实验一、进程控制

一、实验目的

1、加深对进程的理解,进一步认识并发执行的实质;

2、分析进程争用资源现象,学习解决进程互斥的方法;

3、掌握Linux进程基本控制;

4、掌握Linux系统中的软中断和管道通信。

二、实验内容

编写程序，演示多进程并发执行和进程软中断、管道通信。

- 父进程使用系统调用pipe( )建立一个管道,然后使用系统调用fork()创建两个子进程，子进程1和子进程2；
- 子进程1每隔1秒通过管道向子进程2发送数据:

  I send you x times. （x初值为1，每次发送后做加一操作）

  子进程2从管道读出信息，并显示在屏幕上。
- 父进程用系统调用signal()捕捉来自键盘的中断信号（即按Ctrl+C键）；当捕捉到中断信号后，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 Process 1 is Killed by Parent!

  Child Process 2 is Killed by Parent!
- 父进程等待两个子进程终止后，释放管道并输出如下的信息后终止

  Parent Process is Killed!

三、预备知识

1、Linux文件编辑

- vi ：Linux古老的、功能强大的全屏幕编辑器
  - 启动方式：

    --$vi 文件名　　打开已有的文件或编辑新文件

    --$vi　　先编辑，之后命名存盘
  - Vi的三种模式：命令模式、输入模式和末行模式
- gedit：图形编辑器

2、编辑、编译、执行/调试

```
$vi
$cc  -o  test  -g test.c
$cc  -o sub1 sub1.c
$gdb
$./test
```

3、Linux进程管理命令——进程查看

- ps命令：报告进程标识、用户、CPU时间消耗及其他属性
  - 命令单独使用可以看到前台执行的进程；后台进程可以使用带参数的ps命令（如ps -ax）
  - 提供进程的一次性查看，结果不连续
  - 结果数据很精确，但数据量庞大
- top命令：显示CPU占用率为前几位的进程
  - 动态显示，输出结果连续
  - 消耗较多的系统资源
- pstree命令：列出当前的进程，以及它们的树状结构
  - 将当前的执行程序以树状结构显示，弥补ps命令的不足
  - 支持指定特定程序(PID)或使用者(USER)作为显示的起始

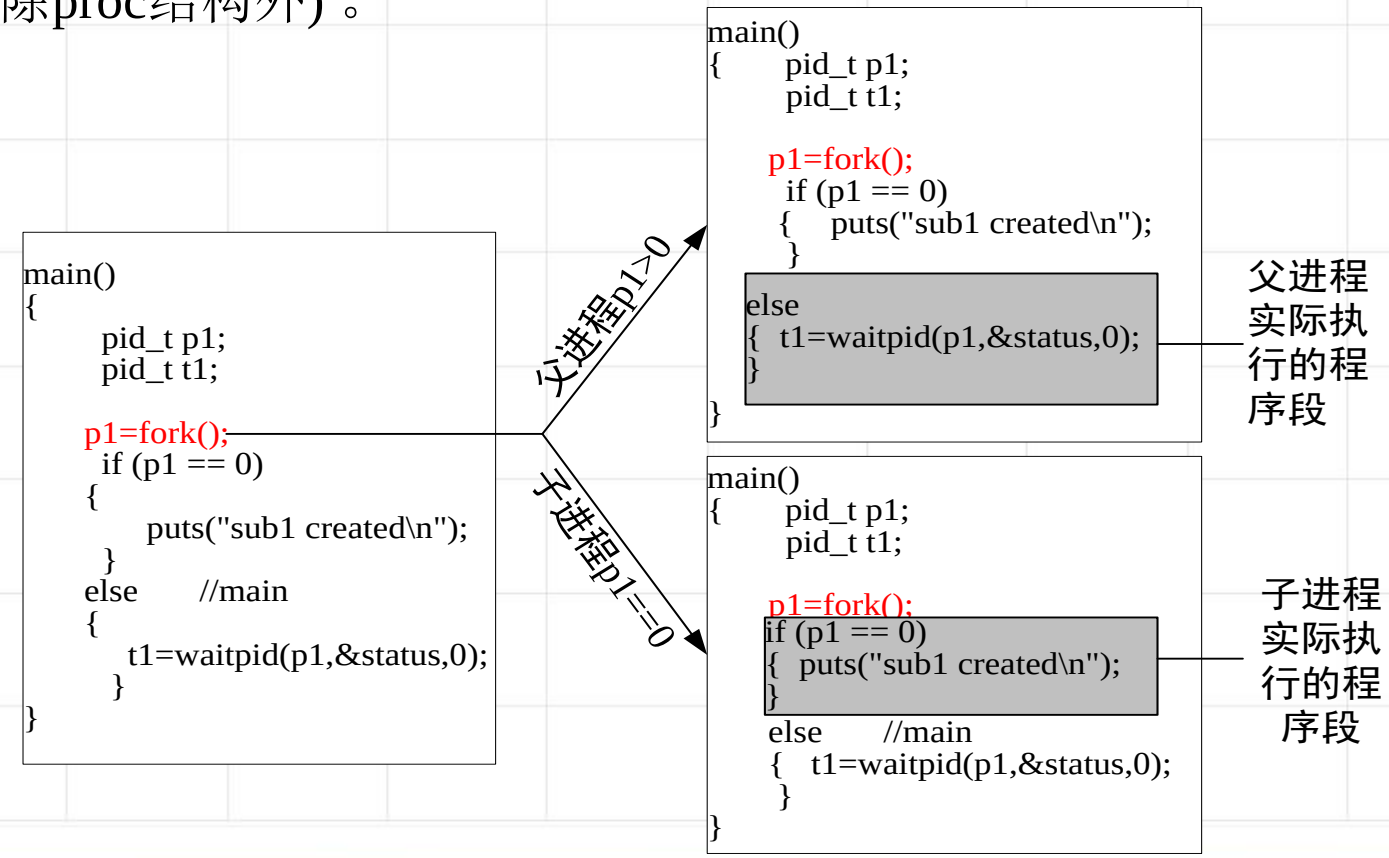
3、Linux进程管理命令—进程终止

- 终止一个进程或终止一个正在运行的程序
  - kill命令：根据PID向进程发送信号，缺省操作是停止进程
  - 如果进程启动了子进程，只终止父进程，子进程运行中将仍消耗资源成为“僵尸”进程，可用kill -9强制终止退出
  - pkill命令：终止同一进程组内的所有进程。允许指定要终止的进程名称，而非PID
  - Killall命令：与pkill应用方法类似，直接杀死运行中的程序
  - 数据库服务器的父进程不能用这些命令杀死（容易产生更多的文件碎片导致数据库崩溃）

4、Linux进程控制函数——进程创建

pid=fork();

创建一个子进程，被创建的子进程是父进程的进程映像的一个副本(除proc结构外)。



4、 Linux进程控制函数—修改进程

- 函数族exec() ：启动另外的进程取代当前的进程
  - #include <unistd.h>
  - int execl(const char *path, const char *arg, ...);
  - int execlp(const char *file, const char *arg, ...);
  - int execle(const char *path, const char *arg, const char *envp[]);
  - int execv(const char *path, const char *argv[]);
  - int execve(const char *path, const char *argv[], const char *envp[];
  - int execvp(const char *file, const char *argv[]);

实例：

```
pid_t  p1;
if ((p1=fork()) == 0) {
     execv("./get",NULL);
} else {  //main
     ……
}
```

4、 Linux进程控制函数—进程属性操作

- 设置进程属性
  - nice()：改变进程执行的优先级
  - setpgid()：将指定进程的组进程设为指定的组识别码
  - setpgrp()：将目前进程的组进程识别码设为目前进程的进程识别码，等价于setpgid(0,0)
  - setpriority()：设置进程、进程组和用户的执行优先权
- 获取进程属性
  - getpid()：获取目前进程的进程标识
  - getpgid()：获得参数pid指定进程所属的组识别码
  - getpgrp()：获得目前进程所属的组识别号，等价于getpgid(0)
  - getpriotity()：获得进程、进程组和用户的执行优先权

4、 Linux控制函数—进程退出

- 正常退出：在main()函数中执行return、调用exit()函数或_exit()函数
- 异常退出：调用abort()函数、进程收到信号而终止
- 区别
  - exit是一个函数，有参数，把控制权交给系统

    return是函数执行完后的返回，将控制权交给调用函数
  - exit是正常终止进程，abort是异常终止
  - exit中参数为0代表进程正常终止，为其他值表示程序执行过程中有错误发生
  - exit()在头文件stdlib.h中声明，先执行清除操作，再将控制权返回给内核

    _exit()在头文件unistd.h中声明，执行后立即返回给内核

4. Linux控制函数—等待进程终止

wait(); waitpid();

① wait() 语法格式： pid=wait(stat_addr);

wait()函数使父进程暂停执行，直到它的一个子进程结束为止，该函数的返回值是终止运行的子进程的PID。参数status所指向的变量存放子进程的退出码，即从子进程的main函数返回的值或子进程中exit()函数的参数。如果status不是一个空指针，状态信息将被写入它指向的变量。

② waitpid() 语法格式：waitpid(pid_t pid,int * status,int options)

用来等待子进程的结束，但它用于等待某个特定进程结束。

参数pid指明要等待的子进程的PID，参数status的含义与wait()函数中的status相同。

5. 进程的软中断通信

- 即信号机制，提供一种简单的处理异步事件的方法，在一个或多个进程之间传递异步信号

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

- 当某个信号出现时，系统有三种处理方式：
  - 忽略信号：大多数信号使用，但SIGKIL和SIGSTOP不能被忽略
  - 捕捉信号：通知内核在某种信号发生时，调用一个用户函数
  - 执行系统默认动作：异常终止(abort)、退出(exit)、忽略(ignore)、停止(stop)或继续(continue)

- 功能
  - 发送信号：发送进程把信号送到指定进程信号域的某一位上，如目标进程正在一个可被中断的优先级上睡眠，核心便将其唤醒
  - 预置对信号的处理方式：进程处于核心态时，即使受到软中断也不予理睬；只有当它返回到用户态后，才处理软中断信号
  - 收受信号的进程按事先规定完成对相应事件的处理

5. 进程的软中断通信——函数的使用

- 向一个进程或一组进程发送一个信号：**int kill(pid, sig)**

  pid>0时，核心将信号发送给进程pid

  pid<0时，核心将信号发送给与发送进程同组的所有进程

  pid=-1时，核心将信号发送给所有用户标识符真正等于发送进程的有效用户标识号的进程

- 预置信号接收后的处理方式：**signal(sig, function)**

  function=1时，屏蔽该类信号

  function=0时，收到sig信号后终止自己

  function为非0、非1类整数时，执行用户设置的软中断处理程序

```
include<stdio.h>
#include<stdlib.h>
#include<signal.h>
void my_func(int sig_no) {
    if(sig_no == SIGUSR1)
        printf("Receive
SIGUSR1.\n");
    if(sig_no == SIGUSR2)
        printf("Receive
SIGUSR2.\n");
    if(sig_no == SIGINT) {
        printf("Receive
        SIGINT.\n");
        exit(0);
    }
}
```

```
int main(void) {
   if(signal(SIGUSR1, my_func) ==
SIG_ERR)
            printf("can't catch
SIGUSR1.\n'");
   if(signal(SIGUSR2, my_func) ==
SIG_ERR)
            printf("can't catch
SIGUSR2.\n'");
   if(signal(SIGINT, my_func) ==
SIG_ERR)
            printf("can't catch
SIGINT.\n'");

   kill(getpid(),SIGINT);

   while(1);
   return 0;
}
```

6. Linux进程间通信—管道和有名管道

- 管道用于具有亲缘关系进程间的通信
  - 管道是半双工的，数据只能单向流动（双方通信需建立两个管道）
  - 管道只能用于父子进程或兄弟进程之间
  - 管道对于管道两端的进程而言就是一个文件，并单独构成一种文件系统，存在于内存中
  - 写管道的内容添加在管道缓冲区的末尾，读管道则从缓冲区头部读出
- 有名管道在普通管道具备功能基础上，通过给管道命名的方法变成管道文件，允许无亲缘关系进程间通过访问管道文件进行通信

6. 管道通信的使用—无名管道的使用

- int pipefd[2]; int pipe(pipefd); /*创建无名管道*/ pipefd[0]只能用于读; pipe[1]只能用于写
- 将数据写入管道：write()
  - 管道长度受到限制，管道满时写入操作将被阻塞，直到管道中的数据被读取
  - fcntl()可将管道设置为非阻塞模式
- 从管道读取数据：read()
  - 当数据被读取后，数据将自动被管道清除
  - 不能由一个进程向多个进程同时传递同一个数据
  - fcntl()可将管道读模式设置为非阻塞模式
- 关闭管道：close()
  - 关闭读端口时，在管道上进行写操作的进程将收到SIGPIPE信号
  - 关闭写端口时，进行读操作的read()函数将返回0

6. 管道通信的使用—命名管道的创建与读写

- 创建命名管道：

  int mknod(const char *path, mode_t mod, dev_t dev);

  int mkfifo(const char *path, mode_t mode);

- 命名管道必须先调用open()将其打开
  - 同时用读写方式(O_RDWR)打开时，一定不会导致阻塞
  - 以只读方式(O_RDONLY)打开时，调用open()函数的进程将会被阻塞直到有写方打开管道
  - 以写方式(O_WRONLY)打开时，阻塞直到有读方打开管道

四、实验指导

```
main( ) {
    创建无名管道;
    设置软中断信号SIGINT;
    创建子进程1、2;
    等待子进程1、2退出;
    关闭管道;
}
```

```
父进程信号处理 {
    发SIGUSR1给子进程1;
    发SIGUSR2给子进程2;
}
```

```
子进程1 {
    设置忽略信号SIGINT;
    设置信号SIGUSR1;
    while(1) {
        发送数据至管道数据;
        计数器++;
        睡眠1秒;
    }
}
```

```
子进程2 {
    设置忽略信号SIGINT;
    设置信号SIGUSR1;
    while(1) {
        接收管道数据;
        显示数据;
    }
}
```

```
SIGUSR1信号处理 {
    关闭管道;
    显示退出信息;
    退出;
}
```