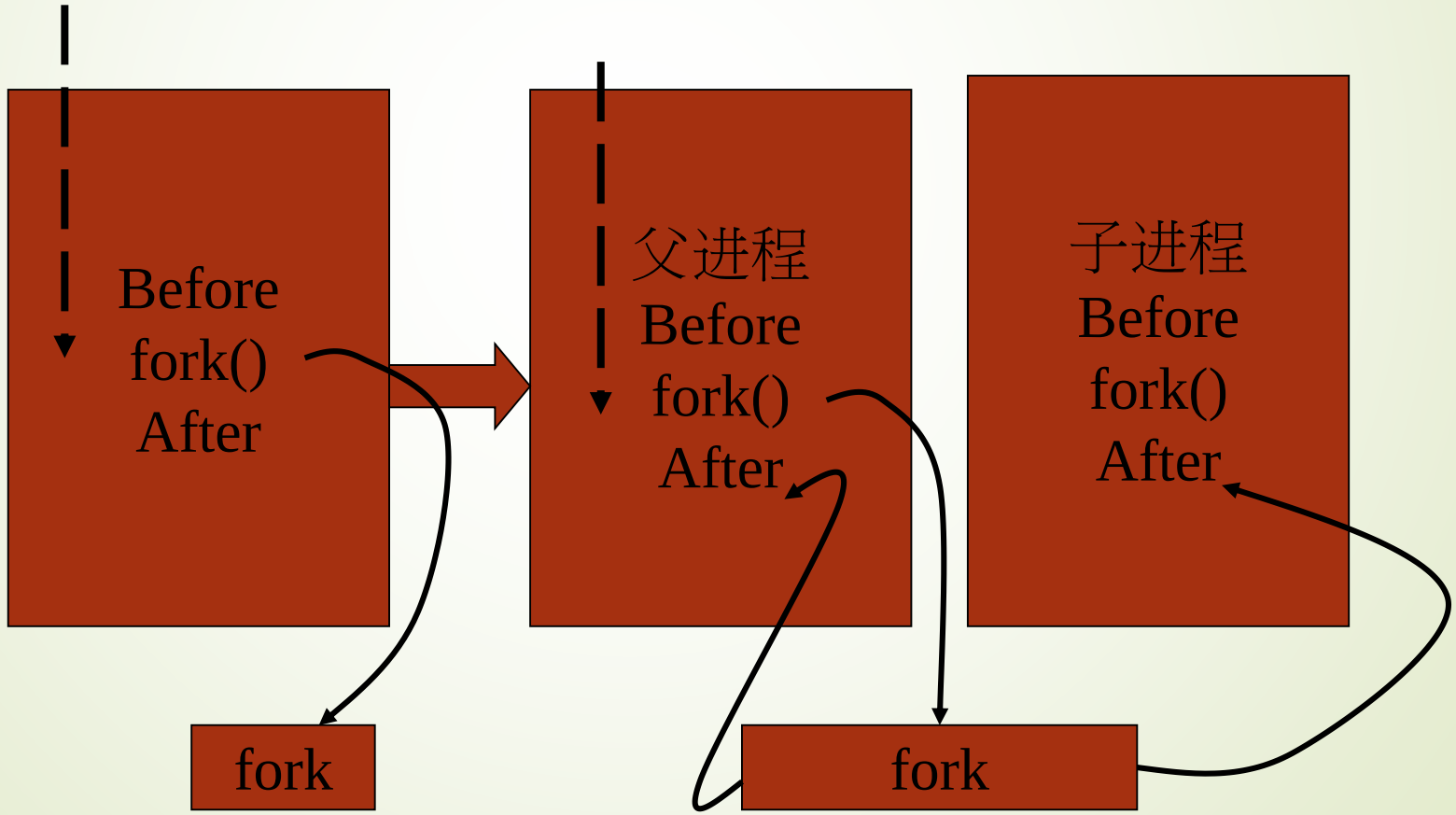
Review: 进程基本概念

- 任务
- Linux进程的主要类型：
  - 交互进程：由Shell启动的进程。可在前台或者后台运行。前台可通过Shell与用户交互
  - 批处理进程：该类进程和终端没有联系，由多个进程按照指定的方式执行
  - 守护进程：在后台运行的与任何终端无关的进程。

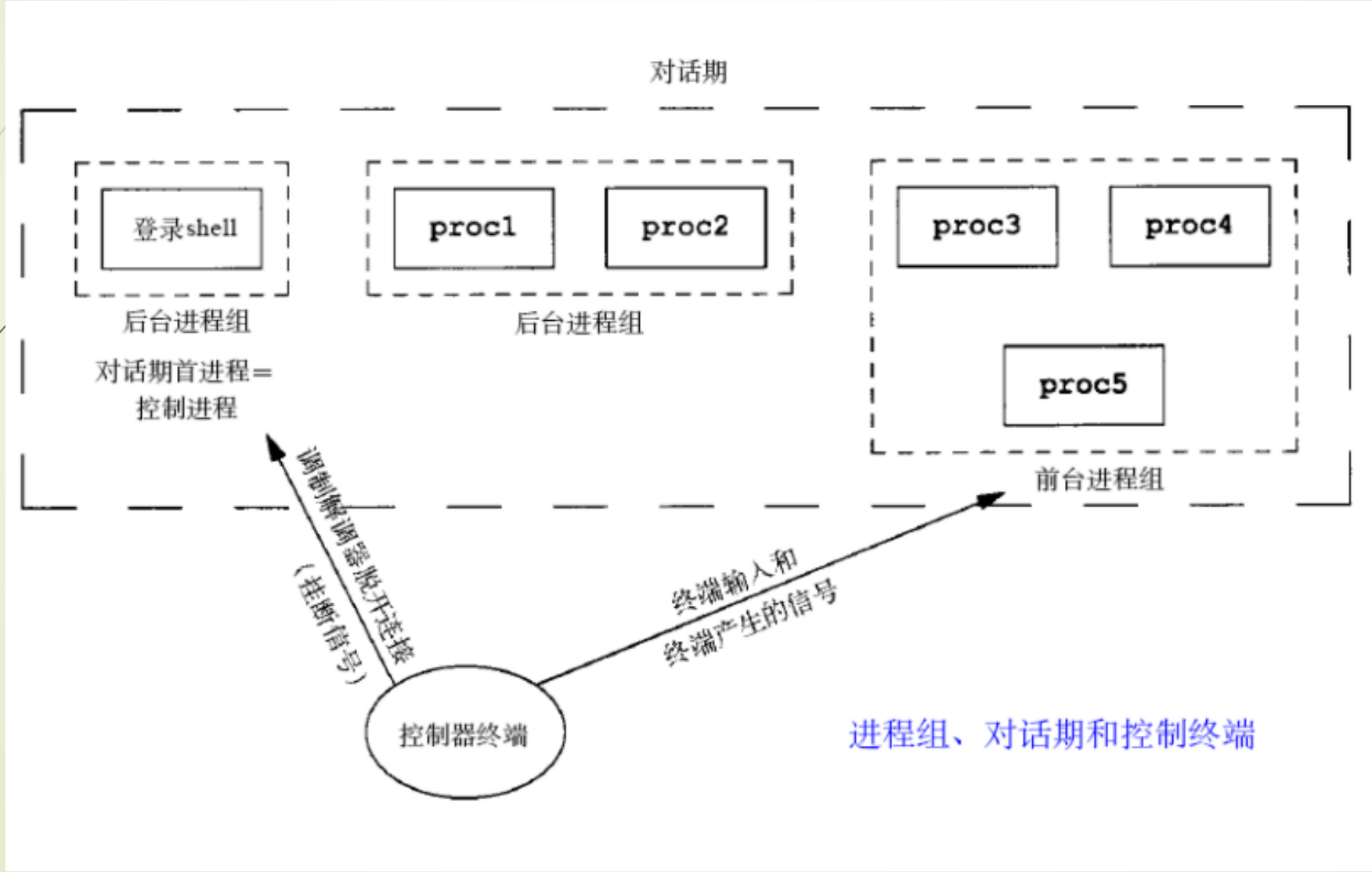
Review: 进程创建


Before
fork()
After
父进程
Before
fork()
After
子进程
Before
fork()
After
fork
fork

Review: execvp系统调用

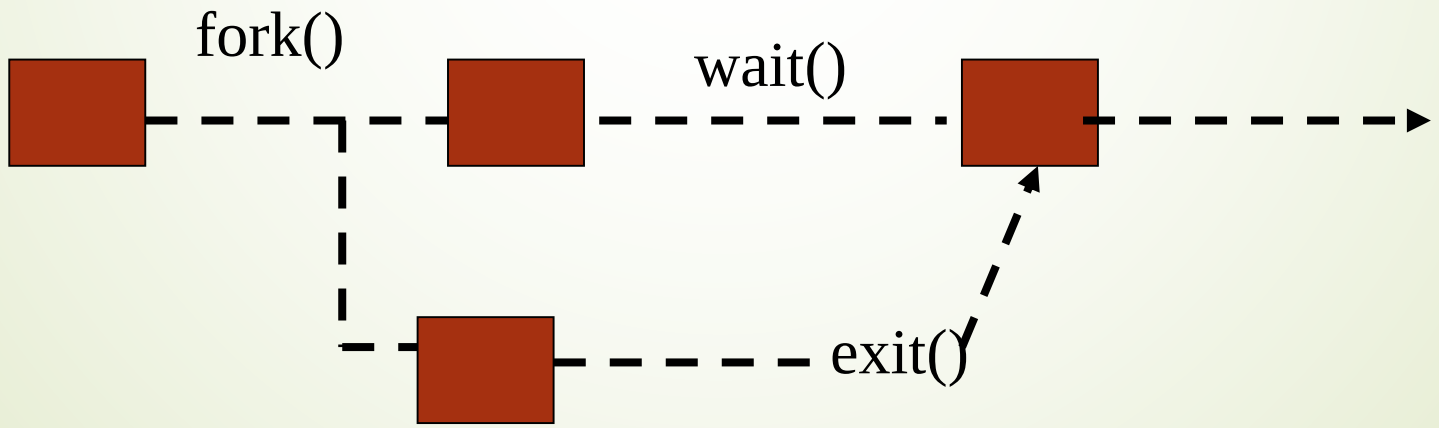
目标	在指定路径中查找并执行一个文件
头文件	#include <unistd.h>
函数原型	result=execvp(const char *file,const char *argv[]);
参数	file 要执行的文件名 argv 字符串数组
返回值	-1 如果出错 成功，execvp没有返回值

Review:进程组、会话期和控制终端关系

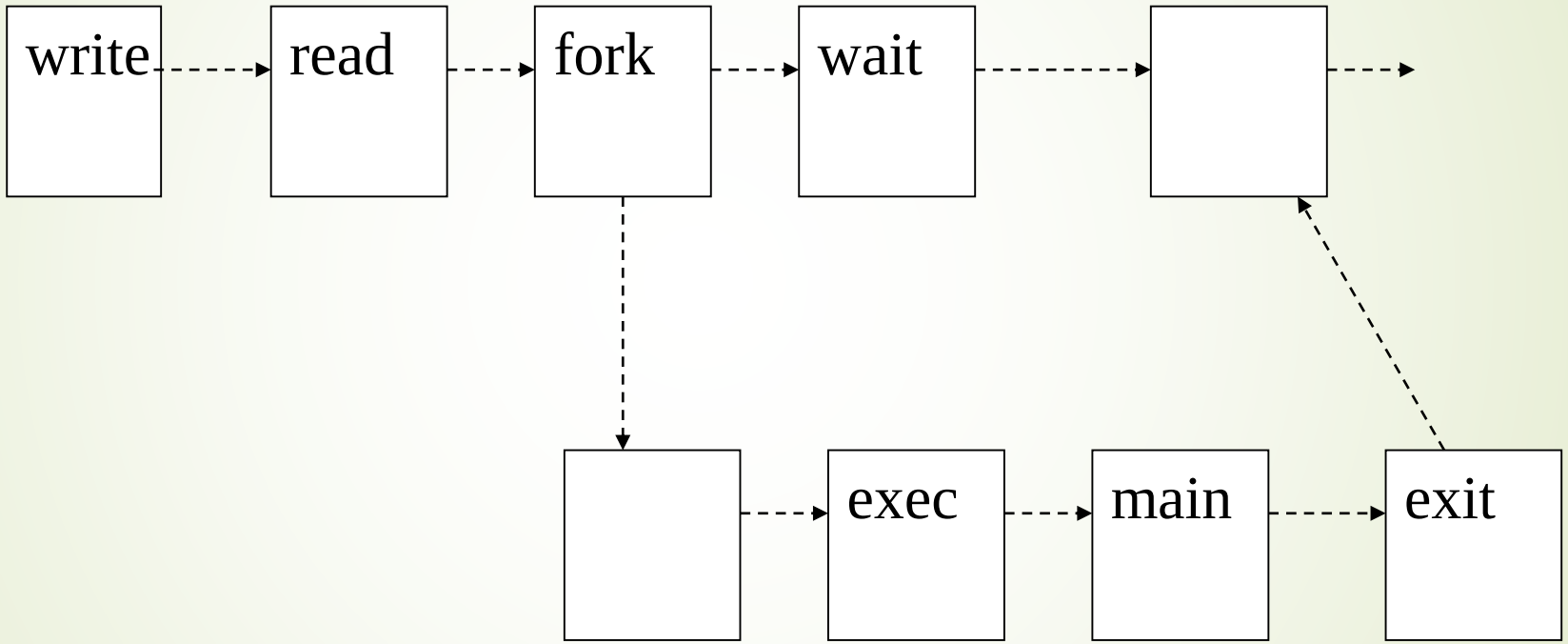


Review:父进程如何等待子进程的退出

- 进程调用wait等待子进程的退出
- pid=wait(&status);
- wait做两件事：首先暂停调用它的进程直到子进程结束，然后取得子进程结束时传给exit的值



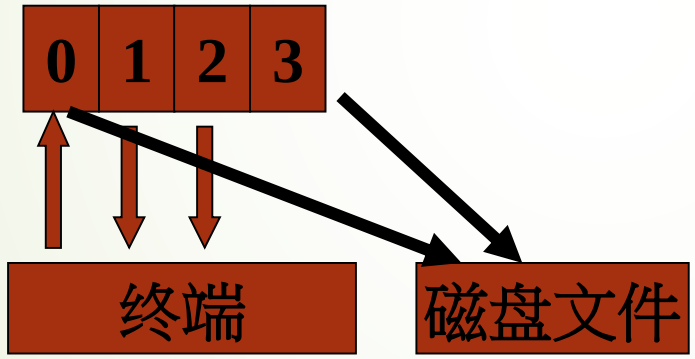
Review: shell如何运行程序？



例5-8

Review:重定向

- 方法1： close 然后 open
- 方法2：open close dup close
- 方法3：open…dup2..close



open(“f”,O_RDONLY)

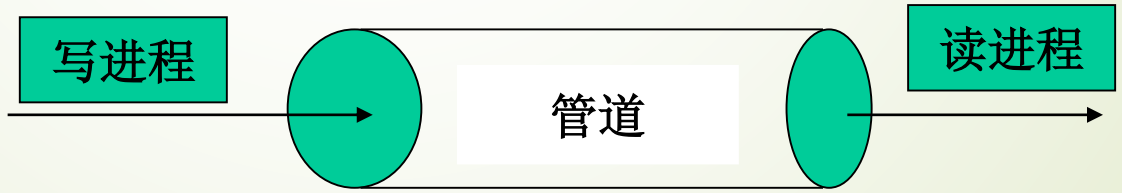
close(0)

dup(fd)

close(fd)

Review:管道通信

- 管道是Linux的一种最简单的通信机制。它在进程之间建立一种逻辑上的管道，一端为流入端，一端为流出端
- 一个进程在流入端写入数据，另外进程在流出段按照流入顺序读出数据，从而实现进程间的通信。
- 匿名管道
- 有名管道



第6讲 信号与信号处理

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
- 信号处理的应用

信号的基本概念

- 信号是Linux操作系统中进程之间通信的一种方式
- 信号传递一种信息，接收方根据该信息进行相应的动作
- 可用于控制信息的传递，例如当发生某种情况时通知进程进行处理
- 在进行Linux编程时，需要知道信号的基本概念，信号可能的发生原因，在信号发生时的处理等情况

信号的本质

- 信号本质是一种软中断
- 信号的两个目的：
  - (1)让进程知道发生了某种事件；
  - (2)根据该事件执行相应的动作即执行它自己代码中的信号处理程序。
- 进程在运行时由自身产生或由进程外部发过来的消息或者事件
  - 进程执行除0指令时产生SIGDIV出发错误信号
  - 非法访问某内存地址时产生SIGSEGV信号
  - 管理员通过kill命令或者进程通过sigsend调用发送。

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
- 信号处理的应用

信号种类

- 每个信号用整型常量宏表示，以SIG开头
  - 比如SIGCHLD、SIGINT等，
- 它们在系统头文件<signal.h>中定义
- 可以通过在shell下键入kill –l查看信号列表
- 或者键入man 7 signal查看更详细的说明。

信号的来源

- 信号均由内核发送
- 生成信号的请求来自3个地方：
  - ✓用户--通过输入Ctrl-C、 Ctrl-\等请求内核产生信号
  - ✓内核--进程执行出错时，内核向进程发送一个信号，例如非法段访问、浮点数溢出等，也可通知进程特定事件的发生。
  - ✓进程--通过系统调用kill给另一个进程发送信号。进程之间可通过信号通信

信号的状态

- 信号的递送(delivery)：当进程对信号采取动作（执行信号处理函数或忽略）时称为递送。
- 信号产生和递送之间的时间间隔内称信号是未决的（pending）。
- 信号递送阻塞(block)：进程可指定对某个信号采用递送阻塞。
  - 若此时信号处理为默认或者捕捉的，该信号就会处于未决的状态。

信号的分类

- 根据信号的来源，可将其分为同步信号和异步信号。
- 由进程的某个操作产生的信号称为同步信号，例如被零处。
  - 该信号的产生和操作同步产生。
- 用户击键这样的进程外的事件引起的信号称为异步信号，该信号产生的事件进程是不可控。

信号的分类

- 根据信号的处理情况，将信号分为
  - 不可靠信号和可靠信号。
- 前者当同时有多个信号产生时，无法及时处理，造成信号的丢失，称为不可靠信号
  - 早期Unix系统中的信号机制比较简单和原始，把那些建立在早期机制上的信号叫做“不可靠信号
- 信号值小于SIGRTMIN为不可靠信号，在SIGRTMIN-SIGRTMAX之间的为可靠信号

可靠信号与不可靠信号

- 收到信号的速度超过进程处理的速度的时候，不可靠信号将多余的丢弃掉
- 可靠信号将来不及处理的信号就会排入进程的队列

实时信号与非实时信号

- 实时信号与非实时信号：
  - Linux目前定义了64种信号（将来可能会扩展）
    - 前面32种为非实时信号
    - 后32种为实时信号。
- 非实时信号都不支持排队，都是不可靠信号
- 实时信号都支持排队，都是可靠信号

常见信号

信号名称	信号说明	默认处理
SIGABRT	调用 abort时产生该信号，程序异常结束	进程终止并且产生core文件
SIGALRM	由alarm或者setitimer设置的定时器到期	进程终止
SIGBUS	总线错误，地址没对齐等，取决于具体硬件	进程终止并产生core文件
SIGCHLD	子进程停止或者终止时，父进程收到该信号	忽略该信号
SIGCONT	让停止的进程继续执行	继续执行或者忽略
SIGFPE	算术运算异常，除0等	进程终止并且产生core文件
SIGHUP	进程的控制终端关闭时产生这个信号	进程终止
SIGILL	代码中有非法指	进程终止并产生core文件
SIGINT	终端输入了CTRL+c信号(下面用^c表示)	进程终止
SIGIO	异步I/O，跟SIGPOLL一样	进程终止
SIGIOT	执行I/O时产生硬件错	进程终止并且产生core文件
SIGKILL	该信号用户不能去捕捉和忽略它	进程终止

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
- 信号处理的应用

进程处理信号的方法

- 进程通过signal系统调用告诉内核如何处理信号
- 进程有3个选择：
  - (1)接受默认处理
    - SIGINT默认处理为消亡，进程通过系统调用signal(SIGINT,SIG_DFL)恢复默认处理
  - (2)忽略信号
    - signal(SIGINT,SIG_IGN)系统调用告诉内核忽略该信号
  - (3)调用一个函数，这是3种方法中最强大的一个。
    - 程序能够告诉内核，当信号来时调用哪个函数，signal(SIGINT,function);
- 信号到来时所调用的函数称为信号处理函数

signal系统调用

目标	简单的信号处理
头文件	**#include <signal.h>**
函数原型	**result=signal (int signum,void (*action)(int));**
参数	**signum** 需响应的信号 **action** 如何响应
返回值	**-1** 遇到错误 **prevaction** 返回之前的处理函数指针

signal系统调用

- 其中**action**可以是函数名也可以是如下两种特殊值之一：
  - **SIG_IGN**,忽略信号
  - **SIG_DFL** 将信号恢复为默认处理
  - **signal**返回前一个处理函数。值为指向该函数的指针

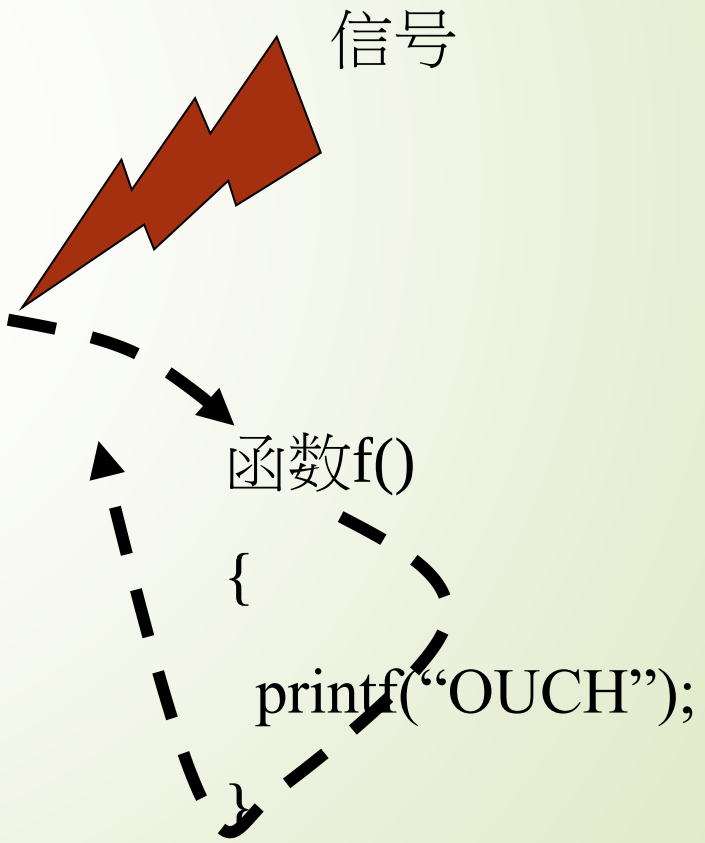
信号处理的例子sigdemo1.c

```
#include    <stdio.h>
#include    <signal.h>
main()
{
            void       f(int);                  /* declare the handler    */
            int   i;
            signal( SIGINT, f );                /* install the handler */
            for(i=0; i<5; i++ ){        /* do something else       */
                  printf("hello\n");
                  sleep(1);
            }
}
void f(int signum)                      /* this function is called */
{
      printf("OUCH!\n");
}
```

信号处理过程

正常控制流

```
main()
{
  signal(SIGINT,f);
  for(i=0;i<5;i++) {
     printf( “hello\n” );
    sleep(1); }
}
```



sigdemo1.c执行结果

- hello
- hello
- hello
- hello
- ^COUCH!
- hello

忽略信号sigdemo2.c

```
#include <stdio.h>
#include <signal.h>
main()
{
    signal( SIGINT, SIG_IGN );

    printf("you can't stop me!\n");
    while( 1 )
    {
        sleep(1);
        printf("haha\n");
    }
}
```

sigdemo2.c程序执行结果

- you can't stop me!
- haha
- haha
- ^Chaha
- haha
- ^Chaha
- haha
- ^\退出

- sigdemo2.c调用signal忽略中断信号，可以随意按Ctrl-C而不会对程序产生影响

不可靠的信号

- 例6-1
- 当多次执行Ctrl+\时，程序退出
- 这是早期不可靠信号处理机制造成的。当执行完一次信号处理函数之后，系统的信号处理就恢复为默认处理，如果想让信号处理函数继续有效，必须重新设置。

不可靠信号

为了解决上述问题，可将**sigHandler**函数改为如下形式：

```
void sigHandler(int signalNum)
{
    printf("The sign no is:%d\n", signalNum);
    signal(SIGINT,sigHandler);  //重新设置
}
```

- 在信号处理函数执行结束和重新设置期间，有可能有信号产生，但未能来得及处理。
  - 被称为捕鼠器问题

signal面临的问题

- 信号处理函数正在执行，没结束时，又产生一个同类型的信号，这时该怎么处理；
- 信号处理函数正在执行，没结束时，又发生了一个不同类型的信号，这时该怎么处理；
- 进程执行一个阻塞系统调用如**read()**时，发生了一个信号，这时是让该阻塞系统调用返回错误再接着进入信号处理函数，还是先跳转到信号处理函数，等信号处理完毕后，系统调用再返回。此时，如何能让**read**和**write**继续操作呢？

6-2

【例 6-2】multisignal.c 程序演示了 signal 所面临的上述问题。

```
//multisignal.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#define INPUTLEN 20
char input[INPUTLEN];
void inthandler(int s)
{
 printf(" I have Received signal %d .. waiting\n", s );
 sleep(2);
 printf(" I am Leaving inthandler \n");
    signal( SIGINT,  inthandler );
}
void quithandler(int s)
{
 printf(" I have Received signal %d .. waiting\n", s );
 sleep(3);
 printf("I am Leaving quithandler \n");
   signal( SIGQUIT, quithandler );
}
void main()
{
signal( SIGINT,  inthandler );  /* set ^C handler */
signal( SIGQUIT, quithandler );  /* set ^\ handler */
do {
 printf("please input a message\n");
 nchars = read(0, input, (INPUTLEN-1));  //从键盘读取输入
 if ( nchars == -1 )
  perror("read returned an error");
 else {
  input[nchars] = '\0';  //存放字符串结束符
  printf("You have inputed: %s\n", input);
       }
}  while( strncmp( input , "quit" , 4 ) != 0 ) ;
}
```

不可靠信号

- 例6-2
  - 捕获SIGINT（^C）
  - 捕获SIGQUIT（^\）

```
please input a message
^C I have Received signal 2 .. waiting
^C^C I am Leaving inthandler
 I have Received signal 2 .. waiting
 I am Leaving inthandler
```

不可靠信号

- 对于第二种情况，连续输入^C^C^C和^\

```
please input a message
^C I have Received signal 2 .. waiting
^C^C^\ I have Received signal 3 .. waiting
I am Leaving quithandler
 I am Leaving inthandler
 I have Received signal 2 .. waiting
 I am Leaving inthandler
```

不可靠信号

- 输入hello^C

```
please input a message
hello^C I have Received signal 2 .. waiting
 I am Leaving inthandler

You have inputed:
```

- 输入hello Return ^C时，可接收到hello字符串
- hel^Clo Return时，程序只接收到lo。
- ^\ ^\ hello^C时，无法得到hello输入，同时信号处理执行完后，重新开始执行read操作等待用户输入

sigaction可以处理多个信号

目标	**指定信号的处理函数**
头文件	**#include <signal.h>**
函数原型	**result=signaction (int signum,const struct sigaction *action struct sigaction *prevaction));**
参数	**signum 需处理的信号** **action 指向描述操作的结构的指针** **Prevaction指向描述被替换操作的结构指针**
返回值	**-1 遇到错误** **0 成功**

定制信号处理struct sigaction

```
struct sigaction{
    void (*sa_handler)();
    void (*sa_sigaction)(int,siginfo_t *,void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

其中**sa_handler**可以为**SIG_DFL,SIG_IGN**或者函数名称,它是老的信号处理方式

- sa_sigaction为新的信号处理机制
  - 它可获得信号编号及被调用的原因及产生问题的上下文的相关信息
- 使用旧的处理机制：
  - struct sigaction action;
  - action.sa_handler=handler_old;
- 使用新的处理机制：
  - struct sigaction action;
  - action.sa_sigaction=handler_new;
  - 将sigaction中的标志位sa_flags置为：SA_SIGINFO

sa_flags标志

标记	含义
SA_RESETHAND	当处理函数被调用时重置，即捕鼠器模式
SA_NODEFER	处理信号时关闭信号自动阻塞，因此允许递归调用信号处理函数
SA_RESTART	当系统调用是针对一些慢速的设备或类似的系统调用，重新开始而不是返回
SA_SIGINFO	指明使用**sa_sigaction**的处理函数值。如果它未设置，则使用旧处理机制，若设置，则传给处理函数的包括信号编号、信号产生的原因和条件等信息

进程的阻塞信号

- 任何时候进程都有一些信号被阻塞，这个信号的集合被称为信号挡板
- 系统调用**sigprocmask**可修改这个被阻塞的信号集
- **sigprocmask**是一个原子操作，根据所给的信号集来修改当前被阻塞的信号集

sigprocmask

目标	修改当前的信号挡板
头文件	#include <signal.h>
函数原型	int result=sigprocmask (int how,const sigset_t *sigs,sigset_t *prev);
参数	how 如何修改信号挡板 sigs 指向使用的信号列表的指针 prev指向之前的信号挡板列表的指针或者为null
返回值	-1 遇到错误 0 成功

- how参数：SIG_BLOCK、SIG_UNBLOCK或者SIG_SET
- SIG_SET表示设置信号集合
- *sigs所指定的信号将被添加、删除或者替换
- 之前的信号挡板设置将被复制到*prev中

sigsetops构造信号集

- **sigset_t是一个抽象的信号集**
  - 通过函数添加或删除其中的信号
- **sigemptyset(sigset_t *setp)**
  - 清除由**setp**指向的列表中的所有信号
- **sigfillset(sigset_t *setp)**
  - 添加所有的信号到**setp**指向的列表中
- **sigaddset(sigset_t *setp，int signum)**
  - 添加信号**signum**到**setp**指向的列表中
- **sigdelset(sigset_t *setp，int signum)**
  - 从**setp**指向的列表中删除信号**signum**

阻塞用户信号的例子

- sigset_t sigs,prevsigs;
- sigemptyset(&sigs);
- sigaddset(&sigs,SIGINT);
- sigaddset(&sigs,SIGQUIT);
- sigprocmask(SIG_BLOCK,&sigs,&prevsigs);
- ……
- sigprocmask(SIG_SET,*prevsigs,NULL);
- 例6-3

```c
int main(int argc,char**argv)
{
    struct sigaction act;
    sigset_t newmask, oldmask;
    int rc;
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
    sigaddset(&newmask, SIGRTMIN);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    act.sa_sigaction = sig_handler;
    act.sa_flags = SA_SIGINFO;
    if(sigaction(SIGINT, &act, NULL) < 0)
    {
        printf("install sigal error\n");
    }
    if(sigaction(SIGRTMIN, &act, NULL) < 0)
    {
        printf("install sigal error\n");
    }
     printf("pid = %d\n", getpid());


    sleep(60);

    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return 0;
} ? end main ?
void sig_handler(int signum,siginfo_t *info,void *myact)
{
    if(signum == SIGINT)
        printf("Got a common signal\n");
    else
        printf("Got a real time signal\n");
}
```

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
  - 信号发送函数
  - 可重入函数
  - 父子进程的信号处理
- 信号处理的应用

kill系统调用

目标	向一个进程发送信号
头文件	#include <sys/types.h> #include <signal.h>
函数原型	int kill (pid_t pid, int sig);
参数	pid 目标进程 sig要被发送的信号
返回值	-1 遇到错误 0 成功

raise调用

目标	向自身进程发送信号
头文件	#include<sys/types.h> #include<signal.h>
函数原型	int raise(intsig);
参数	sig被发送信号
返回值	-1遇到错误 0成功

sigqueue调用

目标	向进程发送信号
头文件	#include<sys/types.h> #include<signal.h>
函数原型	int sigqueue(pid_t pid,int sig,const union sigval value);

参数	pid目标进程的pid
	sig被发送信号
	参数value为一整型与指针类型的联合体：
	unionsigval{
	int sival_int;
	void* sival_ptr;
	};

返回值	-1遇到错误 0成功

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
  - 信号发送函数
  - 可重入函数
  - 父子进程的信号处理
- 信号处理的应用

可重入函数

- 某个函数可被多个任务并发使用，而不会造成数据错误，则该函数具有可重入性（**reentrant**）
- 信号处理函数中，避免使用不可重入函数，因为信号处理函数有可能被调用多次。
- 若处理函数使用了不可重入函数而变成不可重入时，则必须阻塞信号，若阻塞信号，则信号有可能丢失。
- 可重入函数中不能使用静态变量，不能使用**malloc/free**函数和标准I/O库，使用全局变量时也应小心

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
  - 信号发送函数
  - 可重入函数
  - 父子进程的信号处理
- 信号处理的应用

父子进程的信号处理

- 父进程创建子进程时，子进程继承了父进程信号处理方式，直到子进程调用exec函数。
- 子进程调用exec函数后，exec将父进程中设置为捕捉的信号变为默认处理方式。
- 例如父进程设置捕捉SIGINT信号，创建子进程时，子进程与父进程执行相同的SIGINT处理函数，当子进程执行exec后，SIGINT设置为终止子进程。
- 例6-4

```c
void intsig_handler (int signumber,siginfo_t *siginfo,void *empty)
{
   printf("int_handler, my pid=%d\n",getpid());
}
int main()
{
    int pid;
    struct sigaction act;
    char *arg[]={"-l"};
       act.sa_sigaction = intsig_handler;
     act.sa_flags = SA_SIGINFO;
      if(sigaction(SIGINT, &act, NULL) < 0)     {
            printf("install sigal error\n");      }
    printf("The parent pid =%d\n",getpid());
    pid=fork();
    if (pid<0) { perror("fork failed!\n"); exit(0);}
    printf("The return fork =%d\n",pid);
    if (pid==0) execvp("ls",arg);
    else
        while(1);
}
```

主要内容

- 信号的基本概念
- 信号的分类
- 信号的处理
  - 信号发送函数
  - 可重入函数
  - 父子进程的信号处理
- 信号处理的应用

信号处理机制的使用

- 一般情况下，当进程正在执行某个系统调用，那么在该系统调用返回前信号是不会被递送的。
- 但对于慢速设备的系统调用除外，例如读写终端、文件、网络、磁盘等操作。
- 采用**sigaction**设置信号处理函数，当使用了**SA_RESTART**选项时，像**read**、**write**、**ioctl**等系统调用都会自动重启
- 若未使用**SA_RESTART**选项，则返回**-1**，**errno**设置为**EINTR**。

信号机制的应用

- 利用信号机制防止僵尸进程的产生
- 子进程在退出程序时，会向父进程发送**SIGCHLD**信号
- 父进程在该信号的处理函数中调用**wait**或者**waitpid**获取子进程的退出状态
- 默认情况下，父进程是忽略该信号的。
- 例**6-5**

```c
void sigchld_handler(int sig)
{
    int status;
    waitpid(-1,&status, 0) ;
 if  ( WIFEXITED(status) )  printf("child process exit normally\n");
 else if ( WIFSIGNALED(status) ) printf("child process exit abnormally\n");
 else if ( WIFSTOPPED(status) ) printf("child process is stopped\n");
  else printf("else");
}
int main()
{
   pid_t pid;
      signal(SIGCHLD,sigchld_handler) ;
   pid=fork() ;
   if (pid==0) abort() ;
   else if (pid>0) {sleep(2); printf("parent process\n");}
   else exit(0);
}
```

睡眠函数

- 1、alarm
- 如果不要求很精确的话，用alarm()和signal()就够了
  - unsigned int alarm(unsigned int seconds)
  - 函数说明: alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds为0，则之前设置的闹钟会被取消，并将剩下的时间返回。
  - 返回值: 返回之前闹钟的剩余秒数，如果之前未设闹钟则返回0。
  - alarm()执行后，进程将继续执行，在后期(alarm以后)的执行过程中将会在seconds秒后收到信号SIGALRM并执行其处理函数。
- Linux中并没有提供系统调用sleep()，sleep()是在库函数中实现的，它是通过调用alarm()来设定报警时间，调用sigsuspend()将进程挂起在信号SIGALARM上，sleep()只能精确到秒级上。
  - unsigned int sleep(unsigned int seconds);

睡眠函数

- usleep()----以微秒为单位
  - unsigned int usleep(unsigned int useconds);
  - usleep的时间单位为us，肯定不是由alarm实现的，但都是linux用的，而window下不能用，因为都是sleep和usleep都是在unistd.h下定义的。
  - return：若进程暂停到参数seconds 所指定的时间，成功则返回0，若有信号中断则返回剩余微秒数。
- nanosleep( )---------以纳秒为单位
  - struct timespec { time_t tv_sec; /* 秒seconds */ long tv_nsec; /* 纳秒 nanoseconds */ };
  - int nanosleep(const struct timespec *req, struct timespec *rem);
  - #include<time.h>这个函数功能是暂停某个进程直到你规定的时间后恢复，参数req就是你要暂停的时间，其中req->tv_sec是以秒为单位，而tv_nsec以毫微秒为单位（10的-9次方秒）。由于调用nanosleep是是进程进入TASK_INTERRUPTIBLE,这种状态是会相应信号而进入TASK_RUNNING状态的
  - 若没有等到你规定的时间就因为其它信号而唤醒，此时函数返回-1，还剩余的时间会被记录在rem中。
  - return: 若进程暂停到参数*req所指定的时间，成功则返回0，若有信号中断则返回-1，并且将剩余微秒数记录在*rem中。

计时器

- setitimer()
  - int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue));
  - setitimer()比alarm功能强大，支持3种类型的定时器：
    - ITIMER_REAL： 以系统真实的时间来计算，它送出SIGALRM信号。
    - ITIMER_VIRTUAL：-以该进程在用户态下花费的时间来计算，它送出SIGVTALRM信号。
    - ITIMER_PROF： 以该进程在用户态下和内核态下所费的时间来计算，它送出SIGPROF信号。
  - setitimer()第一个参数which指定定时器类型（上面三种之一）；第二个参数是结构itimerval的一个实例；第三个参数可不做处理。
  - setitimer()调用成功返回0，否则返回-1。

6-7计时器

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>


void timeChange(int ms,struct timeval *ptVal){
        ptVal->tv_sec=ms/1000;
        ptVal->tv_usec=(ms%1000)*1000;
}
void  alarmsign_handler(int SignNo){
    printf("%d seconds\n",++i);
}
int i=0;
int main()
{
    struct itimerval tval;
    signal(SIGALRM,alarmsign_handler);

        timeChange(1,&(tval.it_value));
    timeChange (1000,&(tval.it_interval));
    setitimer(ITIMER_REAL,&tval,NULL);
    while(getchar()!='#');
   return 0;
}
```

思考题

- 教材P141 习题 1，2，3，5