 Z  
 // if(waitpid(pid,NULL,0)==0) 可用于测试效果 因为会返回子进程id,所以子进程不会被kill,将一直处于暂停T状态.  
 if(waitpid(pid,NULL,WNOHANG)==0) //waitpid发现没有已退出的子进程可收集，则返回0  
 {  
 kill(pid,9);  
 }  
 wait(NULL);// wait函数如果存在,则会回收资源,清除僵尸进程.则子进程在8秒后会经历 kill 并被wait函数回收的过程  
 while(1);  
 }  
 if(pid==0){  
 printf("raise function before\n");  
 raise(SIGTSTP);  
 printf("raise function after\n");  
 exit(0);  
 }  
 return 0;  
}  
  
pe -ajx 查看验证效果

7 alarm函数

闹钟信号 SIGALRM==>内核可发送 默认是终止进程  
进程调用kill 和 raise函数成功的时候,内核会立即发送对应信号  
进程调用alarm函数成功的时候,内核会在定时器时间到了的时候,才会发出信号==>定时一段时间才发送信号  
  
函数分析  
头文件 #include "unistd.h"  
函数原型: unsigned int alarm (unsigned int seconds) // 因为是固定只发送一种信号(SIGALRM),所以不用像kill -9 process 一样告诉内核发送 9 这种信号  
函数传入值:seconds 定时秒数  
  
函数返回值:  
成功 如果调用alarm前,进程中已经设置了闹钟时间,则返回上一个闹钟时间的剩余时间,否则返回0.  
失败 -1

用法  
代码逻辑 进程运行处理的时间,超过alarm的定时时长.这样alarm就会提前终止进程,以便查看效果.  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
 int i;  
 i=0;  
 printf("alarm before\n");  
 alarm(9);  
 while(i<20){  
  
 i++;  
 sleep(1);  
 printf("process things,i=%d\n",i);  
 }  
 return 0;  
}

###### 7 pause函数

所需头文件  
#include"unistd.h"  
函数原型:int pause(void);  
函数返回值:成功 0 出错 -1  
pause:进程状态为S ==>和sleep函数一样 sleep 是sleep一段时间,pause会一直sleep  
  
  
pause\_1.c   
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
 int i;  
 i=0;  
 printf("pause before\n");  
 pause();  
 // this sentence won't be printed  
 printf("pause after\n");  
 while(i<20){  
  
 i++;  
 sleep(1);  
 printf("process things,i=%d\n",i);  
 }  
 return 0;  
}  
  
  
  
ps -ajx  
 37576 38897 38897 37576 pts/0 38897 S+ 1000 0:00 ./pause\_1  
  
  
在pause-1.c睡眠的terminal中,键盘输入ctrl + c 键盘输入符,会发送信号给内核,让内核发送一个终止信号给pause\_1.c

###### 8 signal函数

实际运行中,a进程调用函数成功,成功让内核发送一个信号给b进程,b进程处理完信号任务之后,会继续等待或者运行自己其他的数据.  
  
函数分析:  
所需头文件:   
#include"signal.h"  
函数原型:  
void(\*signal(int signum,void(\*handler)(int)))(int);  
函数传入值:  
signum 指定信号 ===>处理哪一个信号  
handler:  
1 SIG\_IGN:忽略该信号===>signal ignore  
2 SIG\_DFL:采用系统默认方式处理信号 ===>signal default  
3 自定义的信号处理方式   
函数返回值.(按照相同的函数形式,声明自定义处理函数) ====>怎样处理信号  
成功:设置之前的信号处理方式  
失败:-1  
  
  
详解void(\*signal(int signum,void(\*handler)(int)))(int);  
  
1. 令 A === void(\*handler)(int)===>函数指针变量,指向一个函数,包括一个整型的参数,无返回值的一个函数.  
2 void(\*signal(int signum,A))(int);  
3 B === \*signal(int signum,A)===>函数指针变量,只想一个函数,包括一个整型的参数,一个指针变量.返回值则是一个函数指针.  
4 void(B)(int)===>无返回值,参数为一个整型的函数  
  
测试函数signal\_1.c函数  
  
代码思路:首先是通过alarm函数.过9秒之后,发送SIGALRM信号给当前进程.  
当此进程收到信号之后,会跳转到myfun()打印十条语句,在返回main函数继续运行.  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
void myfun(int signum){  
 int i;  
 i=0;  
 while (i<10){  
 printf("process signal signum = %d\n",i);  
 sleep(1);  
 i++;  
 }  
 return; // return main function  
}  
int main(){  
 int i;  
 i=0;  
 // when reviece 14 ==> SIGALRM ===>alarm ==>signal let myfun run  
 signal(14,myfun);  
 printf("alarm before\n");  
 alarm(9); // after 9seconds work,tell linux core send SIGALRM  
 while(i<20){  
 i++;  
 sleep(1);  
 printf("process things,i=%d\n",i);  
 }  
 return 0;  
}

signal函数的三种处理方式之  
1 SIG\_IGN:忽略该信号   
2 SIG\_DFL:采用系统默认方式处理信号   
  
signal\_2.c ===>SIG\_IGN  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
void myfun(int signum){  
 int i;  
 i=0;  
 while (i<10){  
 printf("process signal signum = %d\n",i);  
 sleep(1);  
 i++;  
 }  
 return; // return main function  
}  
int main(){  
 int i;  
 i=0;  
 // when reviece 14 ==> SIGALRM ===>alarm ==>signal let myfun run  
 signal(14,myfun);  
 printf("alarm before\n");  
 alarm(9); // after 9seconds work,tell linux core send SIGALRM  
 signal(14,SIG\_IGN);// now this is signal is the newest ,so the signal is SIGIGN ====>not is 14   
 while(i<20){  
 i++;  
 sleep(1);  
 printf("process things,i=%d\n",i);  
 }  
 return 0;  
}  
  
代码会忽略alarm定时到了后发出的信号  
  
  
signal\_3.c ===>SIG\_DEL  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
void myfun(int signum){  
 int i;  
 i=0;  
 while (i<10){  
 printf("process signal signum = %d\n",i);  
 sleep(1);  
 i++;  
 }  
 return; // return main function  
}  
int main(){  
 int i;  
 i=0;  
 // when reviece 14 ==> SIGALRM ===>alarm ==>signal let myfun run  
 signal(14,myfun);  
 printf("alarm before\n");  
 alarm(9); // after 9mimutes work,tell linux core send SIGALRM  
 signal(14,SIG\_IGN);  
 signal(14,SIG\_DFL);// now this is signal is the newest ,so the signal is SIGIGN ====>not is 14 or SIG\_IGN  
 while(i<20){  
 i++;  
 sleep(1);  
 printf("process things,i=%d\n",i);  
 }  
 return 0;  
}  
  
内核默认的信号就是alarm函数定时后发送的终止信号,所以signal也继承执行了终止信号,进程定时结束.

###### 9 综合实例1

signal\_4.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
void myfun(int signum){  
 int i;  
 i=0;  
 while (i<5){  
 printf("receive signum=%d,i=%d\n",signum,i);  
 sleep(1);  
 i++;  
 }  
 return;  
}  
  
int main(){  
 pid\_t pid;  
 pid = fork();  
 if(pid>0){  
 int i;  
 i=0;  
 signal(10,myfun);  
 while (1){  
 printf("parent process thing ,i=%d\n",i);  
 sleep(1);  
 i++;  
 }  
 }  
 if(pid==0){  
 sleep(10);  
 kill(getppid(),10); //getparentpid send 10signal  
 sleep(10);  
 exit(0);  
 }  
 return 0;  
}

10 综合实例2

综合实例1bug优化  
解决在signal\_4.c中子进程的僵尸状态问题  
signal\_5.c   
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/wait.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
void myfun(int signum){  
 int i;  
 i=0;  
 while (i<5){  
 printf("receive signum=%d,i=%d\n",signum,i);  
 sleep(1);  
 i++;  
 }  
 return;  
}  
void myfun1(int signum){  
 printf("receive signum = %d\n",signum);  
 wait(NULL);  
 return;  
}  
  
int main(){  
 pid\_t pid;  
 pid = fork();  
 if(pid>0){  
 int i;  
 i=0;  
 signal(10,myfun);  
 // signal(17,myfun1); // way 1  
 waitpid(getpid(),NULL,0); // way 2  
 while (1){  
 printf("parent process thing ,i=%d\n",i);  
 sleep(1);  
 i++;  
 }  
 }  
 if(pid==0){  
 sleep(10);  
 kill(getppid(),10); //getparentpid send 10signal  
 sleep(10);  
 exit(0); //kill SIGCHILD(17) parent===>kill(getppid(),17)  
 }  
 return 0;  
}

##### 六 共享内存

###### 1 概述

创建或打开共享内存对象  
共享内核在内核是什么样子?  
一块缓存,就类似于用户空间的数组或malloc函数分配的空间一样

###### 2 shmget函数

所需头文件  
#include "sys/types.h"  
#include "sys/ipc.h"  
#include "sys/shm.h"  
函数原型  
int shmget(key\_t key,int size,int shmflg);  
函数参数  
key:IPC\_PRIVATE或 ftok函数的返回值  
size:共享内存区的大小  
shmflg:同open函数的权限位,也可以用八进制表示法  
函数返回  
成功:共享内存标识符---ID--文件描述符  
失败: -1  
  
测试shmget函数创建共享内存  
shm\_1.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
  
 int shmid;  
 //0777 rwx rwx rwx  
 shmid = shmget(IPC\_PRIVATE,128,0777);  
 if(shmid<0){  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n",shmid);  
 system("ipcs -m");  
 return 0;  
}  
  
  
查看IPC对象 ipcs -m==> memory, ==> -q queue, ==> -s signal   
删除IPC对象 ipcrm -m id   
返回值:共享内存段标识符 IPC的ID号

以下是shmget以IPC\_PRIVATE为key创建的共享内存的详细情况  
键 shmid 拥有者 权限 字节 连接数 状态   
0x00000000 32826 syj 777 128 0   
0x00000000 32827 syj 777 128 0   
可以看到见都是0x00000000 ==> 0  
  
使用ftok的返回值作为key来创建共享内存  
  
关于ftok函数，先不去了解它的作用来先说说为什么要用它，共享内存，消息队列，信号量它们三个都是找一个中间介质，来进行通信的，这种介质多的是。就是怎么区分出来，就像唯一一个身份证来区分人一样。你随便来一个就行，就是因为这。只要唯一就行，就想起来了文件的设备编号和节点，它是唯一的，但是直接用它来作识别好像不太好，不过可以用它来产生一个号。ftok()就出场了。ftok函数具体形式如下：  
key\_t ftok(const char \*pathname, int proj\_id);  
其中参数fname是指定的文件名，这个文件必须是存在的而且可以访问的。id是子序号，它是一个8bit的整数。即范围是0~255。当函数执行成功，则会返回key\_t键值，否则返回-1。在一般的UNIX中，通常是将文件的索引节点取出，然后在前面加上子序号就得到key\_t的值。  
  
1、ftok根据路径名，提取文件信息，再根据这些文件信息及project ID合成key，该路径可以随便设置。  
2、该路径是必须存在的，ftok只是根据文件inode在系统内的唯一性来取一个数值，和文件的权限无关。  
3、proj\_id是可以根据自己的约定，随意设置。这个数字,有的称之为project ID; 在UNIX系统上,它的取值是1到255;  
  
IPC操作时IPC\_CREAT和IPC\_EXCL选项的说明  
当只有IPC\_CREAT选项打开时,不管是否已存在该块共享内存，则都返回该共享内存的ID，若不存在则创建共享内存  
当只有IPC\_EXCL选项打开时，不管有没有该快共享内存，shmget()都返回-1  
所以当IPC\_CREAT | IPC\_EXCL时, 如果没有该块共享内存，则创建，并返回共享内存ID。  
  
shm\_2.c  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
  
 int shmid;  
 int key;  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 printf("create key success key = %X\n",key);  
 shmid = shmget(key,128,IPC\_CREAT|0777);  
 if(shmid<0){  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n",shmid);  
 system("ipcs -m");  
 return 0;  
}  
  
2 新建一个a.c  
touch a.c在同一层级的目录  
  
代码结果  
键 shmid 拥有者 权限 字节 连接数 状态   
0x00000000 32826 syj 777 128 0   
0x61012b0d 32828 syj 777 128 0   
  
  
ftok的参数不一样,返回的key值就不一样.  
使用IPC\_PRIVATE创建的共享内存就类似于无名管道一样,只允许有亲缘关系的进程之间的通信  
使用ftok函数的key创建的共享内存则类似于有名管道,可以允许无亲缘关系的进程之间的通信

###### 3 shmat函数

之前每次的读写,都会进入内核,如何不进入内核呢?  
使用shmat函数,将内核中的共享内存映射到用户空间地址中.在用户空间有一块和内核中一样的共享内存存在,  
这样可以方便用户空间对共享内存的操作,使用地址映射的方式.  
  
函数分析  
void \*shmat(int shmid,const void \*shmaddr,int shmflg); // 函数类似于malloc  
参数:  
 第一个参数: ID号  
 第二个参数: 映射到的地址,设定为NULL的时候,系统自动完成映射  
 第三个参数: shmflg; SHMRDONLY共享内存只可读,默认是0,表示共享内存可读写  
返回值:  
成功:映射后的地址  
失败:NULL  
  
代码测试 创建映射空间  
shm\_3.c   
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
  
int main(){  
  
 int shmid;  
 int key;  
 char \*p;  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 printf("create key success key = %X\n",key);  
 shmid = shmget(key,128,IPC\_CREAT|0777);  
 if(shmid<0){  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n",shmid);  
 system("ipcs -m");  
  
 // create a mapping for the share memory in userspace  
 p=(char \*)shmat(shmid,NULL,0);  
 if(p==NULL){  
 printf("shmat function failure\n");  
 return -3;  
 }  
 // write share memory stdin datasource from => Keyboard input  
 fgets(p,128,stdin);  
  
 //start read share memory  
 printf("read share memory data:%s",p);  
  
 //second read share memory  
 printf("second read share memory data:%s",p);  
   
 return 0;  
}  
运行结果  
  
x00000000 12 syj 600 524288 2 目标   
0x00000000 15 syj 600 524288 2 目标   
0x00000000 16 syj 600 524288 2 目标   
0x00000000 17 syj 600 524288 2 目标   
0x00000000 32808 syj 600 524288 2 目标   
0x61012b0d 32830 syj 777 128 0   
  
hello linux // 手动输入内容  
read share memory data:hello linux  
second read share memory data:hello linux

###### 4 shmdt&shmctl函数

共享内存创建之后,一直存在于内核中,直到被删除或系统关闭   
共享内存和管道不一样,读取后,内容仍在其共享内存中   
   
当共享缓存使用完成之后,需要释放内核中的共享内存,需要使用到shmctrl函数   
用户空间由shmat创建的映射缓存也需要删除.释放用户空间的缓存就需要使用到shmdt函数   
   
   
shmctl 释放内核中的共享缓存   
shmdt 释放用户空间中的共享缓存   
   
shmdt删除用户空间中的地址映射   
函数分析   
int shmdt(const void \*shmaddr);   
参数:   
shmaddr共享内存映射后的地址   
返回值:   
成功:0   
出错:-1   
   
   
测试shmdt函数   
   
#include "stdio.h"   
#include "sys/types.h"   
#include "sys/shm.h"   
#include "signal.h"   
#include "stdlib.h"   
#include "unistd.h"   
#include "string.h"   
   
int main(){   
   
 int shmid;   
 int key;   
 char \*p;   
 key = ftok("./a.c",'a');   
 if(key<0){   
 printf("create key failure\n");   
 return -2;   
 }   
 printf("create key success key = %X\n",key);   
 shmid = shmget(key,128,IPC\_CREAT|0777);   
 if(shmid<0){   
 printf("create share memory failure\n");   
 return -1;   
 }   
 printf("create share memory success shmid=%d\n",shmid);   
 system("ipcs -m");   
   
 // create a mapping for the share memory in userspace   
 p=(char \*)shmat(shmid,NULL,0);   
 if(p==NULL){   
 printf("shmat function failure\n");   
 return -3;   
 }   
 // write share memory stdin datasource from => Keyboard input   
 fgets(p,128,stdin);   
   
 //start read share memory   
 printf("first read share memory data:%s",p);   
   
 //second read share memory   
 printf("second read share memory data:%s",p);   
   
 shmdt(p);   
 // shared memory in userspace has been deleted   
   
 memcpy(p,"abcd",4);   
   
 return 0;   
}   
   
运行结果   
hello linux // 手动输入   
read share memory data:hello linux   
second read share memory data:hello linux   
段错误 (核心已转储) //报错，使用不存在的内存操作，会出现段错误

shmctl 释放内核中的共享缓存  
函数原型：  
int shmctrl(int shmid,int cmd,struct shmid\_ds \*buf);  
函数参数：  
 shmid 要操作的共享内存标识符  
 cmd:  
 IPC\_STAT (获取内核空间对象属性，将属性读到shmid\_ds 结构体中去) ---实现了命令ipcs-m  
 IPC\_SET(设置内核空间对象属性，修改，使用第三个参数传入)  
 IPC\_RMID(删除对象，第三个参数设定为NUULL) 实现了命令ipcrm -m  
 buf:指定IPC\_STAT/IPC\_SET时用于保存/设置属性  
函数返回值  
 成功：0   
 失败： -1  
   
   
测试代码：  
shmctl\_1.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
int main(){  
 int shmid;  
 int key;  
 char \*p;  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 printf("create key success key = %X\n",key);  
 shmid = shmget(key,128,IPC\_CREAT|0777);  
 if(shmid<0){  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n",shmid);  
 system("ipcs -m");  
   
 // create a mapping for the share memory in userspace  
 p=(char \*)shmat(shmid,NULL,0);  
 if(p==NULL){  
 printf("shmat function failure\n");  
 return -3;  
 }  
   
 // write share memory stdin datasource from => Keyboard input  
 fgets(p,128,stdin);  
  
 //start read share memory  
 printf("first read share memory data:%s",p);  
  
 //second read share memory  
 printf("second read share memory data:%s",p);  
   
 shmdt(p);  
 // shared memory in userspace has been deleted   
 //memcpy(p,"abcd",4);  
 shmctl(shmid,IPC\_RMID,NULL);  
 system("ipcs -m");  
 return 0;  
}

自定义 ipcrm -m 函数  
注意：C 库函数 int atoi(const char \*str) 把参数 str 所指向的字符串转换为一个整数（类型为 int 型）。  
  
函数 myipcsrm.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
int main(int argc,char \*argv[]){  
 int shmid;  
  
 if(argc < 3){  
 printf("please input param\n");  
 return -1;  
 }  
  
 if(strcmp(argv[1],"-m") == 0){  
 printf("delete share memory\n");  
 }  
 else{  
 return -2;  
 }  
  
 shmid=atoi(argv[2]);  
 printf("shmid = %d\n",shmid);  
 shmctl(shmid,IPC\_RMID,NULL);  
 system("ipcs -m");  
  
 return 0;  
}  
  
命令行运行： ./myipcsrm -m 20   
  
再使用ipcs -m 即可查看到 对应shmid的课程已经被停止清除了。  
  
  
同理 也可以实现   
自定义 ipcs -m函数 ===》 shmctl +s IPC\_STAT函数的使用

###### 5 pause函数

1.函数原型：  
int pause(void);  
2.功能解释：  
暂停进程，把当前进程置成就绪态，让出CPU，直到收到任意一个信号后终止，并且当处理完该信号之后，直接执行pause()函数下面的语句  
3.返回值：  
只返回-1  
4.参数解释：  
不需要传参，如：pause();printf("pause over\n");//当前进程暂停，直到系统任意发出一个信号，pause()才被终止（打断），开始继续执行printf()。

###### 6 综合实例1

回顾：  
1 shmget 内核中开辟一段共享内存  
2 shmat 用户空间中映射一段共享内存  
3 shmdt 删除用户空间中的共享内存  
4 shmctl 功能之一：删除内核中的共享内存  
5 ftok 提供key给shmget函数，有两种方式 IPC\_PRIVATE 也可以自定义key的模式  
  
  
  
实例1   
代码思路  
基于共享内存，实现父进程不停的写，子进程不停的读。单向通信。  
  
注意：  
1 fork函数需要在shmget函数之后，保证是对同一个内核对象进行的操作。  
2 在fork函数新建了一样的父子进程之后，然后整体的代码运行的时候，不确定是具体哪一个进程先运行，  
如果是子进程先运行，子进程的signal函数不会运行，但会进入pause函数状态，然后父进程就会开始运行。  
如果是父进程先运行，则会创建映射内存，并使用kill函数给子进程发送信号，然后进入暂停状态。  
  
3 singal函数处理转换信号，这个函数的位置需要保证在pause前。  
  
  
  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
void myfun(int signum){  
 printf("this is myfun\n");  
 return;  
}  
int main() {  
 int shmid;  
 int key;  
 char \*p;  
 int pid;  
 shmid = shmget(IPC\_PRIVATE, 128, IPC\_CREAT | 0777);  
 if (shmid < 0) {  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n", shmid);  
 // 创建一个和自己进程一样的进程，使用的内核共享内存才会是一样的  
 pid = fork();  
  
 if (pid > 0)  
 {  
 signal(SIGUSR2,myfun);// to deal with the signal sended by child  
 //创建一个关于公用 内核共享内存的 映射在自己的进程缓存中  
 p = (char \*) shmat(shmid, NULL, 0);  
 // 是否映射成功  
 if (p == NULL) {  
 printf("parent process:shmat function failure\n");  
 return -3;  
 }  
  
 // 准备好了需要的两个缓存之后 内核缓存和用户空间缓存  
 while (1) {  
 // write share memory  
 printf("parent process start write share memory:\n");  
 fgets(p, 128, stdin);  
 // 当代码运行在这个地方的时候，已经确认了自己是父进程  
 // 对于pid的值，在父进程内fork函数返回的是子进程pid  
 kill(pid, SIGUSR1);// tell child process read data  
 // 暂停等待子进程读完之后，发送信号了中断pause信号  
 pause();// wait for child process starting read  
 }  
 }  
 if (pid == 0) { // child process  
 // 处理 父进程发送的信号，拿到这个信号之后，将默认的终止进程的操作替代为自定义的函数，  
 // 避免函数被终止，自定义函数，只是简单的跳过，没有进行任何处理  
 // 如果是子进程先运行的话，就不会处理直接跳过  
 signal(SIGUSR1,myfun);  
 while (1) {  
 // 有任何信号来了都会停止暂停  
 pause();// after parent process write completely,it will kill(pid, SIGUSR1) and child pause status will change  
 // map the same share memory  
 p = (char \*) shmat(shmid, NULL, 0);  
 if(p == NULL){  
 printf("child process shmat function failure\n");  
 return -3;  
 }  
 // start read share memory  
 printf("share memory data:%s", p);  
 // child change parent pause status  
 kill(getppid(), SIGUSR2);  
 }  
 }  
  
 shmdt(p);  
 shmctl(shmid, IPC\_RMID, NULL);  
 system("ipcs -m");  
 return 0;  
}  
  
  
运行结果：  
parent process start write share memory:  
123  
child start read share memory data:123

###### 7 综合实例2

实现无亲缘关系进程之间的双向通信  
ftok创建key 相同的key 来保证无亲缘关系两个进程之间对同一个共享内存的映射操作  
  
注意：在代码中使用了kill函数来发送信息，kill函数的参数中，需要指明发送目标进程的pid,所以父进程和子进程都需要提前把自己的pid写道共享内存中去，以便进程读取共享内存的时候能拿到发送方的pid,用于在完成相应的操作之后，使用kill函数发回信号给发送方。  
因此在代码中定义了一个 结构体  
struct mybuf{  
 int pid;// 携带发送方自己的pid给接收方  
 char buf[124];// 携带数据  
};  
  
server.c 用于发送代码到client  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
struct mybuf{  
 int pid;  
 char buf[124];  
};  
  
void myfun(int signum) {  
 return;  
}  
  
int main() {  
  
 int shmid;  
 int key;  
 struct mybuf \*p;  
 int pid;  
 key = ftok("./a.c", 'a');  
 if (key < 0) {  
 printf("create key failure\n");  
 return -2;  
 }  
 printf("create key success\n");  
 // parent and child should use the same share memory  
 shmid = shmget(key, 128, IPC\_CREAT | 0777);  
 if (shmid < 0) {  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n", shmid);  
 signal(SIGUSR1, myfun);// to change the signal sended by client  
 p = (struct mybuf \*) shmat(shmid, NULL, 0);  
 if (p == NULL) {  
 printf("server process:shmat function failure\n");  
 return -3;  
 }  
 // get client pid  
 p->pid=getpid();//write server pid to share memory  
  
 pause();// wait client read server pid;  
  
 // when pause changed ,get client pid  
 pid = p->pid;  
  
 //write  
 while (1) {  
 // write share memory  
 printf("server process start write share memory:\n");  
 fgets(p->buf, 128, stdin);  
 kill(pid, SIGUSR2);// tell client stop pause status and read data  
 pause();// wait client process read completely  
 }  
  
 shmdt(p);  
 shmctl(shmid, IPC\_RMID, NULL);  
 system("ipcs -m");  
 return 0;  
}  
  
  
  
client.c 用于接受server的代码  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/shm.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
struct mybuf{  
 int pid;  
 char buf[124];  
};  
  
void myfun(int signum) {  
 return;  
}  
  
int main() {  
  
 int shmid;  
 int key;  
 struct mybuf \*p;  
 int pid;  
 key = ftok("./a.c", 'a');  
 if (key < 0) {  
 printf("create key failure\n");  
 return -2;  
 }  
 printf("create key success\n");  
 // parent and child should use the same share memory  
 shmid = shmget(key, 128, IPC\_CREAT | 0777);  
 if (shmid < 0) {  
 printf("create share memory failure\n");  
 return -1;  
 }  
 printf("create share memory success shmid=%d\n", shmid);  
  
 signal(SIGUSR2, myfun);// to deal with the signal sended by server  
 p = (struct mybuf \*) shmat(shmid, NULL, 0);  
 if (p == NULL) {  
 printf("parent process:shmat function failure\n");  
 return -3;  
 }  
  
 // get server pid  
 pid = p->pid;  
 // write client pid  
 p->pid=getpid();  
 // change server paused status  
 kill(pid,SIGUSR1);  
  
 while (1) {  
 pause();// wait server write data to share memory;  
 printf("client process start receive share memory:%s\n",p->buf);//read data  
 kill(pid, SIGUSR1);// server may write share memory  
 }  
  
 shmdt(p);  
 shmctl(shmid, IPC\_RMID, NULL);  
 system("ipcs -m");  
 return 0;  
}

##### 七 消息队列

###### 1 概述

1 管道中的队列 ===》 顺序队列，顺序读写  
消息队列===》 链式队列，单独一个消息是结构体,因此信息可以相互不一样。  
msqid\_ds 内核维护消息队列的结构体，队列的第一个消息指针msg-first,最后一个消息指针为msg-last  
  
  
2 相关函数  
消息对列的创建或打开 msgget ==>open函数  
消息队列的关闭 msgctl ==> close函数  
消息队列的发送/写入 msgsnd ==> write函数 ==》 队列的插入操作  
消息队列的接受函数 msgrcv ==》 read 函数 ==》队列的删除操作？ 后期验证是否 删除之后还有数据（共享内存删除之后还存在，管道删除之后不存在）（答案是和管道一样）

###### 2 msgget函数

函数分析   
所需头文件  
#include<sys/types.h>  
#include<sys/ipc.h>  
#include<sys/msg.h>  
函数原型  
int msgget(key\_t key ,int flag);  
函数参数  
key 和消息队列关联的key值 IPC\_PRIVATE或者ftok  
flag 消息队列的访问权限  
函数返回值  
成功：消息队列的id  
出错： -1

测试代码msg\_1.c

#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
int main(){  
 int msgid;  
 msgid = msgget(IPC\_PRIVATE,0777);  
 if (msgid<0){  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n",msgid);  
 system("ipcs -q");  
 return 0;  
}  
  
代码结果  
 creat msg queue success msgid=1  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 0 syj 777 0 0   
0x00000000 1 syj 777 0 0

###### 2 msgctl函数

函数分析  
所需头文件  
#include<sys/types.h>  
#include<sys/ipc.h>  
#include<sys/msg.h>  
函数原型  
int msgctl(int msqid,int cmd,struct msqid\_ds \*buf);  
函数参数  
msqid 消息队列的id  
cmd :  
 IPC\_STAT:读取消息队列的属性，并将其保存在buf指向的缓冲区中。  
 IPC\_SET: 设置修改消息队列的属性，这个值取自buf参数  
 IPC\_RMID:从系统中删除消息队列  
buf:消息队列缓冲区  
函数返回值  
成功：0  
出错：-1

测试代码msg\_2.c

#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
int main(){  
 int msgid;  
 msgid = msgget(IPC\_PRIVATE,0777);  
 if (msgid<0){  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n",msgid);  
 system("ipcs -q");  
   
 // delete msg queue  
 msgctl(msgid,IPC\_RMID,NULL);  
 system("ipcs -q");  
 return 0;  
}  
  
测试结果  
 creat msg queue success msgid=2  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 2 syj 777 0 0   
  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息

###### 3 msgsnd函数

函数分析  
所需头文件  
#include<sys/types.h>  
#include<sys/ipc.h>  
#include<sys/msg.h>  
函数原型  
int msgsnd(int msqid,const void \*msgp,size\_t size,int flag);  
函数参数  
msqid 消息队列的id  
msgp 指向消息的指针。常用消息结构msgbuf如下：  
 struct msgbuf{  
 long mytype; //消息类型  
 char mtext[N];//消息正文  
 };  
size 发送消息正文的字节数  
flag :  
 IPC\_NOWAIT 消息没有发送完成 函数也会立即返回==》非阻塞式  
 0 知道发送完成 函数才返回 ==》 阻塞式  
函数返回值  
成功：0  
出错： -1

测试代码 msg\_3.c

#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 struct msgbuf sendbuf;  
 msgid = msgget(IPC\_PRIVATE, 0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
 //init sendbuf  
 sendbuf.type = 100;  
 printf("please input msg:\n");  
 fgets(sendbuf.voltage, 124, stdin);  
 //start write message to message queue  
 msgsnd(msgid, (void \*) &sendbuf, strlen(sendbuf.voltage), 0);  
 while (1);  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}  
  
测试结果  
写入  
 creat msg queue success msgid=3  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 3 syj 777 0 0   
  
please input msg:  
123  
新窗口  
ipcs -q 查看  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 3 syj 777 4 1

###### 4 msgrcv函数

函数分析  
所需头文件  
#include<sys/types.h>  
#include<sys/ipc.h>  
#include<sys/msg.h>  
函数原型  
int msgsnd(int msqid,const void \*msgp,size\_t size,long msgtype,int flag);  
函数参数  
msqid 消息队列的id  
msgp 接受消息的缓冲区  
size 要接受消息的字节数  
msgtype:  
 0 接受消息队列中的第一个消息  
 大于0： 接受消息队列中的第一个类型为msgtype的消息  
 小于0 接受消息队列中类型不大于msgtype的绝对值且类型值又最小的消息  
flag :  
 IPC\_NOWAIT 没有消息 函数也会返回==》非阻塞式  
 0若无消息函数会一直读 ==》 阻塞  
函数返回值  
成功：接受到消息的长度  
出错： -1

测试代码 msg\_4.c

#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 int recvlen;  
 struct msgbuf sendbuf,recvbuf;  
 msgid = msgget(IPC\_PRIVATE, 0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
 //init sendbuf  
 sendbuf.type = 100;  
 printf("please input msg:\n");  
 fgets(sendbuf.voltage, 124, stdin);  
 //start write message to message queue  
 msgsnd(msgid, (void \*) &sendbuf, strlen(sendbuf.voltage), 0);  
   
 // start read message from message queue  
 memset(recvbuf.voltage,0,124);  
 recvlen =msgrcv(msgid,(void \*)&recvbuf,124,100,0);  
 printf("recvbuf:%s\n",recvbuf.voltage);  
 printf("recvlen:%d\n",recvlen);  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}  
  
运行结果  
 creat msg queue success msgid=4  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 3 syj 777 4 1   
0x00000000 4 syj 777 0 0   
  
please input msg:  
1  
recvbuf:1  
  
recvlen:2 ===》 还有一个回车键 被统计了  
  
--------- 消息队列 -----------  
键 msqid 拥有者 权限 已用字节数 消息   
0x00000000 3 syj 777 4 1

可以自行测试读完之后，再次读入数据，可以发现数据已经不存在了，所以消息队列和管道中的顺序队列一样，数据读完之后，就不存在了。

###### 5 综合实战1

基于消息队列，实现两个无亲缘关系进程之间的单向通信。  
  
write.c  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 int recvlen;  
 struct msgbuf sendbuf,recvbuf;  
 int key;  
 // 实现在同一级目录创建a.c文件==> touch a.c 只需要创建a.c即可，空内容也行  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 msgid = msgget(key, IPC\_CREAT|0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
  
 sendbuf.type = 100;  
 //write msg queue  
 while (1){  
 memset(sendbuf.voltage,0,124);  
 printf("please input msg:\n");  
 fgets(sendbuf.voltage, 124, stdin);  
 //start write message to message queue  
 msgsnd(msgid, (void \*) &sendbuf, strlen(sendbuf.voltage), 0);  
 }  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}  
  
read.c  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 int recvlen;  
 struct msgbuf sendbuf,recvbuf;  
 int key;  
 // 实现在同一级目录创建a.c文件==> touch a.c 只需要创建a.c即可，空内容也行  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 msgid = msgget(key, IPC\_CREAT|0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
  
  
 //read msg queue  
 while (1){  
 memset(recvbuf.voltage,0,124);  
 //start read message from message queue  
 msgrcv(msgid, (void \*) &recvbuf,124,100, 0);  
 printf("receive data from msg queue:%s",recvbuf.voltage);  
 }  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}  
  
在不用的terminal中运行之后，wirte进程中写的数据，read会从消息队列中读出来。

###### 6 综合实例2

基于消息队列的无亲缘关系进程之间的双向通信（同时收发）  
注意 双向的消息类型需要定义为不同 a发给b的时候 type=100,b发给a的时候 type=200

测试代码

write\_2.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 int recvlen;  
 struct msgbuf sendbuf,recvbuf;  
 int key;  
 int pid;  
 // 实现在同一级目录创建a.c文件==> touch a.c 只需要创建a.c即可，空内容也行  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 msgid = msgget(key, IPC\_CREAT|0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
 pid = fork();  
  
 if(pid>0){// parent write type =100  
 sendbuf.type = 100;  
 //write msg queue  
 while (1){  
 memset(sendbuf.voltage,0,124);  
 printf("please input msg:\n");  
 fgets(sendbuf.voltage, 124, stdin);  
 //start write message to message queue  
 msgsnd(msgid, (void \*) &sendbuf, strlen(sendbuf.voltage), 0);  
 }  
 }  
 if(pid==0){// child process read type = 200  
 recvbuf.type = 200;  
 //read msg queue  
 while (1){  
 memset(recvbuf.voltage,0,124);  
 //start write message to message queue  
 msgrcv(msgid, (void \*) &recvbuf, 124,200, 0);  
 printf("receive msg from msg queue:%s",recvbuf.voltage);  
 }  
 }  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}

read\_2.c  
  
#include "stdio.h"  
#include "sys/types.h"  
#include "sys/msg.h"  
#include "signal.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "string.h"  
  
struct msgbuf {  
 long type;  
 char voltage[124];  
 char ID[4];  
};  
  
int main() {  
 int msgid;  
 int recvlen;  
 struct msgbuf sendbuf,recvbuf;  
 int key;  
 int pid;  
 // 实现在同一级目录创建a.c文件==> touch a.c 只需要创建a.c即可，空内容也行  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -2;  
 }  
 msgid = msgget(key, IPC\_CREAT|0777);  
 if (msgid < 0) {  
 printf(" creat msg queue failure\n");  
 return -1;  
 }  
 printf(" creat msg queue success msgid=%d\n", msgid);  
 system("ipcs -q");  
 pid = fork();  
  
 if(pid==0){// child write type =200  
 sendbuf.type = 200;  
 //write msg queue  
 while (1){  
 memset(sendbuf.voltage,0,124);  
 printf("please input msg:\n");  
 fgets(sendbuf.voltage, 124, stdin);  
 //start write message to message queue  
 msgsnd(msgid, (void \*) &sendbuf, strlen(sendbuf.voltage), 0);  
 }  
 }  
 if(pid>0){// parent process read type = 100  
 recvbuf.type = 100;  
 //read msg queue  
 while (1){  
 memset(recvbuf.voltage,0,124);  
 //start write message to message queue  
 msgrcv(msgid, (void \*) &recvbuf, 124,100, 0);  
 printf("receive msg from msg queue:%s",recvbuf.voltage);  
 }  
 }  
 // delete msg queue  
 msgctl(msgid, IPC\_RMID, NULL);  
 system("ipcs -q");  
 return 0;  
}

write*2 和 read*2之间可以互相双工通信。

##### 八 信号灯

###### 1.概述

1 IPC通信框架  
 共享内存  
 消息队列  
 信号灯  
2 信号灯和多线程中遇到的posix函数得到的信号量之间的区别  
 posix 相关都是单个信号量，信号灯则是信号量的集合，含有多个信号量  
  
3 相关函数  
信号灯的创建或打开 semget ==>open函数  
信号灯的关闭 semctl ==> close函数  
消息队列的发送/写入 msgsnd ==> write函数 ==》 队列的插入操作  
消息队列的接受函数 msgrcv ==》 read 函数 ==》队列的删除操作？ 后期验证是否 删除之后还有数据（共享内存删除之后还存在，管道删除之后不存在）（答案是和管道一样）  
4 信号灯集内部是以数组的形式进行维护的。====》数组

###### 2 semget函数

所需头文件  
#include "sys/types.h"  
#include "sys/ipc.h"  
#include "sys/sem.h"  
函数原型  
int semget(key\_t key,int nsems,int semflg);  
函数参数  
key:和信号灯集关联的key值  
nsems:信号灯集中包含的信号灯数目  
semflg:信号灯集的访问权限  
函数返回  
成功:信号灯集ID  
失败: -1

测试代码sem\_1.c

#include "sys/sem.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
int main(){  
 int semid;  
 semid = semget(IPC\_PRIAVTE,3,0777);  
 if (semid<0){  
 printf("create semaphore failure\n");  
 return -1;  
 }  
 printf("create semaphore success semid=%d\n",semid);  
 system("ipcs -s");  
 while (1);  
  
 system("ipcs -s");  
 return 1;  
}  
  
  
代码结果  
create semaphore success semid=1  
  
--------- 信号量数组 -----------  
键 semid 拥有者 权限 nsems   
0x00000000 1 syj 777 3

###### 3 semctl函数

所需头文件  
#include "sys/types.h"  
#include "sys/ipc.h"  
#include "sys/sem.h"  
函数原型  
int semctl(int semid,int semnum,int cmd,....union(不是地址));  
函数参数  
semid:信号灯集  
semnum:要修改的信号灯编号  
cmd:   
 GETVAL:获取信号灯的值  
 SETVAL: 设置信号灯的值  
 IPC\_RMID:从系统中删除信号灯集合  
....union:联合体 ===》 Linux 命令行使用 man semctl即可查看  
union semun {  
 int val; /\* Value for SETVAL \*/  
 struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/  
 unsigned short \*array; /\* Array for GETALL, SETALL \*/  
 struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO (Linux-specific) \*/  
};  
  
函数返回  
 成功: 0  
 失败: -1

测试代码sem\_2.c

#include "sys/sem.h"  
#include "sys/types.h"  
#include "signal.h"  
#include "unistd.h"  
#include "stdlib.h"  
#include "stdio.h"  
int main(){  
 int semid;  
 semid = semget(IPC\_PRIAVTE,3,0777);  
 if (semid<0){  
 printf("create semaphore failure\n");  
 return -1;  
 }  
 printf("create semaphore success semid=%d\n",semid);  
 system("ipcs -s");  
 while (1);  
  
 semctl(semi,0,IPC\_RMID,NULL);  
 system("ipcs -s");  
 return 1;  
}  
  
运行结果  
create semaphore success semid=3  
  
--------- 信号量数组 -----------  
键 semid 拥有者 权限 nsems   
0x00000000 0 syj 777 3   
0x00000000 1 syj 777 3   
0x00000000 2 syj 777 3   
0x00000000 3 syj 777 3   
  
  
--------- 信号量数组 -----------  
键 semid 拥有者 权限 nsems   
0x00000000 0 syj 777 3   
0x00000000 1 syj 777 3   
0x00000000 2 syj 777 3

###### 4 多线程信号量(posix)

测试多线程代码之间运行的随机性（没有实现通信，保证运行的顺序）  
  
thread\_1.c  
  
#include "stdio.h"  
#include "stdlib.h"  
#include "pthread.h"  
#include "unistd.h"  
void \*fun(void \*var){ //child thread code  
 int j;  
 //while (thread\_inter == 0);// child thread won't code until thread\_inter!=0  
 for (j = 0; j < 10; j++) {  
 usleep(100);  
 printf("this is fun j =%d\n",j);  
 }  
}  
int main(){ // main thread  
 int i;  
 char str[] = " hello linux\n";  
 pthread\_t tid;  
 int ret;  
 ret = pthread\_create(&tid,NULL,fun,(void \*)str);  
 if(ret<0){  
 printf(" create thread failure\n");  
 return -1;  
 }  
 for (i = 0; i < 10; i++) { // parent thread run first  
 usleep(100);  
 printf("this is main fun i=%d\n",i);  
 }  
 while (1);  
 return 0;  
}  
  
编译命令 gcc -o thread thread.c -lpthread  
运行命令 ./thread  
  
运行结果  
syj@syj-virtual-machine:~/my/sem$ ./thread   
this is main fun i=0  
this is fun j =0  
this is main fun i=1  
this is fun j =1  
this is fun j =2  
this is main fun i=2  
this is fun j =3  
this is main fun i=3  
this is main fun i=4  
this is fun j =4  
this is fun j =5  
this is main fun i=5  
this is main fun i=6  
this is fun j =6  
this is main fun i=7  
this is main fun i=8  
this is fun j =7  
this is main fun i=9  
this is fun j =8  
this is fun j =9

###### 5 posix函数

1、信号量--semaphore  
信号量是一个表示资源的实体，是一个与队列有关的整型变量，信号量只支持P操作和V操作。   
P(proberen)操作, P操作代表的是对资源的申请。  
V(verhogen)操作, V操作是释放资源的操作。  
2 该函数用于多线程，此处仅仅涉及使用多线程完成通信，未涉及到函数解析。  
3 代码逻辑（基于posix信号量原理）  
 定义信号量变量  
 初始化信号量变量  
 找到需要同步代码的地方，执行p操作和v操作

测试代码 thread\_2.c

#include "stdio.h"  
#include "stdlib.h"  
#include "pthread.h"  
#include "unistd.h"  
#include "semaphore.h"  
sem\_t sem; // 1 define  
void \*fun(void \*var){ //child thread code  
 int j;  
 sem\_wait(&sem);//sleep  
 for (j = 0; j < 10; j++) {  
 usleep(100);  
 printf("this is fun j =%d\n",j);  
 }  
}  
int main(){ // main thread  
 int i;  
 char str[] = " hello linux\n";  
 pthread\_t tid;  
 int ret;  
 sem\_init(&sem,0,0);// 2 init  
 ret = pthread\_create(&tid,NULL,fun,(void \*)str);  
 if(ret<0){  
 printf(" create thread failure\n");  
 return -1;  
 }  
 for (i = 0; i < 10; i++) { // parent thread run first  
 usleep(100);  
 printf("this is main fun i=%d\n",i);  
 }  
 sem\_post(&sem);  
 while (1);  
 return 0;  
}  
  
  
代码运行结果  
syj@syj-virtual-machine:~/my/sem$ gcc -o thread\_2 thread\_2.c -lpthread  
syj@syj-virtual-machine:~/my/sem$ ./thread\_2   
this is main fun i=0  
this is main fun i=1  
this is main fun i=2  
this is main fun i=3  
this is main fun i=4  
this is main fun i=5  
this is main fun i=6  
this is main fun i=7  
this is main fun i=8  
this is main fun i=9  
this is fun j =0  
this is fun j =1  
this is fun j =2  
this is fun j =3  
this is fun j =4  
this is fun j =5  
this is fun j =6  
this is fun j =7  
this is fun j =8  
this is fun j =9

###### 6 综合实战1

基于posix函数，先运行父进程，再运行子进程。  
进程管理--P V操作  
1 代码逻辑  
 基于IPC 信号灯集来完成进程之间的通信操作  
2 函数讲解===>均可以在linux 终端使用 man 函数名来查看  
semctl   
 联合体 union semun mysemun;  
semop   
 结构体 struct sembuf mysembuf;

代码测试 sem\_3.c

#include "stdio.h"  
#include "stdlib.h"  
#include "pthread.h"  
#include "unistd.h"  
#include "sys/ipc.h"  
#include "sys/sem.h"  
int semid;  
union semun {  
 int val; /\* Value for SETVAL \*/  
 struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/  
 unsigned short \*array; /\* Array for GETALL, SETALL \*/  
 struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO (Linux-specific) \*/  
};  
union semun mysemun;  
struct sembuf mysembuf;  
  
void \*fun(void \*var){ //child thread code  
 int j;  
 mysembuf.sem\_op = -1;  
 semop(semid,&mysembuf,1);  
 for (j = 0; j < 10; j++) {  
 usleep(100);  
 printf("this is fun j =%d\n",j);  
 }  
}  
int main(){ // main thread  
 int i;  
 char str[] = " hello linux\n";  
 int ret;  
 pthread\_t tid;  
 semid = semget(IPC\_PRIVATE,3,0777);  
 if(semid<0)  
 {  
 printf("create semaphore failure\n");  
 return -2;  
 }  
 printf("create semaphore success,semid = %d",semid);  
 system("ipcs -s");  
 mysemun.val = 0;  
 semctl(semid,0,SETVAL,mysemun); // sem init  
 mysembuf.sem\_num = 0;  
 mysembuf.sem\_flg = 0;  
 ret = pthread\_create(&tid,NULL,fun,(void \*)str);  
 if(ret<0){  
 printf(" create thread failure\n");  
 return -1;  
 }  
 for (i = 0; i < 10; i++) { // parent thread run first  
 usleep(100);  
 printf("this is main fun i=%d\n",i);  
 }  
 mysembuf.sem\_op = 1;  
 semop(semid,&mysembuf,1);  
 while (1);  
 return 0;  
}  
  
编译命令行   
gcc -o sem\_3 sem\_3.c -lpthread  
  
运行命令行  
./sem\_3  
  
代码运行结果  
syj@syj-virtual-machine:~/my/sem$ ./sem\_3  
  
--------- 信号量数组 -----------  
键 semid 拥有者 权限 nsems   
0x00000000 10 syj 777 3   
  
create semaphore success,semid = 10this is main fun i=0  
this is main fun i=1  
this is main fun i=2  
this is main fun i=3  
this is main fun i=4  
this is main fun i=5  
this is main fun i=6  
this is main fun i=7  
this is main fun i=8  
this is main fun i=9  
this is fun j =0  
this is fun j =1  
this is fun j =2  
this is fun j =3  
this is fun j =4  
this is fun j =5  
this is fun j =6  
this is fun j =7  
this is fun j =8  
this is fun j =9

###### 7 综合实战2

基于IPC信号灯集来完成进程之间的通信操作，实现两个无亲缘关系进程之间的通信。  
  
client先运行，初始化信号灯  
server运行打开信号灯，server打印内容  
client打印自己的内容。

测试代码 client.c

#include "stdio.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "sys/ipc.h"  
#include "sys/sem.h"  
int semid;  
union semun {  
 int val; /\* Value for SETVAL \*/  
 struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/  
 unsigned short \*array; /\* Array for GETALL, SETALL \*/  
 struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO (Linux-specific) \*/  
};  
union semun mysemun;  
struct sembuf mysembuf;  
int main(){ // main thread  
 int i;   
 int key;  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -1;  
 }  
 printf("create key success,key=%d\n",key);  
 semid = semget(key,3,IPC\_CREAT|0777);  
 if(semid<0)  
 {  
 printf("create semaphore failure\n");  
 return -2;  
 }  
 printf("create semaphore success,semid = %d\n",semid);  
 system("ipcs -s");  
 // init sem  
 mysemun.val = 0;  
 semctl(semid,0,SETVAL,mysemun); // sem init  
 mysembuf.sem\_num = 0;  
 mysembuf.sem\_flg = 0;  
 // p operation  
 mysembuf.sem\_op = -1;  
 semop(semid,&mysembuf,1);  
 for (i = 0; i < 10; i++) { //second  
 printf("this is client main fun i=%d\n",i);  
 }  
  
   
 while (1);  
 return 0;  
}

测试代码 server.c

#include "stdio.h"  
#include "stdlib.h"  
#include "unistd.h"  
#include "sys/ipc.h"  
#include "sys/sem.h"  
int semid;  
union semun {  
 int val; /\* Value for SETVAL \*/  
 struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/  
 unsigned short \*array; /\* Array for GETALL, SETALL \*/  
 struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO (Linux-specific) \*/  
};  
union semun mysemun;  
struct sembuf mysembuf;  
int main(){ // main thread  
 int i;   
 int key;  
 key = ftok("./a.c",'a');  
 if(key<0){  
 printf("create key failure\n");  
 return -1;  
 }  
 printf("create key success,key=%d\n",key);  
 semid = semget(key,3,IPC\_CREAT|0777);  
 if(semid<0)  
 {  
 printf("create semaphore failure\n");  
 return -2;  
 }  
 printf("create semaphore success,semid = %d\n",semid);  
 system("ipcs -s");  
 // init sem  
 // mysemun.val = 0;  
 // semctl(semid,0,SETVAL,mysemun); // sem init  
 mysembuf.sem\_num = 0;  
 mysembuf.sem\_flg = 0;  
 for (i = 0; i < 10; i++) {   
 printf("this is server main fun i=%d\n",i);  
 }  
 // v operation  
 mysembuf.sem\_op = 1;  
 semop(semid,&mysembuf,1);  
 while (1);  
 return 0;  
}