重定向与管道

进程与文件描述符

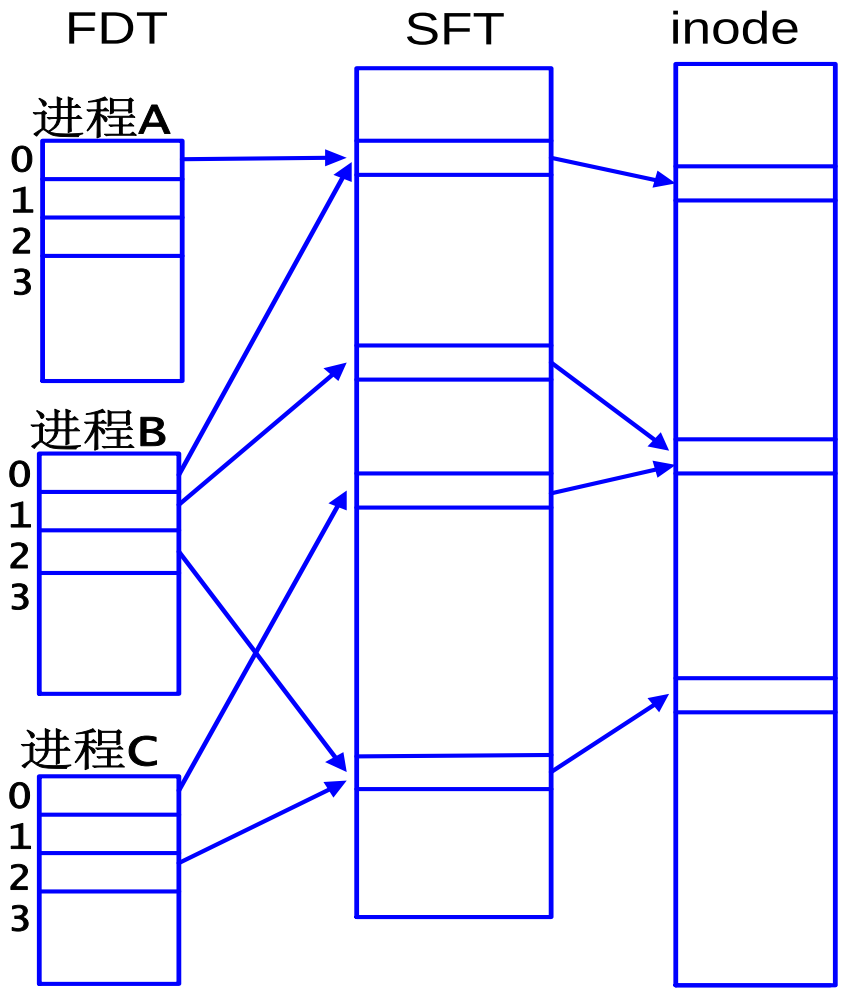
活动文件目录AFD

- **磁盘文件目录(分两级)**
  - 文件名，i节点
- **活动文件目录(分三级)**
  - **文件描述符表FDT**：每进程一张，PCB的user结构中
    - user结构中整型数组u_ofile记录进程打开的文件
    - 文件描述符fd是u_ofile数组的下标
  - **系统文件表SFT**：整个核心一张，file结构

```
struct file {
    char f_flag;    /* 读、写操作要求 */
    char f_count;   /* 引用计数 */
    long f_offset; /* 文件读写位置指针 */
    int  f_inode;   /* 内核中inode数组的下标 */
};
```

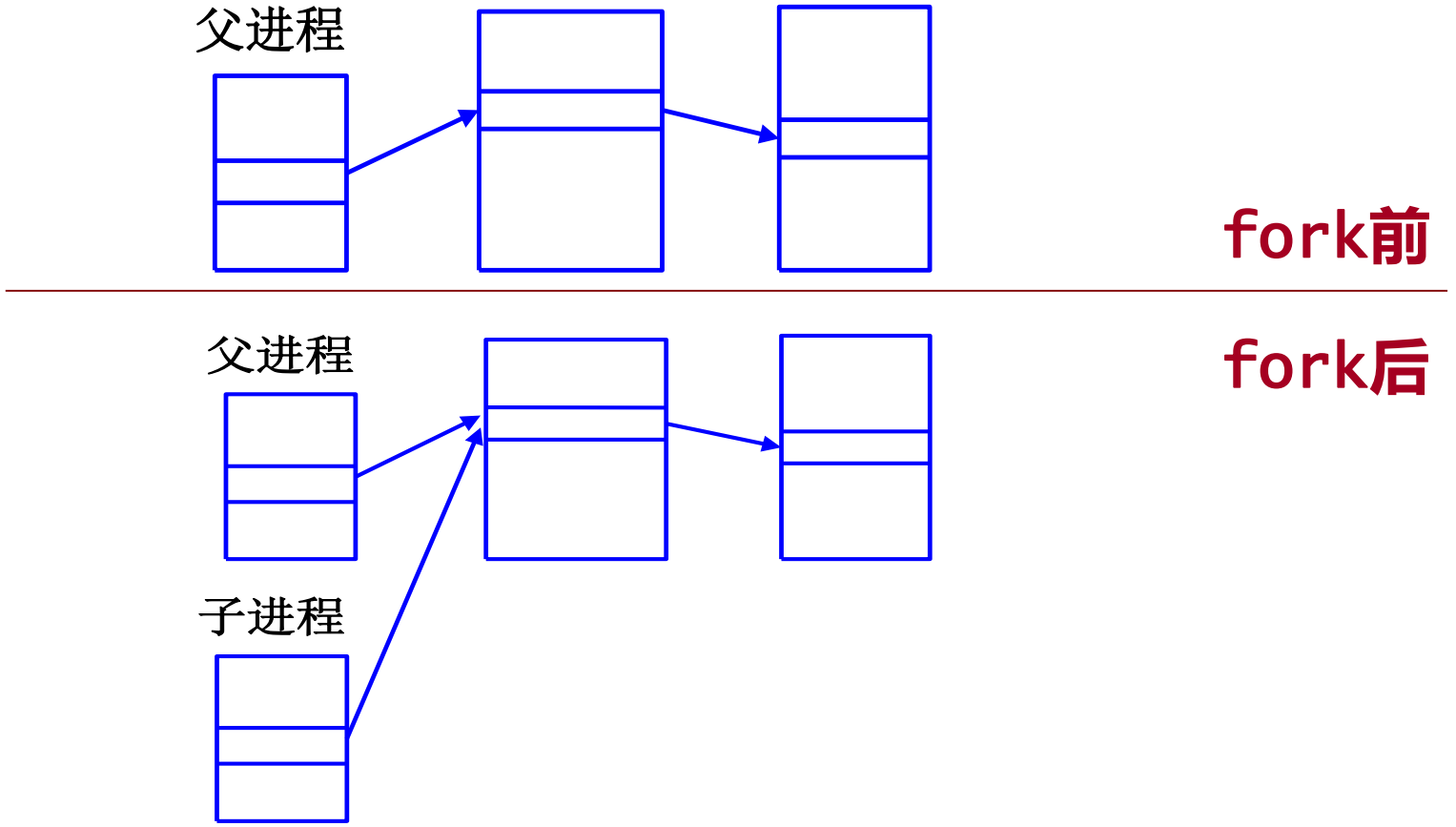
  - **活动i节点表**：整个核心一张，inode结构
    - 内存中inode表是外存中inode的缓冲
    - 内存inode表里也有个专用的引用计数

活动文件目录AFD(图)



文件描述符的继承与关闭

- fork创建的子进程继承父进程的文件描述符表
- 父进程在fork前打开的文件，父子进程有相同的文件偏移



例:文件描述符的继承与关闭(1)

文件f1. c

```
void main()
{
    int fd;
    fd = open("xxf1f2.txt", O_CREAT | O_WRONLY, 0666);
    if (fork() > 0) {
        char *str = "Message from process F1\n";
        int i;
        for (i = 0; i < 200; i++) {
            write(fd, str, strlen(str));
            sleep(1);
        }
        close(fd);
    } else {
        char fdstr[16];
        sprintf(fdstr, "%d", fd);
        execlp("./f2", "f2", fdstr, 0);
        printf("failed to start 'f2': %m\n");
    }
}
```

例:文件描述符的继承与关闭(2)

文件f2. c

```
int main(int argc, char **argv)
{
    int fd, i;
    static char *str = "Message from process F2\n";
    fd = strtol(argv[1], 0, 0);
    for (i = 0; i < 200; i++) {
        if (write(fd, str, strlen(str)) < 0)
            printf("Write error: %m\n");
        sleep(1);
    }
    close(fd);
}
```

close-on-exec标志

- **文件设置了close-on-exec标志，执行exec()系统会自动关闭这些文件**
  - 在open()调用的第三个参数里可以加O_CLOEXEC属性
  - 通过系统调用fcntl()设置
- **函数**

  #include <fcntl.h>

  int fcntl (*fd*, *cmd*, *arg*);

  cmd: F_GETFD 获取文件*fd*的控制字flag，控制字的比特0为close-on-exec标志位

  flag = fcntl(fd, F_GETFD, 0);

  cmd: F_SETFD 设置文件*fd*的控制字 fcntl(fd, F_SETFD, flag);
- **例**

```
flags = fcntl(fd, F_GETFD, 0);
flags |= FD_CLOEXEC;
fcntl(fd, F_SETFD, flags);
```

重定向

文件描述符的复制

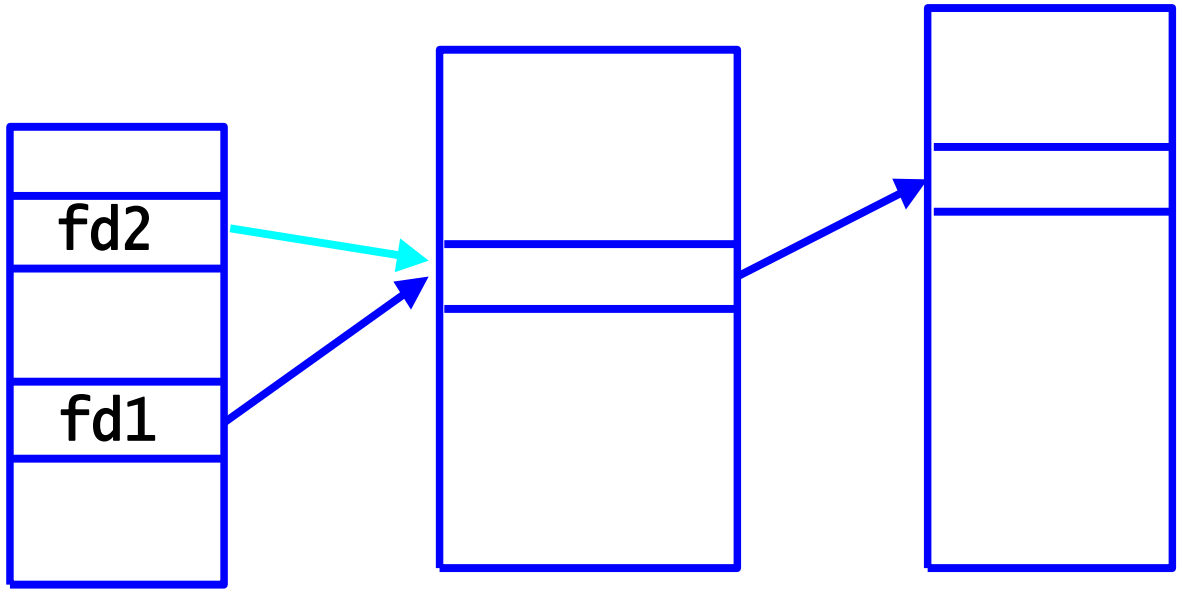
- **系统调用**

  int dup2(int *fd1*, int *fd2*);

- **功能**

  复制文件描述符*fd1*到*fd2*

  - *fd2*可以是空闲的文件描述符
  - 如果*fd2*是已打开文件，则关闭已打开文件



xsh1:输入输出重定向(1)

```
void main(void)
{
     char buf[256], *argv[256], **p, *in, *out;
     int sv;
     for (;;) {
          printf("=> ");
          if (fgets(buf, sizeof(buf), stdin) == NULL) exit(0);
          in = strstr(buf, "<");
          out = strstr(buf, ">");
          if (in != NULL) {
               *in++ = '\0';
               in = strtok(in, " \t\n");
          }
          if (out != NULL) {
               *out++ = '\0';
               out = strtok(out, " \t\n");
          }
          for (p = &argv[0], *p = strtok(buf, " \t\n"); *p;
             *++p = strtok(NULL, " \t\n"));
          if (argv[0] == NULL)
               continue;
          if (strcmp(argv[0], "exit") == 0)
               exit(0);
```


buf
sort -fr < /etc/passwd > xxout.txt \n \0
Infile
outfile
\0
argv
0
1
2
NULL
infile

xsh1:输入输出重定向(2)

```
            if (fork() == 0) {
                int fd0 = -1, fd1 = -1;
                if (in != NULL)
                    fd0 = open(in, O_RDONLY);
                if (fd0 != -1) {
                    dup2(fd0, 0);
                    close(fd0);
                }
                if (out != NULL)
                    fd1 = open(out, O_CREAT | O_WRONLY, 0666);
                if (fd1 != -1) {
                    dup2(fd1, 1);
                    close(fd1);
                }
                execvp(argv[0], argv);
                fprintf(stderr, "** Bad command\n");
                exit(1);
            }
            wait(&sv);
        }
    }
```

管 道

管道操作(1)

创建管道

int pipe(int *pfd*[2]);

int pipe(int **pfd*);

int pipe(int *pfd*[]);

- 创建一个管道，*pfd*[0]和*pfd*[1]分别为管道两端的文件描述字，*pfd*[0]用于读，*pfd*[1]用于写

管道写

ret = write(pfd[1], buf, n)

- 若管道已满，则被阻塞，直到管道另一端read将已进入管道的数据取走为止
- 管道容量：某一有限值，如8192字节，与操作系统的实现相关

管道操作(2)

管道读

ret = read(pfd[0], buf, n)

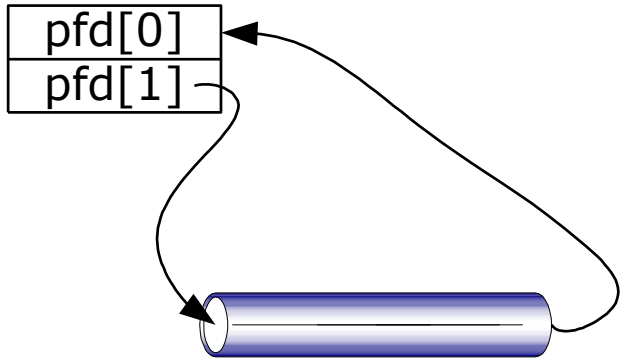
- 若管道写端已关闭，则返回0
- 若管道为空，且写端文件描述字未关闭，则被阻塞
- 若管道不为空(设管道中实际有m个字节)
  - n≥m，则读m个;
  - 如果n < m则读取n个
- 实际读取的数目作为read的返回值。
- 注意：管道是无记录边界的字节流通信

关闭管道close

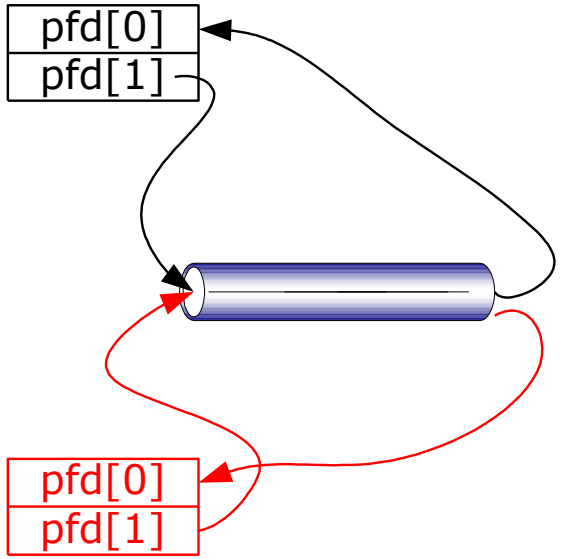
- 关闭写端则读端read调用返回0。
- 关闭读端则写端write导致进程收到SIGPIPE信号(默认处理是终止进程，该信号可以被捕捉)
  - 写端write调用返回-1，errno被设为EPIPE

进程间使用管道通信

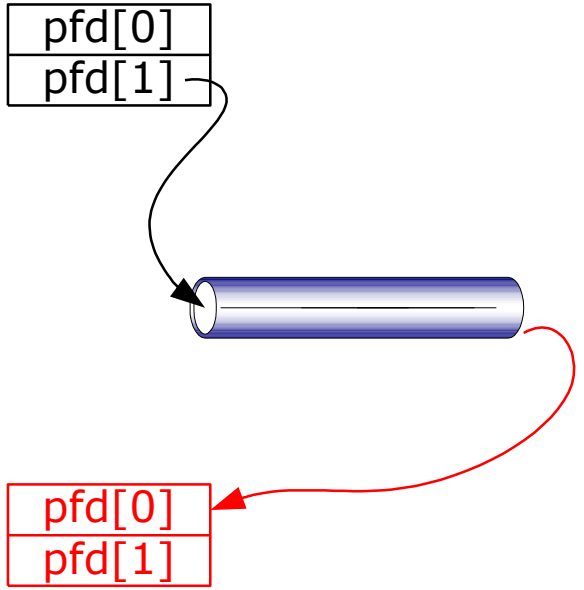
- 父进程



- fork后



- 父进程关闭读端，子进程关闭写端



管道通信：写端

```c
int main(void) /* File name : pwrite.c  */
{
    int pfd[2], i;
    pipe(pfd);
    if (fork() == 0) { /* child */
        char fdstr[10];
        close(pfd[1]);
        sprintf(fdstr, "%d", pfd[0]);
        execlp("./pread", "pread", fdstr, NULL);
        printf("Execute pread file error: %m\n");
    } else { /* parent */
        FILE *f = fdopen(pfd[1], "w");
        close(pfd[0]);
        fprintf(f, "Alice 95\n");
        fprintf(f, "Bob 87\n");
        fprintf(f, "Mallory 79\n");
        fclose(f);
    }
}
```

管道通信：读端

```c
/* File name : pread.c */
int main(int argc, char **argv)
{
    int score;
    char name[128];
    FILE *f;

    f = fdopen(atoi(argv[1]), "r");
    while (fscanf(f, "%s%d", name, &score) != EOF)
        printf("name: %s, score: %d\n", name, score);
    fclose(f);
}
```

管道通信：应注意的问题

- **管道传输是一个无记录边界的字节流**
  - 写端一次write所发数据读端可能需多次read才能读取
  - 写端多次write所发数据读端可能一次read就全部读出
- **父子进程需要双向通信时，应采用两个管道**
  - 用一个管道，进程可能会收到自己刚写到管道去的数据
  - 仅使用一个管道并增加其他同步方式也可以，太复杂
- **父子进程使用两个管道传递数据，有可能死锁**
  - 父进程因输出管道满而写，导致被阻塞
  - 子进程因要向父进程写回足够多的数据而导致写也被阻塞，产生死锁
  - 多进程通信问题必须仔细分析流量控制和死锁问题
- **管道的缺点：没有记录边界**

xsh2:管道(1)

```
int main(void)
{
    char buf[256], *argv1[256], **p, *cmd2, *argv2[256];
    int sv, fd[2];

    for (;;) {
        printf("=> ");
        if (fgets(buf, sizeof(buf), stdin) == NULL)
            exit(0);

        if ((cmd2 = strstr(buf, "|")) == NULL)
            exit(0);
        *cmd2++ = '\0';

        for (p = &argv1[0], *p = strtok(buf, " \t\n"); *p != NULL;
            *++p = strtok(NULL, " \t\n"));
        for (p = &argv2[0], *p = strtok(cmd2, " \t\n"); *p != NULL;
            *++p = strtok(NULL, " \t\n"));

        if (argv1[0] == NULL || argv2[0] == NULL)
            exit(0);
```

xsh2:管道(2)

```
            pipe(fd);
            if (fork() == 0) {
                dup2(fd[1], 1);
                close(fd[1]);
                close(fd[0]);
                execvp(argv1[0], argv1);
                fprintf(stderr, "** bad command 1: %m\n");
                exit(1);
            } else if (fork() == 0) {
                dup2(fd[0], 0);
                close(fd[0]);
                close(fd[1]);
                execvp(argv2[0], argv2);
                fprintf(stderr, "** bad command 2: %m\n");
                exit(1);
            }
            close(fd[0]);
            close(fd[1]);
            wait(&sv);
            wait(&sv);
        }
}
```

命名管道

- **pipe创建的管道（匿名管道）的缺点**

  只限于同祖先进程间通信

- **命名管道**

  允许不相干的进程(没有共同的祖先)访问FIFO管道

- **命名管道的创建**
  - 用命令 mknod pipe0 p
  - 创建一个文件，名字为pipe0
  - 用ls -1列出时，文件类型为p

- **发送者**

```
fd = open(”pipe0”, O_WRONLY);
write(fd, buf, len);
```

- **接收者**

```
fd = open(”pipe0”, O_RDONLY);
read(fd, buf, sizeof(buf));
```

重定向与管道：小结

- fork/exec与文件描述符关系
- 三级活动文件目录
- 设置三级活动文件目录的原因
- 文件描述符的继承与关闭
- close-on-exec标志
- 文件描述符的复制
- 输入输出重定向：xsh1.c
- 管道
- 管道与进程状态
- 管道的读写模型
- 命名管道
- 管道操作：xsh2.c

信号

信号的产生及类型

命令kill

用法与功能

kill –*signal PID-list*

kill命令用于向进程发送一个信号

举例

- kill 1275
  - 向进程1275的进程发送信号，默认信号为15(SIGTERM)，一般会导致进程死亡
- kill -9 1326
  - 向进程1326发送信号9(SIGKILL)，导致进程死亡

进程组

进程组

- 进程在其PCB结构中有p_pgrp域
- p_pgrp都相同的进程构成一个“进程组”
- 如果p_pgrp=p_pid则该进程是组长
- setsid()系统调用将PCB中的p_pgrp改为进程自己的PID，从而脱离原进程组，成为新进程组的组长
- fork创建的进程继承父进程p_pgrp，与父进程同组

举例

- kill命令的PID为0时，向与本进程同组的所有进程发送信号

信号机制

功能

- 信号是送到进程的“软件中断”，通知进程出现了非正常事件

信号的产生

- 用户态进程：自己或者其他进程发出的
  - 使用kill()或者alarm()调用
- 操作系统内核产生信号（往往由中断引发，也有软件触发）
  - 段违例信号SIGSEGV：当进程试图存取它的地址空间以外的存贮单元时，内核向进程发送段违例信号
  - 浮点溢出信号SIGFPE：零做除数时，内核向进程发送浮点溢出信号
  - 信号SIGPIPE：关闭管道读端则写端write导致进程收到信号SIGPIPE

信号类型

- **在<sys/signal.h>文件中定义宏（只有几十种）**

SIGTERM　软件终止信号。kill命令默认信号

SIGHUP　挂断。用户从注册shell中退出时，同组的进程都收到SIGHUP

SIGINT　中断。用户按Ctrl-C键（或Del键）时产生

SIGQUIT　退出。按Ctrl-\时产生，产生core文件

SIGALRM　闹钟信号。计时器时间到，与alarm()有关

SIGCLD　进程的一个子进程终止。

SIGKILL　无条件终止，该信号不能被捕获或忽略。

SIGUSR1，SIGUSR2　用户定义的信号

SIGFPE　浮点溢出

SIGILL　非法指令

SIGSEGV　段违例

SIGWINCH　终端窗口大小发生变化

```
/* ISO C99 signals.  */
#define SIGINT          2       /* Interactive attention signal.  */
#define SIGILL          4       /* Illegal instruction.  */
#define SIGABRT         6       /* Abnormal termination.  */
#define SIGFPE          8       /* Erroneous arithmetic operation.  */
#define SIGSEGV         11      /* Invalid access to storage.  */
#define SIGTERM         15      /* Termination request.  */

/* Historical signals specified by POSIX. */
#define SIGHUP          1       /* Hangup.  */
#define SIGQUIT         3       /* Quit.  */
#define SIGTRAP         5       /* Trace/breakpoint trap.  */
#define SIGKILL         9       /* Killed.  */
#define SIGBUS          10      /* Bus error.  */
#define SIGSYS          12      /* Bad system call.  */
#define SIGPIPE         13      /* Broken pipe.  */
#define SIGALRM         14      /* Alarm clock.  */

/* New(er) POSIX signals (1003.1-2008, 1003.1-2013).  */
#define SIGURG          16      /* Urgent data is available at a socket.  */
#define SIGSTOP         17      /* Stop, unblockable.  */
#define SIGTSTP         18      /* Keyboard stop.  */
#define SIGCONT         19      /* Continue.  */
#define SIGCHLD         20      /* Child terminated or stopped.  */
#define SIGTTIN         21      /* Background read from control terminal.  */
#define SIGTTOU         22      /* Background write to control terminal.  */
#define SIGPOLL         23      /* Pollable event occurred (System V).  */
#define SIGXCPU         24      /* CPU time limit exceeded.  */
#define SIGXFSZ         25      /* File size limit exceeded.  */
#define SIGVTALRM       26      /* Virtual timer expired.  */
#define SIGPROF         27      /* Profiling timer expired.  */
#define SIGUSR1         30      /* User-defined signal 1.  */
#define SIGUSR2         31      /* User-defined signal 2.  */

/* Nonstandard signals found in all modern POSIX systems
   (including both BSD and Linux).  */
#define SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
```

进程对信号的处理

进程对信号的处理

- **进程对到达的信号可以在下列处理中选取一种**
  - 设置为缺省处理方式(大部分处理是程序中止，有的会产生core文件)
  - 信号被忽略
  - 信号被捕捉
    - 用户事先注册好一个函数，当信号发生后就去执行这一函数
- **信号被设为缺省处理方式**

  signal(SIGINT, SIG_DFL);
- **信号被忽略**

  signal(SIGINT,SIG_IGN);
  - 在执行了这个调用后，进程就不再收到SIGINT信号
  - 注意：“某信号被忽略”作为进程的一种属性被它的子进程所继承

信号被忽略：举例

```c
/* File name : foo.c */

int main(void)
{
    signal(SIGTERM, SIG_IGN);
    signal(SIGINT, SIG_IGN);
    if (fork()) {
        for(;;) sleep(100);
    } else
        execlp("./bar", "bar", NULL);
    return 0;
}
```

```c
/* File name : bar.c */

int main(void)
{
    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
    return 0;
}
```

- 单独运行**bar**时，使用**kill -15**命令或者Ctrl-C按键可杀死该进程
- 由**foo**进程来启动的bar进程就不能用**kill -15**命令或按键杀死

信号的捕捉

- 信号被捕捉并由一个用户函数来处理
- 信号到达时，这个函数将被调用来处理那个信号

```
/* Filename: Ctrl-c.c */

#include <stdio.h>
#include <sys/signal.h>
#include <unistd.h>

void sig_handle(int sig)
{
  printf("HELLO! Signal %d catched.\n",sig);
}

int main(void)
{
    signal(SIGINT, sig_handle);
    signal(SIGTERM, sig_handle);
    for(;;) sleep(500);
}
```

僵尸进程

- **僵尸子进程**
  - 子进程终止，僵尸进程(defunct或zombie)出现，父进程使用wait系统调用收尸后消除僵尸
  - 僵尸进程不占用内存资源等但占用内核proc表项，僵尸进程太多会导致proc表耗尽而无法再创建新进程
- **子进程中止后的异步通知机制**
  - 子进程中止后，系统会向父进程发送信号SIGCLD
- **不导致僵尸子进程出现的方法**
  - 忽略对SIGCLD信号的处理

    signal(SIGCLD,SIG_IGN);
  - 捕获SIGCLD信号，执行wait系统调用

发送信号

系统调用kill

int kill(int *pid*, int *sig*)

返回值： 0--成功　　-1--失败

kill调用分几种情况

- 当pid>0时，向指定的进程发信号
- 当pid=0时，向与本进程同组的所有进程发信号
- 当pid<0时，向以-pid为组长的所有进程发信号
- 当sig=0时，则信号根本就没有发送，但可据此判断一个已知PID的进程是否仍然运行
  - kill(pid, 0)；如果函数返回值为-1就可根据errno判断：errno=ESRCH 说明不存在pid进程

系统调用与信号

进程睡眠

- 系统调用执行时会导致进程处于睡眠状态
  - 如：scanf()，sleep()，msgrcv()，操作外设的read()，write()，等等。

睡眠进程收到信号后处理

- 进程正在睡眠时收到信号，进程就会从睡眠中被惊醒，系统调用立即被半途终止，返回值-1， errno一般被置为EINTR

注意：有的系统调用在特殊情况（与内核中的代码相关尤其是驱动程序）下睡眠很深，信号到达不能将它惊醒(有的进程用kill -9也杀不死)

例：sleep(1000)返回，有可能只睡眠了10秒

pause与alarm系统调用

- **pause()**
  - 等待信号，进程收到信号前一直处于睡眠状态
- **设置进程报警时钟(闹钟)**

  int alarm(int *secs*)
  - 进程报警时钟存贮在它内核系统数据中，报警时钟到时，进程收到SIGALRM信号
    - 子进程继承父进程的报警时钟值。报警时钟在exec执行后保持这一设置
    - 进程收到SIGALRM后的默认处理是终止进程

      (可以利用这一功能，fork后exec前做设置，限制加载程序的执行时间)
  - alarm参数为*secs*
    - 当*secs*>0时，将时钟设置成***secs***指定的秒数
    - 当*secs*=0时，关闭报警时钟。

alarm系统调用举例

```
static char cmd[128];
void default_cmd(int sig)
{
    strcpy(cmd, "CMD_A");
}

int main(void) /* Filename alarm.c */
{
    siginterrupt(SIGALRM, 1);
    signal(SIGALRM, default_cmd);

    alarm(3);
    printf("Input command : ");
    scanf("%s", cmd);
    alarm(0);
    printf("\ncmd=[%s]\n", cmd);
}
```

全局跳转

案例

```
struct database …
int main(void)
{
    int c;
    for (;;) {
        printf("Input your choice:");
        scanf("%d", &c);
        switch(c) {
            case 0: return 0;
            case 1: func_1();   break;
            case 2: func_2();   break;
                    .
                    .
                    .
        }
    }
}
```

func_1()运行时间很长，用户在输入了命令1后反悔，希望中止对命令1的处理，重新选择另一条命令，而不是中止整个程序的运行

将主循环封装为子程序，利用信号解决问题

```
struct database …
void main_control(int sig)
{
    int c;
    for (;;) {
        printf("Input your choice:");
        scanf("%d", &c);
        switch (c) {
            case 0: return;
            case 1: func_1();  break;
            case 2: func_2();  break;
                    .
                    .
        }
    }
}
int main(void)
{
    signal(SIGINT, main_control);
    main_control();
}
```

问题

上述程序存在的问题

- 每次按下中断键，程序停留在信号捕捉函数中，堆栈没有清理，嵌套越来越深浪费越来越大
- main_control()一旦返回，进程的执行将弹回到刚才被SIGINT中断的地方恢复刚才的执行
  - 会让用户感到迷惑不解，刚才的动作已打断而且又已经开始了新的工作，可过一段时间后又开始运行
- C语言中的goto语句只限于一个函数体内使用，不能解决上述的问题

其他问题

- 零做除数问题

全局跳转

解决方法

- 把栈恢复为进程某一留存的状态，程序执行也跳转到此

有两个函数用于这个目的

```
#include <setjmp.h>

int sigsetjmp(jmp_buf jmpenv, int savemask);

    /* 返回值为零，或者是siglongjmp提供的val, savemask一般设为1 */

void siglongjmp(jmp_buf jmpenv, int val)

    /* val是提供给sigsetjmp作返回值*/
```

解决方案

```
struct database …
static jmp_buf env;
void intr_proc(int sig)
{
    printf("\n...INTERRUPTED by signal %d\n", sig)
    siglongjmp(env, 1);
    printf("siglongjmp Failed\n");
}
main()
{
    int c;
    sigsetjmp(env, 1);
    signal(SIGINT, intr_proc);
    for(;;) {
        printf("Input a command:");
        scanf("%d", &c);
        switch(c) {
            case 0: return;
            case 1: func_1();  break;
            case 2: func_2();  break;
        }
    }
}
```

全局跳转举例：0做除数

```c
/* file name quiz1.c */
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

#define N 10

static jmp_buf env;

static void div_error(int sig)
{
    siglongjmp(env, 1);
}

static int func(int a, int b)
{
    int answer;
    /*if (a - b == 0)
        siglongjmp(env, 1);*/
    printf("Try to calculate the answer\n");
    answer = (a + b) / (a - b);
    printf("Done\n");
    return answer;
}
```

```c
static int do_test(void)
{
    int x, y, z, ok;
    printf("Input x, y and yuor answer: ");
    scanf("%d%d%d", &x, &y, &z);
    ok = func(x, y) == z;
    printf("%s\n", ok ? "CORRECT" : "WRONG");
    return ok;
}

int main(void)
{
    int i, score = 0;
    signal(SIGFPE, div_error);
    for (i = 0; i < N; i++) {
        sigsetjmp(env, 1);
        printf("No.%d ", i + 1);
        score += do_test();
    }
    printf("Score: %d\n", score * 100 / N);
}
```

关于信号

- **破坏了“程序顺序执行”的模型，导致重入**
  - 例如：程序中有printf，在信号处理程序中也有printf
- **两种常见应用**
  - 进程收到信号后，做完相关后事处理，进程终止运行
  - 后台运行的网络服务进程等借用SIGHUP信号捕获以接受新编辑好的配置文件中的配置信息（重读配置文件）
- **函数sigaction()**
  - signal()的升级版本,使用起来更复杂
- **其他编程语言中的try-catch-finally结构**

信号:小结

- kill命令
- 进程组，分组的目的
- 信号机制
- 常用的信号
- 忽略信号
- 捕捉信号
- 信号的发送：kill的三种用法
- 系统调用和进程状态与信号关系
- sleep,pause,alarm
- 全局跳转的必要性
- 全局跳转的两个主要调用