联航精英训练营

UNIGRESS ELITE TRAINING CAMP

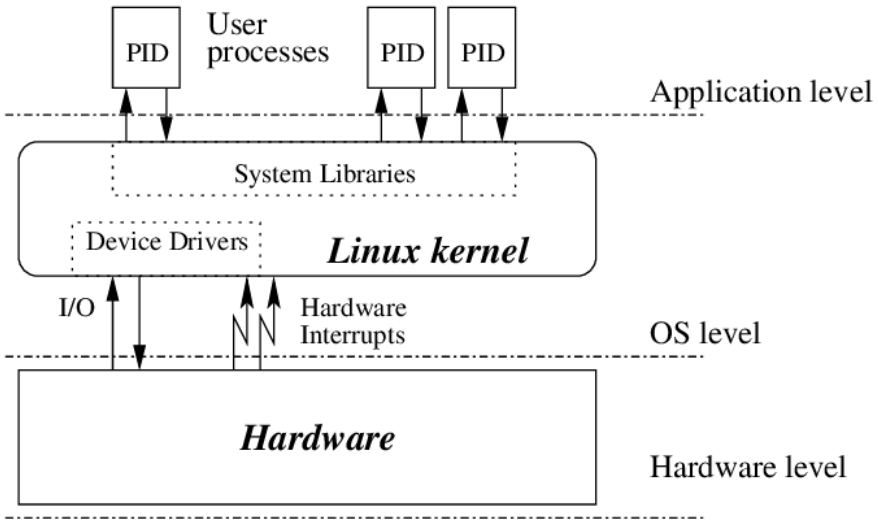
Linux高级编程（四）

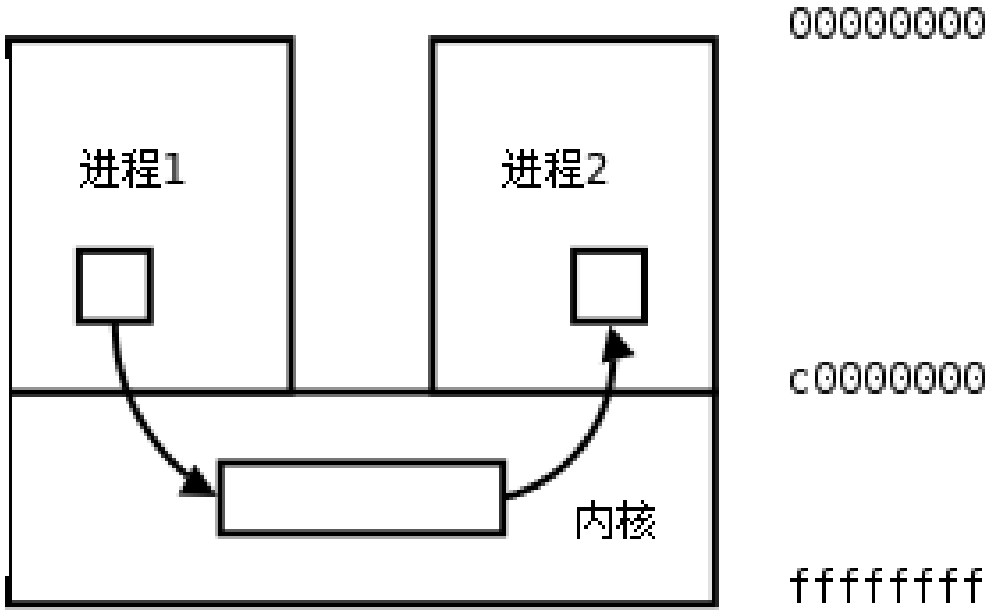
张勇涛

进程间通讯

进程间通信

- 每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种**机制**称为**进程间通信（IPC，InterProcess Communication）。**





进程间通讯的目的

- **数据传输**：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
- **共享数据**：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
- **通知事件**：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
- **资源共享**：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
- **进程控制**：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

linux进程间通信（IPC）的历史渊源

早期UNIX进程间通信、基于System V进程间通信、基于Socket进程间通信和POSIX进程间通信。

早期UNIX进程间通信方式包括：管道、FIFO、信号。

System V进程间通信方式包括：System V消息队列、System V信号灯、System V共享内存。

POSIX进程间通信包括：posix消息队列、posix信号灯、posix共享内存。

Linux下进程间通信概述

- 继承了UNIX平台上进程通信方式
- 两大主力
  - AT&T的贝尔实验室，对Unix早期的进程间通信手段进行了系统的改进和扩充，形成了"system V IPC"，其通信进程主要局限在单个计算机内
  - BSD(加州大学伯克利分校的伯克利软件发布中心)，跳过了该限制，形成了基于套接口(socket)的进程间通信机制
- Linux则把两者的优势都继承了下来

6

Linux下进程间通信概述

- 使用较多的进程间通信方式
  - **管道(Pipe)及有名管道(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，它还允许无亲缘关系进程间的通信
  - **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，用于通知接受进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求在效果上可以说是一样的
  - **消息队列**：消息队列是消息的链表，包括Posix消息队列systemV消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以向消息队列中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息

Linux下进程间通信概述

- 使用较多的进程间通信方式
  - **共享内存：**可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以即时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等
  - **信号量：**主要作为进程间以及同一进程不同线程之间的同步手段
  - **套接字(Socket)：**这是一种更为一般的进程间通信机制，它可用于不同机器之间的进程间通信，应用非常广泛

8

管道

管道通信

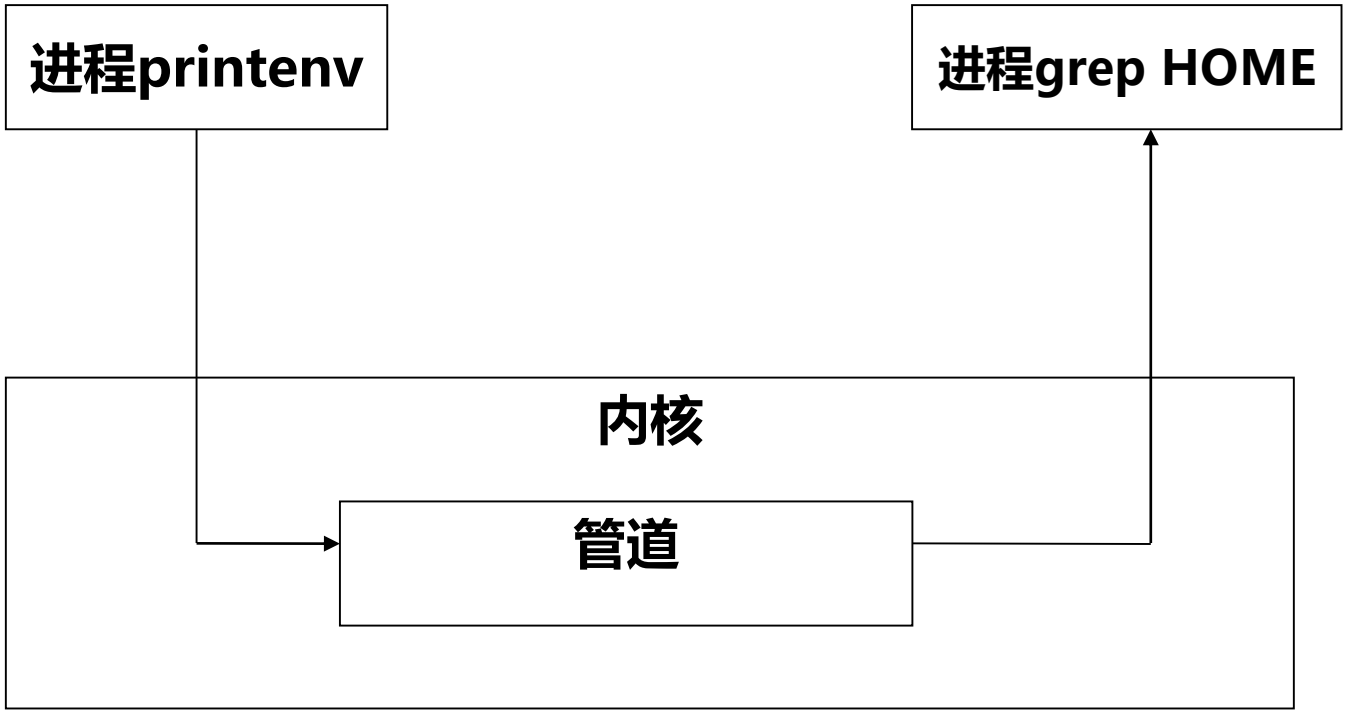
- 普通的Linux shell都允许重定向，而重定向使用的就是管道。例如：

```
ps | grep ps
```

- 管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的首端读出数据。数据读出后将从管道中移走，其它读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程将一直阻塞。同样，管道已经满时，进程再试图写管道，在其它进程从管道中移走数据之前，写进程将一直阻塞。

- 管道主要用于不同进程间通信。

管道通信



Linux下进程间通信概述

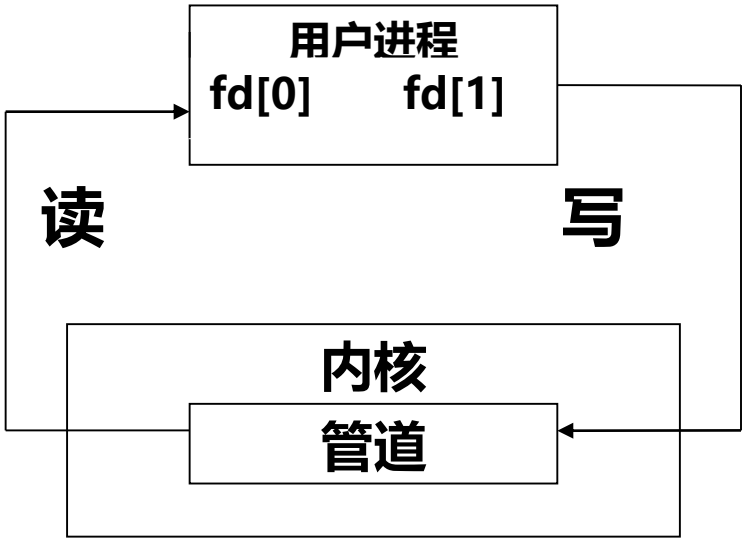
- 这里所说的管道主要指无名管道，它具有如下特点：
  - 只能用于**具有亲缘关系**的进程之间的通信
  - **半双工**的通信模式，具有固定的读端和写端
  - 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write等函数。但是它并不是普通的文件，不属于其他任何文件系统，并且只存在于内存中

12

管道创建与关闭

- 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fds[0]和fds[1]，其中fds[0]固定用于读管道，而fd[1]固定用于写管道

- 构成了一个半双工的通道。



管道创建与关闭

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，之后就可以直接操作这两个文件描述符
函数返回值	成功：0
	出错：-1

管道

```
#include <unistd.h>
int pipe(int filedes[2]);
```

pipe函数调用成功返回0，调用失败返回-1

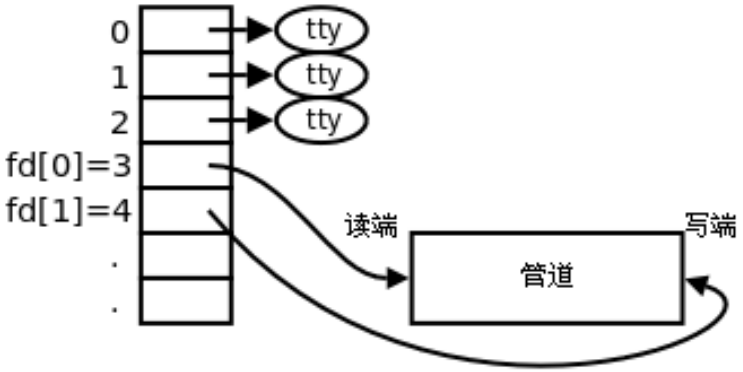
errno = EMFILE (没有空闲的文件描述符)

EMFILE (系统文件表已满)

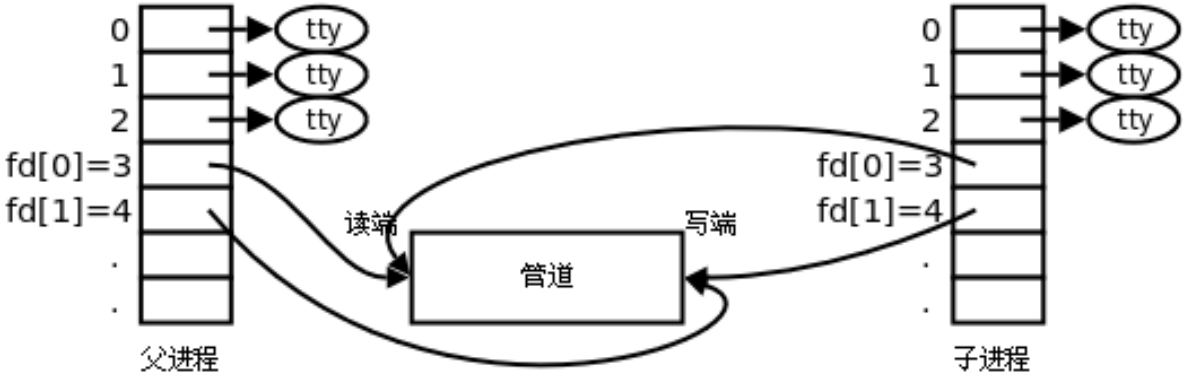
EFAULT (fd数组无效)

- 调用pipe函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个写端，然后通过filedes参数传出给用户程序两个文件描述符，filedes[0]指向管道的读端，filedes[1]指向管道的写端（很好记，就像0是标准输入1是标准输出一样）。所以管道在用户程序看起来就像一个打开的文件，通过read(filedes[0]);或者write(filedes[1]);向这个文件读写数据其实是在读写内核缓冲区。

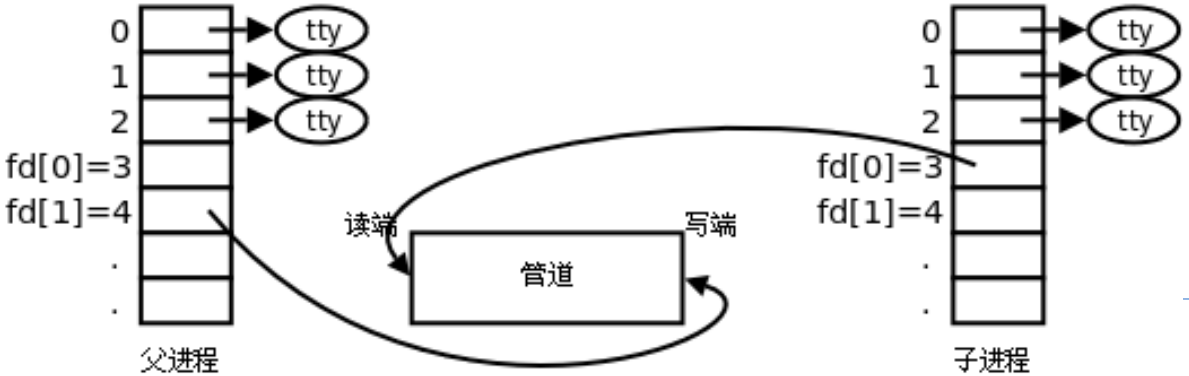
1. 父进程创建管道



2. 父进程*fork*出子进程



3. 父进程关闭*fd[0]*，子进程关闭*fd[1]*



管道调用过程

- 父进程调用pipe开辟管道，得到两个文件描述符指向管道的两端。
- 父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。
- 父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。

管道例子

- 见pipe.c

使用管道的限制

- 两个进程通过一个管道只能实现单向通信，比如上面的例子，父进程写子进程读，如果有时候也需要子进程写父进程读，就必须另开一个管道。

- 管道的读写端通过打开的文件描述符来传递，因此要通信的两个进程必须从它们的公共祖先那里继承管道文件描述符。

- 上面的例子是父进程把文件描述符传给子进程之后父子进程之间通信，也可以父进程fork两次，把文件描述符传给两个子进程，然后两个子进程之间通信，总之需要通过fork传递文件描述符使两个进程都能访问同一管道，它们才能通信。

使用管道的特殊情况

- 如果所有指向管道写端的文件描述符都关闭了（管道写端的引用计数等于0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。
- 如果有指向管道写端的文件描述符没关闭（管道写端的引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。
- 如果所有指向管道读端的文件描述符都关闭了（管道读端的引用计数等于0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。在信号会讲到怎样使SIGPIPE信号不终止进程。
- 如果有指向管道读端的文件描述符没关闭（管道读端的引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

标准流管道

popen

与linux中文件操作有文件流的标准I/O一样，管道的操作也支持基于文件流的模式。接口函数如下

库函数：popen();

原型： FILE *popen (char *command, char *type);

返回值：如果成功，返回一个新的文件流。如果无法创建进程或者管道，返回NULL。

管道中数据流的方向是由第二个参数type控制的。此参数可以是r或者w。w，r分别代表读或写。但不能同时为读和写。在Linux系统下，管道将会以参数type中第一个字符代表的方式打开。所以，如果你在参数type中写入rw，管道将会以读的方式打开

pclose

使用popen()创建的管道必须使用pclose( )关闭。其实，popen/pclose和标准文件输入/输出流中的fopen() / fclose()十分相似。

库函数： pclose();

原型： int pclose(FILE *stream);

返回值： 返回系统调用wait4( )的状态。

如果stream无效，或者系统调用wait4( )失败，则返回 -1。

- 注意此库函数等待管道进程运行结束，然后关闭文件流。

库函数pclose( )在使用popen( )创建的进程上执行wait4( )函数。当它返回时，它将破坏管道和文件系统。

例子

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUFSIZE 1024

int main()
{
        FILE *fp;
        char *cmd = "printenv";
        char buf[BUFSIZE];
        buf[BUFSIZE-1] = '\0';

        if((fp=popen(cmd,"r"))==NULL)
                perror("popen");

        while((fgets(buf,BUFSIZE,fp))!=NULL)
                printf("%s",buf);

        pclose(fp);

        exit(0);
}
```

命名管道（FIFO）

命名管道（FIFO）

- 无名管道只能用于具有亲缘关系的进程之间，这就大大限制了管道的使用
- 有名管道可以使互不相关的两个进程实现彼此通信。该管道可以通过路径名来指出，并且在文件系统中是可见的
- 两个进程就可以把FIFO当作普通文件一样进行读写操作，使用非常方便
- FIFO是严格遵循先进先出规则的，对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾
- 不支持如lseek()等文件定位操作

26

命名管道（FIFO）基本概念

命名管道和一般的管道基本相同，但也有一些显著的不同：

- 命名管道是在文件系统中作为一个特殊的设备文件而存在的。
- 不同祖先的进程之间可以通过管道共享数据。
- 当共享管道的进程执行完所有的I／O操作以后，命名管道将继续保存在文件系统中以便以后使用。

管道只能由相关进程使用，它们共同的祖先进程创建了管道。但是，通过FIFO，不相关的进程也能交换数据。

FIFO

所需头文件	#include <sys/types.h> #include <sys/state.h>	
函数原型	int mkfifo(const char *filename,mode_t mode);	
函数传入值	filename：要创建的管道。	
	mode：	O_RDONLY：读管道
		O_WRONLY：写管道
		O_RDWR：读写管道
		O_NONBLOCK：非阻塞
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三的参数为其设置权限。
		O_EXCL：如果使用O_CREAT时文件存在，则可返回错误消息。这一参数可测试文件是否存在。
函数返回值	成功：0	
	出错：-1	

一旦已经用mkfifo创建了一个FIFO，就可用open打开它。确实，一般的文件I / O函数（close、read、write、unlink等）都可用于FIFO。

O_NONBLOCK的影响

■ 当打开一个FIFO时，非阻塞标志（O_NONBLOCK）产生下列影响：

(1) 在一般情况中（没有说明O_NONBLOCK），只读打开要阻塞到某个其他进程为写打开此FIFO。类似，为写而打开一个FIFO要阻塞到某个其他进程为读而打开它。

(2) 如果指定了O_NONBLOCK，则只读打开立即返回。但是，如果没有进程已经为读而打开一个FIFO，那么只写打开将出错返回，其errno是ENXIO。

mode可以写作：O_CREAT|O_EXCL|0777

■ 类似于管道，若写一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE。若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志。

FIFO出错信息

EACCESS	参数filename所指定的目录路径无可执行的权限
EEXIST	参数filename所指定的文件已存在
ENAMETOOLONG	参数filename的路径名称太长
ENOENT	参数filename包含的目录不存在
ENOSPC	文件系统的剩余空间不足
EROFS	参数filename指定的文件存在于只读文件系统内

例子

- 见fifo_write.c fifo_read.c

信号通信

信号的基本概念

- 信号是软件中断。
- 信号提供了一种处理异步事件的方法
- 信号(signal)机制是Unix系统中最为古老的进程之间的通信机制。
- 它用于在一个或多个进程之间传递异步信号。

信号通信

- 信号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。它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；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

产生信号

- **通过终端按键产生信号**

  SIGINT(Ctrl-C)的默认处理动作是终止进程，SIGQUIT (Ctrl-\)的默认处理动作是终止进程并且Core Dump

- **调用系统函数向进程发信号**

  kill -SIGSEGV pid

  #include <signal.h>

  int kill(pid_t pid, int signo);

  int raise(int signo);

- **由软件条件产生信号**

  见alarm.c例子

信号的产生

- kill()和raise()
  - kill函数同读者熟知的kill系统命令一样，可以发送信号给进程或进程组(实际上，kill系统命令只是kill函数的一个用户接口)。这里要注意的是，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
  - kill –l查看系统支持的信号列表
  - raise函数允许进程向自身发送信号

信号的产生

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid,int sig);	
函数传入值	pid：	正数：要发送信号的进程号
		0：信号被发送到所有和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(除了进程号最大的进程外)
	sig：信号	
函数返回值	成功：0	
	出错：-1	

信号的产生

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
函数返回值	成功：0
	出错：-1

产生信号的条件

1. 用户在终端按下某些键时，终端驱动程序会发送信号给前台进程，例如Ctrl-C产生SIGINT信号，Ctrl-\产生SIGQUIT信号，Ctrl-Z产生SIGTSTP信号
2. 硬件异常产生信号，这些条件由硬件检测到并通知内核，然后内核向当前进程发送适当的信号。例如当前进程执行了除以0的指令，CPU的运算单元会产生异常，内核将这个异常解释为SIGFPE信号发送给进程。
3. 一个进程调用kill(2)函数可以发送信号给另一个进程。
4. 可以用kill(1)命令发送信号给某个进程，kill(1)命令也是调用kill(2)函数实现的，如果不明确指定信号则发送SIGTERM信号，该信号的默认处理动作是终止进程。
5. 当内核检测到某种软件条件发生时也可以通过信号通知进程，例如闹钟超时产生SIGALRM信号，向读端已关闭的管道写数据时产生SIGPIPE信号。

主要的信号来源

异常：进程运行过程中出现异常；
其它进程：一个进程可以向另一个或一组进程发送信号；
终端中断：Ctrl-C，Ctrl-\等；
作业控制：前台、后台进程的管理；
分配额：CPU超时或文件大小突破限制；
通知：通知进程某事件发生，如I/O就绪等；
报警：计时器到期。

信号列表

- kill -l命令可以察看系统定义的信号列表
- $ 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

...

名　字	说　明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止(abort)	•	•	•	•	终止w/core
SIGALRM	超时(alarm)		•	•	•	终止
SIGBUS	硬件故障			•	•	终止w/core
SIGCHLD	子进程状态改变		作业	•	•	忽略
SIGCONT	使暂停进程继续		作业	•	•	继续/忽略
SIGEMT	硬件故障			•	•	终止w/core
SIGFPE	算术异常	•	•	•	•	终止w/core
SIGHUP	连接断开		•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	终止w/core
SIGINFO	键盘状态请求				•	忽略
SIGINT	终端中断符	•	•	•	•	终止
SIGIO	异步I/O			•	•	终止/忽略
SIGIOT	硬件故障			•	•	终止w/core
SIGKILL	终止		•	•	•	终止
SIGPIPE	写至无读进程的管道		•	•	•	终止
SIGPOLL	可轮询事件(poll)			•		终止
SIGPROF	梗概时间超时(setitimer)			•	•	终止
SIGPWR	电源失效/再起动			•		忽略
SIGQUIT	终端退出符		•	•	•	终止w/core
SIGSEGV	无效存储访问	•	•	•	•	终止w/core
SIGSTOP	停止		作业	•	•	暂停进程

名　字	说　明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGSYS	无效系统调用			•	•	终止w/core
SIGTERM	终止	•	•	•	•	终止
SIGTRAP	硬件故障			•	•	终止w/core
SIGTSTP	终端挂起符		作业	•	•	停止进程
SIGTTIN	后台从控制tty读		作业	•	•	停止进程
SIGTTOU	后台向控制tty写		作业	•	•	停止进程
SIGURG	紧急情况			•	•	忽略
SIGUSR1	用户定义信号		•	•	•	终止
SIGUSR2	用户定义信号		•	•	•	终止
SIGVTALRM	虚拟时间闹钟（setitimer）			•	•	终止
SIGWINCH	终端窗口大小改变			•	•	忽略
SIGXCPU	超过CPU限制（setrlimit）			•	•	终止w/core
SIGXFSZ	超过文件长度限制（setrlimit）			•	•	终止w/core

Core Dump

- 当一个进程要异常终止时，可以选择把进程的用户空间内存数据全部保存到磁盘上，文件名通常是core，这叫做Core Dump。

- 进程异常终止通常是因为有Bug，比如非法内存访问导致段错误，事后可以用调试器检查core文件以查清错误原因，这叫做Post-mortem Debug。一个进程允许产生多大的core文件取决于进程的Resource Limit（这个信息保存在PCB中）。默认是不允许产生core文件的，因为core文件中可能包含用户密码等敏感信息，不安全。

- 在开发调试阶段可以用ulimit命令改变这个限制，允许产生core文件。

如何查看Core Dump

- 首先用ulimit命令改变Shell进程的Resource Limit，允许core文件最大为1024K：

- $ ulimit -c 1024
- ulimit -c unlimited
- 编译执行 -g
- ./a.out
- ulimit命令改变了Shell进程的Resource Limit，a.out进程的PCB由Shell进程复制而来，所以也具有和Shell进程相同的Resource Limit值，这样就可以产生Core Dump了

- 调式core文件

- gdb ./文件名 core

- (gdb)where

产生Core Dump的信号

名字	说明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止(abort)	.	.	.	.	终止w/core
SIGBUS	硬件故障		.	.	.	终止w/core
SIGEMT	硬件故障			.	.	终止w/core
SIGFPE	算术异常	.	.	.	.	终止w/core
SIGILL	非法硬件指令	.	.	.	.	终止w/core
SIGIOT	硬件故障			.	.	终止w/core
SIGQUIT	终端退出符		.	.	.	终止w/core
SIGSEGV	无效存储访问	.	.	.	.	终止w/core
SIGSYS	无效系统调用			.	.	终止w/core
SIGTRAP	硬件故障			.	.	终止w/core
SIGXCPU	超过CPU限制(setrlimit)			.	.	终止w/core
SIGXFSZ	超过文件长度限制(setrlimit)			.	.	终止w/core

信号的关联动作

可以要求系统在某个信号出现时按照下列三种方式中的一种进行操作:

- **忽略此信号。**

  SIGKILL和SIGSTOP永远不能被忽略

  忽略硬件异常,结果是未定义的

- **执行该信号的默认处理动作。**

  每个信号都有其默认动作,如SIGINT的默认动作是终止进程

- 提供一个信号处理函数，要求内核在处理该信号时切换到用户态执行这个处理函数，这种方式称为**捕捉**（Catch）一个信号。

标准C库函数出现在信号处理函数中是不安全的

5个缺省的动作

每一个信号都有一个缺省动作，它是当进程没有给这个信号指定处理程序时，内核对信号的处理。有5种缺省的动作：

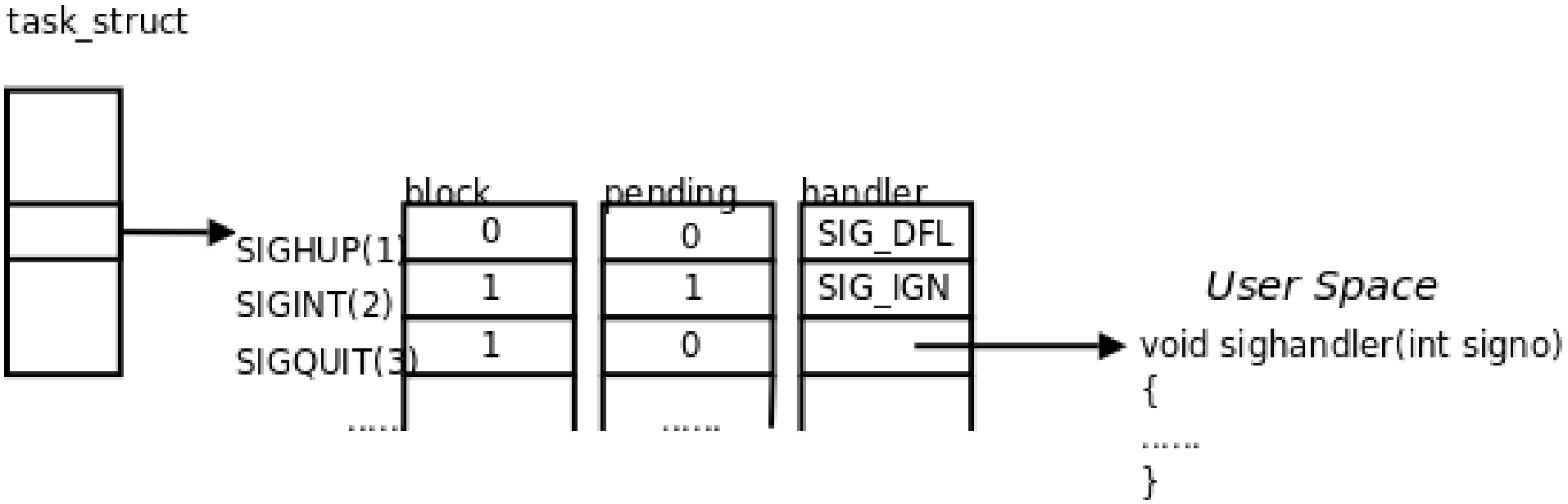
- **异常终止**（abort）：在进程的当前目录下，把进程的地址空间内容、寄存器内容保存到一个叫做core的文件中，而后终止进程。
- **退出**（exit）：不产生core文件，直接终止进程。
- **忽略**（ignore）：忽略该信号。
- **停止**（stop）：挂起该进程。
- **继续**（continue）：如果进程被挂起，则恢复进程的运行。否则，忽略信号。

信号在内核中的状态

- 实际执行信号的处理动作称为信号**递达**（Delivery），
- 信号从产生到递达之间的状态，称为信号**未决**（Pending）。
- 进程可以选择**阻塞**（Block）某个信号。被阻塞的信号产生时将保持在未决状态，直到进程解除对此信号的阻塞，才执行递达的动作。

- 注意: 阻塞和忽略是不同的，只要信号被阻塞就不会递达，而忽略是在递达之后可选的一种处理动作。

信号在内核中的表示



信号在内核中的表示

每个信号都有两个标志位分别表示阻塞和未决，还有一个函数指针表示处理动作。信号产生时，内核在进程控制块中设置该信号的未决标志，直到信号递达才清除该标志。

- SIGHUP信号未阻塞也未产生过，当它递达时执行默认处理动作。
- SIGINT信号产生过，但正在被阻塞，所以暂时不能递达。虽然它的处理动作是忽略，但在没有解除阻塞之前不能忽略这个信号，因为进程仍有机会改变处理动作之后再解除阻塞。
- SIGQUIT信号未产生过，一旦产生SIGQUIT信号将被阻塞，它的处理动作是用户自定义函数sighandler。
- 如果在进程解除对某信号的阻塞之前这种信号产生过多次，将如何处理？POSIX.1允许系统递送该信号一次或多次。

Linux是这样实现的：常规信号在递达之前产生多次只计一次，而实时信号在递达之前产生多次可以依次放在一个队列里。

在linux中共有64个信号.前32个为常规信号. 后32个为实时信号.实时信号与常规信号的唯一区别就是实时信号会排队等候.

信号在内核中的表示

- 每个信号只有一个bit的未决标志，非0即1，不记录该信号产生了多少次，阻塞标志也是这样表示的。

- 未决和阻塞标志可以用相同的数据类型sigset_t来存储，sigset_t称为**信号集**，这个类型可以表示每个信号的“有效”或“无效”状态，
- 在**阻塞信号集**中“有效”和“无效”的含义是该信号是否被阻塞
- 在**未决信号集**中“有效”和“无效”的含义是该信号是否处于未决状态。

- 阻塞信号集也叫做当前进程的**信号屏蔽字**（Signal Mask），这里的“屏蔽”应该理解为阻塞而不是忽略。

信号集操作函数

- **#include <signal.h>**

- **int sigemptyset(sigset_t *set);**

  函数sigemptyset初始化set所指向的信号集，使其中所有信号的对应bit清零，表示该信号集不包含任何有效信号。

- **int sigfillset(sigset_t *set);**

  函数sigfillset初始化set所指向的信号集，使其中所有信号的对应bit置位，表示该信号集的有效信号包括系统支持的所有信号。

注意：在使用sigset_t类型的变量之前，一定要调用sigemptyset或sigfillset做初始化，使信号集处于确定的状态。

信号集操作函数

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

初始化sigset_t变量之后就可以在调用sigaddset和sigdelset在该信号集中添加或删除某种有效信号。

这四个函数都是成功返回0，出错返回-1。

int sigismember(const sigset_t *set, int signo);

sigismember是一个布尔函数，用于判断一个信号集的有效信号中是否包含某种信号，若包含则返回1，不包含则返回0，出错返回-1。

sigprocmask

- 调用函数sigprocmask可以读取或更改进程的信号屏蔽字。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
sigset_t *oset);
```

返回值：若成功则为0，若出错则为-1

如果oset是非空指针，则读取进程的当前信号屏蔽字通过oset参数传出。

如果set是非空指针，则更改进程的信号屏蔽字，参数how指示如何更改。

如果oset和set都是非空指针，则先将原来的信号屏蔽字备份到oset里，然后根据set和how参数更改信号屏蔽字。

sigprocmask 中how参数的含义

- 假设当前的信号屏蔽字为mask，下表说明了how参数的可选值:

SIG_BLOCK	set包含了我们希望添加到当前信号屏蔽字的信号，相当于mask=mask|set
SIG_UNBLOCK	set包含了我们希望从当前信号屏蔽字中解除阻塞的信号，相当于mask=mask&~set
SIG_SETMASK	设置当前信号屏蔽字为set所指向的值，相当于mask=set

sigpending

```
#include <signal.h>
```

int sigpending(sigset_t *set);

sigpending读取当前进程的未决信号集，通过set参数传出。调用成功则返回0，出错则返回-1。

例子

- 见signalset.c

捕捉信号

- 如果信号的处理动作是用户自定义函数，在信号递达时就调用这个函数，这称为**捕捉信号**。

信号的处理

- 特定的信号是与一定的进程相联系的
- 一个进程可以决定在该进程中需要对哪些信号进程什么样的处理
- 信号处理的主要方法有两种
  - 使用简单的**signal**函数
  - 使用信号集函数组
- signal()
  - 使用signal函数处理时，只需把要处理的信号和处理函数列出即可
  - 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
  - 使用简单、易于理解，比较常用

信号的处理

所需头文件	#include <signal.h>	

函数原型	void (*signal(int signum, void (*handler)(int)))(int);
	typedef void (*sighandler_t)(int);
	sighandler_t signal(int signum, sighandler_t handler);	

函数传入值	signum：指定信号	
	handler:	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
函数返回值	成功：以前的信号处理配置	
	出错：-1	

信号的处理

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void my_func(int sign_no)
{
        if(sign_no==SIGINT)
                printf("I have get SIGINT\n");
        else if(sign_no==SIGQUIT)
                printf("I have get SIGQUIT\n");
}

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);
        pause();
        exit(0);
}
```

内核如何实现信号的捕捉

1. 用户程序**注册**了SIGQUIT信号的处理函数sighandler。

2. 当前正在执行main函数，这时发生中断或异常**切换到内核态**。

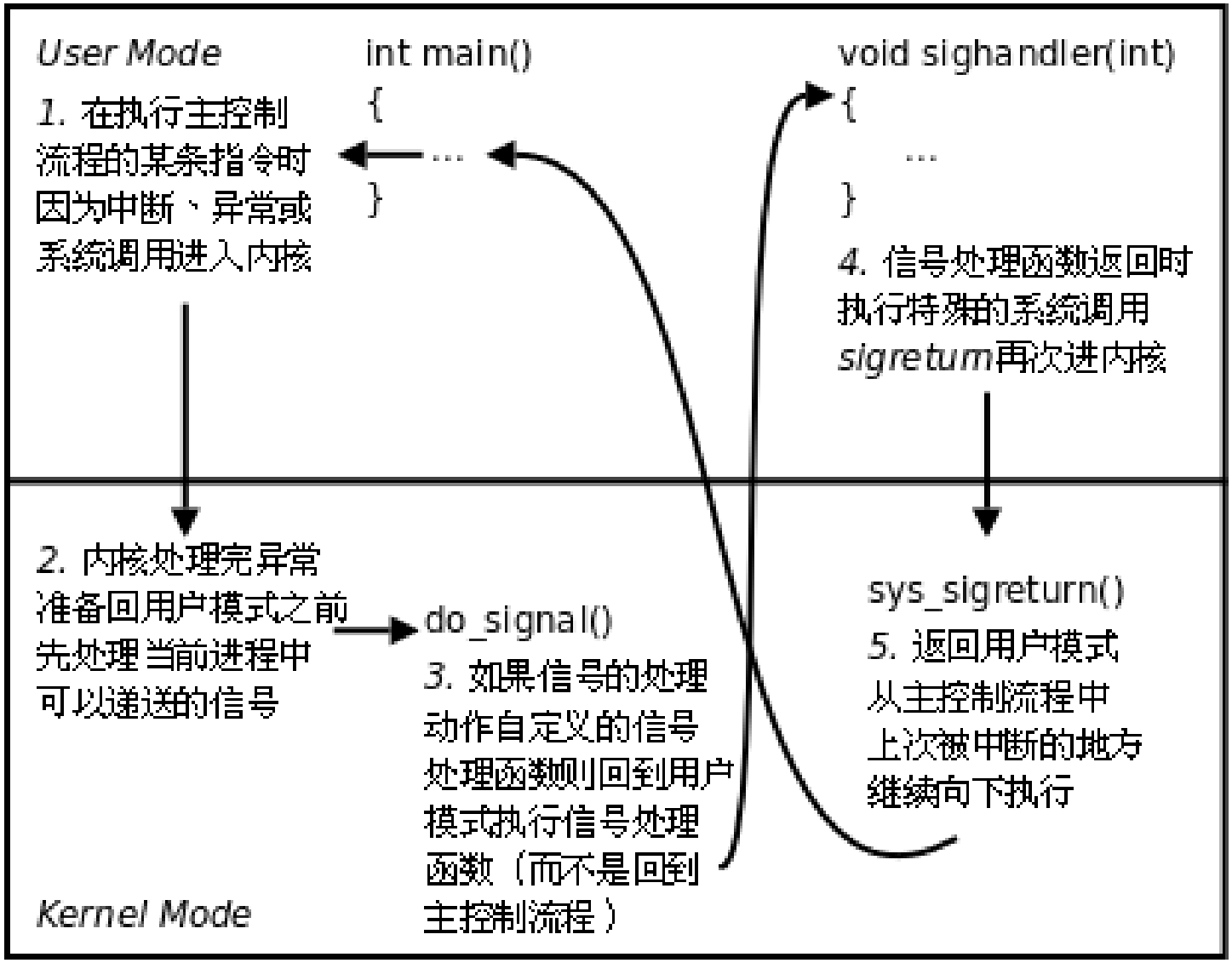
3. 在中断处理完毕后要**返回用户态的main函数**之前检查到有信号SIGQUIT递达。

4. 内核决定返回用户态后**不是恢复main函数的上下文继续执行，而是执行sighandler函数**，sighandler和main函数使用不同的堆栈空间，它们之间不存在调用和被调用的关系，是两个独立的控制流程。

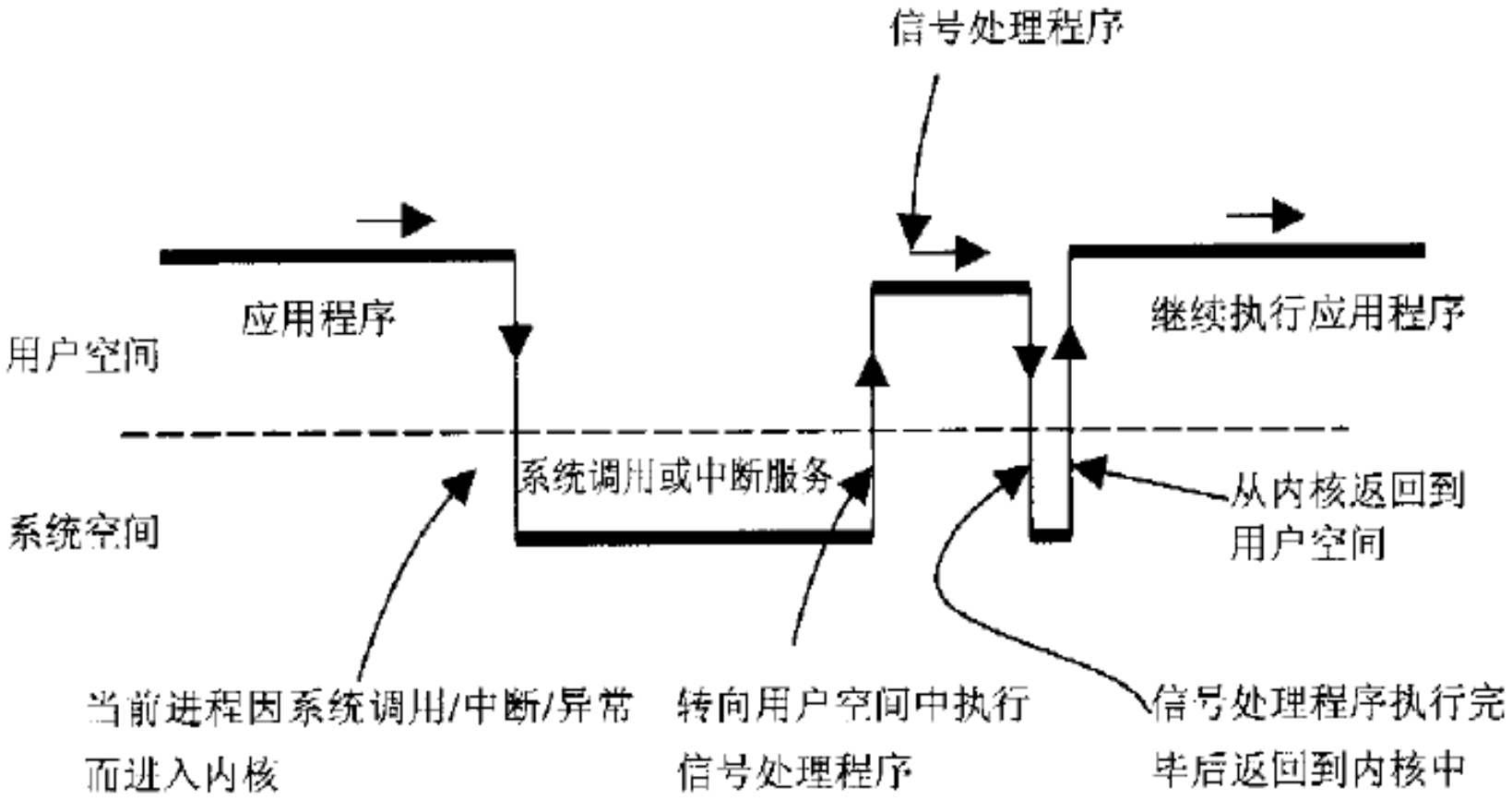
5. sighandler函数返回后自动执行特殊的系统调用sigreturn再次进入内核态。

6. 如果没有新的信号要递达，这次再返回用户态就是恢复main函数的上下文继续执行了。

信号的捕捉



信号检测与处理流程



信号的检测与处理流程图

sigaction

#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);

sigaction函数可以读取和修改与指定信号相关联的处理动作。调用成功则返回0，出错则返回-1。

signo是指定信号的编号。

若act指针非空，则根据act修改该信号的处理动作。

若oact指针非空，则通过oact传出该信号原来的处理动作。

sigaction

act和oact指向sigaction结构体：

```
 struct sigaction
{
     void      (*sa_handler)(int);   /* addr of signal handler, */
                                      /* or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask;                /* additional signals to block */
    int      sa_flags;               /* signal options */

    /* alternate handler */
    void     (*sa_sigaction)(int, siginfo_t *, void *);
   };
```

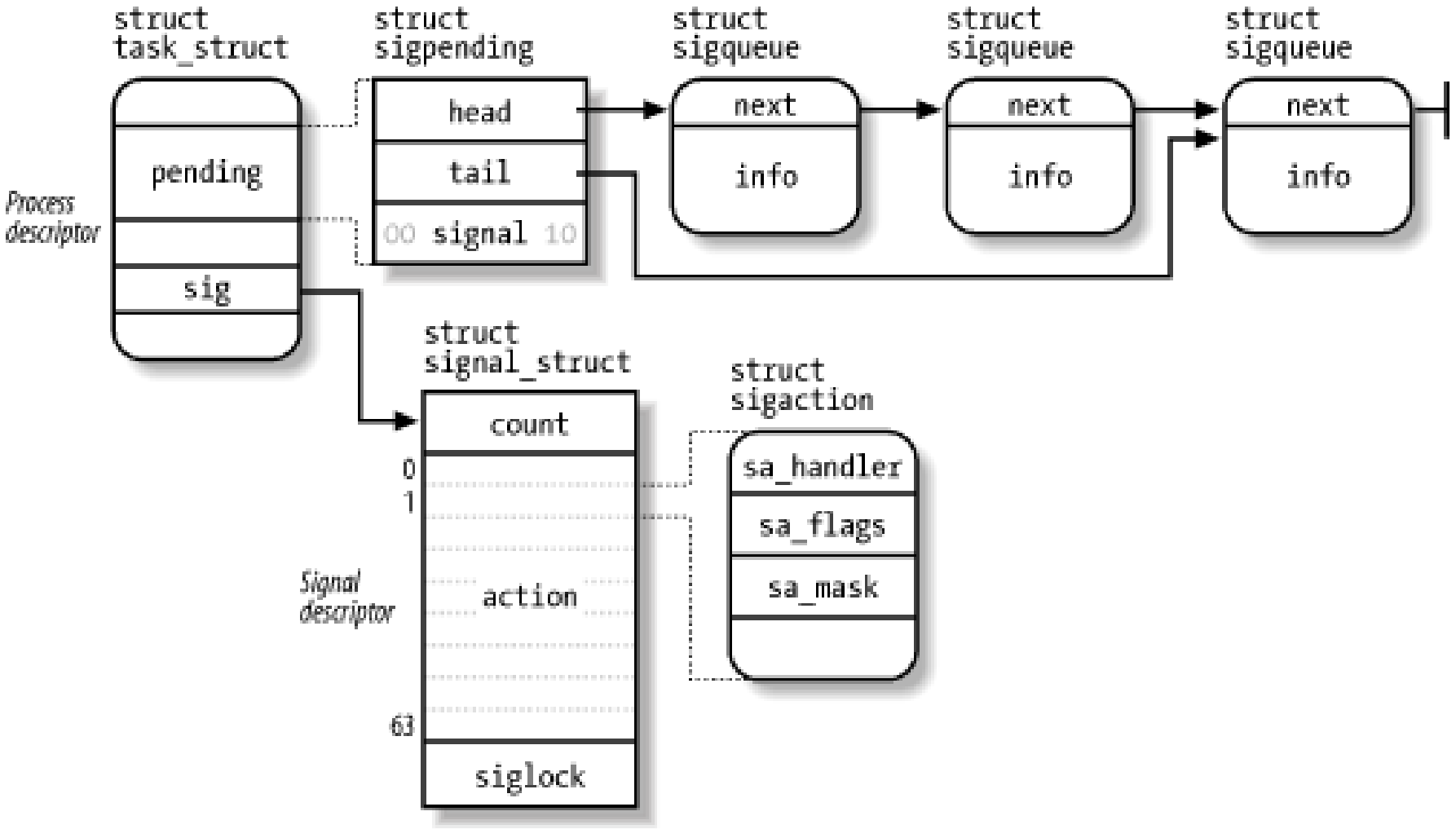
sigaction

- 将sa_handler赋值为常数SIG_IGN传给sigaction表示忽略信号，赋值为常数SIG_DFL表示执行系统默认动作，赋值为一个函数指针表示用自定义函数捕捉信号，或者说向内核注册了一个信号处理函数，该函数返回值为void，可以带一个int参数，通过参数可以得知当前信号的编号，这样就可以用同一个函数处理多种信号。显然，这也是一个回调函数，不是被main函数调用，而是被系统所调用。

- 当某个信号的处理函数被调用时，内核自动将当前信号加入进程的信号屏蔽字，当信号处理函数返回时自动恢复原来的信号屏蔽字，这样就保证了在处理某个信号时，如果这种信号再次产生，那么它会被阻塞到当前处理结束为止。如果在调用信号处理函数时，除了当前信号被自动屏蔽之外，还希望自动屏蔽另外一些信号，则用sa_mask字段说明这些需要额外屏蔽的信号，当信号处理函数返回时自动恢复原来的信号屏蔽字。

- sa_flags字段包含一些选项，这里的代码都把sa_flags设为0，sa_sigaction是实时信号的处理函数

和信号有关的数据结构



pause

- #include <unistd.h>

- **int pause(void);**

pause函数使调用进程挂起直到有信号递达。

如果信号的处理动作是终止进程，则进程终止，pause函数没有机会返回；

如果信号的处理动作是忽略，则进程继续处于挂起状态，pause不返回；

如果信号的处理动作是捕捉，则调用了信号处理函数之后pause返回-1，errno设置为EINTR，所以pause只有出错的返回值.

错误码EINTR表示“被信号中断”。

信号发送与捕捉

- **alarm()和pause()**
- alarm也称为闹钟函数，它可以在进程中设置一个定时器，当定时器指定的时间到时，它就向进程发送SIGALARM信号。要注意的是，一个进程只能有一个闹钟时间，如果在调用alarm之前已设置过闹钟时间，则任何以前的闹钟时间都被新值所代替
- pause函数是用于将调用进程挂起直到捕捉到信号为止。这个函数非常常用，通常可以用于判断信号是否已到

例子

- mysleep.c

1. main函数调用mysleep函数，后者调用sigaction注册了SIGALRM信号的处理函数sig_alrm。
2. 调用alarm(nsecs)设定闹钟。
3. 调用pause等待，内核切换到别的进程运行。
4. nsecs秒之后，闹钟超时，内核发SIGALRM给这个进程。
5. 从内核态返回这个进程的用户态之前处理未决信号，发现有SIGALRM信号，其处理函数是sig_alrm。
6. 切换到用户态执行sig_alrm函数，进入sig_alrm函数时SIGALRM信号被自动屏蔽，从sig_alrm函数返回时SIGALRM信号自动解除屏蔽。然后自动执行系统调用sigreturn再次进入内核，再返回用户态继续执行进程的主控制流程（main函数调用的mysleep函数）。
7. pause函数返回-1，然后调用alarm(0)取消闹钟，调用sigaction恢复SIGALRM信号以前的处理动作。

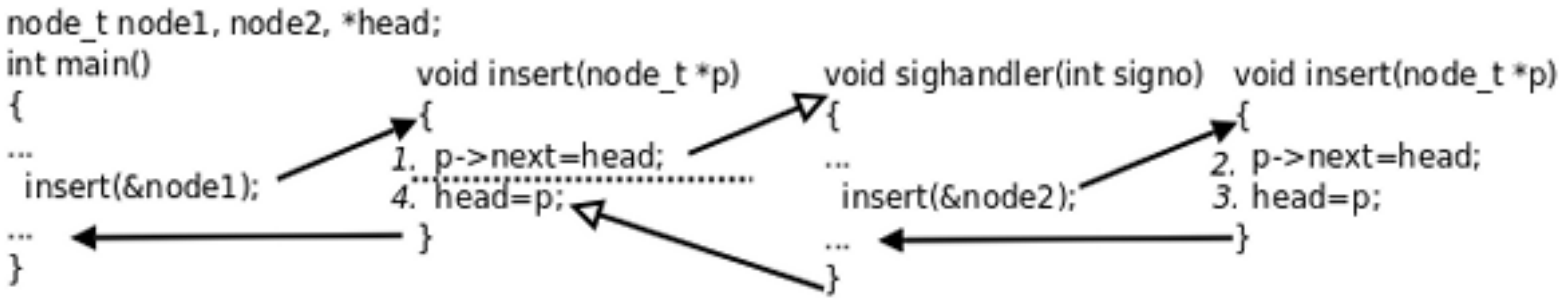
异步程序

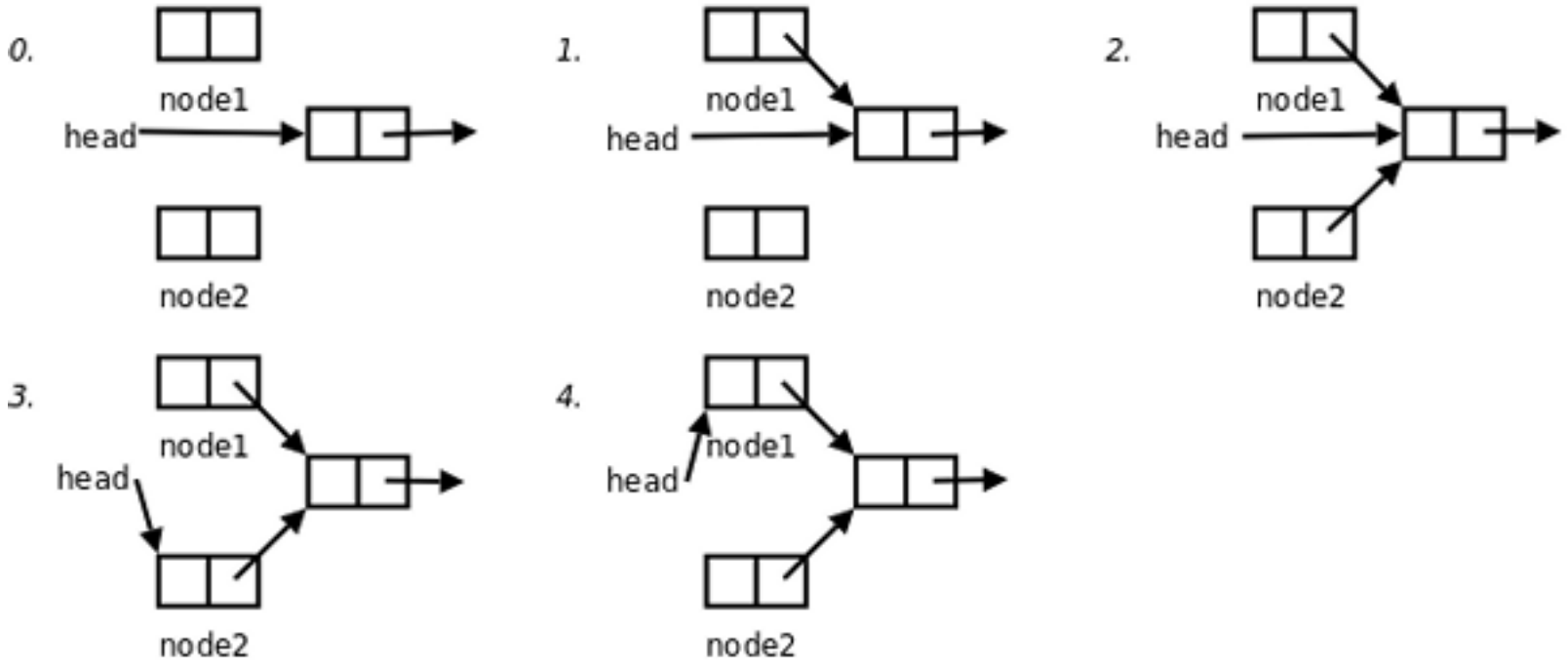
- 当捕捉到信号时，不论进程的主控制流程当前执行到哪儿，都会先跳到信号处理函数中执行，从信号处理函数返回后再继续执行主控制流程。

- 信号处理函数是一个单独的控制流程，因为它和主控制流程是异步的，二者不存在调用和被调用的关系，并且使用不同的堆栈空间。

- 引入了信号处理函数使得一个进程具有多个控制流程，如果这些控制流程访问相同的全局资源（全局变量、硬件资源等），就有可能出现冲突

不可重入函数





不可重入函数

- main函数调用insert函数向一个链表head中插入节点node1

- 插入操作分为两步，刚做完第一步的时候，因为硬件中断使进程切换到内核，再次回用户态之前检查到有信号待处理，于是切换到sighandler函数，sighandler也调用insert函数向同一个链表head中插入节点node2，插入操作的两步都做完之后从sighandler返回内核态，

- 再次回到用户态就从main函数调用的insert函数中继续往下执行，先前做第一步之后被打断，现在继续做完第二步。结果是，main函数和sighandler先后向链表中插入两个节点，而最后只有一个节点真正插入链表中了。

可重入函数

- 像上例这样，insert函数被不同的控制流程调用，有可能在第一次调用还没返回时就再次进入该函数，这称为**重入**，insert函数访问一个全局链表，有可能因为重入而造成错乱，像这样的函数称为**不可重入函数**，反之，如果一个函数只访问自己的局部变量或参数，则称为**可重入（Reentrant）函数**。

不可重入函数

如果一个函数符合以下条件之一则是不可重入的：

- 调用了malloc或free，因为malloc也是用全局链表来管理堆的。
- 调用了标准I/O库函数。标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

竞态条件

- 现在重新审视 mysleep.c，设想这样的时序：

1. 注册SIGALRM信号的处理函数。
2. 调用alarm(nsecs)设定闹钟。
3. 内核调度优先级更高的进程取代当前进程执行，并且优先级更高的进程有很多个，每个都要执行很长时间
4. nsecs秒钟之后闹钟超时了，内核发送SIGALRM信号给这个进程，处于未决状态。
5. 优先级更高的进程执行完了，内核要调度回这个进程执行。SIGALRM信号递达，执行处理函数sig_alrm之后再次进入内核。
6. 返回这个进程的主控制流程，alarm(nsecs)返回，调用pause()挂起等待。
7. 可是SIGALRM信号已经处理完了，还等待什么呢？

出现这个问题的根本原因是系统运行的时序（Timing）并不像我们写程序时所设想的那样。虽然alarm(nsecs)紧接着的下一行就是pause()，但是无法保证pause()一定会在调用alarm(nsecs)之后的nsecs秒之内被调用。由于异步事件在任何时候都有可能发生（这里的异步事件指出现更高优先级的进程），如果我们写程序时考虑不周密，就可能由于时序问题而导致错误，这叫做竞态条件（Race Condition）。

解决方法？

- 在调用pause之前屏蔽SIGALRM信号使它不能提前递达就可以了？

1. 屏蔽SIGALRM信号;
2. alarm(nsecs);
3. 解除对SIGALRM信号的屏蔽;
4. pause();

要是“解除信号屏蔽”和“挂起等待信号”这两步能合并成一个原子操作就好了，这正是sigsuspend函数的功能。sigsuspend包含了pause的挂起等待功能，同时解决了竞态条件的问题，在对时序要求严格的场合下都应该调用sigsuspend而不是pause。

sigsuspend

- #include <signal.h>

- **int sigsuspend(const sigset_t *sigmask);**
- 和pause一样，sigsuspend没有成功返回值，只有执行了一个信号处理函数之后sigsuspend才返回，返回值为-1，errno设置为EINTR。

- 调用sigsuspend时，进程的信号屏蔽字由sigmask参数指定，可以通过指定sigmask来临时解除对某个信号的屏蔽，然后挂起等待，当sigsuspend返回时，进程的信号屏蔽字恢复为原来的值，如果原来对该信号是屏蔽的，从sigsuspend返回后仍然是屏蔽的。

例子

■ new_mysleep.c

如果在调用mysleep函数时SIGALRM信号没有屏蔽：

1. 调用sigprocmask(SIG_BLOCK, &newmask, &oldmask);时屏蔽SIGALRM。
2. 调用sigsuspend(&suspmask);时解除对SIGALRM的屏蔽，然后挂起等待待。
3. SIGALRM递达后suspend返回，自动恢复原来的屏蔽字，也就是再次屏蔽SIGALRM。
4. 调用sigprocmask(SIG_SETMASK, &oldmask, NULL);时再次解除对SIGALRM的屏蔽。

关于SIGCHLD信号

- 进程中讲过用wait和waitpid函数清理僵尸进程，父进程可以阻塞等待子进程结束，也可以非阻塞地查询是否有子进程结束等待清理（也就是轮询的方式）。

- 其实，子进程在终止时会给父进程发SIGCHLD信号，该信号的默认处理动作是忽略，父进程可以自定义SIGCHLD信号的处理函数，这样父进程只需专心处理自己的工作，不必关心子进程了，子进程终止时会通知父进程，父进程在信号处理函数中调用wait清理子进程即可。

- 请编写一个程序完成以下功能：父进程fork出子进程，子进程调用exit(2)终止，父进程自定义SIGCHLD信号的处理函数，在其中调用wait获得子进程的退出状态并打印。

消除僵尸进程

- 事实上，由于UNIX的历史原因，要想不产生僵尸进程还有另外一种办法：父进程调用sigaction将SIGCHLD的处理动作置为SIG_IGN，这样fork出来的子进程在终止时会自动清理掉，不会产生僵尸进程，也不会通知父进程。

- 系统默认的忽略动作和用户用sigaction函数自定义的忽略通常是没有区别的，但这是一个特例。此方法对于Linux可用，但不保证在其它UNIX系统上都可用。

- 请编写程序验证这样做不会产生僵尸进程。

例子程序

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(void)
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0)
    {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0)
    {
        message = "This is the child\n";

    }else
    {
        signal(SIGCHLD,SIG_IGN);
        while (1);
    }

    return 0;

}
```

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.