实验 4 Linux 的进程管理

-补充管道、信号和 lockf 等相关内容

一、 Linux 下进程间通信概述

进程是一段程序的执行过程。这里所说的进程一般是指运行在**用户态（对应于内核态）**的进程，而由于处于用户态的不同进程之间是彼此隔离的，就像处于不同城市的人们，它们必须通过某种方式来进行通信，例如人们使用手机等方式相互通信。

- 用户态：user mode，非特权状态，每个进程都在各自的用户空间中运行，而不允许存取其他程序的用户空间。
- 用户空间：Linux系统将自身划分为两部分，一部分为核心软件，即kernel，也称作内核空间；另一部分为普通应用程序，称为用户空间。

就像人们有多种通信方式一样，进程之间的通信也有多种形式，目前在 Linux 中使用较多的进程间通信方式主要有以下几种：管道（pipe）、信号（Signal）、消息队列（Messge Queue）、共享内存（Shared memory）、信号量和套接字。

本节课我们主要介绍**管道和信号**的使用。

1) 管道（pipe）和有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信；有名管道，除具有管道所具有的功能外，还允许无亲缘关系进程间的通信。

- 管道（pipe）：在进程中产生，进程结束也随之结束，如“ls | grep abc”；
- 有名管道（named pipe）：通过mkfifo或mknod创建的管道文件。

2) 信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。

二、 管道和有名管道

1. 管道概述

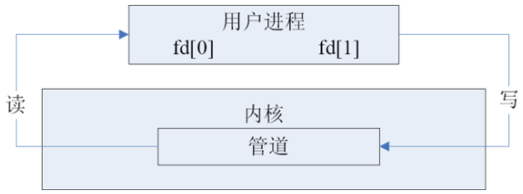
管道是Linux中一种很重要的通信方式，它是把一个程序的输出直接连接到另一个程序的输入。这里以**无名管道**为例，它具有如下特点：

- 管道可以看成是一种特殊的文件，对于它的读写也可以使用普通的read()和write()等函数。但是它不是普通的文件，不属于任何文件系统，只存在于内核的内存空间中；
- 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）；
- 它具有固定的读端和写端；
- 进程退出，管道也随之关闭。

2. 管道系统调用

1) 管道创建与关闭说明

管道是基于文件描述符的通信方式，当一个管道建立时，它会创建两个文件描述符fd[0]和fd[1]，其中fd[0]固定用于读管道，而fd[1]固定用于写管道，如下图所示，这样就构成了一个通道。



管道关闭时只需将这两个文件描述符关闭即可，可使用普通的close()函数逐个关闭各个文件描述符。

2) 管道创建函数

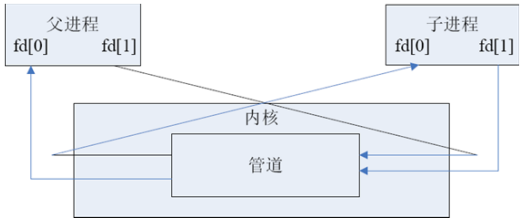
创建管道可以通过调用 pipe()来实现，表1列出了pipe()函数的语法要点。

表1 pipe()函数语法要点

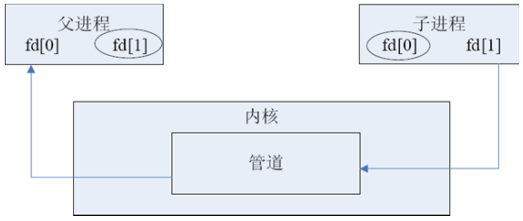
所需头文件	#include <unistd.h>
函数原型	int pipe(int fd[2])
函数传入值	fd[2]：管道的两个文件描述符，pipe之后就可以直接操作这两个文件描述符
返回值	成功：0
	出错：-1

3) 管道读写说明

用pipe()函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是创建一个管道，再通过fork()函数创建一个子进程，该子进程会继承父进程所创建的管道，这时，父子进程管道的文件描述符对应关系如下图所示：



此时的关系看似非常复杂，实际上却已经给不同进程之间的读写创造了很好的条件。父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。例如在下图中将父进程的写端 fd[1]和子进程的读端 fd[0]关闭。此时，父子进程之间就建立起了一条“子进程写入父进程读取”的通道。



同样，也可以关闭父进程的 fd[0]和子进程的 fd[1]，这样就可以建立一条“父进程写入子进程读取”的通道。另外，父进程还可以创建多个子进程，各个子进程都继承了相应的 fd[0]和 fd[1]，这时，只需要关闭相应端口就可以建立其各子进程之间的通道。

4) 管道使用实例

本例主要建立一条父进程写、子进程读的管道。首先创建管道，然后父进程使用fork()函数创建子进程，再关闭父进程的读描述符和子进程的写描述符，建立起它们之间的管道通

信。

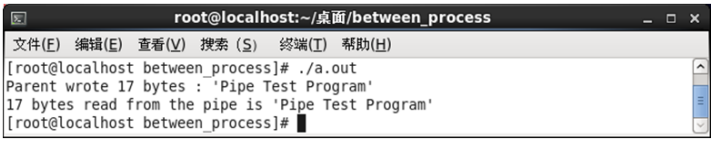
```
/* pipe.c */
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_DATA_LEN 256
#define DELAY_TIME 1
int main()
{
pid_t pid;
int pipe_fd[2];
char buf[MAX_DATA_LEN];
const char data[] = "Pipe Test Program";
int real_read, real_write;
memset((void*)buf, 0, sizeof(buf));
/* 创建管道 */
if (pipe(pipe_fd) < 0)
{
printf("pipe create error\n");
exit(1);
}
/* 创建一子进程 */
if ((pid = fork()) == 0)
{
/* 子进程关闭写描述符，并使子进程暂停1s等待父进程已关闭相应的读描述符 */
close(pipe_fd[1]);
sleep(DELAY_TIME);
/* 子进程读取管道内容 */
if ((real_read = read(pipe_fd[0], buf, MAX_DATA_LEN)) > 0)
{
printf("%d bytes read from the pipe is '%s'\n", real_read, buf);
}
/* 关闭子进程读描述符 */
close(pipe_fd[0]);
exit(0);
}
else if (pid > 0)
{
/* 父进程关闭读描述符，写入相应信息 */
close(pipe_fd[0]);
if((real_write = write(pipe_fd[1], data, strlen(data))) != -1)
{
```

批注 [U1]: 内存清零。
【语法】void *memset(void *s, int ch, size_t n);
函数解释：将中前 n 个字节用 ch 替换并返回 s。
本例表示把 buf 中所有的数据清零。

批注 [U2]: memset 的操作对象是内存，所以须将 buf 转换为指针类型。该操作可以省略。

```
printf("Parent wrote %d bytes : '%s'\n", real_write, data);
}
/*关闭父进程写描述符*/
close(pipe_fd[1]);
/*收集子进程退出信息*/
waitpid(pid, NULL, 0);
exit(0);
}
}
```

该程序的运行结果如下所示：



5) 管道读写注意点

- 只有在管道的读端存在时，向管道写入数据才有意义。
- 向管道写入数据时，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。
- 父子进程在运行时，它们的先后次序并不能保证，因此，在这里为了保证父子进程已经关闭了相应的文件描述符，可在两个进程中调用sleep()函数，当然这种调用不是很好的解决方法，在后面会学习进程之间的同步与互斥机制。

三、 lockf()函数

lockf()函数允许将文件区域用作信号量（监视锁），或用于控制对锁定进程的访问（强制模式记录锁定）。试图访问已锁定资源的其他进程将返回错误或进入休眠状态，直到资源解除锁定为止。当关闭文件时，将释放进程的所有锁定，即使进程仍然有打开的文件。当进程终止时，将释放进程保留的所有锁定。

表2 lockf()函数语法要点

所需头文件	#include <unistd.h>
函数原型	int lockf(int fd, int cmd, off_t len);
函数传入值	fd：打开的文件描述符
	cmd：是指定要采取的操作的控制值，允许的值在中定义 # define F_ULOCK 0 //解锁 # define F_LOCK 1 //互斥锁定区域
	len：要锁定或解锁的连续字节数。 ● 要锁定的资源从文件中当前偏移量开始，对于正len将向前扩展，对于负len则向后扩展（直到但不包括当前偏移量的前面的字节数）。 ● 如果 len 为零，则锁定从当前偏移量到文件结尾的区域（即从当前偏移量到现有或任何将来的文件结束标志）。
返回值	成功：0
	出错：-1

四、 软中断通信

1. 信号概述

信号是UNIX中所使用的进程通信的一种**最古老**的方法。它是在**软件层次上对中断机制的一种模拟**，是一种异步通信方式。信号可以直接进行**用户空间进程和内核进程之间的交互**，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。

信号可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它为止；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

kill命令的“–l”选项可以列出该系统所支持的所有信号的列表，如下图所示。

```
[root@centoscc cc]# 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
```

在Linux中，信号值在32 之前的有不同的名称，而信号值在 32 以后的都是用“ SIGRTMIN”或“ SIGRTMAX”开头的（RT表示Real Time），这就是两类典型的信号。前者是从 UNIX 系统中继承下来的信号，为不可靠信号（也称为非实时信号）；后者是为了解决前面“不可靠信号”的问题而进行了更改和扩充的信号，称为“可靠信号”（也称为实时信号）。那么为什么之前的信号不可靠呢？这里首先要介绍一下信号的生命周期。

一个完整的信号生命周期由4 个重要事件来刻画：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数。

一个不可靠信号的处理过程是这样的：如果发现该信号已经在进程中注册，那么就忽略该信号。因此，若前一个信号还未注销又产生了相同的信号就会产生信号丢失。而当可靠信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此信号就不会丢失。

用户进程对信号的响应可以有 3 种方式。

- 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即SIGKILL和SIGSTOP。
- 捕捉信号，定义信号处理函数，当信号发生时，执行相应的自定义处理函数。
- 执行缺省操作， Linux 对每种信号都规定了默认操作。

Linux 中的大多数信号是提供给内核的，表6列出了 Linux 中最为常见信号的含义及其默认操作。

信号名	含义	默认操作
SIGHUP	该信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制终端不再关联	终止
SIGINT	该信号在用户键入INTR字符（通常是Ctrl-C）时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程	终止
SIGQUIT	该信号和SIGINT 类似，但由QUIT字符（通常是Ctrl-\）来控制，类似于一个程序错误信号	终止

SIGILL	该信号在一个进程企图执行一条非法指令时（可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时）发出	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为0 等其他所有的算术错误	终止
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理或忽略	终止
SIGALRM	该信号当一个定时器到时的时候发出	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略	暂停进程
SIGTSTP	该信号用于暂停进程，用户键入TSTP 字符时（通常是Ctrl+Z）发出这个信号，可以被阻塞、处理或忽略	停止进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略
SIGCONT	恢复暂停的进程	恢复进程
SIGABORT	进程异常终止时发出	终止

2. 信号发送与捕捉

发送和捕捉信号的函数主要有 kill()、 raise()、 alarm()以及 pause()，下面就依次对其进行介绍。

1) kill()和raise()

① 函数说明。

kill()函数kill系统命令一样，可以发送信号给进程或进程组（kill系统命令只是kill()函数的一个用户接口）。需要注意的是，它不仅可以中止进程（实际上发出SIGKILL信号），也可以向进程发送其他信号。

与kill()函数所不同的是，raise()函数仅允许进程向自身发送信号。

② 函数格式。

表3列出了kill()函数的语法要点：

表3 kill()函数语法要点		
所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
	pid	正数：要发送信号的进程号
		0：信号被发送到所有和当前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
		<-1：信号发送给进程组号为-pid的每一个进程
	sig：信号	
返回值	成功：0	
	出错：-1	

表4列出了raise()函数的语法要点。

表4 raise()函数语法要点	
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
返回值	成功：0
	出错：-1

③ 函数实例。

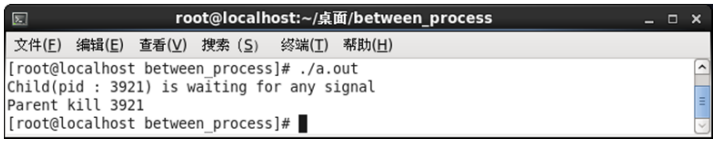
下面这个示例首先使用fork()创建了一个子进程，接着为了保证子进程不在父进程调用kill()之前退出，在子进程中使用raise()函数向自身发送SIGSTOP信号，使子进程暂停。接下来再在父进程中调用kill()向子进程发送信号，在该示例中使用的是SIGKILL，有兴趣的同学可以使用其他信号进行练习。

```
/* kill_raise.c */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main()
{
pid_t pid;
int ret;
 /* 创建一子进程 */
 if ((pid = fork()) < 0)
 {
printf("Fork error\n");
 exit(1);
 }
 if (pid == 0)
 {
 /* 在子进程中使用 raise() 函数发出 SIGSTOP 信号，使子进程暂停 */
printf("Child(pid : %d) is waiting for any signal\n", getpid());
 raise(SIGSTOP);
 exit(0);
 }
 else
 {
 /* 在父进程中收集子进程发出的信号，并调用 kill() 函数进行相应的操作 */
sleep(1);
 if ((waitpid(pid, NULL, WNOHANG) ) == 0)
 {
 if ((ret = kill(pid, SIGKILL) ) == 0)
 {
printf("Parent killed %d\n",pid);
 }
 }
 exit(0);
 }
}
```

批注 [U3]: 如果去掉该句，看看执行结果有什么不同。
可在程序运行时，在另一个终端运行 ps -u 命令查看结果。

批注 [U4]: 保证子进程能执行。否则因为父进程先执行，会直接执行 kill 函数，子进程将不执行。

该程序运行结果如下所示：



3. 信号的处理

在了解了信号的产生与捕获之后，接下来就要对信号进行具体的操作了。特定的信号是与一定的进程相联系的，也就是说，一个进程可以决定在该进程中需要对哪些信号进行什么样的处理。例如，一个进程可以选择忽略某些信号而只处理其他一些信号，另外，一个进程还可以选择如何处理信号。因此，首先就要建立**进程与其信号之间的对应关系，这就是信号的处理**。信号处理可以使用signal()函数。

1) 信号处理函数

① 函数说明。

使用signal()函数处理时，只需要指出要处理的信号和处理函数即可。它主要是用于前32 种非实时信号的处理，不支持信号传递信息，但是由于使用简单、易于理解，因此也受到很多程序员的欢迎。

② 函数格式。

signal()函数的语法要点如表5所示。

表5 signal()函数语法要点

所需头文件	#include <signal.h>	
函数原型	void (*signal(int signum, void (*func)(int)))(int)	
函数传入值	signum：指定信号代码，signal一旦获取到signum指定的信号，就开始执行func所指定的函数。	
	mode：	SIG_IGN：忽略该信号
		SIG_DFL：采用系统默认方式处理信号
		自定义的信号处理函数指针，此函数必须在 signal()被调用前申明，func 就是这个函数的名字。当接收到一个类型为 sig 的信号时，就执行 func 所指定的函数。这个函数应有如下形式的定义： void func(int sig);
返回值	成功：以前的信号处理配置	
	出错：-1	

这个函数原型“void (*signal(int signum, void (*func)(int)))(int)”很复杂，这里不占用太多时间说明，简单说明其使用方式：

- 最后一个“int” 是获取的信号，由进程自动获取；
- signal函数的第一个参数signum是一个整数，是指定的信号代码（在signal.h头文件中引用的“bits/signum.h”中定义，如信号SIGINT的代码是2），signal函数把由进程获取到的信号和signum对比，如果一致，再将该信号传递给signal的第二个参数，即函数指针func，这样func函数所使用的参数（倒数第二个“int”）其实就是由进程获取到的信号（最后一个“int”）；
- 以获取到的信号作为参数执行func函数；

③ 使用实例。

实例表明了如何使用signal()函数捕捉相应信号，并做出给定的处理。这里，my_func就是信号处理的函数指针。

```
/* signal.c */
```

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义信号处理函数*/
void my_func(int sign_no)
{
if (sign_no == SIGINT)
{
printf("I have get SIGINT %d\n",sign_no);
}
else if (sign_no == SIGQUIT)
{
printf("I have get SIGQUIT %d\n",sign_no);
}
}
int main()
{
printf("Waiting for signal SIGINT or SIGQUIT...\n");
/* 发出相应的信号，并跳转到信号处理函数处 */
signal(SIGINT, my_func);
signal(SIGQUIT, my_func);
signal(SIGTSTP,SIG_IGN);    /* 忽略SIGTSTP（ctrl+z）信号 */
pause();
exit(0);
}
```

批注 [U5]: 键入INTR字符（通常是Ctrl-C）时发出该信号。SIGINT对应的值是2（在signal.h中有定义），所以此处将“SIGINT”改为“2”，即语句改为“`if (sign_no == 2)`”，效果一样。

批注 [U6]: 键入 QUIT 字符（通常是Ctrl-\）时发出该信号. SIGQUIT 对应的值是 3，所以此处将“SIGQUIT”改为“3”,即语句改为“`if (sign_no == 3)`”，效果一样。

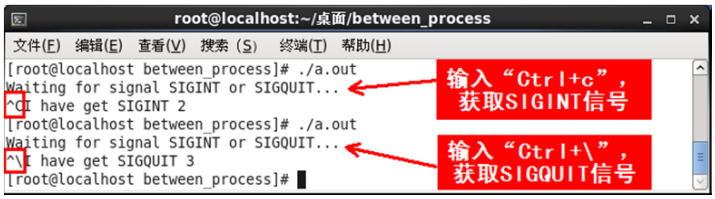
批注 [U7]: 指向 my_func 函数的入口地址

批注 [U8]: 等待获取信号，一旦获取到信号，将传给 signal 函数。

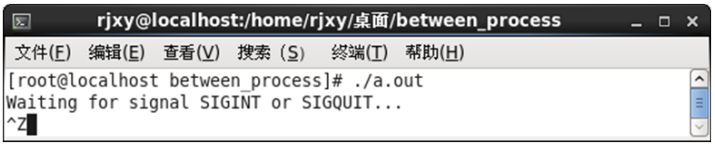
pause()获取一次信号后即退出，所以此处虽然可以输出 my_func 函数的结果，但最终还会退出程序；此处可以用 while(1)替换 pause()，则只输出 my_func 函数的结果，而不退出程序。

运行结果如下所示：

【测试1】在程序运行的终端里直接输入“Ctrl+c”或“Ctrl+\”：

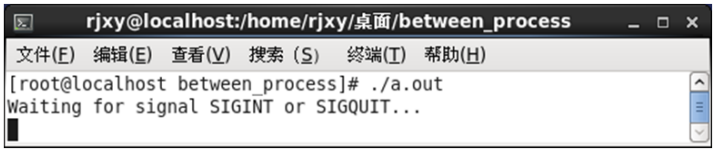


输入“Ctrl+z”后程序将忽略该信号。



【测试2】在另一个终端中使用kill命令给该进程相应的信号，命令格式“kill -signo PID”，其中signo是要发送的信号值，可通过“kill -l”查看，SIGINT对应的信号值是2，SIGQUIT对应的信号值是3；PID是程序“a.out”所对应的PID。

- 在第一个终端中运行程序



- 在第2个终端中查看程序的PID，并通过kill命令向其发送SIGINT信号

- 第1个终端中的程序接收到信号，做出相应的操作。