Unix 环境高级编程

一．Unix 基础知识

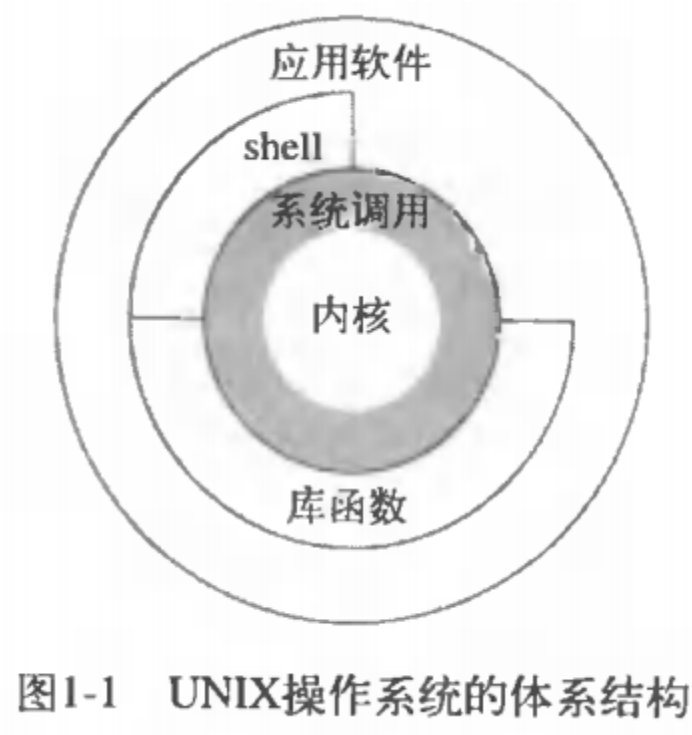


图1-1　UNIX操作系统的体系结构

内核的接口，被称为系统调用。公用函数库（可能指标准 io 库函数）构建在系统调用接口之上，应用软件既可使用公用函数库，也可使用系统调用。

Shell 是一种特殊的应用程序，它为运行其他应用程序提供了一个接口。当用户登录时，某些系统会启动一个视窗管理程序，但最终总会有一个 shell 程序运行在一个视窗中。Shell 是一个命令解释器，它读取用户输入，然后执行命令。用户通常用终端，或 shell 脚本向 shell 进行输入。

文件属性——指文件类型，大小，所有者，权限以及文件最后的修改时间等。

文件名——不能出现在文件名中的字符只有斜线（/）和空操作符（null）两个。

工作目录——每个进程都有一个工作目录，也称当前工作目录。是进程的属性，所以在子进程中更改工作目录，不会影响到父进程。

起始目录——登录时，工作目录设置为起始目录，即家目录，从口令文件中取得。是登录名的一个属性。

文件描述符——通常是一个非负整数，内核用它标识一个特定进程正在访问的文件。

每当运行一个新程序，所有的 shell 都为其打开三个文件描述符：标准输入，输出，出错。它们都重定向到某个文件。

系统调用函数——提供了不用缓冲的 i/o，不用缓冲即直接进行系统调用。

标准 I/O 函数——提供一种对不用缓冲 i/o 函数的带缓冲的接口。优点：①无需考虑选取最佳的标准 I/O 缓冲区（并非系统调用函数中的 BUFFSIZE，在一个流上执行第一次 I/O 操作时，由标准 I/O 函数调用 malloc 获得需使用的缓冲区）大小,若流是带缓冲的，而用户并未指定标准 I/O 缓冲区大小（如调用 setvbuf 函数时，参数 buf 是 NULL），则标准 I/O 库将自动为该流分配适当长度的缓冲区，其大小一般由 stat 结构中的成员 st_blksize 指定（详见 P112）；②简化了对输入行的处理，如 fgets 能读一完整的行，read 只能读指定字节。

程序——存放在磁盘上，处于某个目录中的一个可执行文件，如 a.out。使用 6 个 exec 函数中的一个由内核将程序读入存储器，并使其执行。

程序文件——指程序源码，如 a.c。

进程——即程序的执行实例。有些系统成为任务。

进程 ID——每个进程都有的唯一的一个数字标识符，是一个非负整数。

用于进程控制的三个主要函数——fork、exec 和 waitpid.(exec 函数有六种，统称 exec)

线程

出错处理：

对于 errno 应当知道两条规则：①如果没出错，其值不会被一个例程清楚。仅当函数的指明出错时，才检验其值。②任一函数都不会将 errno 设置为 0。

C 标准定义了两个函数，它们帮助打印出错信息——strerror（），perror（）。但此书中的实例都不直接调用这两个函数，而是使用附录 B 中的出错函数。

出错恢复：分为致命性与非致命性的，对于非致命性的，恢复动作是延迟一会。

用户 ID——用于系统标识各个不同的用户，ID 为 0 的用户为根用户，即超级用户。是一个数值。

组 ID——也是一个数值。对于用户来说，使用名字比使用数值方便，所以口令文件包含了登录名和用户 ID 之间的映射关系，而组文件则包含了组名和组 ID 之间的映射关系。

附加组 ID——一个账号所属的所有组。

/etc/passwd 中：包含用户 ID，初始用户组 ID（可能就是有效用户组 ID）

/etc/group 中：包含一个用户所属的所有组（附加组）。

信号

时间值：（可用 time（1）指令取得）

日历时间——该值是自 1970 年一月一日 00:00:00 以来国际标准时间所经过的秒数累计值。系统基本数据类型 time_t 用于保存这种时间值。

进程时间——即 cpu 时间（用户 cpu 时间与系统 cpu 时间之和），用以度量进程使用的中央处理机资源。以时钟滴答计算，系统基本数据类型 clock_t 用于保存这种时间值。当度量一个进程的执行时间时，unix 系统使用以下三个进程时间值：

①时钟时间——又称墙上时钟时间，是进程运行的时间总量，以时钟滴答计算。

②用户 cpu 时间——执行用户指令所用的时间（系统调用之前）。

③系统 cpu 时间——为该进程执行内核程序所经历的时间。

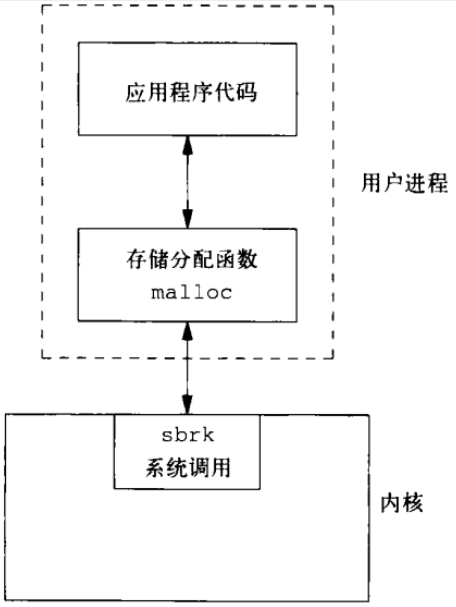


图1-2 malloc函数和sbrk系统调用

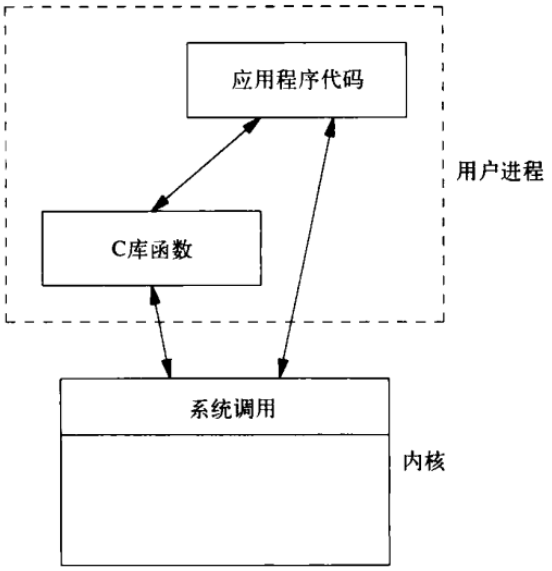


图1-3 C库函数和系统调用之间的差别

系统调用与库函数——两者职责不同；系统调用通常提供一种最小接口，而库函数通常提供比较复杂的功能。我们可以替换库函数，但不能替换系统调用。

三．文件 I/O

各种 shell 以及很多应用程序使用的惯例：文件描述符 0——进程的标准输入
文件描述符 1——进程的标准输出
文件描述符 2——进程的标准出错

文件描述符的变化范围是 0~OPEN_MAX。

在依从 POSIX 的应用程序中，幻数 0、1、2 应当替换成符号常量 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO。

_POSIX_NO_TRUNC 决定了是要截短过长的文件名或路径名，还是返回一个出错。若其有效，则不允许截短，直接返回出错状态。

open 函数——用于打开或创建一个文件。由其返回的文件描述符一定是最小的未用描述符值。见 P48。

creat 函数——用于创建一个新文件。其不足之处是它是以只写方式打开所创建的文件。见 P49。

close 函数——关闭一个打开的文件。当一个进程终止时，内核会自动关闭它所有打开的文件。见 P50

lseek 函数——显式的为一个打开的文件设置其偏移量。仅将当前的文件偏移量记录在内核中，并不引起任何 I/O 操作。见 P50。

每个打开的文件都有一个与其相关联的“当前文件偏移量”，是一非负整数，用以度量从文件开始处计算的字节数。按系统默认，当打开一个文件，除非指定 O_APPEND 选项，否则该偏移量被设置为 0。文件偏移量可以大于文件的当前长度，在这种情况下，对该文件的下一次写将加长该文件，并在文件中构成一个空洞，未写过字节被读为 0。空洞不需要在磁盘上占用存储区，即不需要分配磁盘块。（这会造成两文件长度可能相同，但由于占用的磁盘块数不同，文件大小不一样。）

read 函数——从打开文件中读数据。读操作从文件的当前位移量处开始，在成功返回之

前，该位移量增加实际读得的字节数。见 P53。

有多种情况可使实际读到的字节数少于要求读字节数：

①读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有 30 个字节，而要求读 100 个字节，则 read 返回 30，下一次再调用 read 时，它将返回 0(文件尾端)。

②当从终端设备读时，通常一次最多读一行。

③当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。

④当从管道或 FIFO 读时，如若管道包含的字节少于所需的数量，那么 read 将只返回实际可用的字节数。

⑤某些面向记录的设备，例如磁带，一次最多返回一个记录。

⑥当某一信号造成中断，而已经读了部分数据量时。

write 函数——向打开的文件写数据。见 P54。

文件共享：

内核使用三种数据结构表示打开的文件：

①每个进程在进程表中都有一个记录项，其中包含一张打开文件描述符表。

②内核为所有打开文件维持一张文件表。

③每个打开文件（或设备）都有一个 V 节点结构。

V 节点（与文件系统类型无关的 i 节点，与虚拟文件系统有关）

V 节点信息——包含了文件类型和对此文件进行各种操作的函数的指针。这些信息是在打开文件时从磁盘上读入内存的，快速供使用。

i 节点（依赖于文件系统类型的 i 节点，与实际文件系统有关）

i 节点信息——包含了文件的所有者、文件长度、文件所在的设备、指向文件实际数据块在磁盘上所在位置的指针等等。

Linux 操作系统没有使用 V 节点，而是使用了通用 i 节点结构，采用了一个独立于文件系统的 i 节点和一个依赖于文件系统的 i 节点。虽然实现有所不同，但在概念上，V 节点指针与 i 节点指针一样，都指向文件系统特有的 i 节点结构。

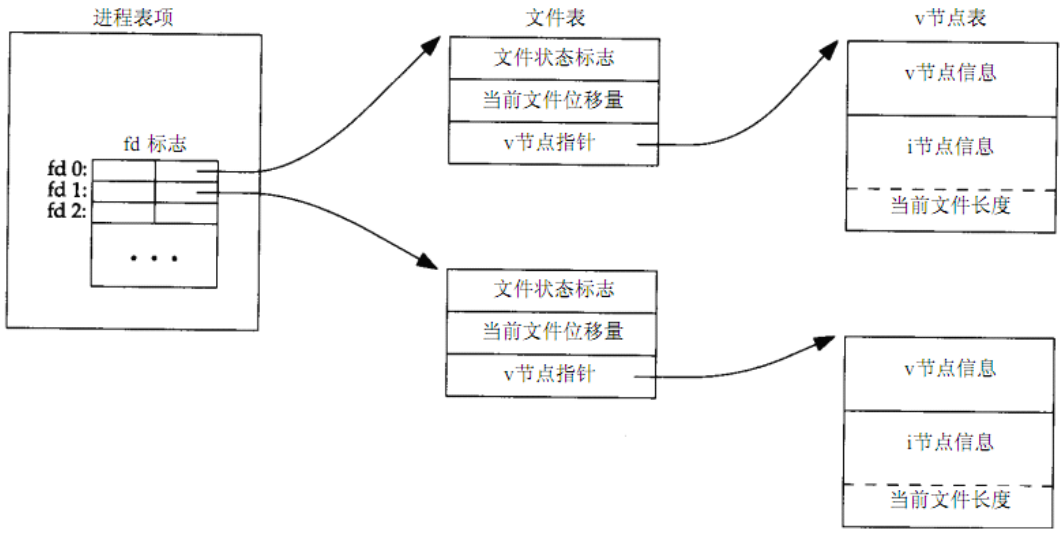


图3-1　打开文件的内核数据结构

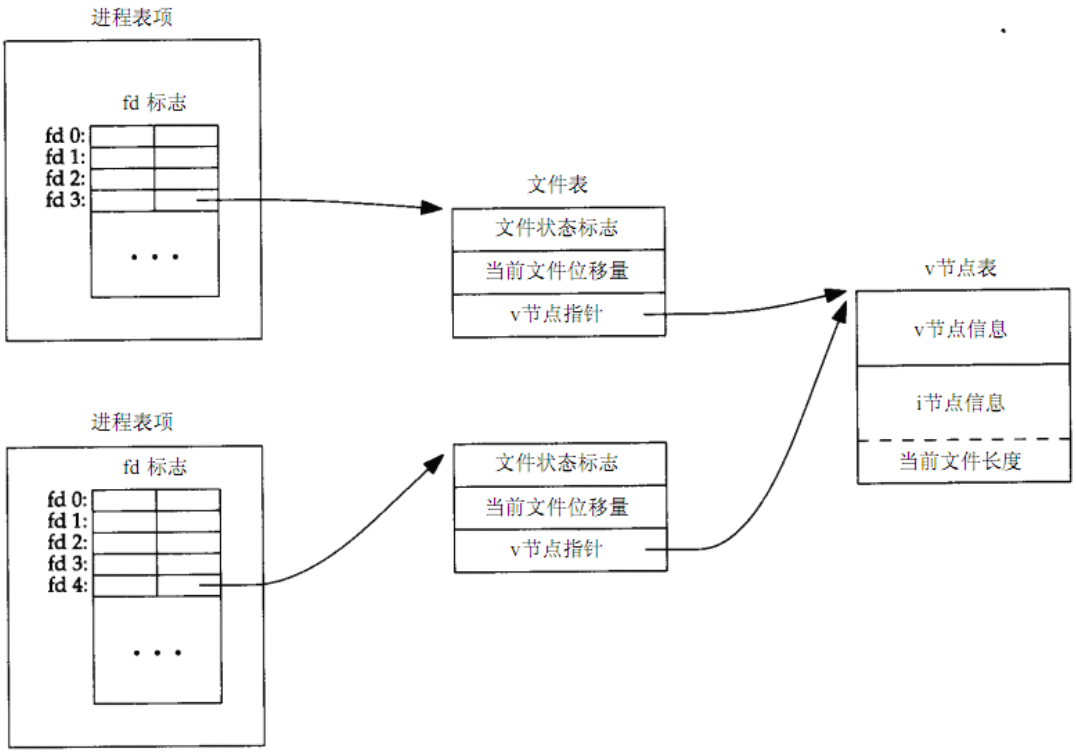


图3-2　两个独立进程各自打开同一个文件

文件描述符标志与文件状态标志的区别：前者只用于一个进程的一个描述符，而后者则适用于指向该给定文件表项的任何进程中的所有描述符。

原子操作：

①添写至一个文件：使用 open 函数的选项 O_APPEND

②pread 函数——顺序调用 lseek 与 read；pwrite 函数——顺序调用 lseek 与 write

③创建一个文件：使用 open 函数的选项 O_CREAT 和 O_EXCL。

dup 函数，dup2 函数——复制一个现存的文件描述符。见 P60。

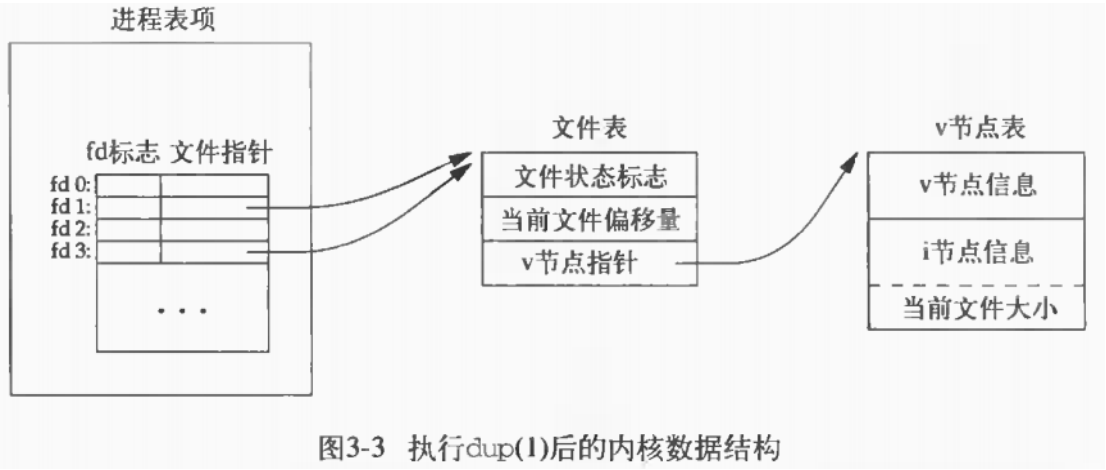


图3-3　执行dup(1)后的内核数据结构

sync、fsync 和 fdatasync 函数——保证磁盘上实际文件系统与缓冲区高速缓冲中内容的一致性。见 P61。fsync 和 fdatasync 函数与 0_SYNC 标志比较，fsync 和 fdatasync 在我们需要时（调用）即可更新文件内容，O_SYNC 标志则在我们每次写（write）至文件时就更新文件内容。Ext2 文件系统并未真正实现 O_SYNC 标志功能。

fcntl 函数——可以改变已打开的文件的性质。见 P62。

ioctl 函数——I/O 操作的杂物箱。见 P67。每个设备驱动程序都可以定义它自己专用的一组 ioctl 命令。系统则为不同种类的设备提供通用的 ioctl 命令。

/dev/fd：较新的系统都提供此目录，其目录项是名为 0、1、2 等的文件。在下列调用中 fd=open（"/dev/fd/0",mode）；大多数系统忽略它所指定的 mode，而另外一些则要求 mode 必须是所涉及的文件原先打开时所使用 mode 的子集。

某些系统提供路径名/dev/stdin、/dev/stdout 和/dev/stderr。这些等效于/dev/fd/0、/dev/fd/1 和/dev/fd/2。

“-”号——可解释为标准输入，但不建议使用，建议还是使用/dev/fd/0。

四．文件和目录

stat、fstat、lstat 函数——返回与给定文件相关的信息结构。lstat 函数不跟随符号链接。其中的参数 buf 指向返回的信息结构，其基本形式如下：

```
struct stat{
    mode_t    st_mode;      //包含文件类型信息，设置用户 ID 位，设置组 ID
                            位及针对文件的 9 个访问权限位
    ino_t     st_ino;
    dev_t     st_dev;       //文件系统的设备号,分主次
    dev_t     st_rdev;      //实际设备的设备号，分主次，只有字符特殊文件和块
                            特殊文件才有
    nlink_t   st_nlink;     //链接计数，其值是指向该文件对应 i 节点的目录项数
    uid_t     st_uid;       //包含文件的所有者
    gid_t     st_git;       //包含文件的组所有者
    off_t     st_size;      //以字节为单位的文件长度，只对普通文件、目录文件
                            和符号链接有意义
    time_t    st_atime;
    time_t    st_mtime;
    time_t    st_ctime;
    blksize_t st_blksize;   //对文件 I/O 较合适的块长度，标准 I/O 库也试图一次
                            读写 st_blksize 个字节
    blkcnt_t  st_blocks;    //为此文件所分配的实际 512 字节块数量
```

};

表4-5 9个访问权限位，取自<sys/stat.h>

st_mode屏蔽	意 义
S_IRUSR	用户-读
S_IWUSR	用户-写
S_IXUSR	用户-执行
S_IRGRP	组-读
S_IWGRP	组-写
S_IXGRP	组-执行
S_IROTH	其他-读
S_IWOTH	其他-写
S_IXOTH	其他-执行

文件类型：有以下几种

①普通文件——至于它是文本还是二进制，对于 unix 内核没有区别，对其内容的解释由处理该文件的应用程序进行。

②目录文件——它包含了其他文件的名字以及指向与这些文件有关信息的指针。只有内核才能直接写目录文件，进程不可以，进程必须使用本章说明的函数才能更改目录。

③块特殊文件（即块设备文件）——提供对设备带缓冲的访问。每次访问以固定长度为单位进行。

④字符特殊文件（即字符设备文件）——提供对设备不带缓冲的访问。每次访问长度可变。

（系统的所有设备要么是字符特殊文件，要么是块特殊文件。）

⑤FIFO（命名管道）——用于进程间通信。

⑥套接字——用于进程间网络通信，也可用于一台宿主机上进程间的非网络通信。

⑦符号链接——指向另一个文件。

表4-1 <sys/stat.h>中的文件类型宏

宏	文 件 类 型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件
S_ISFIFO()	管道或FIFO
S_ISLNK()	符号链接
S_ISSOCK()	套接字

上表中的宏用于检测一个文件的文件类型，参数是 stat 结构中的 st_mode。

实际用户 ID 和实际组 ID（唯一）——取自口令文件/etc/passwd，标识我们究竟是谁。

有效用户 ID，有效组 ID 以及附加组 ID——决定了我们的访问权限。属于进程的性质。

保存的设置用户 ID 和保存的设置组 ID（唯一）——在执行一个程序时包含了有效用户 ID 和有效组 ID 的副本，由 exec 函数保存。

文件的所有者 ID 和文件的组所有者 ID（唯一）——每个文件都存在。属于文件的性质。

文件的文件模式字（st_mode）中：

设置用户 ID 位——当执行此文件时，将进程的有效用户 ID 设置为文件所有者的 ID。可用 S_ISUID 宏测试此位是否设置，参数为 st_mode。

设置组 ID 位——当执行此文件时，将进程的有效组 ID 设置为文件组所有者的 ID。可用 S_ISGID 宏测试此位是否设置，参数为 st_mode。

（通常有效用户 ID 等于实际用户 ID，有效组 ID 等于实际组 ID。）

进程每次打开、创建或删除一个文件时，内核进行的测试：

①若进程的有效用户 ID 是 0(超级用户)，则允许访问。

②若进程的有效用户 ID 等于文件的所有者 ID，那么：若所有者适当的访问权限位被设置，则允许访问，否则拒绝访问。适当的访问权限位是指，若进程为读而打开该文件，则用户读位应为 1。

③若进程的有效组 ID 或进程的附加组 ID 之一等于文件的组 ID，那么：若组适当的访问权限位被设置，则允许访问，否则拒绝访问。

④若其他用户适当的访问权限位被设置，则允许访问，否则拒绝访问。

以上四步按顺序执行，只要有一条满足，则后面几条不执行。

当我们创建一个新文件（指文件或目录）时：

新文件的所有者 ID=进程的有效用户 ID

新文件组所有者 ID=以下两者之一

①进程的有效组 ID。 (当该文件所在目录的设置组 ID 位未设置)

②新文件所在目录的组 ID。 (当该文件所在目录的设置组 ID 位已设置)

access 函数——测试文件是否具有某种权限或文件是否存在。见 P78。

与 open 函数的区别在于 open 函数在使用上述四步进行访问权限测试时，采用有效用户 ID 和有效组 ID 进行测试，而 access 函数使用实际用户 ID 和实际组 ID 进行测试。

umask 函数——为进程设置文件模式创建屏蔽字，并返回以前的值。参数是 9 个权限位中的若干个按位“或”。见 P79。

当我们使用 open 函数的 mode 参数指定新文件的访问权限位时，对于任何在文件模式创建屏蔽字中为 1 的位，在文件 mode 中的相应位则一定被关闭。大多数用户并不处理 umask 值，通常在登录时，由 shell 的启动文件设置一次，然后从不改变。尽管如此，当编写创建新文件的程序时，如果我们想确保指定的访问权限位已经激活，那么必须在进程运行时修改 umask 值。更改进程的 umask 值并不影响其父进程的屏蔽字。

chmod 函数、fchmod 函数——可更改现有文件的访问权限，条件是进程的有效用户 ID 必须等于文件的所有者 ID，或该进程具有超级用户权限。fchmod 函数的操作对象必须是打开的文件。参数 mode 为下面若干位按位“或”。见 P81。

表4-8 chmod函数的*mode*常量，取自<sys/stat.h>

mode	说　明
S_ISUID	执行时设置用户ID
S_ISGID	执行时设置组ID
S_ISVTX	保存正文（粘住位）
S_IRWXU	用户（所有者）读、写和执行
S_IRUSR	用户（所有者）读
S_IWUSR	用户（所有者）写
S_IXUSR	用户（所有者）执行
S_IRWXG	组读、写和执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IRWXO	其他读、写和执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

如果新创建的文件的组ID不等于进程的有效组ID或进程附加组ID中的一个，以及进程没有超级用户权限，那么当调用chmod函数时，新文件的设置组ID位将自动被关闭。如果没有超级用户特权的进程写一个文件，则文件的设置用户ID位和设置组ID位将自动被清除。

chown、fchown、lchown函数——用于更改文件的用户ID和组ID，lchown函数不跟随符号链接。见P84。但具备以下条件限制。

若_POSIX_CHOWN_RESTRICTED对指定的文件起作用（linux默认一直起作用），则：（1）只有超级用户进程能更改该文件的用户ID。

（2）若满足下列条件，一个非超级用户进程就可以更改该文件的组ID

（a）参数owner等于-1或文件的用户ID，并且参数group等于进程的有效组ID或进程的附加组ID之一。

（b）进程拥有此文件（其有效用户ID等于该文件的用户ID）。

即意味着非超级用户只能更改你所拥有的文件的组ID，但只能改到你所属的组。

上述这些函数若被非超级用户进程调用，则在成功返回时，该文件的设置用户ID位和设置组ID位都会被清除。

文件长度：

①对于普通文件，其文件长度可以是 0，在读这种文件时，将得到文件结束指示。

②对于目录文件，其长度通常是一个数（例如 16 或 512）的倍数。

③对于符号链接，文件长度是该文件（一级）指向的文件名中的实际字节数。

“ls -l core”指令显示的文件长度是包括空洞一起的字节数。

“du -s core”指令显示的是不包括文件空洞在内的实际数据所占有的块数。

正常的 I/O 操作读整个文件长度，所以当我们复制文件 core 的内容时，系统会将其中的空洞先用 0 填充，变成占用磁盘块的空间，再将其所有数据复制到另外一个文件 core1 中。再用“du -s”指令查看时，会发现文件 core1 所占用的块数比 core 中包括空洞在内所占用的块数还多一些，主要是因为文件系统多使用了若干块以存放指向实际数据块（可能是指文件中除空洞外的其他数据块）的各个指针。

truncate 函数、ftruncate 函数——把现有的文件长度截短为 length 字节，length 为其参数。见 P86。若以前长度短于 length，遵循 XSI 的系统将增加该文件的长度，即在该文件尾端后面创建一个空洞，空洞部分数据将读作 0。

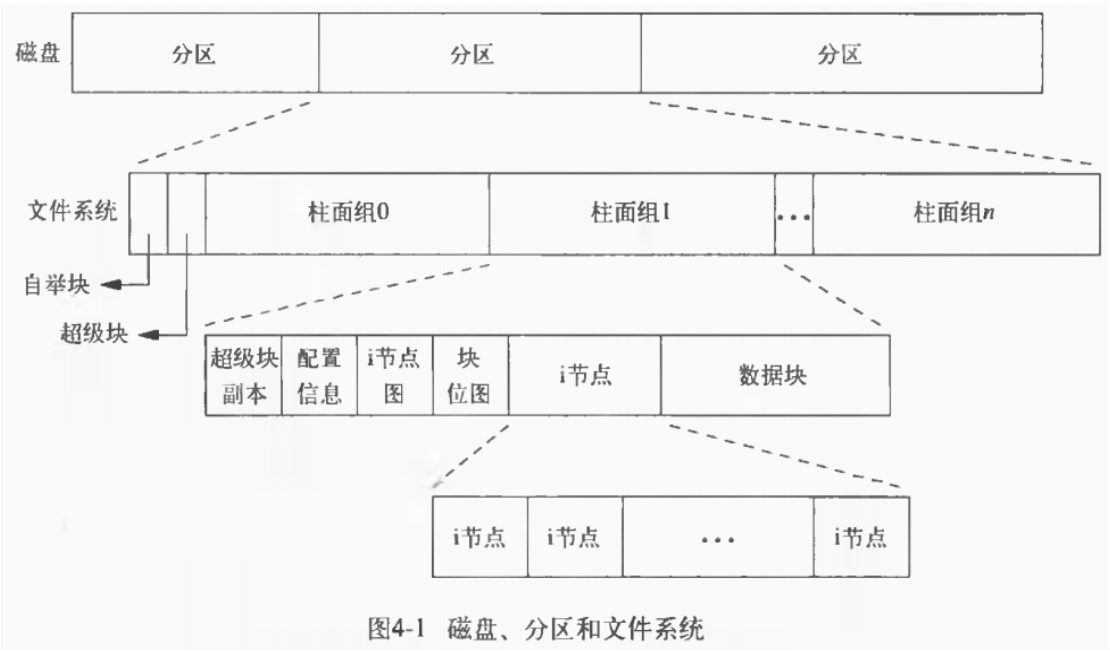


图4-1 磁盘、分区和文件系统

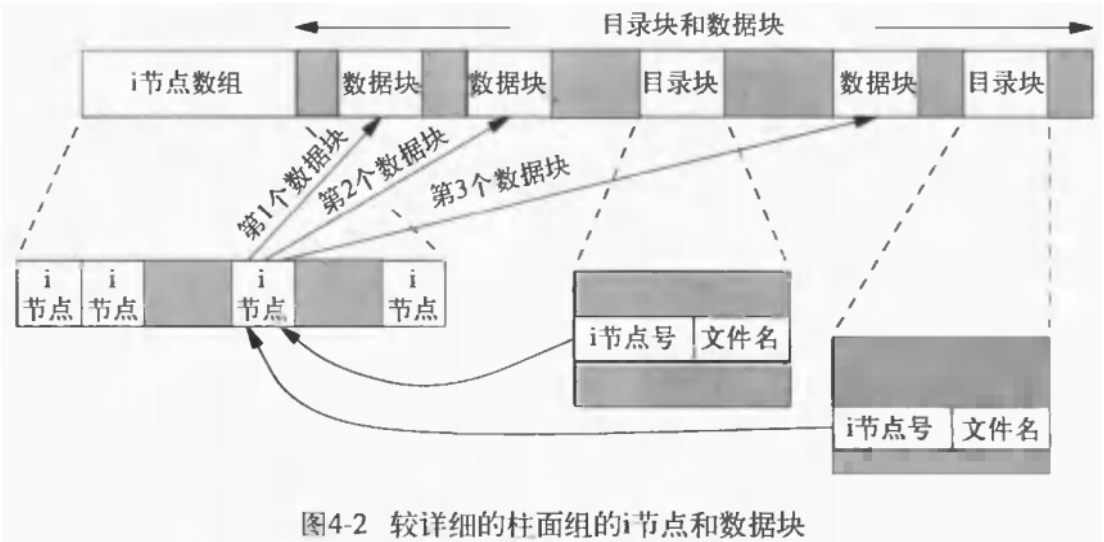


图4-2 较详细的柱面组的i节点和数据块

上图显示的是一种硬链接。而对于符号链接，该文件的实际内容（在数据块中）包含了

该符号链接所指向的文件的名字。

上图中的 i 节点包含了大多数与文件有关的信息，如链接计数、文件访问权限位、文件长度等，估计是前面的 V 节点信息与 i 节点信息的结合，stat 结构中的大多数信息都取自 i 节点。只有 i 节点号和文件名放在目录项中，不能使一个目录项指向另一个文件系统的 i 节点（所以 ln 指令不能跨越文件系统）。

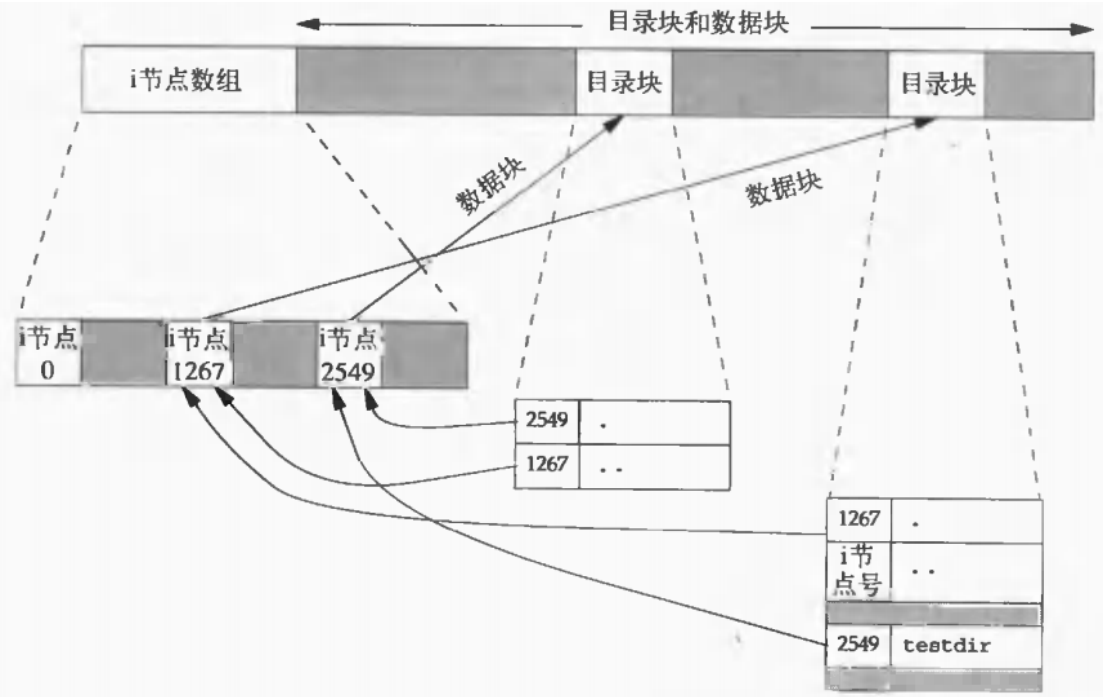


图4-3 创建了目录testdir后的示例柱形组

上图是针对目录文件的，任何一个叶目录（不包含任何其他目录和文件的目录）的链接计数总是 2。

link 函数——创建一个指向现有文件的链接，即创建硬链接。原子性的创建新目录项以及增加链接计数。见 P89。

但硬链接有两个限制：①不能跨越文件系统。

②只有超级用户才能创建指向一个目录的硬链接。

unlink 函数——原子性的删除一个现有的目录项并将由参数 pathname 所引用文件的链接计数减 1。条件是必须对包含该目录项的目录具有写和执行权限。见 P89。（超级用户调用 unlink 时，其参数可以指定一个目录。）

只有当链接计数达到 0 时，该文件的内容才可被删除。然而当有进程打开了该文件，即使链接计数为 0，其内容也不能删除。关闭一个文件时，内核首先检查打开该文件的进程数，如果该进程数达到 0，再检查其文件链接数，如果也为 0 就删除该文件内容。

remove 函数——解除对一个文件或目录的链接。对于文件，其函数功能相当于 unlink 函数；对于目录，相当于 rmdir 函数。见 P90。

rename 函数——为文件或目录更名。见 P91。

表4-9 各个函数对符号链接的处理

函 数	不跟随符号链接	跟随符号链接
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

注意：①其中的 chown 函数，linux 系统从 2.1.81 版开始，chown 就跟随符号链接了。

②同时用 O_CREAT 和 O_EXCL 两者调用 open 函数时，若路径名引用符号链接，open 将出错返回。否则当引用符号链接时，open 将跟随符号链接，但若此符号链接所指向的文件并不存在，则 open 返回出错。

symlink 函数——创建一个符号链接。见 P94。

readlink 函数——不跟随符号链接，打开该链接文件本身，并读该链接中的名字。见 P94。

与每个文件相关的三个时间值：

①st_atime——文件数据的最后访问时间。

②st_mtime——文件数据的最后修改时间。

③st_ctime——i 节点状态（即文件属性）的最后更改时间。

表4-11　各种函数对访问、修改和更改状态时间的作用

函　数	引用的文件或目录			所引用文件或目录的父目录			节	备　注
	a	m	c	a	m	c		
chmod、fchmod			•				4.9	
chown、fchown			•				4.11	
creat	•	•	•		•	•	3.4	O_CREAT新文件
creat		•	•				3.4	O_TRUNC现有文件
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	第二个参数的父目录
mkdir	•	•	•		•	•	4.20	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	O_CREAT新文件
open		•	•				3.3	O_TRUNC现有文件
pipe	•	•	•				15.2	
read	•						3.7	
remove			•		•	•	4.15	删除文件 =unlink
remove					•	•	4.15	删除目录=rmdir
rename			•		•	•	4.15	对于两个参数
rmdir					•	•	4.20	
truncate、ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utime	•	•	•				4.19	
write		•	•				3.8	

utime 函数——可更改文件的访问和修改时间的值，而更改状态时间 st_ctime 的值不能指定，在调用此函数时，它会自动更新。见 P96。

mkdir 函数——创建一个空目录，其中 • 和 •• 目录项是自动创建的，所指定的文件访问权限 mode 由进程的文件模式创建屏蔽字修改。见 P97。

rmdir 函数——删除一个空目录，空目录是只包含 • 和 •• 目录项的目录。见 P98。只有当调用此函数使目录的链接计数成为 0，并且也没有其他进程打开此目录，才释放由此目录占用的空间。

读目录：P98

对某个目录具有访问权限的任一用户都可以读该目录，只有内核才能写目录。

下表是用于读目录的相关函数：（具体可参考 linux 常用 C 函数）

①由 opendir 函数返回的 DIR 结构的指针由另外 5 个函数使用。

②DIR 结构是一个内部结构，下述 6 个函数用这个内部结构保存当前正被读的目录的有关信息。

③DIR 结构如下

```
struct dirent {
  ino_t  d_ino;                 /* i-node number */
  char   d_name[NAME_MAX + 1];  /* null-terminated filename */
}
```

```
#include <dirent.h>
DIR *opendir(const char *pathname);
                                        返回值：若成功则返回指针，若出错则返回NULL
struct dirent *readdir(DIR *dp);
                                    返回值：若成功则返回指针，若在目录结尾或出错则返回NULL
void rewinddir(DIR *dp);
int closedir(DIR *dp);
                                                返回值：若成功则返回0，若出错则返回-1
long telldir(DIR *dp);
                                                    返回值：与dp关联的目录中的当前位置
void seekdir(DIR *dp, long loc);
```

chdir 函数、fchdir 函数——可以更改当前工作目录。见 P102。

getcwd 函数——内核为每个进程只保存指向该目录 V 节点的指针等目录本身的信息，并不保存该目录的完整文件名。而此函数可以根据内核给出的当前工作目录，逐层上移，推到根，最后返回当前工作目录的绝对路径名。见 P103。

fchdir 函数——提供比 getcwd 函数更方便的功能保存当前工作目录。见 P104。

每个文件所在的存储设备都由其主、次设备号表示。主设备号标识设备驱动程序，次设备号标识特定的子设备。我们一般使用宏 major 和 minor 来访问主次设备号。

表4-12 文件访问权限位小结

常 量	说 明	对普通文件的影响	对目录的影响
S_ISUID	设置用户ID	执行时设置有效用户ID	（不使用）
S_ISGID	设置组ID	若组执行位设置，则执行时设置有效组ID，否则使强制性记录锁起作用（若支持）	将在目录中创建的新文件的组ID设置为目录的组ID
S_ISVTX	粘住位	在交换区保存程序正文（若支持）	限制在目录中删除和更名文件
S_IRUSR	用户读	许可用户读文件	许可用户读目录项
S_IWUSR	用户写	许可用户写文件	许可用户在目录中删除和创建文件
S_IXUSR	用户执行	许可用户执行文件	许可用户在目录中搜索给定路径名
S_IRGRP	组读	许可组读文件	许可组读目录项
S_IWGRP	组写	许可组写文件	许可组在目录中删除和创建文件
S_IXGRP	组执行	许可组执行文件	许可组在目录中搜索给定路径名
S_IROTH	其他读	许可其他读文件	许可其他读目录项
S_IWOTH	其他写	许可其他写文件	许可其他在目录中删除和创建文件
S_IXOTH	其他执行	许可其他执行文件	许可其他在目录中搜索给定路径名

五．标准 I/O 库

假设调用 fgetc（或 fgets）和 fputc（或 fputs）函数从标准输入（连至终端）复制数据至标准输出（连至终端），系统进行以下过程：①③③④（其中 a、b、c、d 可想象成四个开关）

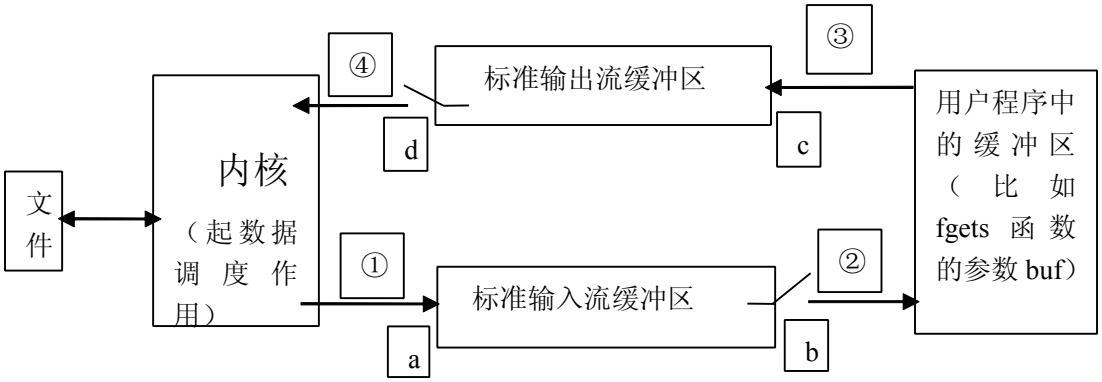
比如在终端输入 aaaa，然后在 enter 之后：

①首先 a 闭合、b 断开，数据会先存入标准输入对应的字符设备文件中，之后由 read 函数尽可能多的将文件中的数据预读到标准输入流缓冲区中（数据通过内核调度）。

②当标准输入流缓冲区已满或里面复制来一个换行符，或流缓冲区已满，则 a 断开、b 闭合，调用 fgetc 或 fgets 将标准输入流缓冲区的数据读到用户程序中的缓冲区中，直至读完，标准输入流缓冲区为空。此时 a 再闭合、b 断开。

③之后 c 闭合、d 断开，调用 fputc 或 fputs 先将用户程序中的缓冲区中的数据往标准输出流缓冲区里送，直到标准输出流缓冲区已满或里面送来了一个换行符，则断开 c、闭合 d。

④最后调用 write 函数将标准输出流缓冲区中的数据写入标准输出对应的字符设备文件中（数据通过内核调度），直至标准输出流缓冲区为空。再断开 d，闭合 c。



本章概括性图示

流缓冲区（即标准 I/O 缓冲区、有三种缓冲类型）和流是针对单一文件的，是文件的属性，而非 I/O 函数的属性，每个文件都有一个流。但是当一个流是带缓冲时，其缓冲区由系统分配（通常调用 malloc），也可由用户通过标准 I/O 库函数分配（如调用 setvbuf 函数）。

对于标准 I/O 库，它们的操作是围绕“流”进行的。当用标准 I/O 库打开或创建一个文件时，我们已使一个流与一个文件相关联。流的定向决定了所读、写的字符是单字节还是多字节。流最初被创建时，无定向，若此时在流上使用一个多字节 I/O 函数，此流的定向会被设置为宽定向（即多字节定向）；若使用一个单字节 I/O 函数，则被设置为字节定向的。

freopen 函数——清除一个流的定向。

fwide 函数——设置流的定向，但并不改变已定向流的定向。见 P109。

当打开一个流，标准 I/O 函数 fopen 返回一个指向 FILE 对象的指针。该 FILE 对象通常是一个结构，包含了标准 I/O 库位管理该流所需要的所有信息，FILE 对象包括：

①用于实际 I/O 的文件描述符。

②指向用于该流缓冲区的指针。

③缓冲区的长度。

④当前在缓冲区中的字符数以及出错标志等等。

文件指针——指向 FILE 对象的指针。为引用一个流，就将该文件指针作为参数传递给每个标准 I/O 函数。

标准输入、标准输出和标准出错通过预定义文件指针 stdin、stdout 和 stderr 加以引用，这些流引用的文件与文件描述符 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 所引用的文件相同，对于一个进程首先预定义了这三个流。

标准 I/O 提供三种类型的缓冲：

①全缓冲——填满标准 I/O 缓冲区后才进行实际 I/O 操作。

在一个流上执行第一次 I/O 操作时，通常 I/O 函数调用 malloc 获得需使用的缓冲区。

冲洗（flush）——在标准 I/O 库方面，冲洗意味着将缓冲区的内容写到磁盘上。在终端驱动程序方面，则表示丢弃已存储在缓冲区中的数据。

②行缓冲——当在输入和输出中遇到换行符时，执行 I/O 操作。

对于行缓冲有两个限制：第一，因为标准 I/O 库用来收集每一行的缓冲区的长度是固定的，所以只要填满了缓冲区，即使还没有写一个换行符，也进行 I/O 操作。第二，任何时候只要通过标准 I/O 库要求从（a）一个不带缓冲的流，或者（b）一个行缓冲的流（它要求从内核得到数据）得到输入数据，那么就会造成冲洗所有行缓冲输出流。

③不带缓冲——标准 I/O 库不对字符进行缓冲存储。

很多系统默认下列类型的缓冲：

①标准出错是不带缓冲的。

②如若是涉及终端设备的其他流（如标准输入、标准输出），则它们是行缓冲的；否则是全缓冲的。

③当标准输入、标准输出连至终端时，它们是行缓冲的，当将这两个流重定向到普通文件时，它们就变成是全缓冲的（其缓冲长度是该文件系统优先选用的 I/O 长度，即 st_blksize）。标准错误总是不带缓冲的，不论其连至的是终端还是普通文件。

setbuf 函数、setvbuf 函数——改变流的缓冲类型。必须在流打开后才能调用。见 P111。

如果在一个函数内分配一个自动变量类的标准 I/O 缓冲区，则从该函数返回之前，必须关闭该流；有些实现将缓冲区的一部分用于存放它自己的管理操作信息，所以可以存放在缓冲区中的实际数据字节数小于 size（size 为用户调用 setvbuf 函数分配的流缓冲区长度）；一般来说，应由系统选择缓冲区的长度，并自动分配缓冲区，在这种情况关闭流时，标准 I/O 库将自动释放缓冲区。

fflush 函数——强制冲洗一个流，使该流所有未写的数据都被传送至内核。若其参数 fp 是 NULL，则此函数导致所有输出流被冲洗。见 P112。

fopen 函数——打开一个指定的文件。见 P112。

freopen 函数——在一个指定的流上打开一个指定的文件。见 P112。一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。

fdopen 函数——获取一个现有的文件描述符，并使一个标准的 I/O 流与该描述符结合。见 P112。此函数常用于由创建管道和网络通信通道函数返回的描述符。

fclose 函数——关闭一个打开的流。见 P114。在该文件被关闭之前，会自动冲洗缓冲区中的输出数据，丢弃缓冲区中的任何输入数据。如果是标准 I/O 库会该流自动分配的一个缓冲区，则释放此缓冲区。

一个进出正常终止时，所有带未写缓冲数据的标准 I/O 流都会被冲洗，进程打开的所有标准 I/O 流都会被关闭。

非格式化 I/O：一旦打开了一个流，可用以下三种不同类型的非格式化 I/O 中进行选择，

对其进行读写操作：

①每次一个字符的 I/O——每次读或写一个字符，如果流是带缓冲的，则标准 I/O 函数会处理所有缓冲。

getc、fgetc 和 getchar 函数——用于从流缓冲区一次读一个字符。见 P115。由于不管是出错还是到达文件尾端，这三个函数都返回同样的值，为区分这两种情况，需调用 ferror 和 feof 函数。

clearerr 函数——大多数实现都为每个流在 FILE 对象中维持了两个标志：出错标志和文件结束标志。此函数则用于清除这两个标志。见 P115。

ungetc 函数——从流中读取数据以后，此函数可将字符再压送回流缓冲区中。见 P115。一次只送回一个字符，而回送的字符不必是上次读到的字符。不能回送 EOF。但是当已经到达文件尾端时，仍可以回送一个字符，下次读将返回该字符，再次读则返回 EOF，原因是一次成功的 ungetc 调用会清除该流的文件结束标志。压送回流中的字符以后又可从流中读出，但读出的顺序与压送回的顺序相反。

putc、fputc 和 putchar 函数——用于一次写一个字符到流缓冲区中。见 P116。对应 get、fgetc 和 getchar 函数。

②每次一行的 I/O——每次读写一行，每行都以换行符终止，比如 fgets 和 fputs。

fgets 和 gets 函数——用于从流缓冲区每次读出一行到参数 buf 中。见 P116。gets 函数不推荐使用。

fputs 和 puts 函数——用于每次输出一行到流缓冲区中。但并不一定每次都输出一行。见 P117。

标准 I/O 库与直接调用 read 和 write 函数相比并不慢很多。使用每次一行 I/O 版本其速度大约是每次一个字符版本的两倍。

③直接 I/O（即二进制 I/O）——每次 I/O 操作读或写某种数量的对象，而每个对象具有指定的长度。如 fread 和 fwrite。使用直接 I/O 的基本问题是，它只能用于读在同一系统上已写的数据，在不同系统之间交换二进制数据的实际解决方法是使用较高级的协议。

fread 和 fwrite 函数——读或写一个二进制数组或一个结构，还可结合起来读写一个结构数组。

有三种方法可定位标准 I/O 流：

①ftell 和 fseek 函数——它们都假定文件的位置（偏移量）可放在一个长整形中。见 P120。（rewind 函数仅可将一个流设置到文件的起始位置。）

②ftello 和 fseeko 函数——见 P121。

③fgetpos 和 fsetpos 函数——使用一个抽象数据类型 fpos_t 记录文件的位置。见 P121。需要移植到非 UNIX 系统上运行的应用程序应当使用这两个函数。

格式化 I/O：

①格式化输出——printf、fprintf、sprintf 和 snprintf 函数。见 P121。

四种 printf 族的变体：vprintf、vfprintf、vsprintf 和 vsnprintf 函数

②格式化输入——scanf、fscanf 和 sscanf 函数。见 P125。

fileno 函数——通过参数指定一个流，返回其对应的文件描述符。见 P125。

tmpnam 函数——产生一个与现有文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名。见 P127。

tmpfile 函数——创建一个临时二进制文件（类型 wb+）。内部具体的过程：先调用 tmpnam 产生一个唯一的路径名，然后用该路径名创建一个文件，并立即 unlink 它。在关闭该文件或程序结束时将自动删除这种文件。见 P127。

XSI 为处理临时文件定义了另外两个函数：

①tempnam 函数——是 tmpnam 的一个变体，它允许调用者为所产生的路径名指定目录和前缀。见 P128。

②mkstemp 函数——类似于 tmpfile，但是该函数返回的不是文件指针，而是临时文件的打开文件描述符。见 P129。

七. 进程环境

C 程序总是从 main 函数开始执行。Main 函数的原型是

int main（int argc, char *argv[]）；

其中，argc 是命令行参数的数目，argv 是指向参数的各个指针所构成的数组。ISO C 和 POSIX.1 都要求 argv[argc]是一个空指针。在内核执行 C 程序时，先调用一个特殊的启动例程，启动例程再调用 main 函数。C 程序将此启动例程指定为程序的起始地址——这是有连接器设置的。启动例程从内核取得命令行参数和环境变量值，为调用 main 函数做准备。启动例程用 C 代码表示——exit（main（argc，argv））；它使得从 main 函数返回后立即调用 exit 函数。

有 8 种方式可使进程终止，其中 5 种为正常终止，它们是

①从 main 返回。即在 main 函数内执行 return 语句，等效于调用 exit。

②调用 exit。

③调用_exit 或_Exit。

④最后一个线程从其启动例程返回。

⑤最后一个线程调用 pthread_exit。

异常终止有 3 种方式，它们是

⑥调用 abort。

⑦接到一个信号并终止。

⑧最后一个线程对取消请求做出响应。

不管进程如何终止，最后都会执行内核中的同一段代码，这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等。

对于上述任意一种终止情形，终止进程能够通知其父进程它是如何终止的方法如下：

①对于三个终止函数（exit、_exit 和_Exit），将其退出状态作为参数传递给函数，最后调用_exit 时，内核将退出状态转换成终止状态。

②对于异常终止情况，内核（不是进程本身）产生一个指示其异常终止原因的终止状态。

③在子进程正常终止情况下，父进程可以获得子进程的退出状态。

下列三个 exit 函数用于正常终止一个程序：

_exit 和_Exit 函数——立即进入内核，不进行其他操作。见 P148。这两个函数同义，属于系统调用函数。

exit 函数——先进行一些清理处理（首先调用执行各终止处理程序，然后按需多次调用 fclose，关闭所有打开的标准 I/O 流等），然后进入内核。见 P148。此函数是一个库函数，当它执行时，会调用系统调用_exit（如下图中所示）。

三个 exit 函数都带一个整形参数，称之为退出状态（exit status），如果遇到下列情况之一：

（a）若调用这些函数时不带退出状态。

（b）main 执行了一个无返回值的 return 语句。

（c）main 没有声明返回类型为整形。

则该进程的终止状态是为定义的。但若 main 的返回类型是整形，并且 main 执行到最后一条语句时返回（隐式返回），那么该进程的终止状态为 0。

由于 main 函数返回一整形值与用该值调用 exit 是等价的。所以在 main 函数中：

exit（0）；　　≡　　return（0）；所以 return 也会进行关闭各种流等的操作。

一个进程可以登记多大 32 个函数，这些函数被称为终止处理函数。atexit 函数用于登记这些函数。见 P149。登记之后，它们将由 exit 自动调用执行。调用顺序与登记顺序相反。

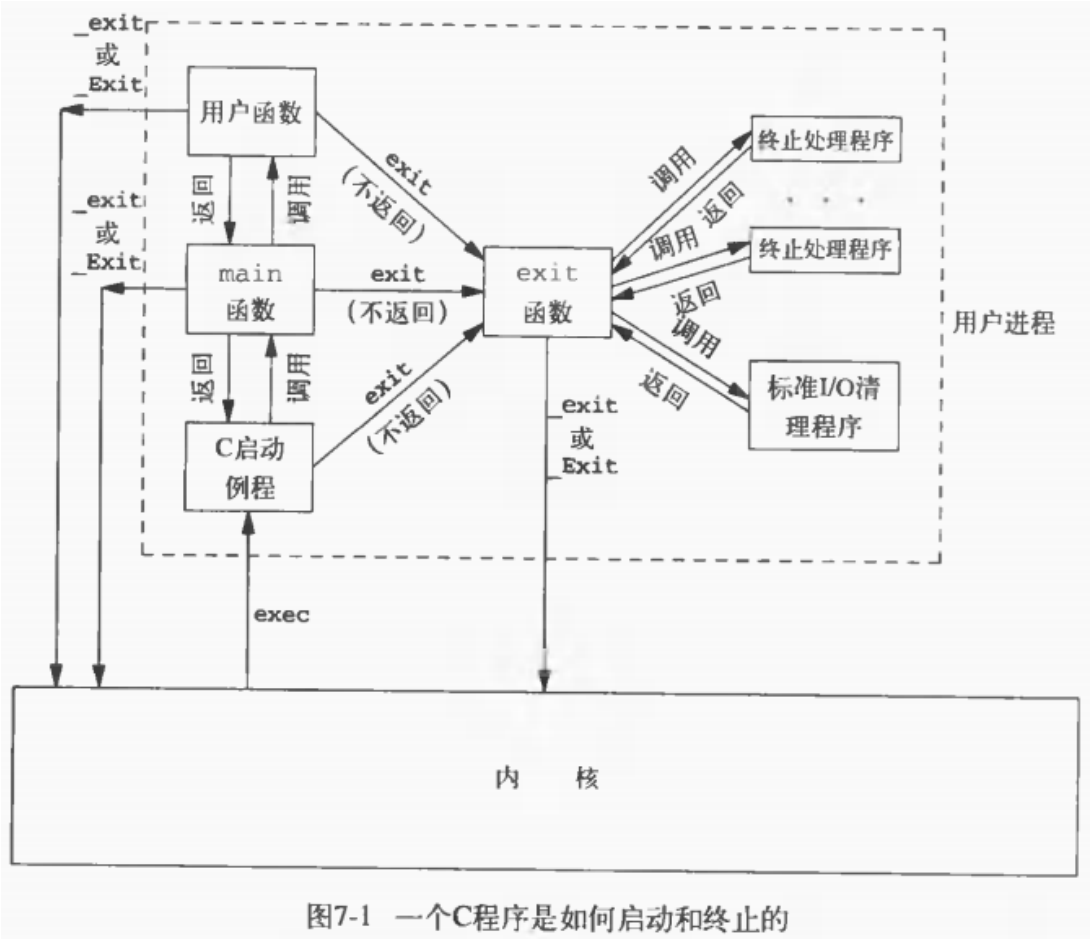


图7-1　一个C程序是如何启动和终止的

从上图可看出，内核使程序执行的唯一方法是调用一个 exec 函数（若程序调用 exec 函数族中的任一函数，则将清除所有已安装的终止处理程序）。进程自愿终止的唯一方法是显式的或隐式的（通过调用 exit）调用_exit 或_Exit。进程也可非自愿的由一个信号使其终止。

每个程序都会接收到一张环境表。环境表是一个字符指针数组，其中每个指针包括一个以 null 结束的 C 字符串的地址。全局变量 environ 则包含了该指针数组的地址：

extern　char **environ；我们称 environ 为环境指针。

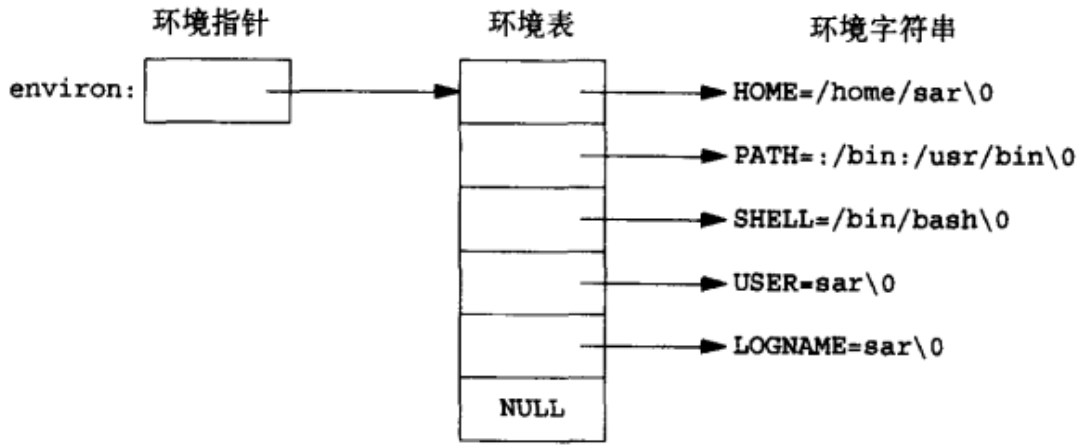


图7-2 由5个C字符串组成的环境

环境字符串格式：name=value，name 一般用大写字母。

一般用 getenv 函数和 putenv 函数来访问特定的环境变量，但要查看整个环境，则必须使用 environ 指针。

getenv 函数——用于取环境变量的值，此函数返回一个指针，它指向 name=value 字符串中的 value。见 P157。

表7-1 Single UNIX Specification定义的环境变量

变 量	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
COLUMNS	•	•	•	•	•	终端宽度
DATEMSK	XSI		•		•	getdate(3)模板文件路径名
HOME	•	•	•	•	•	起始目录
LANG	•	•	•	•	•	本地名
LC_ALL	•	•	•	•	•	本地名
LC_COLLATE	•	•	•	•	•	本地排序名
LC_CTYPE	•	•	•	•	•	本地字符分类名
LC_MESSAGES	•	•	•	•	•	本地消息名
LC_MONETARY	•	•	•	•	•	本地货币编辑名
LC_NUMERIC	•	•	•	•	•	本地数字编辑名
LC_TIME	•	•	•	•	•	本地日期/时间格式名
LINES	•	•	•	•	•	终端高度
LOGNAME	•	•	•	•	•	登录名
MSGVERB	XSI	•			•	fmtmsg(3)处理的消息组成部分
NLSPATH	XSI	•	•	•	•	消息类模板序列
PATH	•	•	•	•	•	搜索可执行文件的路径前缀列表
PWD	•	•	•	•	•	当前工作目录的绝对路径名
SHELL	•	•	•	•	•	用户首选的shell名
TERM	•	•	•	•	•	终端类型
TMPDIR	•	•	•	•	•	在其中创建临时文件的目录路径名
TZ	•	•	•	•	•	时区信息

除了去环境变量值之外，有时我们需要更改现有变量的值，或者增加新的环境变量（通常我们能影响的只是当前进程及调用的任何子进程的环境，但不能影响父进程的环境）。

clearenv 不是 XSI 的组成部分，它被用来删除环境表中的所有项。

putenv 函数——取形式为 name=value 的字符串，将其放到环境表中。如果 name 已经存在，则先删除其原来的定义。见 P158。

setenv 函数——将 name 的值设置为 value。如果 name 不存在，增加此环境字符串。如果 name 已经存在，那么（a）若参数 rewrite 非 0，则首先删除其现有的定义；（b）若 rewrite

为 0，则不删除其现有定义（name 不设置为新的 value，而且也不出错）。见 P158。

（关于 putenv 与 setenv 的区别，见 P158。）

unsetenv 函数——删除 name 的定义，即使 name 不存在也不出错。见 P158。

对环境字符串的相关操作：

①删除一个字符串：只要先在环境表中找到该指针，然后将所有后续指针都向环境表首部顺次移动一个位置。

②如果修改一个现有的 name 的值：

（a）如果新 value 的长度少于或等于现有 value 的长度，则只要在原字符串所用空间中写入新字符串。

（b）如果新 value 的长度大于原长度，则必须调用 malloc 为新字符串分配空间，然后将新字符串复制到该空间中，接着使环境表中针对 name 的指针指向新分配区。

③如果要增加一个新的 name，首先调用 malloc 为 name=value 字符串分配空间，然后将该字符串复制到此空间中，则：

（a）如果这是第一次增加一个新 name，则必须调用 malloc 为新的指针表分配空间。接着，将原来的环境表复制到新分配区，并将指向新 name=value 字符串的指针存放在该指针表的表尾，然后又将一个空指针存放在其后。最后使 environ 指向新指针表。

（b）如果这不是第一次增加一个新的 name，则可知以前已调用 malloc 在堆中为环境表分配了空间，所以只要调用 realloc，以分配比原空间多存放一个指针的空间。然后将指向新 name=value 字符串的指针存放在该表表尾，后面跟着一个空指针。

注意下图中最上面的一片区域，里面存有环境表及环境字符串，但此区域即不能上扩，也不能下扩。所以当在环境表中新增加一个 name 时，由于环境表不够装，需要在堆中另外分配一个大点的存储区，将环境表里的指针都复制到堆中，但复制过来的指针仍指向栈顶之上的各 name=value 字符串。

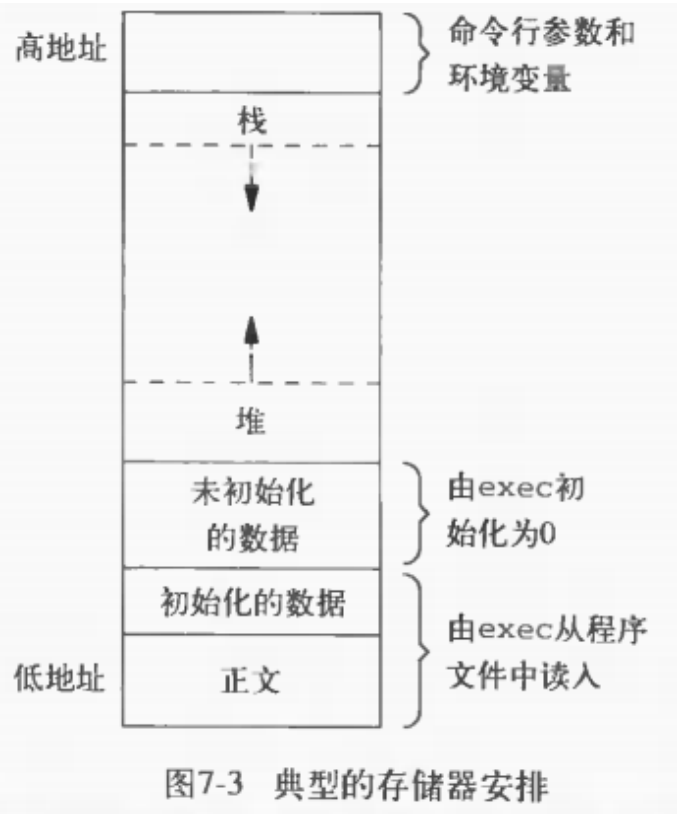


图7-3 典型的存储器安排

上图为进程执行的程序映像，即进程地址空间，堆顶和栈底之间未用的虚地址空间很大。

一个程序（如 a.out）通常由下面几部分组成：

①正文段——由 cpu 执行的机器指定部分。通常正文段是只读的，可共享的。

②初始化数据段——亦称数据段，包含了程序中需明确的赋初值的变量。

③非初始化数据段——亦称 bss 段。在程序开始执行之前，内核将此段中的数据初始化为 0 或空指针。

④栈——自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次调用函数时，其返回地址以及调用者的环境信息（例如某些机器寄存器的值）都存放在栈中。

⑤堆——通常在堆中进行动态存储分配。

a.out 中还有若干其他类型的段，比如包含符号表的段，包含调试信息的段，这些部分并不装载到进程执行的程序映像中。

共享库——使得可执行文件中不再需要包含公用的库例程，而只需在所有进程都可引用的存储区中维护这种库例程的一个副本。这样减少了每个可执行文件的长度，但增加了一些运行时间开销。另一个好处是可以方便的用库函数的新版本代替老版本，而无需对使用该库的程序重新连接编辑。详见 P154。

ISO C 说明了三个用于存储空间动态分配的函数（应该是在堆中分配）。

①malloc 函数——分配指定字节数的存储区。见 P155。

②calloc 函数——为指定数量具指定长度的对象分配存储空间。该空间中每一位都初始化为 0。见 P155。

③更改以前分配区的长度（增加或减少）。见 P155。

free 函数——释放参数 ptr 指向的存储空间。见 P155。释放的空间可供以后再分配，通常将它们保持在 malloc 池中，而不是返回给内核。

这些函数通常调用 sbrk（2）系统调用实现。该系统调用扩充（或缩小）进程的堆。

大多数实现所分配的存储空间比所要求的要稍大一些，额外的空间用来记录管理信息——分配块的长度、指向下一个分配块的指针等等。详见 P155。

alloca 函数——它的调用序列与 malloc 相同，但它是在当前函数的栈帧中分配存储空间，而不是在堆中。优点是：当函数返回时，自动会释放使用的栈帧。缺点是：增加了栈帧的长度。

当 main 函数中的深层嵌套函数中遇到一个非致命性错误时，它可能要求一次性返回到 main 函数中，此时我们就要使用非局部 goto——setjmp 和 longjmp 函数。非局部是指，这不是由普通 C 语言 goto 语句在一个函数内实施的跳转，而是在栈上跳过若干调用帧，返回到当前函数调用路径上的某一个函数中。

setjmp 函数与 longjmp 函数是配合使用的，两者有个共同的参数 env，其类型是一个特殊类型 jmp_buf。这一数据类型是某种形式的数组，其中存放在调用 longjmp 时能用来恢复栈状态的所有信息。因为要在另一个函数中被使用，所以 env 被定义为全局变量。我们在希望返回到的位置调用 setjmp 函数，它会将用来恢复栈状态的信息存入其参数 env 中，之后就可在子函数中通过调用 longjmp 函数而返回到先前使用 setjmp 函数的位置重新执行程序。此外，longjmp 函数还有另一个参数 val，它作为返回之后 setjmp 函数的返回值，这样我们就可以在调用一个 setjmp 之后，使用多个 longjmp 函数，每个 longjmp 的参数 val 设置不同的值，最后通过 setjmp 的返回值而判断是从哪个 longjmp 处返回的。（注意，在 main 中第一次直接执行 setjmp 时，其返回值为 0。）

在调用 longjmp 使栈状态恢复时，声明为全局或静态变量的值在执行 longjmp 时保持不变；而关于自动变量、寄存器变量等如何变动，一般是看情况的，如果你有一个自动变量，而又不想其值回滚，则可定义其具有 volatile 属性。可详见 P162。

每个进程都有一组资源限制，进程的资源限制通常是在系统初始化时由进程 0 建立的，然后由每个后续进程继承。资源限制影响到调用进程并由其子进程继承。

getrlimit 函数——用于查看资源限制。见 P165。

setrlimit 函数——用于更改资源限制。见 P165。

以上两个函数有两个相同的参数，一个用于指定一个资源，另一个指向下列结构的指针。

```
struct rlimit{
    rlim_t rlim_cur;    //软限制
    rlim_t rlim_cur;    //硬限制
}
```

（常量 RLIM_INFINITY 指定了一个无限量的限制）

在更改资源限制时，要遵循下列三条规则：

①任何一个进程都可将一个软限制值更改为小于或等于其硬限制值。

②任何一个进程都可降低其应限制值，但它必须大于或等于其软限制值。

③普通用户进程只可降低其硬限制值，超级用户进程既可降低硬限制值，又可提高硬限制值。

上面两个函数的参数之一 resource，可取下表中变量之一。关于每个变量的具体含义，详见 P165。

表7-3　对资源限制的支持

限　制	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
RLIMIT_AS	•		•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_LOCKS			•		
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
RLIMIT_VMEM		•			•

八．进程控制

对于进程 ID，虽然是唯一的，但是进程 ID 可以重用。当一个进程终止后，其进程 ID 就可以再次使用了。（但大多数实现采用延迟重用算法，使得赋予新建进程的 ID 不同于最近终止进程所使用的 ID。）

进程 ID 为 0 的进程通常是调度进程（即交换进程），该进程是内核的一部分，它并不执行任何磁盘上的程序，因此又被叫做系统进程。

进程 ID 为 1 的进程通常是 init 进程，在自举过程结束时由内核调用。该进程的程序文件在 UNIX 的早期版本中是/etc/init，在较新版本中是/sbin/init。此进程负责在自举内核后启

动一个 UNIX 系统。init 进程绝不会终止，它是一个普通的用户进程（与交换进程不同，它不是内核中的系统进程），但是它是以超级用户特权运行。

每个进程除进程 ID 外，还有其他的一些标识符，可通过下列函数返回：

①getpid 函数——返回调用进程的进程 ID。

②getppid 函数——返回调用进程的父进程 ID。

③getuid 函数——返回调用进程的实际用户 ID。

④geteuid 函数——返回调用进程的有效用户 ID。

⑤getgid 函数——返回调用进程的实际组 ID。

⑥getegid 函数——返回调用进程的有效组 ID。

fork 函数——用于创建一个新进程（称为子进程）。见 P172。此函数被调用一次，但返回两次。对子进程返回 0，对父进程返回子进程的进程 ID。子进程和父进程继续执行 fork 调用之后的指令。

fork 之后，父进程的正文段由父、子进程共享。由于 fork 之后经常跟随 exec，父进程的数据空间、堆和栈暂时由父、子进程共享，而且内核将它们的访问权限改变为只读。如果父、子进程中的任一个试图修改这些区域，则内核只为修改区域的那块内存制作一个副本，通常是虚拟存储器系统中的一“页”。

在 fork 之后，是父进程先执行还是子进程先执行，这是不确定的，取决于内核所使用的调度算法。

在 fork 之后处理文件描述符有两种常见的情况：

①父进程等待子进程完成：父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的文件偏移量已执行了相应更新。

②父、子进程各自执行不同的程序段：父、子进程各自关闭它们不需使用的文件描述符，这样就不会干扰对方使用的文件描述符。

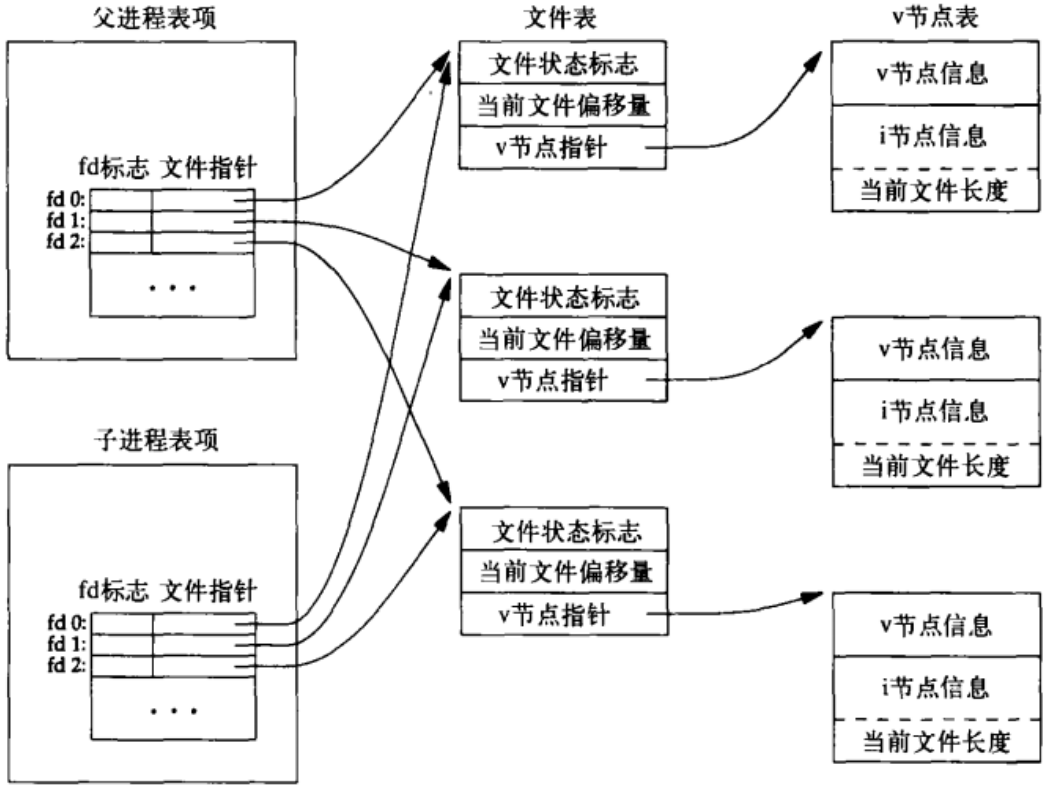


图8-1 调用fork之后父、子进程之间对打开文件的共享

由上图可知，不论对于 fork 还是 vfork，在重定向父进程的标准输出时，子进程的标准输出也被重定向，父进程的所有打开文件描述符都被复制到子进程中，即子进程中的文件描述符是父进程的一个副本。

除了打开文件之外，父进程的很多其他属性也由子进程继承，包括：

- 继承父进程的存储映射区。
- 继承父进程的信号处理方式。
- 实际用户 ID、实际组 ID、有效用户 ID、有效组 ID。
- 附加组 ID。
- 进程组 ID。
- 会话 ID。
- 控制终端。
- 设置用户 ID 标志和设置组- ID 标志。
- 当前工作目录
- 根目录。
- 文件方式创建屏蔽字。
- 信号屏蔽和安排。
- 对任一打开文件描述符的在执行时关闭（close-on-exec）标志。
- 环境。
- 连接的共享存储段。
- 资源限制。

父、子进程之间的区别是：

- fork 的返回值。
- 进程 ID 不同。
- 不同的父进程 ID。
- 子进程的 tms_utime,tms_stime,tms_cutime 以及 tms_ustime 设置为 0。
- 父进程设置的锁，子进程不继承。
- 子进程的未处理的闹钟（alarm）被清除。
- 子进程的未处理信号集设置为空集。

使 fork 失败的两个主要原因：①系统中已经有了太多的进程。

②该实际用户 ID 的进程总数超过了系统限制。

fork 有下面两种用法：

①一个父进程希望复制自己，使父、子进程同时执行不同的代码段。

②一个进程要执行一个不同的程序。这种情况下，子进程从 fork 返回后立即调用 exec。

vfork 函数——同 fork 类似，也是创建一个新的子进程，而该子进程的目的是 exec 一个新程序。因为子进程会立即调用 exec（或 exit），所以此函数并不将父进程的地址空间复制到子进程中。在子进程调用 exec 或 exit 之前，它在父进程的空间中运行。此外，vfork 函数保证子程序先运行，内核会使父进程暂时处于休眠状态，在子程序调用 exec 或 exit 之后，父进程才有可能被调度运行。

（注意：上面的 exec 和 exit 都是数个函数的统称。关于 vfork 之后调用 exit（0），此时 exit（0）只会冲洗所有标准 I/O 流缓冲区，在现代大多数实现中，并不会关闭标准 I/O 流（只是针对 vfork，所以不建议使用 vfork，其他情况还是会关闭标准 I/O 流），详见 P177。）

对于父进程已经终止的所有进程，它们的父进程都改变为 init 进程。对于 init 进程，无论何时，只要其有一个子进程终止，init 就会调用一个 wait 函数取得其终止状态。

如果子进程在父进程之前终止，内核为每个终止子进程保存一定量的信息，所以当终止进程的父进程调用 wait 或 waitpid 时，可以得到这些信息。这些信息至少包括：进程 ID、该进程的终止状态、以及该进程使用的 cpu 时间总量。

进程终止后，内核可以释放其所使用的所有存储区，关闭其所有打开文件。

僵死进程——但若一个进程已经终止，但其父进程尚未对其进行善后处理（获取子进程有关信息，释放其占用的资源）的进程就是僵死进程。

当一个进程正常或异常终止时，内核就像其父进程发送 SIGCHLD 信号。父进程可以选择忽略该信号，或者提供一个该信号发生时即被调用执行的函数（信号处理程序），对该信号，系统默认动作是忽略它。

wait 函数、waitpid 函数——返回终止子进程的进程 ID，取得终止进程的终止状态，通常用于等待一个子进程终止。见 P179。进程的终止状态就存于这两个函数的参数整形指针 statloc 所指向的单元内，如果不关心终止状态，则将该参数指定为空指针。

(这两个函数返回的 statloc 指向的整形状态字中，其中某些位表示退出状态（正常返回），其他位则指示信号编号（异常返回），有一位指示是否产生了一个 core 文件等。)

下表通过四种宏来检测进程的终止状态，确定进程是如何终止的。

表8-1 检查wait和waitpid所返回的终止状态的宏

宏	说 明
WIFEXITED (*status*)	若为正常终止子进程返回的状态，则为真。对于这种情况可执行WEXITSTATUS (*status*)，取子进程传送给exit、_exit或_Exit参数的低8位
WIFSIGNALED (*status*)	若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行WTERMSIG (*status*)，取使子进程终止的信号编号。另外，有些实现（非Single UNIX Specification）定义宏WCOREDUMP (*status*)，若已产生终止进程的core文件，则它返回真
WIFSTOPPED (*status*)	若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行WSTOPSIG (*status*)，取使子进程暂停的信号编号
WIFCONTINUED (*status*)	若在作业控制暂停后已经继续的子进程返回了状态，则为真。(POSIX.1的XSI扩展，仅用于waitpid。)

waitpid 函数中的 pid 参数的作用的解释如下：

①pid=-1　　等待任一子进程。

②pid>0　　等待其进程与 pid 相等的子进程。

③pid=0　　等待其组 ID 等于调用进程组 ID 的任一子进程。

④pid<-1　　等待其组 ID 等于 pid 绝对值的任一子进程。

waitpid 函数中的 options 参数使我们能进一步控制 waitpid 的操作。此参数可以是 0，或者是下表中按位“或”运算的结果。

表8-2 waitpid的*options*常量

常 量	说 明
WCONTINUED	若实现支持作业控制，那么由*pid*指定的任一子进程在暂停后已经继续，但其状态尚未报告，则返回其状态（POSIX.1的XSI扩展）
WNOHANG	若由*pid*指定的子进程并不是立即可用的，则waitpid不阻塞，此时其返回值为0
WUNTRACED	若某实现支持作业控制，而由*pid*指定的任一子进程已处于暂停状态，并且其状态自暂停以来还未报告过，则返回其状态。WIFSTOPPED宏确定返回值是否对应于一个暂停子进程

两个函数的出错情况：

①对于 wait，其唯一的出错是调用进程没有子进程（函数调用被一个信号中断时，

也可能返回另一种出错）。

②对于 waitpid，如果指定的进程或进程组不存在，或者参数 pid 指定的进程不是调用进程的子进程则都将出错。

当进程调用这两个函数时，会出现以下情况：

①如果其所有子进程都还在运行，则阻塞，直到任意一个子进程终止。

②如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回。

③如果它没有任何子进程，则立即出错返回。

这两个函数的区别：

①在一个子进程终止前，wait 使其调用者阻塞，而 waitpid 有一个选项，可使调用者不阻塞。

②waitpid 并不等待在其调用之后的第一个终止的子进程终止，它有若干个选项，可以等待它所选择的某个指定的子进程。而且 waitpid 支持作业控制（利用 WUNTRACED 和 WCONTINUED 选项）。

wait3 和 wait4 函数——比 wait 和 waitpid 函数相对多一个功能，因为它们多一个参数 rusage，该参数要求内核返回由终止进程及其所有子进程使用的资源汇总（资源统计信息包括用户 cpu 时间总量、系统 cpu 时间总量、页面出错次数、接受到信号的次数等）。

竞争条件——当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件。解决方法是使用几个宏 TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT 以及 WAIT_CHILD。详见 P185，P273。

exec 函数——进程调用此函数用于执行另一个新程序。新程序从 main 函数开始执行，由于此函数并非创建一个新进程，所以进程 ID 并为改变，它只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。此函数是一个统称，它包含下列 6 种类型：

①execl 函数

②execv 函数

③execle 函数

④execve 函数——系统调用函数，其他是库函数。

⑤execlp 函数

⑥execvp 函数

（具体区别详见 P188。）

调用 exec 后，除进程 ID 未改变外，执行新程序的进程还保持了原进程的下列特征：

- 进程 ID 和父进程 ID。

(对于有效用户 ID：如果新程序的设置用户 ID 位未设置，则不变，还是为原来由父进程继承来的有效用户 ID，并非等于未调用 exec 之前的子进程的实际用户 ID；若已设置，则为新程序的所有者 ID。)

- 实际用户 ID 和实际组 ID。
- 附加组 ID。
- 进程组 ID。
- 会话 ID。
- 控制终端。
- 闹钟尚余留的时间。
- 当前工作目录。
- 根目录。
- 文件模式创建屏蔽字。

- 文件锁。
- 进程信号屏蔽。
- 未处理信号。
- 资源限制。
- tms_utime, tms_stime, tms_cutime 以及 tms_ustime 值。

执行时关闭标志——对打开文件的处理与每个描述符的执行时关闭标志值有关。进程中每个打开描述符都有一个执行时关闭标志。若此标志设置，则在执行 exec 时关闭该描述符，否则该描述符仍打开。除非特地用 fcntl 设置了该标志，否则系统的默认操作是在 exec 后仍保持这种描述符打开。比如 POSIX.1 明确要求在执行 exec 时关闭打开的目录流，它调用 fcntl 函数为对应于打开目录流的描述符设置执行时关闭标志。

注意，在执行 exec 前后实际用户 ID 和实际组 ID 保持不变，而有效 ID 是否改变取决于所执行程序文件的设置用户 ID 位和设置组 ID 位是否设置。

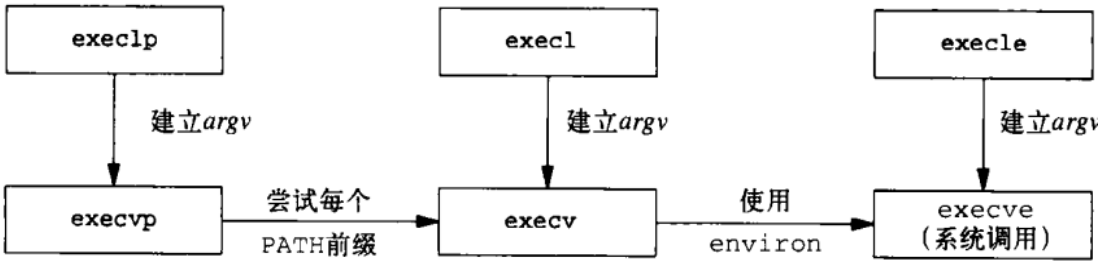


图8-2 6个exec函数之间的关系

setuid 函数——设置实际用户 ID 和有效用户 ID。见 P193。

setgid 函数——设置实际组 ID 和有效组 ID。见 P193。

下面是关于用户 ID（setuid）的说明，其中一切也都适用于组 ID（setgid）。

①若进程具有超级用户特权（即有效用户为 root），则 setuid 函数将实际用户 ID、有效用户 ID，以及保存的设置用户 ID 设置为 uid。（只有超级用户进程可以更改实际用户 ID。）

②若进程无超级用户特权，但是 uid 等于实际用户 ID 或保存的设置用户 ID，则 setuid 只将有效用户 ID 设置为 uid。不改变实际用户 ID 和保存的设置用户 ID。

③若上面两条件都不满足，则将 errno 设置为 EPERM，并返回-1。

保存的设置用户 ID——不论是否设置了文件的设置用户 ID 位，它都是由 exec 复制有效用户 ID 而得来的。如果设置了文件设置用户 ID 位，则在 exec 根据文件的所有者 ID 设置了进程的有效用户 ID 以后，就将这个副本保存起来。如果未设置文件设置用户 ID 位，则将其从父进程继承来的有效用户 ID 直接复制，作为副本，当做保存的设置用户 ID。

（注意，保存的设置用户 ID 只能由 exec 保存，若进程未调用 exec，则此 ID 不存在）

表8-7 改变三个用户ID的不同方法

ID	exec		setuid (*uid*)	
	设置用户ID位关闭	设置用户ID位打开	超级用户	非特权用户
实际用户ID	不变	不变	设为*uid*	不变
有效用户ID	不变	设置为程序文件的用户ID	设为*uid*	设为*uid*
保存的设置用户ID	从有效用户ID复制	从有效用户ID复制	设为*uid*	不变

setreuid 函数——交换实际用户 ID 与有效用户 ID 的值。见 P195。

setregid 函数——交换实际组 ID 与有效组 ID 的值。见 P195。

下面是关于用户 ID（setreuid）的说明，其中一切也都适用于组 ID（setregid）。

①对于特权用户，setreuid 函数用于指定实际用户 ID 和有效用户 ID 的值。

②对于非特权用户，setreuid 总能交换实际用户 ID 和有效用户 ID。此外，还允许其将有效用户 ID 设置为保存的设置用户 ID。

seteuid 函数——设置有效用户 ID。见 P195。

setegid 函数——设置有效组 ID。见 P195。

下面是关于用户 ID（seteuid）的说明，其中一切也都适用于组 ID（setegid）。

①一个非特权用户可将其有效用户 ID 设置为其实际用户 ID 或其保存的设置用户 ID。

②对于一个特权用户，可将其有效用户 ID 设置为参数 uid。

（本章所说明的一切都以类似方式适用于各个组 ID。附加组 ID 不受 setgid、setregid 或 setegid 函数的限制）

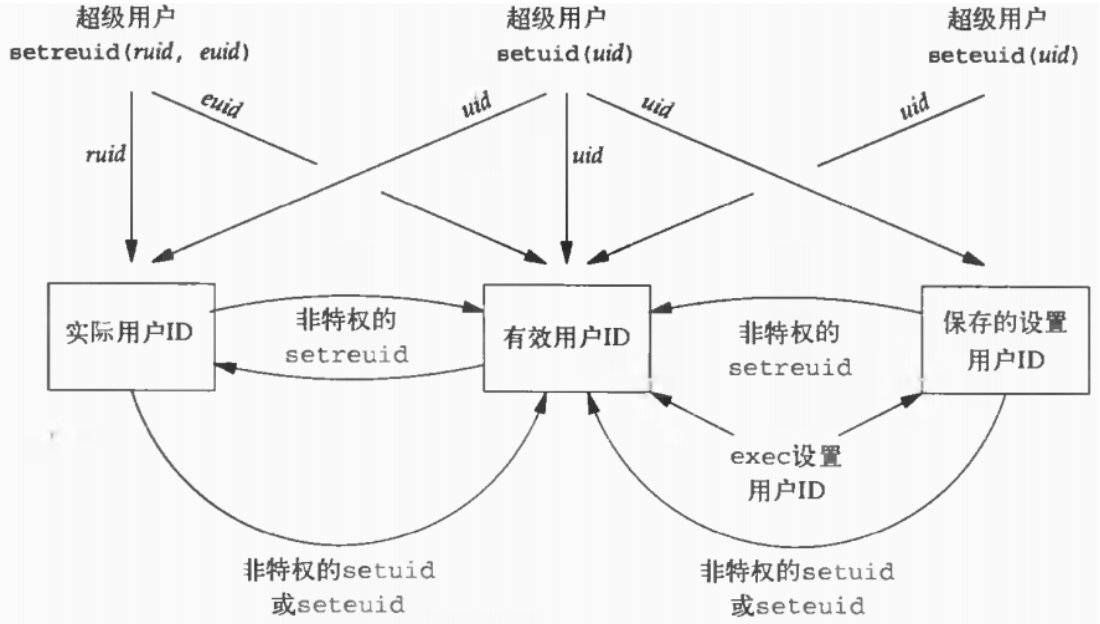


图8-3　设置不同用户ID的各函数

解释器文件——文本文件，它以#！开头。其起始行的形式是：

#！　pathname [optional-argument]（pathname 为绝对路径名）

解释器——由该解释器文件起始行中的 pathname 指定。内核使调用 exec 函数（此时参数是解释器文件，即文本文件，并非二进制可执行文件）的进程实际执行的并不是该解释器文件，而是该解释器文件起始行中 pathname 所指定的文件（此文件为可执行文件）。

关于解释器部分，请详见 P196。

system 函数——生成一个进程执行另一个程序，通过此函数还可使 shell 命令在进程中执行。见 P200。如 system（"date > file"）;

如果一个进程正以特殊的权限（设置用户 ID 或设置组 ID）运行，它又想生成另一个进程执行另一个程序，则它应当直接使用 fork 和 exec（不能直接使用 system，否则 exec 执行的新程序执行时会有特权，从而产生错误），而且在 fork 之后、exic 之前要将子程序的实际用户 ID 和有效用户 ID 改回到普通权限。设置用户 ID 或设置组 ID 程序决不应调用 system 函数。

进程会计——大多数 UNIX 系统提供了一个选项以进行进程会计处理。启用该选项后，每当进程终止时内核就写一个会计记录。会计记录会写到指定的文件中（在 Linux 中，该文

件是/var/account/pacct），即会计文件中记录的顺序对应于进程终止的顺序。会计记录对应于进程而不是程序，它所需的各种数据都由内核保存在进程表中，并在一个新进程被创建时置初值。典型的会计记录包含总量较小的二进制数据，一般包括命令名、所使用的 cpu 时间总量、用户 ID 和组 ID、启动时间等。会计记录结构如下：

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char    ac_flag;     /* flag (see Figure 8.26) */
  char    ac_stat;     /* termination status (signal & core flag only) */
                       /* (Solaris only) */
  uid_t   ac_uid;      /* real user ID */
  gid_t   ac_gid;      /* real group ID */
  dev_t   ac_tty;      /* controlling terminal */
  time_t  ac_btime;    /* starting calendar time */
  comp_t  ac_utime;    /* user CPU time (clock ticks) */
  comp_t  ac_stime;    /* system CPU time (clock ticks) */
  comp_t  ac_etime;    /* elapsed time (clock ticks) */
  comp_t  ac_mem;      /* average memory usage */
  comp_t  ac_io;       /* bytes transferred (by read and write) */
                       /* "blocks" on BSD systems */
  comp_t  ac_rw;       /* blocks read or written */
                       /* (not present on BSD systems) */
  char    ac_comm[8]; /* command name: [8] for Solaris, */
                       /* [10] for Mac OS X, [16] for FreeBSD, and */
                       /* [17] for Linux */
};
```

其中，ac_flag成员记录了进程执行期间的某些事件。这些事件见表8-8。

表8-8 会计记录中的ac_flag值

ac_flag	说　明	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
AFORK	进程是由fork产生的，但从未调用exec	•	•	•	•
ASU	进程使用超级用户特权		•	•	•
ACOMPAT	进程使用兼容模式				
ACORE	进程转储core	•	•	•	
AXSIG	进程由信号杀死	•	•	•	
AEXPND	扩展的会计条目				•

关于结构中的其他一些元素说明，详见 P207。

函数（acct）可用于启用和禁用进程会计，唯一使用这一函数的是 accton（8）命令。

①超级用户执行一个带路径名参数的 accton 命令会启动会计处理。

②超级用户执行不带任何参数的 accton 命令可停止会计处理。

在 fork 之后，内核为子进程初始化一个记录。在 exec 之后并不创建一个新的会计记录，但改变了相应记录中命令名，并且 AFORK 标志会被清除。

getpwuid（getuid（））——用于查找运行该程序的用户登录名（一个登录名对应一个 ID）。

getlogin 函数——成功则返回指向登录名字符串的指针。见 P208。（当多个登录名对应同一个用户 ID。）当我们得到了登录名，就可用 getpwnam 在口令文件中查找用户的相应记录，从而确定其登录 shall 等（对应同一用户 ID 的不同的登录名，其登录 shell 不同）。

times 函数——任一进程都可调用此函数以获得它自己及终止子进程的三个值：墙上时钟时间、用户 cpu 时间和系统 cpu 时间。见 P208。

九．进程关系

终端登录——详见 P213。

网络登录——详见 P216。

进程组——是一个或多个进程的集合。它们与同一作业相关联(可将一个作业看做一个进程组)，可以接受来自同一终端的各种信号。通常是由 shell 的管道线将几个进程编成一组的。每个进程组有一个唯一的进程组 ID。

getpgrp 函数——返回当前进程的进程组 ID。见 P218。

getpgid 函数——有一个参数 pid，用于返回指定进程的进程组 ID。若 pid 为 0，则返回当前进程的进程组 ID。所以 getpgid（0）相当于 getpgrp（）。

组长进程——每个进程组都有一个组长进程，对于组长进程，其进程组 ID 等于其进程 ID。组长进程可以创建一个进程组，创建该组中的进程，然后终止。但进程组还在，进程组中只要有一个进程存在，该进程组就存在。

（组长进程的进程 ID 与其进程组 ID 相等,即组长进程也是这个进程组中的一个进程。）

进程组的生存期——从进程组创建开始到其中最后一个进程离开为止的时间区间。（进程组中的最后一个进程可以终止，也可以转移到另一个进程组中。）

setpgid 函数——用于为某个指定进程设置其进程组 ID。（可使其加入一个现有的进程组或创建一个新进程组。）见 P218。一个进程只能为自己或它的子进程设置进程组 ID，但在子进程调用 exec 之后，就不可以为其设置进程组 ID 了。

会话——是一个或多个进程组的集合。

会话首进程——创建该会话的进程，刚创建该会话时，它是该新会话中的唯一进程。

会话 ID——会话首进程的进程 ID。

setsid 函数——进程（不允许是一个进程组的组长进程）调用此函数来建立一个新会话。会发生下面三件事：

①该进程变成新会话首进程。

②该进程成为一个新进程组的组长进程。

③该进程没有控制终端。

所以对于会话首进程：　　进程 ID=进程组 ID=会话 ID

getsid 函数——有个参数 pid，返回指定进程的会话 ID（即会话首进程的进程 ID）。若 pid 为 0，则返回当前调用进程的会话 ID。见 P220。如果 pid 并不属于调用者所在的会话，那么调用者就不能得到该会话首进程的进程 ID。

（一个会话里的所有进程都有相同的会话 ID,一个进程组里的所有进程都有相同的进程组 ID。）

会话和进程组有一些其他特性：

- 一个会话可以有一个单独的控制终端（controlling terminal）。这通常是我们在其上登录的终端设备（终端登录情况）或伪终端设备（网络登录情况）。登录时将自动建立控制终端。
- 建立与控制终端连接的会话首进程，被称之为控制进程（controlling process）。
- 一个会话中的几个进程组可被分成一个前台进程组（ foreground process group）以及一个或几个后台进程组（background process group）。

• 如果一个会话有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。

• 无论何时键入终端的中断键（常常是 DELETE 或 Ctrl+C）或退出键（常常是 Ctrl+\），就会造成控制终端将中断信号或退出信号送至前台进程组中的所有进程。

• 如果终端界面检测到调制解调器（或网络）已经脱开连接，控制终端则将挂断信号（由终端驱动程序产生，默认动作是终止进程）送至控制进程（会话首进程）。

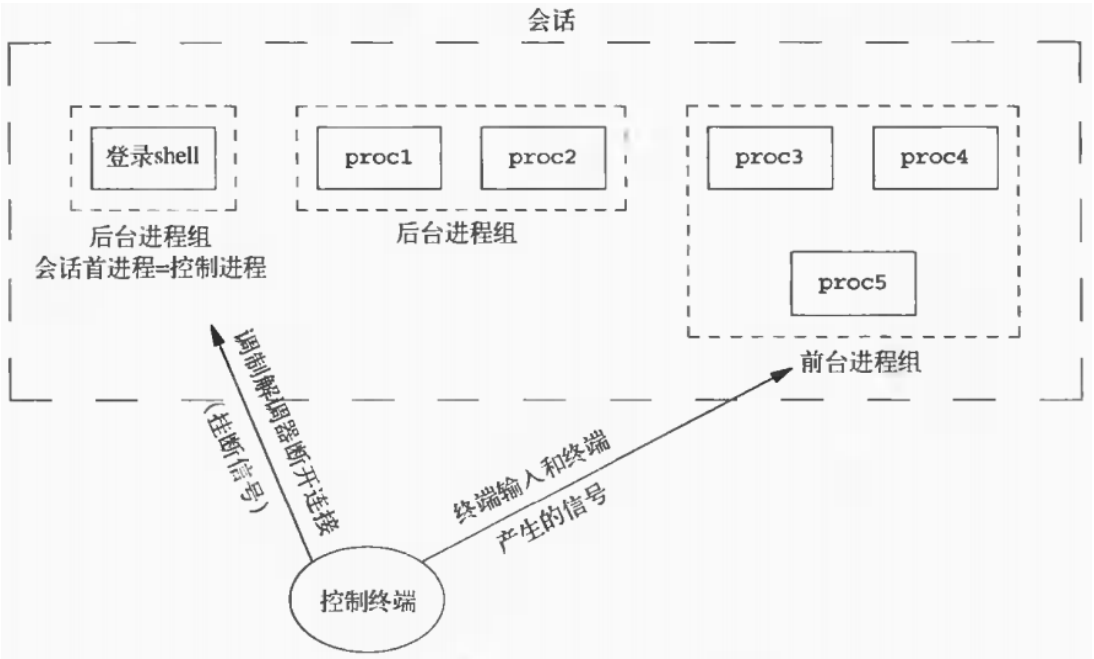


图9-7 显示控制终端的进程组和会话

有时不管标准输入、标准输出是否被重定向，程序都要与控制终端交互。保证程序能读写控制终端的方法是打开文件/dev/tty。如果程序没有控制终端，则打开此设备将失败。

tcgetpgrp 函数——有个参数 filedes，返回（当前会话中）前台进程组的进程组 ID。该前台进程组与 filedes 上打开的终端相关联。见 P221。

tcsetpgrp 函数——如进程有一个控制终端（即该进程必须属于前台进程组，且该前台进程组有控制终端），则可调用此函数将前台进程组的进程组 ID 改为同一会话中其他进程组的 ID，即更换前台进程组。参数 filedes 必须引用该会话的控制终端（即 filedes 为/dev/tty 的文件描述符）。见 P221。

（以上两函数一般不由应用程序直接调用，而是由作业控制 shell 调用。）

tcgetsid 函数——通过参数 filedes（/dev/tty 的文件描述符），获得会话 ID。见 P222。

作业控制——它允许在一个终端上启动多个作业（进程组），它控制哪一个作业可以访问该终端，以及哪些作业在后台运行。作业控制要求下面三种形式的支持：

①支持作业控制的 shell。

②内核中的终端驱动程序必须支持作业控制。

③内核必须提供对某些作业控制信号的支持。

实际上有下面三个特殊字符可使终端驱动程序产生信号，并将它们送至前台进程组中的所有进程，而后台进程组作业不受影响：

①中断字符（一般采用 DELETE 或 Ctrl+C）产生 SIGINT。

②退出字符（一般采用 Ctrl+\）产生 SIGQUIT。

③挂起字符（一般采用 Ctrl+Z）产生 SIGTSTP。

由上图 9-7 可知，只有前台作业能接受终端输入。当后台作业试图读终端时，终端驱动

程序将向后台作业发送一个特定信号 SIGTTIN，该信号通常会暂停此后台作业，然后当用户用 shell 命令将其转为前台作业之后，并将继续信号（SIGCONT，应该是由终端驱动程序提供）送给该作业，使其再运行（注意：若后台作业是孤儿进程组，当其读终端则情况特殊——见 P230）。而后台是否可输出到控制终端是可选的（可根据 stty（1）命令改变其选项确定其是否可输出到控制终端，默认是可输出），如果用户禁止后台作业写到终端，当命令要向终端写数据时，因为终端驱动程序将该写操作标识为来自于后台进程，于是向该作业发送 SIGTTOU 信号（默认也是将该后台作业暂停）。之后就和读终端类似了。

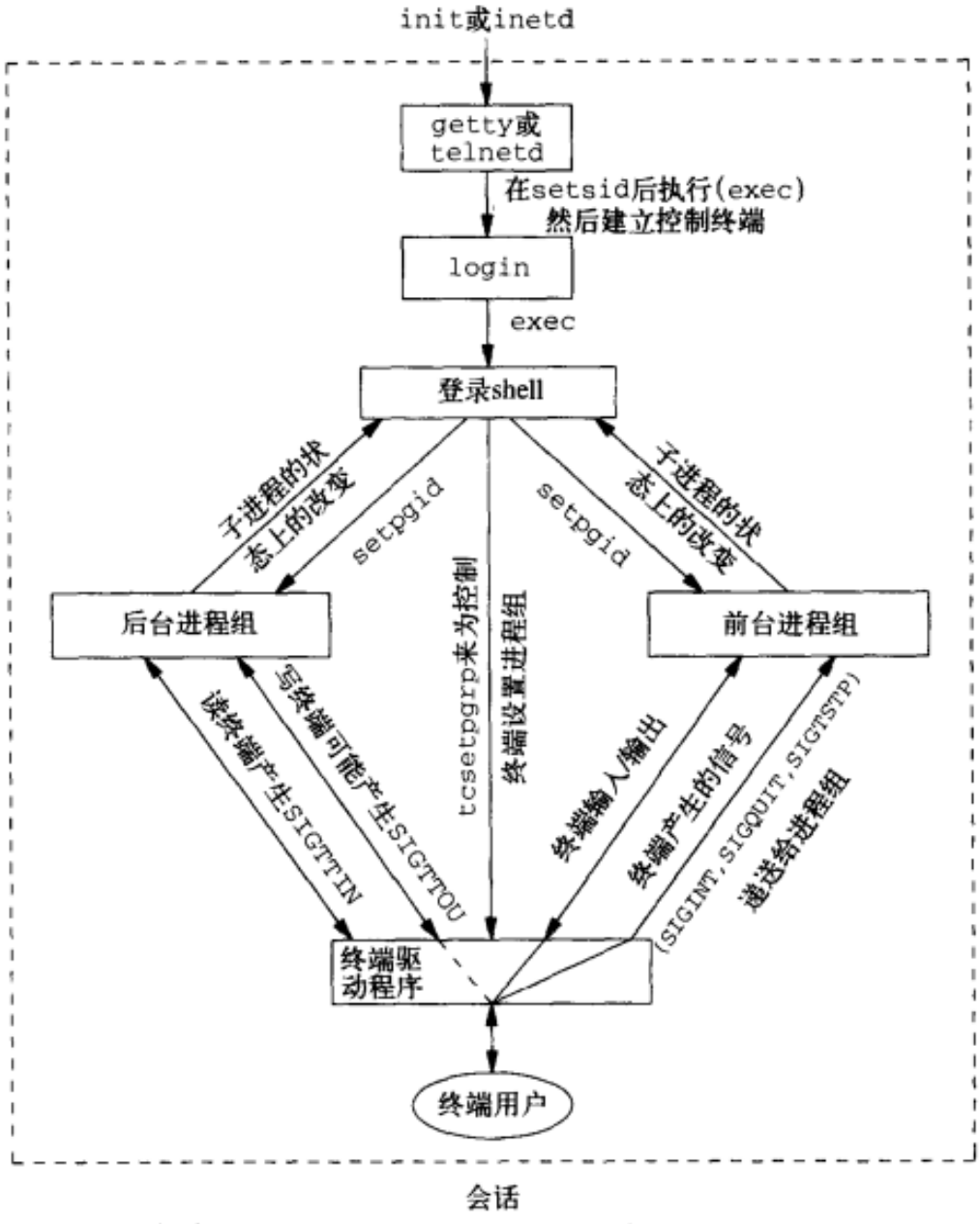


图9-8　具有前台、后台作业以及终端驱动程序的作业控制功能总结

上图中：穿过终端驱动程序框的实线表示——终端 I/O 和终端产生的信号总是从前台进程组连接到实际终端。对应于 SIGTTOU 信号的虚线表示——后台进程组进程的输出是否出现在终端是可选的。

shell 执行程序（针对支持作业控制的 bash）——当在 shell 中执行一个命令时，由 shell 创建的进程会存在于新进程组中（该进程创建的子进程也在这新进程组中），而 shell 则留在

原来的旧进程组中，前台进程组一直是新进程组（当 shell 创建的父进程终止后，无论其子进程是否还在运行，前台进程组都会更换为 shell 所在的旧进程组，所以孤儿进程组不会是前台进程组）。如果在执行的指定后加一个“&”，使其在后台执行，则情况相似，只是前台进程组一直是旧进程组。——详见 P227。

孤儿进程组——该组中每个成员的父进程要么是该组的一个成员，要么不是该组所属会话的成员。如果进程组不是孤儿进程组，那么在属于同一会话的另一个组中的父进程就由机会重新启动该组中停止的进程。

POSIX.1 要求向新的孤儿进程组中处于停止状态的每一个进程发送挂断信号（SIGHUP），接着又向其发送继续信号（SIGCONT）。

十．信号

信号——是软件中断，提供了一种处理异步事件的方法。每个信号都由一个名字，都以 SIG 开头。Linux2.4.22 支持 31 种不同的信号，还支持应用程序额外定义的信号，并将其作为实时扩展。在头文件<signal.h>中，这些信号都被定义为正整数（即信号编号）。不存在编号为 0 的信号（编号为 0 的话，则称作空信号）。

以下几种条件可产生信号：

①当用户按某些终端键时，引发终端产生的信号。

②硬件异常产生信号：除数为 0、无效的内存应用等等。（由硬件检测到，通知内核，由内核为该条件发生时正运行的进程产生适当的信号。）

③进程调用 kill（2）函数可将信号发送给另一个进程或进程组。限制：接收信号进程和发送信号进程的所有者必须相同，或者发送信号进程的所有者必须是超级用户。

④用户可用 kill（1）命令将信号发送给其他进程。

⑤当检测到某种软件条件已经发生，并应将其通知有关进程时也产生信号。

因为信号是随机出现的，所以要告诉内核“当此信号出现时，执行何种操作”。一般要求内核在某个信号出现时以下列三种方式之一进行处理，我们称之为信号的处理或者与信号相关的动作：

①忽略此信号。如果忽略某些由硬件异常产生的信号，则进程的运行行为是未定义的。

②捕捉信号。即通知内核在某种信号发生时调用一个用户函数。

（注意：不能捕捉或忽略 SIGKILL 和 SIGSTOP 信号，因为它们向超级用户提供了使进程终止和停止的可靠方法）

③执行系统默认动作。见 P235 页的表 10-1。

signal 函数——指定当某个信号发生时内核的处理方式。见 P240。从下图可看出，对于 signal 函数的限制：不改变信号的处理方式就不能确定信号的当前处理方式。

很多捕捉这两个信号的交互式程序具有下列形式的代码：

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

这样处理后，仅当信号当前未被忽略时，进程才会捕捉它们。

当执行一个程序时，所有信号的最初状态都是系统默认或忽略。（当用 signal 设置对一

个信号捕捉后，不论信号发生多少次都一直是对其捕捉，除非再用函数将其设置为系统默认或忽略。）对于 exec 函数，它会将原先设置为要捕捉的信号都更改为它们的默认动作，其他信号的状态则不变。

早期 UNIX 实现的不可靠信号机制

不可靠的原因包括：信号丢失、进程难以捕捉到信号等；

主要的问题：系统处理信号时，总是立即将其信号的处理方式复位为系统默认方式，引起难以捕捉的问题；没有阻塞机制，容易导致信号丢失；

现代 Unix 系统都提供了可靠的信号机制。

如果进程在执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行，该系统调用返回出错，其 errno 被设置为 EINTR。

系统调用分两类：

①低速系统调用——是可能会使进程永远阻塞的一类系统调用，它们包括：

- 在读某些类型的文件（管道、终端设备以及网络设备）时，如果数据并不存在则可能会使调用者永远阻塞。
- 在写这些类型的文件时，如果不能立即接受这些数据，则也可能会使调用者永远阻塞。
- 打开某些文件，在某种条件发生之前也可能会使调用者阻塞（例如，打开终端设备，它要等待直到所连接的调制解调器应答了电话） 。
- pause(按照定义，它使调用进程睡眠直至捕捉到一个信号)、wait 和 waitpid 函数。
- 某种 ioctl 操作。
- 某些进程间通信函数（见第 15 章）。

低速系统调用中的一个例外：与磁盘 I/O 有关的系统调用——虽然读、写一个磁盘文件可能会暂时阻塞调用者，但是除非发生硬件错误，I/O 操作总会很快返回，并使调用者不再处于阻塞状态，所以并不能将与磁盘 I/O 有关的系统调用视为“低速”。

对于被中断的 read、write 系统调用，若已处理了部分数据量，POSIX.1 标准处理为部分成功返回。

②其他系统调用

对于中断的系统调用，应当显式的处理出错返回，但 POSIX.1 允许实现重启动由信号中断的系统调用。即中断的系统调用要么自动重启动，要么显式处理出错返回。现代 Unix 系统对因捕捉信号而返回的系统调用函数都有自动重启动的功能。如果拿不准的话，也可以自己通过包装系统调用而实现其自动重启库的版本。或者利用 sigaction(2)函数显式的指定被信号中断时是否重启动系统调用。

表10-2 几种信号实现所提供的功能

函 数	系 统	保持安装信号处理程序	阻塞信号的能力	被中断的系统调用自动重启动?
signal	ISO C、POSIX.1	未说明	未说明	未说明
	V7、SVR2、SVR3、SVR4、Solaris			决不
	4.2BSD	•	•	总是
	4.3BSD、4.4BSD、FreeBSD、Linux、Mac OS X	•	•	默认
sigset	XSI	•	•	未说明
	SVR3、SVR4、Linux、Solaris	•	•	决不
sigvec	4.2BSD	•	•	总是
	4.3BSD、4.4BSD、FreeBSD、Mac OS X	•	•	默认
sigaction	POSIX.1	•	•	未说明
	XSI、4.4BSD、SVR4、FreeBSD、Mac OS X、Linux、Solaris	•	•	可选

（上表中 linux 系统默认是指自动重启动）

进程捕捉到信号并对其进行处理时，进程正在执行的指令序列就被信号处理程序临时中断。它首先执行该信号处理程序中的指令，如果从信号处理程序返回（例如未调用 exit 或 longjmp），则继续执行在捕捉到信号时进程正在执行的正常指令序列。但如果进程在执行 malloc 时被中断，信号处理程序中又调用 malloc，此时就会出问题，也就是说 malloc 并非可重入函数。

可重入函数——即可被中断，稍后可继续执行的函数。进程可在执行此函数中中断，且可在信号处理程序中再调用一次此函数，不会出问题。

（如果进程在执行不可重入函数时被中断，信号处理程序中再调用此不可重入函数，则结果是不可预测的。）

表10-3 信号处理程序可以调用的可重入函数

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit&_exit	listen	sendmsg	socketpair	write

当在信号处理程序中调用的是上表中的可重入函数，应当在其前保存，在其后恢复errno。

未列入上表中的函数大多是不可重入的，原因为：(a) 已知它们使用静态数据结构。(b) 它们调用 malloc 或 free。或 (c) 它们是标准 I/O 函数。(longjmp 和 siglongjmp 也不可重入。)

现代 Unix 系统中，信号 SIGCLD 等价于 SIGCHLD。

当引发信号的事件发生时，为进程产生一个信号。在产生了信号时，内核通常在进程表中设置一个某种形式的标志。当对信号采取了这种动作时，我们说向进程递送了一个信号，在信号产生和递送之间的时间间隔内，称信号是未决的。

进程可以选用信号递送阻塞。如果为进程产生了一个选择为阻塞的信号，而且对该信号的动作是系统默认或捕捉该信号，则为该进程将此信号保持为未决状态，直到该进程（a）对此信号解除了阻塞，或者（b）将对此信号的动作更改为忽略。内核在递送一个原来被阻塞的信号给进程时（而不是在产生该信号时），才决定对它的处理方式。于是进程在信号递送给它之前仍可改变对该信号的动作。

在进程解除对某个信号的阻塞之前，这种信号发生多次，POSIX.1 允许系统递送该信号一次或多次。如果递送该信号多次，则称对这些信号进行了排队，但大多 Unix 系统，包括 linux 并不对信号进行排队。

如果有多个信号要递送给同一个进程，POSIX.1 并未规定顺序，但 POSIX.1 的 Rationale 建议：在其他信号之前递送与进程当前状态有关的信号。

kill 函数——将信号发送给指定进程或进程组。见 P251。

raise 函数——使进程向自身发送信号。见 P251。

当进程将信号发送给其他进程时需要权限。超级用户可将信号发送给任一进程。对于非超级用户，则发送者的实际或有效用户 ID 必须等于接受者的实际或有效用户 ID。若实现支持_POSIX_SAVED_IDS，则检查接受者的保存的设置用户 ID（而不是其有效用户 ID）。一个特例：如果被发送的信号是 SIGCONT，则进程可将它发送给属于同一

会话的任何其他进程。

如果 kill 产生的信号不被阻塞，那么在 kill 返回之前，就会将该信号传送至该进程。

alarm 函数——用于设置一个计时器，在将来某个指定的时间该计时器会超时。当计时器超时时，内核产生 SIGALRM 信号（每调用一次只产生一次信号）。如果不忽略或不捕捉此信号，则其默认动作是终止调用该 alarm 函数的进程。一般用在 signal 函数之后。见 P252。由于进程调度的延迟，所以进程得到控制从而能够处理该信号还需一些时间。

（alarm 还常用于对可能阻塞的操作设置上限值。）

pause 函数——使调用进程挂起直至捕捉到一个信号。见 P252。只有当执行了一个信号处理程序并从其返回时，pause 才返回。

（若要对 I/O 操作设置时间限制，可以使用 longjmp，当然也要清楚它可能使其他信号处理程序被迫终止。另一种选择是使用 select 或 poll 函数。）

信号屏蔽字——每个进程都有一个信号屏蔽字，它规定了当前要阻塞而不能递送到该进程的信号集。对于每种可能的信号，该屏蔽字中都有一位与之对应。某一位设置为 1，则相应的信号在当前是被阻塞的。（注意不能阻塞 SIGKILL 和 SIGSTOP 信号。）

POSIX.1 定义了一个新类型：sigset_t 类型——用于保存一个信号集。并定义了下列五个处理信号集的函数。见 P257。

①sigemptyset 函数——初始化由参数 set 指向的信号集，清除其中所有信号，使每一位都为 0。

②sigfillset 函数——初始化由参数 set 指向的信号集，使其包括所有信号，即每一位都置 1。

（所有应用程序在使用信号集前，要对该信号集调用 sigemptyset 或 sigfillset 一次。）

③sigaddset 函数——将一个信号添加到现有集中，使信号集中的对应位置 1。

（一旦已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号。）

④sigdelset 函数——从信号集中删除一个信号，使信号集中的对应位置 0。

⑤sigismember 函数——测试信号集中是否包含某一指定信号，即测试此信号对应位是 1 还是 0。

sigprocmask 函数——用来检测和更改当前信号屏蔽字，或在一个步骤中同时执行这两个操作。见 P258。在调用此函数后如果有任何未决的、不再阻塞的信号（即此函数使某些已发生的信号从阻塞变为不阻塞），则在 sigprocmask 返回前，至少会将其中一个信号递送给该进程。（此函数仅为单线程的进程定义。）

(为解除对某个信号的阻塞，一般用旧屏蔽字重新设置进程信号屏蔽字（SIG_SETMASK）。)

sigpending 函数——利用参数返回信号集，此信号集中的各个信号对于调用进程是阻塞不能递送的、未决的（即已发生的）。即此函数用来判定哪些信号是设置为阻塞并处于未决状态的。见 P259。

sigaction 函数——检测或修改与指定信号相关联的处理动作（或同时执行这两种操作）。见 P261。

当捕捉到一个信号时，进入信号捕捉函数，此时当前信号被自动的加到进程的信号屏蔽字中。在调用 longjmp 时有个问题，这阻止了后来产生的这种信号中断该信号处理程序。但是如果用 longjmp 跳出信号处理程序，那么此进程的信号屏蔽字该如何处理呢？为了允许两种形式的行为并存。POSIX.1 提供了下面两个函数：

sigsetjmp 和 siglongjmp 函数——它们与 setjmp 和 longjmp 的唯一区别是 sigsetjmp 增加了一个参数，跟当前信号屏蔽字有关。见 P266。

数据类型 sig_atomic_t：写这种类型的变量时不会被中断。详见 P267。

sigsuspend 函数——在一个原子操作中先恢复信号屏蔽字，然后使进程休眠。此函数将进程的信号屏蔽字设置为由参数 sigmask 指向的值，在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则 sigsuspend 返回，并且将该进程的信号屏蔽字设置为调用 sigsuspend 之前的值。见 P269。

此函数的用处——（a）用于保护临界区，使其不被特定信号中断。

（b）等待一个信号处理程序设置一个全局变量。

（c）可以用信号实现父、子进程之间的同步。

如果在等待信号发生时希望去休眠，则使用 sigsuspend 函数是非常适当的。但是如果在等待信号期间希望调用其他系统函数，将会怎样呢？——详见 P273。

abort 函数——使异常程序终止。此函数将 SIGABRT 信号发送给调用进程（进程不应忽略此信号）。如果捕捉到此信号，则信号处理程序不能返回的唯一方法是它调用 exit、_exit、_Exit、longjmp 或 siglongjmp。让进程捕捉 SIGABRT 的意图是：在进程终止之前由其执行所需的清理操作。如果进程并不在信号处理程序中终止自己，POSIX.1 声明当信号处理程序返回时，abort 终止该进程。POSIX.1 还要求，如果 abort 调用终止进程，则它对所有打开标准 I/O 流的效果应当与进程终止前对每个流调用 fclose 相同。见 P274。

system 函数——

sleep 函数——此函数使调用进程被挂起，直到满足以下条件之一：（a）已经过了参数 seconds 所指定的墙上时钟时间，返回值为 0；（b）调用进程捕捉到一个信号并从信号处理程序返回，由于是提前返回，返回值是未睡够的秒数（seconds 减去实际休眠时间）。见 P280。

作业控制信号：

在表 10-1 中有六个 POSIX.1 认为是与作业控制有关的信号。

- SIGCHLD　子进程已停止或终止。
- SIGCONT　如果进程已停止，则使其继续运行。
- SIGSTOP　停止信号（不能被捕捉或忽略）　。
- SIGTSTP　交互式停止信号。
- SIGTTIN　后台进程组的成员读控制终端。
- SIGTTOU　后台进程组的成员写控制终端。

当对某一个进程产生 SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU 信号使之停止执行时，如果有未决的 SIGCONT 信号存在，则将被丢弃。反之亦然，当对某一个进程产生 SIGCONT 信号使之继续执行时，如果有未决的 SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU 信号存在，则将被丢弃。。

对一个停止的进程产生一个 SIGCONT 信号时，不管该信号是否会被进程阻塞或忽略，都会使进程继续执行。

某些系统（包括 linux）提供数组：extern char *sys_siglist[]；数组下标是信号编号，给出一个指向信号字符串名字的指针。

psignal 函数——类似 perror，见 P284。

strsignal 函数——返回说明指定信号的字符串，类似 strerror。见 P284。

十四．高级 I/O

高级 I/O——包括非阻塞 I/O、记录锁、系统 V 流机制、I/O 多路转接（select 和 poll 函数）、readv 和 writev 函数以及存储映像 I/O（mmap）。

非阻塞 I/O——使我们可以调用 open、read 和 write 这样的操作，并使这些操作不会永远阻塞。如果这种操作不能完成，则调用立即出错返回，表示该操作如继续执行将阻塞。

对于一个给定的描述符有两种方法对其指定非阻塞 I/O：

①如果调用 open 获得描述符，则可指定 0_NONBLOCK 标志。

②对于已经打开的一个描述符，则可调用 fcntl，由该函数打开 O_NONBLOCK 文件状态标志。

POSIX.1 要求，对于一个非阻塞的描述符如果无数据可读，则 read 返回-1，并且 errno 被设置为 EAGAIN(对应的值为 11)。

记录锁——其功能是当一个进程正在读或修改文件的某个部分时，它可以阻止其他进程修改同一文件区。

3.14节中已经给出了fcntl函数的原型，为了叙述方便，这里再重复一次。

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
                                    返回值：若成功则依赖于cmd（见下），若出错则返回-1
```

fcntl 函数——对于记录锁，cmd 是 F_GETLK、F_SETLK 或 F_SETLKW。第三个参数（称其为 flockptr）是一个指向 flock 结构的指针。详细用法见 P358。

进程不能使用 fcntl 函数测试它自己是否在文件的某一部分持有一把锁。调用进程的锁只会被其他进程的锁阻塞，若已有自己的锁在文件的某一部分上，则会被替换。

共享读锁和独占写锁：

基本规则——多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定字节上只能有一个进程独用的一把写锁。

表14-2　不同类型锁之间的兼容性

当前区域状态 \ 请求	读锁	写锁
无锁	允许	允许
一个或多个读锁	允许	拒绝
一个写锁	拒绝	拒绝

单个进程对于文件区间要么是有一把读锁，要么是有一把写锁。如果一个进程对一个文件区间已经有了一把锁，后来该进程又企图在同一文件区间再加一把锁，那么新锁将替换老锁。

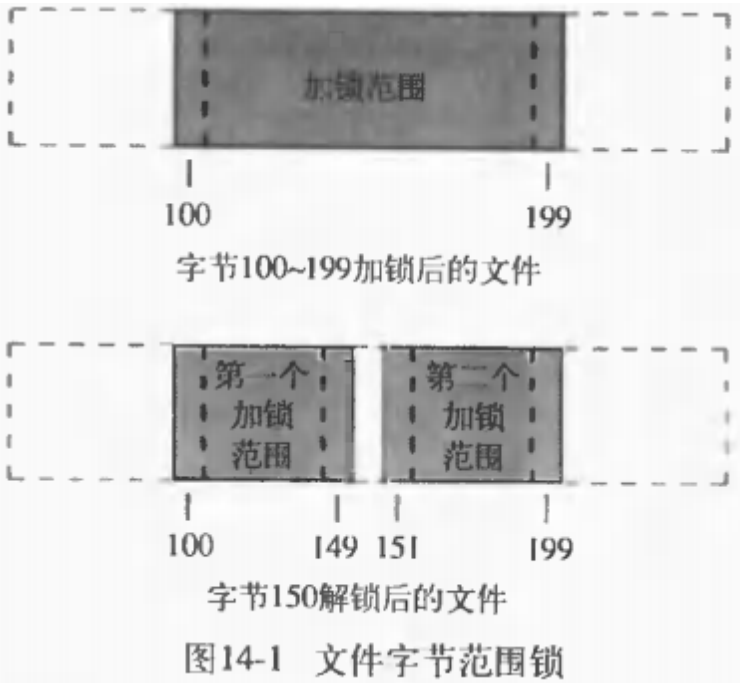


图14-1 文件字节范围锁

死锁状态——如果两个进程相互等待对方持有并且锁定的资源时，则这两个进程就处于死锁状态。检测到死锁时，内核必须选择一个进程接收出错返回。具体选哪个进程与具体系统有关。

记录锁的自动继承与释放有三条规则：

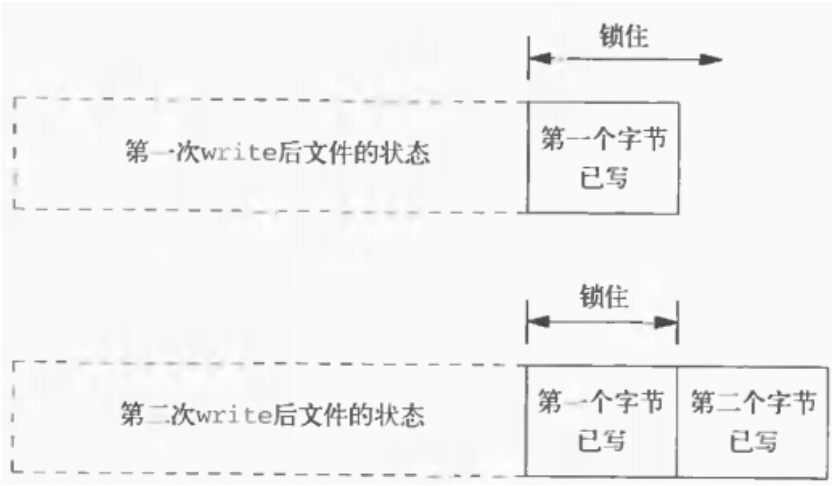
①锁与进程和文件两方面有关。有两重含义：（a）当一个进程终止时，它所建立的锁全部释放。（b）任何时候关闭一个描述符时，则该进程通过这一描述符可以引用的文件上的任何一把锁都被释放（这些锁都是该进程设置的）。

②由 fork 产生的子进程不继承父进程所设置的锁。

③在执行 exec 后，新程序可以继承原执行程序的锁。但是注意：如果对一个文件描述符设置了 close-on-exec 标志，那么当作为 exec 的一部分关闭该文件描述符时，对相应文件的所有锁都被释放了。

在文件尾端加锁：

当对文件的一部分加锁时，内核将指定的偏移量变换成绝对文件偏移量。另外，除了指定一个绝对偏移量（SEEK_SET）之外，fcntl 还允许我们相对于文件中的某个点（当前偏移量（SEEK_CUR）或文件尾端（SEEK_END））指定该偏移量。当前偏移量和文件尾端是可能不断变化的，而这种变化又不应影响现存锁的状态，所以内核必须独立于当前文件偏移量或文件尾端而记住锁，可以使用以下方式：对于 fcntl，可以指定长度为-1。负的长度值表示在指定偏移量之前的字节数。详见 P366。



合作进程——考虑数据库访问例程库，如果该库中所有函数都以一致的方法处理记录锁，则称使用这些函数访问数据库的任何进程集为合作进程。如果这些函数只是用来访问数据库的函数，那么它们使用建议性锁是可行的（但是建议性锁并不能阻止对数据库文件有写权限的任何其他进程对数据库文件进行随意的写操作）。

非合作进程——没有使用被认可的方法（数据库函数库）访问数据库的进程是一个非合作进程。

强制性锁——也称强迫方式锁，它使内核对每一个 open、read 和 write 系统调用都进行检查，检查调用进程对正在访问的文件是否违背了某一把锁的作用。对一个特定文件打开其设置组 ID 位并关闭其组执行位，则对该文件开启了强制性锁机制。

一个别有用心的用户可以对大家都可读的文件加一把读锁（强制性），这样就能阻止任何其他人写该文件。

关于强制锁的详细用法，参考 P366。

STREAMS（流）—— 略。

图14-6 telnet程序概观

I/O 多路转接——上图中所示，telnet 进程有两个输入、两个输出。对这两个输入中的任一个都不能使用阻塞 read，处理这种特殊问题的好方法就是使用 I/O 多路转接。

基本原理：先构造一张有关描述符的列表，然后调用一个函数，直到这些描述符中的一个已准备好进行 I/O 时或函数超时，该函数才返回。在返回时，它告诉进程哪些描述符已准备好可以进行 I/O。执行 I/O 多路转接的函数有 poll、pselect 和 select 函数。

select 函数——用法见 P381。

传向 select 的参数告诉内核：

(a)我们所关心的描述符。

(b)对于每个描述符我们所关心的状态（是否读一个给定的描述符？是否想写一个给定的描述符？是否关心一个描述符的异常状态？）。

(c)愿意等待多长时间（可以永远等待，等待一个固定量时间，或完全不等待）。

从 select 返回时，内核告诉我们：

(a)已准备好的描述符的数量。

(b)对于读、写或异常这三个状态中的每一个，哪些描述符已准备好。

使用这种返回信息，就可调用相应的 I/O 函数（一般是 read 或 write），并且确知该函数不会阻塞。

一个描述符阻塞与否并不影响 select 是否阻塞。也就是说，如果希望读一个非阻塞描述符，并且以超时值为 5 秒调用 select，则 select 最多阻塞 5 秒。相类似，如果指定一个无限的超时值，则 select 阻塞到对该描述符数据准备好，或捕捉到一个信号。

如果在一个描述符上碰到了文件结尾处，则 select 认为该描述符是可读的。然后调用 read，它返回 0，这是 UNIX 指示到达文件结尾处的方法。

pselect 函数——是 select 的变体，与其只有几点区别，见 P384。

poll 函数——类似于 select，但其程序员接口有所不同。见 P384。

select 和 poll 的可中断性——见 P386。

异步 I/O——由 BSD 和系统 V 派生的所有系统提供了使用一个信号的异步 I/O 方法，该信号通知进程某个描述符已经发生了所关心的某个事件。异步 I/O 的一个限制是：每个进程只有一个信号。如果要对几个描述符进行异步 I/O，那么在进程接受到该信号时并不知道这一信号对应于哪一个描述符。

在 BSD 派生的系统中，异步 I/O 是 SIGIO 和 SIGURG 两个信号的组合。前者是通用异步 I/O 信号；后者只是用来通知进程在网络连接上到达了带外的数据，该信号仅对引用支持带外数据的网络连接描述符而产生。

为了接收 SIGIO 信号，需执行下列三步，为了接收 SIGURG，需执行下面的①②两步：

①调用 signal 或 sigaction 为 SIGIO 信号建立信号处理程序。

②以命令 F_SETOWN 调用 fcntl 来设置进程 ID 和进程组 ID，它们将接受对于该描述符的信号。

③以命令 F_SETFL 调用 fcntl 设置 O_ASYNC 文件状态标志，使在该描述符上可以进行异步 I/O。（最后一步只能指向终端或网络的描述符执行。）

readv 和 writev 函数——用于在一次函数调用中读、写多个非连续缓冲区。见 P387。

管道、FIFO 以及某些设备，特别是终端、网络和 STREAMS 设备有下列两种性质：

①一次 read 操作所返回的数据可能少于所要求的数据，即使还没达到文件尾端也可能是这样。这不是一个错误，应当继续读该设备。

②一次 write 操作的返回值也可能少于指定输出的字节数，这可能由若干原因造成。这也不是一个错误，应当继续写余下的数据至该设备。

在读、写磁盘文件时从未见到过上面两种情况，除非文件系统用完了空间，或者我们接近了配额限制，而不能将要求写的数据全部写出。

在读写上面提到的各种文件类型（管道、FIFO 以及某些设备，特别是终端、网络和 STREAMS 设备）时，为解决上面两种性质带来的麻烦，可以调用以下两个函数：

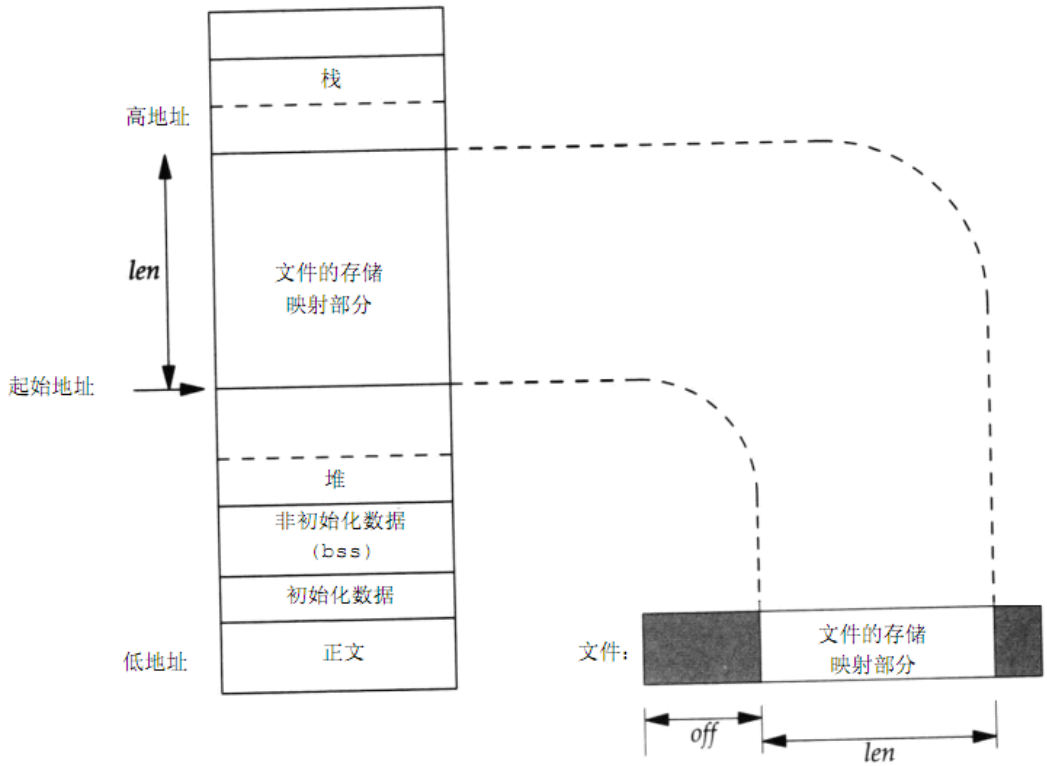
readn 和 writen 函数——它们的功能是读、写指定的 N 字节数据，并处理返回值小于要求值的情况。见 P389。这两个函数只是按需多次调用 read 和 write 直至读、写了 N 字节数据。在要将数据写到上面提到的文件类型上时，就可调用 writen，但是只有当事先就知道要接受数据的数量时，才调用 readn，所以通常还是只调用 read 接收来自这些

设备的数据。

注意：若在已经读、写了一些数据后出错，则这两个函数返回已传输的数据量，而非出错返回。若读时到达文件尾端，已读了一些数据，但未达到要求的量时，与 read 一样，readn 也只返回已读的字节数。

存储映射 I/O——使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节（由内核自动读输入文件到缓冲区中）。与此类似，将数据存入缓冲区，则相应字节就由内核自动地写入文件。这样文件就可以在不使用 read 和 write 的情况下执行 I/O。（数据被写入文件的确切时间依赖于系统的页管理算法。某些系统设置了守护进程，在系统运行期间，它“慢条斯理”地将脏页写到磁盘上。如果想要确保数据安全地写到文件中，则需在进程终止前以 MS_SYNC 标志调用 msync。）

mmap 函数——告诉内核将一个给定的文件映射到一个存储区域中。见 P391。



因为映射文件的起始偏移量受系统虚存页长度的限制，那么如果映射区的长度不是页长度的整数倍时，将如何呢？假定文件长 12 字节，系统页长为 512 字节，则系统通常提供 512 字节的映射区，其中后 500 字节被设为 0。可以修改这 500 字节，但任何变动都不会在文件中反映出来。这样我们就不能利用 mmap 将数据添加到文件中，解决问题的方法就是必须首先加长该文件。见 P394 程序清单 14-12。

与映射存储区相关的信号有 SIGSEGV 和 SIGBUS 两个：

SIGSEGV——通常用于指示进程试图访问对它不可用的存储区。如进程企图存数据到 mmap 指定为只读的映射存储区，那么就产生此信号。

SIGBUS——如果访问存储区的某个部分，而在访问时这一部分实际上已不存在，则产生 SIGBUS 信号。

调用 fork 之后：子进程继承父进程的存储映射区。

调用 exec 之后：新程序不继承原程序的存储映射区。

mprotect 函数——可以更改一个现存映射存储区的权限。见 P393。

msync 函数——如果共享存储映射区的页已被修改，那么调用此函数可将该页冲洗到被

映射的文件中。如果映射是私有的，那么不修改被映射的文件。见 P393。此函数类似于 fsync，但作用于存储映射区。

munmap 函数——进程终止后，或者调用此函数之后，存储映射区就被自动解除映射。但此函数不会影响被映射的对象，即调用 munmap 不会使映射区的内容写到磁盘文件上。见 P393。（关闭文件描述符 filedes 并不解除映射区。）

对于共享存储映射区磁盘文件的更新，在写到存储映射区时按内核虚存算法自动进行。

对于私有存储映射区，在解除了映射后，对其存储映射区的修改被丢弃。

将一个普通文件复制到另一个普通文件中时，存储映射 I/O 比较快。但是有一些限制，如不能用其在某些设备之间（例如网络设备或终端设备）进行复制，并且在对被复制的文件进行映射后，也要注意该文件的长度是否改变。

十五．进程间通信

IPC——各种进程通信方式的统称。

表15-1 UNIX系统IPC摘要

IPC类型	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
半双工管道	•	(全)	•	•	(全)
FIFO	•	•	•	•	•
全双工管道	允许	•,UDS	opt, UDS	UDS	•, UDS
命名全双工管道	XSI 可选	UDS	opt, UDS	UDS	•, UDS
消息队列	XSI	•	•		•
信号量	XSI	•	•	•	•
共享存储	XSI	•	•	•	•
套接字	•	•	•	•	•
STREAMS	XSI 可选		opt		

上表中：

黑点——基本功能得到支持。

UDS——对于全双工管道，经由 UNIX 域套接字支持该特征。

opt——在平台对相应特征以可选择包方式提供支持时，相应位置标示为“opt”。可选择包通常并非是默认安装的。

（前七种只能用于同一台主机的各进程间通信，后两种支持不同主机上各进程间的通信。）

管道：

通常管道有两种局限性：①是半双工的（即数据只能在一个方向上流动）。②只能在具有公共祖先的进程之间使用，比如父子进程之间通信。

半双工管道——有上述两种局限性；FIFO——没有第二种局限性；UNIX 域套接字和命名流管道——两种局限性都没有。

pipe 函数——创建一个半双工管道。见 P398。

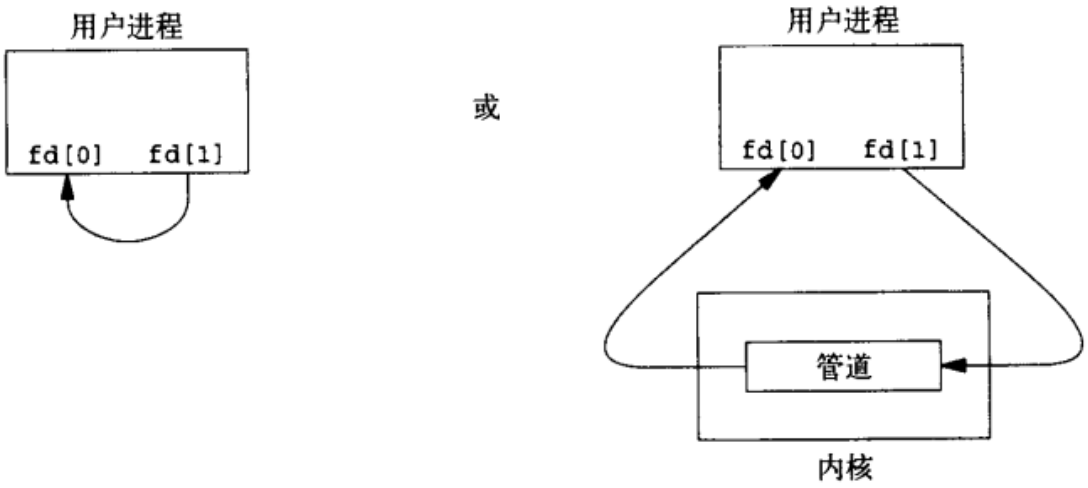


图15-1　观察半双工管道的两种方法

调用 fork 之后做什么取决于我们想要有的数据流的方向。

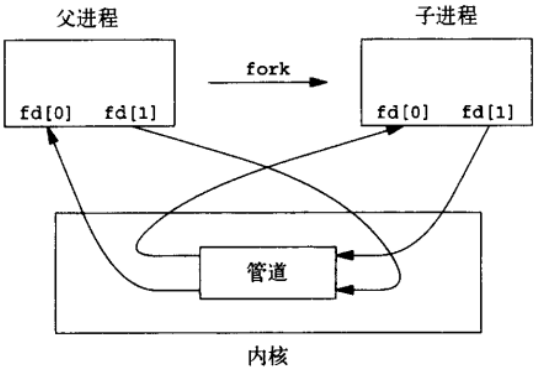


图15-2　调用fork之后的半双工管道

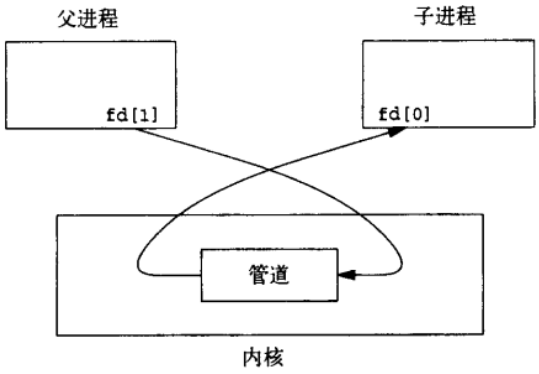


图15-3　从父进程到子进程的管道

上图显示的是父进程关闭读端，子进程关闭写端。

当管道的一端被关闭后，下列两条规则其作用：

①当读一个写端已被关闭的管道时，在所有数据都被读取后，read 返回 0，以指示达到了文件结束处。（可以复制一个管道的描述符，使得有多个进程对它具有写打开文件描述符。）

②如果写一个读端已被关闭的管道，则产生信号 SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则 write 返回-1，errno 设置为 EPIPE。

在写管道（或 FIFO）时，常量 PIPE_BUF 规定了内核中管道缓冲区的大小。

如果对管道调用 write，而且要求写的字节数小于等于 PIPE_BUF，则此操作不会与其他进程对同一管道（或 FIFO）的 write 操作穿插进行。但是，若有多个进程同时写一个管道（或 FIFO），而且有进程要求写的字节数超过 PIPE_BUF 字节数时，则写操作的数据可能相互穿插。PIPE_BUF 确定了可被原子的写到 FIFO 的最大数据量。

pathconf 或 fpathconf 函数——确定 PIPE_BUF 的值。见 P399。

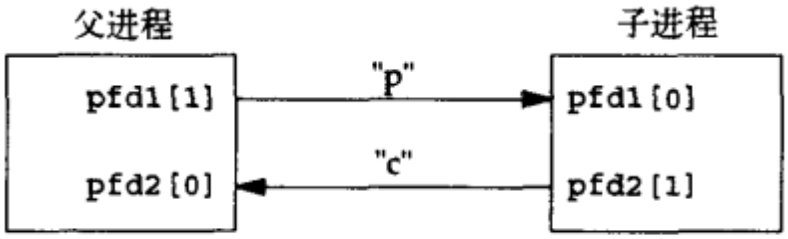


图15-4 用两个管道实现父子进程同步

popen 函数——创建一个管道，调用一个 fork 产生一个子进程，关闭管道的不使用端，然后调用 exec，执行一个 shell 以运行命令，并返回一个标准 I/O 文件指针（流）。见 P403。

（popen 绝不应由设置用户 ID 或设置组 ID 程序调用。）

pclose 函数——关闭标准 I/O 流，等待上面的 shell 命令终止，然后返回 shell 的终止状态。若上面的 shell 不能被执行，则函数返回的终止状态与 shell 已执行 exit（127）一样。见 P403。

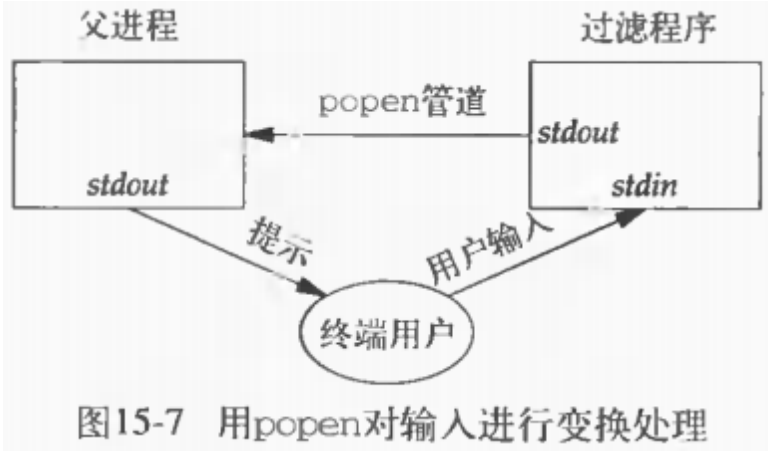


图15-7 用popen对输入进行变换处理

协同进程——注意上图中的过滤程序，当一个程序产生某个过滤程序的输入，同时又读取该过滤程序的输出时，则该过滤程序就成为协同进程。

与 popen 只提供连接到另一个进程的标准输入或标准输出的一个单向管道不同，对于协同进程，它有连接到另一个进程的两个单向管道——一个接到其标准输入，另一个则来自其标准输出。我们先要将数据写到其标准输入，经其处理后，再从其标准输出读取数据。

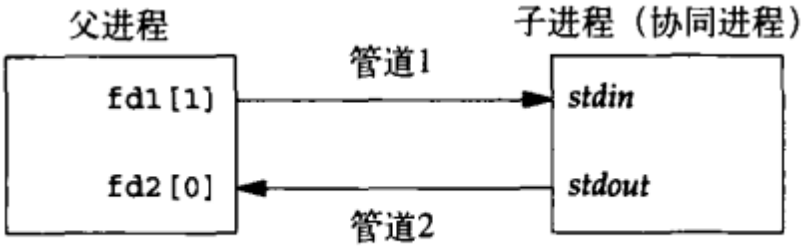


图15-8 写协同进程的标准输入，读它的标准输出

FIFO——被称为命名管道，也是一种文件类型。它与管道不同，管道只能由相关进程使用（这些相关进程的共同的祖先进程创建了管道），而通过 FIFO，不相关的进程也能交换数据。

mkfifo 函数——创建一个 FIFO（大多系统中，mkfifo 调用 mknod 创建 FIFO）。见 P412。

一旦用此函数创建了一个 FIFO，一般的文件 I/O 函数（open、write、read、close）都可用于 FIFO。

当打开一个 FIFO 时，非阻塞标志（O_NONBLOCK）产生下列影响：

①如果没有指定 O_NONBLOCK，只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。

②如果指定了 O_NONBLOCK，只读 open 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 open 将出错返回-1，其 errno 是 ENXIO。

类似于管道，若用 write 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。

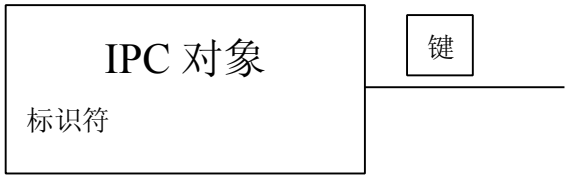
一个 FIFO 有多个写进程是很常见的，要用原子写操作，与管道一样，PIPE_BUF 说明了可被原子的写到 FIFO 的最大数据量。

FIFO 的两种用途：

①FIFO 由 shell 命令使用以便将数据从一条管道线传送到另一条，为此无需创建中间临时文件。

②FIFO 用于客户进程-服务器进程应用程序中，以在客户进程和服务器进程之间传递数据。

XSI IPC——有三种 IPC 我们称作 XSI IPC，即消息队列、信号量以及共享存储器，它们之间有很多相似之处。



IPC 对象：①内部名——标识符；②外部名——键。

ftok 函数——由一个路径名和项目 ID 产生一个键。见 P416。

通常情况下，由于两个路径名引用两个不同的文件，那么，对这两个路径名调用 ftok 通常返回不同的键。但是，因为 i 节点和键通常都存放于长整型中，于是创建键时可能会丢失信息。这意味着如果使用同一项目 ID，那么对于不同文件的两个路径名可能产生相同的键。

作者：zhaolong（安徽理工大学研究生） QQ:419469849 闭门思过

注：本笔记包含书中的部分章节，只是本人对 unix 环境高级编程的个人理解，难免有些理解上的误区，望大家指正。内容只可参考，不可尽信。