

UNIX 操作系统

2016-3-2

作者：朱林峰

目录

UNIX 操作系统	5
1. 概述	5
2. I/O	5
2.1 文件系统	5
2.1.1 文件系统定义	5
2.1.2 常见文件系统	6
2.1.3 文件系统建立过程	6
2.1.4 文件系统	6
2.1.5 文件类型	7
3.2 文件描述符	8
2.3 文件共享	8
2.4 文件信息结构体	9
2.5 文件访问权限	10
2.6 缓冲	10
2.7 I/O 多路转接	10
2.7.1 为什么需要 I/O 多路转接	10
2.7.2 select	11
2.7.3 poll	11
2.7.4 epoll	12
2.7.5 select & epoll 比较	15
2.8 I/O 模型	16
3. 进程	18
3.1 进程环境	18
3.1.1 进程标识符	18
3.1.2 命令行参数	18
3.1.3 环境变量	18
3.1.4 资源限制	18
资源限制可以由下列函数查看或改变：	19
3.1.5 进程空间	19

3.2 创建进程.....	20
3.3 加载程序.....	22
3.3.1 函数接口	22
3.3.2 exec 函数执行过程	22
3.4 程序运行.....	23
3.5 程序退出.....	23
3.6 进程终止状态.....	24
3.7 进程流程.....	24
4. 进程通信	24
4.1 无名管道.....	24
4.2 命名管道 (FIFO).....	25
4.3 XSI IPC.....	25
4.4 消息队列.....	25
4.5 共享内存.....	25
4.6 信号量	26
4.7 UNIX 域套接字	26
5. 线程.....	26
5.1 线程环境.....	26
5.1.1 多线程的好处	26
5.1.2 线程的公共数据	26
5.1.3 线程的私有数据	26
5.1.4 线程的属性.....	27
5.2 线程开始.....	27
5.3 线程终止.....	27
5.3.1 线程终止方式	27
5.3.2 线程终止流程	27
5.3.3 线程终止接口	27
5.4 线程同步.....	28
5.4.1 线程安全函数	28
5.4.2 互斥量.....	28
5.4.3 读写锁.....	28

5.4.4 条件变量	29
5.4.5 避免死锁	29
5.5 线程和信号	29
5.6 线程和fork.....	30
6. 信号.....	30
6.1 信号	30
6.2 产生信号的条件.....	30
6.3 信号的处理.....	30
6.3.1 信号的一般处理方式	30
6.3.2 启动进程与加载程序	30
6.3.3 中断的系统调用	30
6.3.3 可重入函数.....	31
7. 守护进程	32
7.1 编程规则.....	32
7.2 守护进程惯例.....	32

UNIX 操作系统

1. 概述

1. UNIX 系统的体系结构



图1-1 UNIX操作系统的体系结构

2. 系统调用和库函数

UNIX 系统调用接口在 UNIX 系统手册第二册列出，C 库函数在系统手册第三册给出。手册所在目录为：

```
/user/share/man
```

在 mac 上统计：

- 系统调用一共有 512 个
- C 库函数调用一共有 118176 个

在 linux 上统计：

- 系统调用一共有 1192 个
- C 库函数调用一共有 3356 个

C 库函数是对系统调用的封装，很多 C 库函数最终都会调用系统函数，但也有很多 C 库函数不进行任何系统调用。系统调用是操作系统内核的入口点，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。

3. 出错处理

当 UNIX 函数出错时，常常返回一个负值，而且整型变量 errno 通常被设置为包含有附加信息的一个值。

文件 <errno.h> 中定义了符号 errno 以及可以赋值的各种常量。UNIX 系统手册第二部分的第一页 intro(2) 列出了这些常量，可以用 man 命令查看：

```
man 2 intro
```

对于 errno 应知道两点：

- 如果函数调用没有出错，errno 值不会被覆写
- 任何函数都不会将 errno 值设为 0，error 的所有取值都不为 0

```
#include <sdtio.h>
char *strerror(int errnum);
    //返回某个 errno 的字符串描述
void perror(const char *msg);
    //基于当前 errno 的值，在标准出错上产生一条消息，消息以 msg 开头
```

2. I/O

2.1 文件系统

2.1.1 文件系统定义

文件系统定义了把文件存储于磁盘时所必须的数据结构及磁盘数据的管理方式。我们知道，磁盘是由很多个扇区 (Sector) 组成的，如果扇区之间不建立任何的关系，写入其中的文件就无法访问，因为无法知道文件从哪个扇区开始，文件占用多少个扇区，文件有什么属性。为了访问磁盘中的数据，就必须在扇区之间建立联系，

也就是需要一种逻辑上的数据存储结构。建立这种逻辑结构就是文件系统要做的事情，在磁盘上建立文件系统的过程通常称为“格式化”。

2.1.2 常见文件系统

- FAT/FAT32/NTFS
- EXT2/EXT3/ReiserFS
- JFS/UFS/VXFS

上面列举了几种常见的文件系统，Windows 系统下有 FAT、FAT32、NTFS，Linux 下有 EXT2、EXT3、ReiserFS 等，IBM AIX 下有 JFS，SUN solaris 下有 UFS，HP-UX 下有 VXFS，还有很多产商自己开发的文件系统。

表1-1 Linux文件系统

文件系统	描述
ext	Linux扩展文件系统，最早的Linux文件系统
ext2	第二扩展文件系统，在ext的基础上提供了更多的功能
ext3	第三扩展文件系统，支持日志功能
ext4	第四扩展文件系统，支持高级日志功能
hpfs	OS/2高性能文件系统
jfs	IBM日志文件系统
iso9660	ISO 9660文件系统（CD-ROM）
minix	MINIX文件系统
msdos	微软的FAT16
ncp	Netware文件系统
nfs	网络文件系统
ntfs	支持Microsoft NT文件系统
proc	访问系统信息
ReiserFS	高级Linux文件系统，能提供更好的性能和硬盘恢复功能
smb	支持网络访问的Samba SMB文件系统
sysv	较早期的Unix文件系统
ufs	BSD文件系统
umsdos	贮存在msdos上的类Unix文件系统
vfat	Windows 95文件系统（FAT32）
XFS	高性能64位日志文件系统

2.1.3 文件系统建立过程

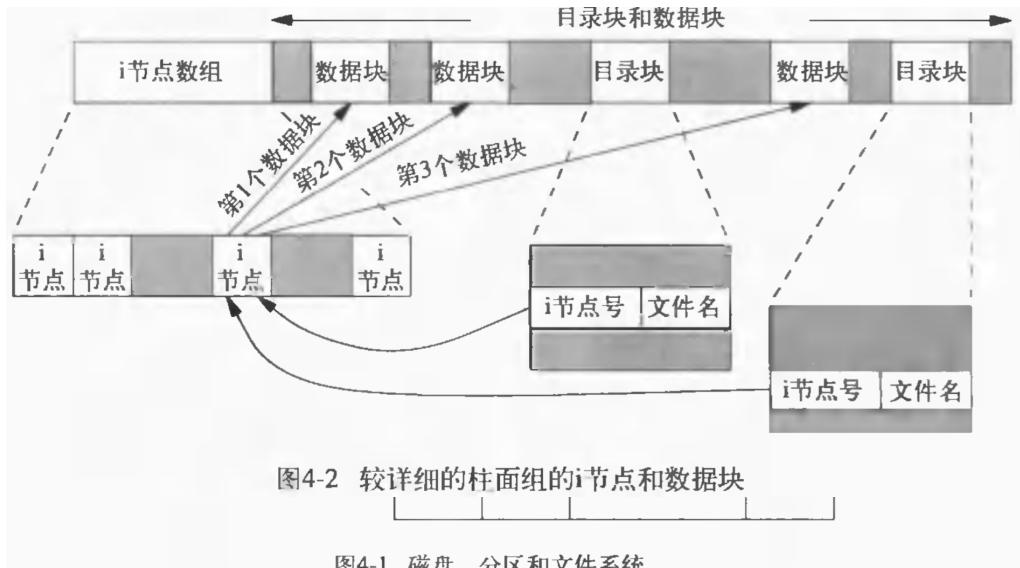
物理磁盘初始化 → 分区 → 格式化(高级) → 装载

- 一块物理磁盘，一般在出厂后就已经初始化了，主要是对盘片划分道、扇区，以及磁头放置初始位置等。
- 初始化后，我们利用分区工具对磁盘进行分区，就是在一块物理磁盘上进行逻辑划分成一个一个分区，便于进行管理。在 Linux 下，可以使用如 Fdisk 工具进行分区。
- 分区后，就可以在上面建立文件系统，建立文件系统就是根据特定的文件系统格式，对分区上的扇区块进行编号和重新组织，这个建立文件系统的过程通常叫高级格式化，与高级格式化对应的是低级格式化，比如将所有扇区清零就是一种低级格式化。
- 建立文件系统后，对于 Linux 和 UNIX 系统，需要进行装载到内核，以便系统进行统一管理，使用完之后要进行卸载。

2.1.4 文件系统

我们可以把一个磁盘分成一个或多个区，每个分区可以包含一个文件系统。

1. 图4-2中有两个目录项指向同一 i 节点。每个 i 节点中都有一个链接计数，其值是指向该 i 节点的目录项数。只有当链接计数减少至 0 时，才可删除该文件。这就是为什么“解除对一个文件的链接”操作并不总意味着“释放该文件占用的磁盘块”的原因。这也是为什么删除一个目录项的函数被称为 unlink 而不是 delete 的原因。在 stat 结构中，链接计数包含在 st_nlink 中，其基本数据类型是 nlink_t。这种链接类型称为硬链接。
2. 另外一种链接类型称为符号链接。对于这种链接，该文件的内容包含了该符号链接所指向文件的名字。
3. i 节点包含了大多数与文件有关的信息：文件类型、文件访问权限位、文件长度和指向该文件所占用的数据块的指针等等。stat 结构中的大多数信息都取值 i 节点，只有两项数据存放在目录项中：文件名和 i 节点编号。i 节点编号的数据类型是 ino_t。



4. 每个文件系统各自对它们的 i 节点编号，因此目录项中的 i 节点编号指向同一个文件系统中的相应 i 节点，不能使一个目录项指向另一个文件系统的 i 节点。这就是为什么 ln 命令不能跨越文件系统的原因。
5. 当在不更换文件系统的情况下为一个文件更名时，该文件的实际内容并未移动，只需构造一个指向现有 i 节点的新目录项，并解除与旧目录项的链接。这就是 mv 命令的通常操作方式。

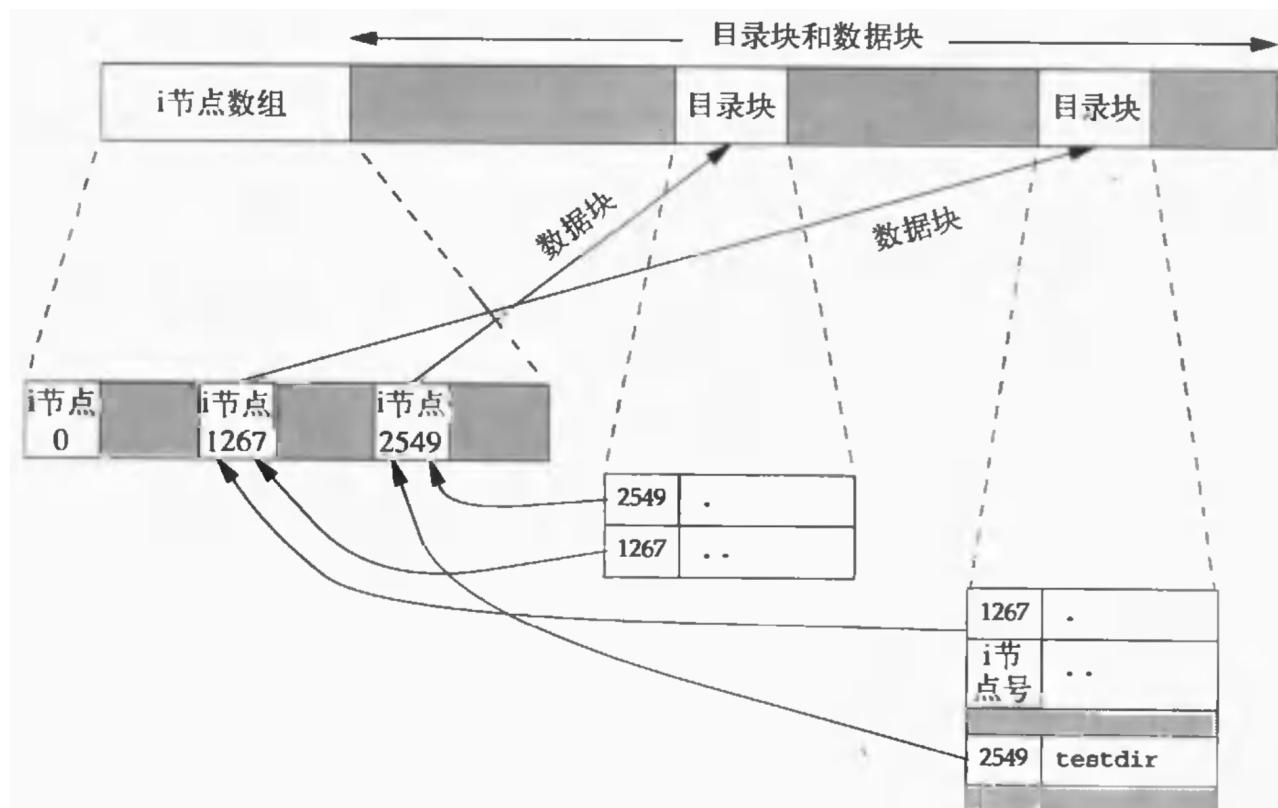


图4-3 创建了目录 testdir 后的示例柱形组

6. 图4.3中，对于编号为 2549 的节点，其类型字段表示它是一个目录，而链接计数为2。任何一个叶目录（不包含其他目录的目录）的链接计数总是 2，数值 2 来自于命名该目录的目录项以及在该目录中的 \. 项。对于编号为 1267 的 i 节点，其类型字段表示它是一个目录，而其链接计数大于或等于 3，因为至少有三个目录指向它。

2.1.5 文件类型

1. 普通文件。这是最常用的文件类型，这种文件包含了某种形式的数据。至于这种数据是文本还是二进制，对于 UNIX 内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序进行。

可执行的二进制文件是个例外，为了执行程序，内核必须理解其格式。所有二进制可执行文件都遵循一种格式，这种格式使内核能够确定程序文本和数据的加载位置。

2. 目录文件。这种文件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。
3. 块特殊文件。这种文件类型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。这种文件类型提供对设备不带缓冲的访问，访问长度可变。系统中的所有设备，要么是字符特殊设备，要么是块特殊设备。
5. FIFO。这种类型文件用于进程间通信，有时也将其称为命名管道。
6. 套接字。这种文件类型用于进程间的网络通信。套接字也可用于进程之间的非网络通信。
7. 符号链接。这种文件类型指向另一个文件。

表4-3 不同类型文件的统计值和百分比

文件类型	统计值	百分比 (%)
普通文件	226 856	88.22
目录	23 017	8.95
符号链接	6 442	2.51
字符特殊	447	0.17
块特殊	312	0.12
套接字	69	0.03
FIFO	1	0.00

3.2 文件描述符

对内核而言，所有打开的文件都通过文件描述符引用。文件描述符是一个非负整数。当打开一个现有文件或创建一个新文件时，内核向进程返回一个文件描述符。

通常，STDIN_FILENO = 0, STDOUT_FILENO = 1, STDERR_FILENO = 2 这三个文件描述符在进程被创建时默认是打开的。

试图操作一个没有打开的文件描述符，或是以不恰当的方式操作一个需要相应权限的文件描述符，都会返回错误，errno = 9：

```
errno = 9 : BADF Bad file descriptor. A file descriptor argument was out of range, referred to no open file, or a read (write) request was made to a file that was only open for writing(reading).
```

2.3 文件共享

内核使用三种数据结构表示打开的文件，它们之间的关系决定了在文件共享方面一个进程对另一个进程可能产生的影响。

8. 每个进程在进程表中都有一个记录项，记录项中包含有一张打开文件描述符的表，每个描述符占用一项。与每个描述符相关联的是：
 - 文件描述符标志
 - 指向一个文件表项的指针
9. 内核为每个打开文件维持一张文件表，每个文件表包含：
 - 文件状态标志（读、写、同步、阻塞等）
 - 当前文件偏移量
 - 指向该文件 v 节点表项的指针
10. 每个打开文件（或设备）都有一个 v 节点（v-node）结构。v 节点包含了：
 - 文件类型
 - 此文件进行各种操作的函数的指针。
 - i 节点（i-node，索引节点）。
11. i 节点包含：
 - 文件所有者
 - 文件长度

- 文件所在设备
- 指向文件实际数据块在磁盘上所在位置的指针等

这些信息是在打开文件时从磁盘上读入内存的，所以所有关于文件的信息都是快速可供使用的。

如果两个独立进程各自打开了同一文件，打开文件的每个进程都得到一个文件表项，但对一个给定的文件只有一个 v 节点表项。每个进程都有自己的文件表项的一个理由是：每个进程都有自己对该文件的当前偏移量。然而当多个进程同时访问一个文件时，可能产生意外。

2.4 文件信息结构体

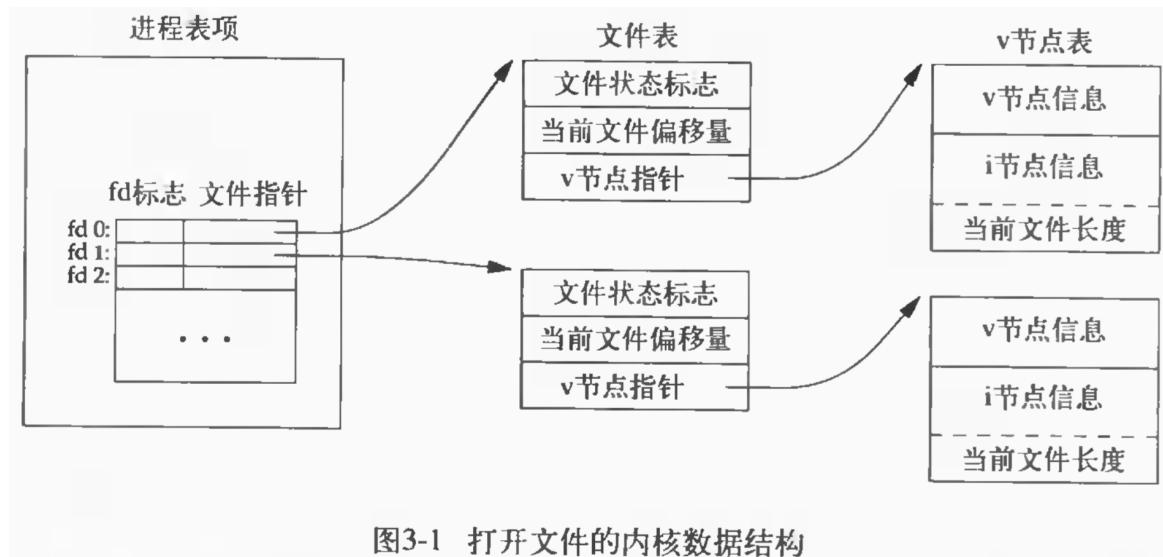


图3-1 打开文件的内核数据结构

```

stat.h

struct stat {
    dev_t      st_dev;      /* [XSI] ID of device containing file */
    ino_t      st_ino;      /* [XSI] File serial number */
    mode_t     st_mode;     /* [XSI] Mode of file (see below) */
    nlink_t    st_nlink;    /* [XSI] Number of hard links */
    uid_t      st_uid;      /* [XSI] User ID of the file */
    gid_t      st_gid;      /* [XSI] Group ID of the file */
    dev_t      st_rdev;     /* [XSI] Device ID */
#ifndef _POSIX_C_SOURCE || defined(_DARWIN_C_SOURCE)
    struct timespec st_atimespec; /* time of last access */
    struct timespec st_mtimespec; /* time of last data modification */
    struct timespec st_ctimespec; /* time of last status change */
#else
    time_t      st_atime;    /* [XSI] Time of last access */
    long       st_atimensec; /* nsec of last access */
    time_t      st_mtime;    /* [XSI] Last data modification time */
    long       st_mtimensec; /* last data modification nsec */
    time_t      st_ctime;    /* [XSI] Time of last status change */
    long       st_ctimensec; /* nsec of last status change */
#endif
    off_t      st_size;     /* [XSI] file size, in bytes */
    blkcnt_t   st_blocks;   /* [XSI] blocks allocated for file */
    blksize_t  st_blksize;  /* [XSI] optimal blocksize for I/O */
    __uint32_t st_flags;    /* user defined flags for file */
    __uint32_t st_gen;     /* file generation number */
    __int32_t  st_lspare;   /* RESERVED: DO NOT USE! */
    __int64_t  st_qspare[2];/* RESERVED: DO NOT USE! */
};


```

2.5 文件访问权限

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

最后9个常量分成3组，因为

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR  
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP  
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

2.6 缓冲

内核提供的 I/O 是不带缓冲的。标准 I/O 库提供缓冲的目的是尽可能减少使用 read 和 write 的次数。它也对每个 I/O 流自动进行缓冲管理。标准 I/O 提供了三种类型的缓冲：

1. 全缓冲。这种情况下，在填满标准 I/O 缓冲区后才进行实际的 I/O 操作。对于驻留在磁盘上的文件通常是由标准 I/O 库实施全缓冲的。
2. 行缓冲。在这种情况下，当输入和输出中遇到换行符时，标准 I/O 库执行 I/O 操作。这允许我们一次输出一个字符，但只有在写了一行之后才进行实际 I/O 操作。当流涉及一个终端时，通常使用行缓冲。
行缓冲有个限制，因为标准 I/O 库用来收集每一行的缓冲区是固定的，所以只要填满了缓冲区，那么即使还没有换行符，也进行 I/O 操作。
3. 不带缓冲。标准 I/O 不对字符进行缓冲存储。如 fputs 函数被调用后，立即执行实际的 I/O。

标准出错是不带缓冲的；标准输入和标准输出若涉及交互式设备，则是带行缓冲的，否则是全缓冲的。

2.7 I/O 多路转接

2.7.1 为什么需要 I/O 多路转接

问题描述：程序同时有多个输入输出 I/O，该怎么处理？

1. 对每个 I/O 都使用阻塞 I/O。那么可能长时间阻塞在一个描述符上，而另一个描述符虽然有很多数据却不能得到及时处理。

2. 多进程处理。每个进程都处理一个阻塞 I/O。这一是增加了程序的复杂性，二是多个 I/O 之间的数据无法得到有效的交流共享。
3. 多线程。每个线程处理一个 I/O。这样线程之间的同步问题也会使程序变得复杂。
4. 使用非阻塞 I/O。将 I/O 都设置为非阻塞的，没有数据时就立即返回。但是不可避免的需要在不同的描述符之间来回轮询，浪费 CPU 时间。
5. 使用异步 I/O。告诉内核，当描述符可读或可写时，通知进程进行处理。但并非所有系统都支持这种机制；同时这种信号对每个进程也只有一个，当该信号对多个描述符起作用，那么接到此信号无从判断是哪一个描述符已准备好进行 I/O。

因此需要 poll, select, pselect, epoll 等函数进行 I/O 多路转接。

2.7.2 select

```
#include <sys/select.h>

int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

1. 工作原理

传向 select 的参数告诉内核：

- 我们所关心的描述符
- 对于每个描述符我们所关心的状态（是否读一个给定描述符？是否写一个给定描述符？是否关心一个描述符的异常状态？）
- 愿意等待多长时间（完全不等待，等待一个固定量时间，永远等待）

从 select 函数返回时，内核告诉我们：

- 已准备好的描述符的数量
- 对于读、写或异常这三个状态中的一个，哪些描述符已准备好

使用这些返回信息，就可调用相应的 I/O 函数，并且确知该函数会阻塞。

2. 参数介绍

- timeval *tvptr
 - tvptr == NULL：永远等待，直到其中一个描述符准备好或捕捉到一个信号
 - tvptr->tv-sec == 0：完全不等待，测试所有指定的描述符并立即返回。这是得到多个描述符的状态而不阻塞 select 函数的轮询方法。
 - tvptr->tv-sec != 0 || tvptr->tv-usec != 0：等待指定时间。如果有描述符准备好或有信号或超时则返回。
- readfds、writefds、exceptfds：指向描述符集的指针，每个描述符集都是个很大的位图（有1024位），某一位置 1 即表示关心某个描述符。如果这三个参数全为 NULL，那么 select 函数就是个比 sleep 更精准的计时器。
- maxfdp1：最大描述符加 1。如果描述符集中的最大描述符为 100，那么 maxfdp1 = 101。它指明了在描述符集中进行搜索的范围，内核只需要在此范围内寻找打开的位，而不必对三个描述符集中的几千个位搜索。

3. 返回值

- -1：表示出错，比如捕捉到信号。这种情况下，将不修改其中的任何描述符集，对信号处理完毕可立即再次调用 select。
- 0：指定时间超时而没有描述符准备好。此时，所有描述符集皆被清零。
- >0：表示已经准备好的描述符数，该值是三个描述符集已准备好的描述符之和，所以如果同一描述符已准备好读和写，那么返回值将 +2. 这种情况下，已准备好的描述符不会被清零。

对于准备好的语义，有一些说明：

- 若对于读集中的一个描述符的 read 操作不会被阻塞，则此描述符是准备好的
- 若对于写集中的一个描述符的 write 操作不会被阻塞，则此描述符是准备好的
- 若异常状态集中的一个描述符有一个未决异常状态，则此描述符是准备好的。异常状态包括：a 在网络连接上到达的带外数据，b 在处于数据包模式的伪终端上发生了某些状态。对于读、写异常状态，普通描述符总是返回准备好。

2.7.3 poll

```
#include <poll.h>

struct pollfd {
    int fd;           //file descriptor to check or < 0 to ignore
    short events;    //events of interest on fd
    short revents;   //events that occurred on fd
};

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
    //准备就绪的描述符数, 超时返回 0, 出错返回 -1
```

我们通过 events 成员告诉内核我们关心什么, 然后内核通过 revents 成员告诉我们实际发生了什么。

1. 参数介绍

events & revents 取值表

表14-6 poll的events和revents标志

标志名	输入至 events?	从revents 得到结果?	说明
POLLIN	•	•	不阻塞地可读除高优先级外的数据 (等效于 POLLRDNORM POLLRBAND)
POLLRDNORM	•	•	不阻塞地可读普通数据 (优先级波段为0)
POLLRBAND	•	•	不阻塞地可读非0优先级波段数据
POLLPRI	•	•	不阻塞地可读高优先级数据
POLLOUT	•	•	不阻塞地可写普通数据
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	不阻塞地可写非0优先级波段数据
POLLERR		•	已出错
POLLHUP		•	已挂断
POLLNVAL		•	描述符不引用一打开文件

2.7.4 epoll

```
#include <sys/epoll.h>

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} poll_data_t;

struct epoll_event {
    uint32_t events;      /* Epoll events */
    epoll_data_t data;    /* User data variable */
};

int epoll_create(int size);
    //On success, these system calls return a nonnegative file descriptor.
    //On error, -1 is returned, and errno is set to indicate the error.
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
    //success 0, error -1
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
    //success, number of file descriptors ready for the requested I/O
    //timeout, 0
    //error, -1
```

epoll 是 Linux 内核为处理大批句柄而作改进的 poll (好像只有 linux 有 epoll) , 是 Linux 下多路复用IO接口 select/poll 的增强版本, 它能显著的减少程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。因为它会复用文件描述符集合来传递结果而不是迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合, 另一个原因就是获取事件的时候, 它无须遍历整个被侦听的描述符集, 只要遍历那些被内核 IO 事

件异步唤醒而加入 Ready 队列的描述符集合就行了。epoll 除了提供 select\poll 那种IO事件的电平触发(Level Triggered)外，还提供了边沿触发(Edge Triggered)，这就使得用户空间程序有可能缓存 IO 状态，减少 epoll_wait epoll_pwait 的调用，提供应用程序的效率。

2. 工作方式

LT(level triggered): 水平触发，缺省方式，同时支持 block 和 no-block socket，在这种做法中，内核告诉我们一个文件描述符是否被就绪了，如果就绪了，你就可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错的可能性较小。传统的select\poll都是这种模型的代表。

ET(edge-triggered): 边沿触发，高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪状态时，内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如：你在发送、接受或者接受请求，或者发送接受的数据少于一定量时导致了一个EWOULDBLOCK错误)。但是请注意，如果一直不对这个fs做IO操作(从而导致它再次变成未就绪状态)，内核不会发送更多的通知。

区别：LT事件不会丢弃，而是只要读buffer里面有数据可以让用户读取，则不断的通知你。而ET则只在事件发生之时通知。

3. epoll_create

创建一个epoll句柄，参数size用来告诉内核监听的数目。

其实目前 linux 的处理方式是忽略 size 参数，直接进行动态存储分配。但 size 参数还是得传入，且要大于零，以保证兼容性。

4. epoll_ctl

epoll 事件注册函数。

- epfd 为 epoll 的句柄
- op 表示动作，用 3 个宏来表示：EPOLL_CTL_ADD(注册新的fd到epfd)，EPOLL_CTL_MOD(修改已经注册的fd的监听事件)，EPOLL_CTL_DEL(从epfd删除一个fd)；
- fd 为需要监听的描述符
- event告诉内核需要监听的事件

其中events可以用以下几个宏的集合：

EPOLLIN：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）

EPOLLOUT：表示对应的文件描述符可以写

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）

EPOLLERR：表示对应的文件描述符发生错误

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET：将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

5. epoll_wait

等待事件的产生，类似于 select() 调用。参数 events 用来从内核得到事件的集合，maxevents 告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create() 时的 size，参数 timeout 是超时时间（毫秒，0 会立即返回，-1 将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时

- events：返回的数据和注册时传入的数据是一样的

6. 示例代码

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;

/* Set up listening socket, 'listen_sock' (socket(),
bind(), listen()) */

epollfd = epoll_create(10);
if (epollfd == -1)
{
    perror("epoll create");
```

```

    exit(EXIT_FAILURE);
}

ev.events = EPOLLIN;
ev.data.fd = listen_sock;
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1)
{
    perror("epoll_ctl: listen_sock");
    exit(EXIT_FAILURE);
}

for (;;)
{
    nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
    if (nfds == -1)
    {
        perror("epoll_pwait");
        exit(EXIT_FAILURE);
    }

    for (n = 0; n < nfds; ++n)
    {
        if (events[n].data.fd == listen_sock)
        {
            conn_sock = accept(listen_sock,
                               (struct sockaddr *) &local, &addrlen);
            if (conn_sock == -1)
            {
                perror("accept");
                exit(EXIT_FAILURE);
            }
            setnonblocking(conn_sock);
            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = conn_sock;
            if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock, &ev) == -1)
            {
                perror("epoll_ctl: conn_sock");
                exit(EXIT_FAILURE);
            } else
            {
                do_use_fd(events[n].data.fd);
            }
        }
    }
}
}

```

7. 回答

- Q1 如果多次将同一个描述符注册到同一个 epoll 实例会怎样?

A1 会返回错误。但你可以复制一个描述符，然后将复制的描述符注册到同一个 epoll 实例。每个描述符都可以只关心个别事件，这样可以为一个 I/O 的各种不同事件分别注册不同的处理函数。

- Q2 可以两个 epoll 实例同时监听一个描述符吗?

A2 可以。并且会向两个 epoll 都发送事件通知，但编程要小心。

- Q3 如果将 epoll 描述符注册到自身会怎样?

A3 不允许。但可以将一个 epoll 描述符注册到另一个 epoll。

- Q4 关闭一个文件描述符，会自动将它从 epoll 的描述符集合撤除吗?

A4 Yes, but be aware of the following point. A file descriptor is a reference to an open file description (see open(2)). Whenever a descriptor is duplicated via dup(2), dup2(2), fcntl(2) F_DUPFD, or fork(2), a new file

descriptor referring to the same open file description is created. An open file description continues to exist until all file descriptors referring to it have been closed. A file descriptor is removed from an epoll set only after all the file descriptors referring to the underlying open file description have been closed (or before if the descriptor is explicitly removed using epoll_ctl(2) EPOLL_CTL_DEL). This means that even after a file descriptor that is part of an epoll set has been closed, events may be reported for that file descriptor if other file descriptors referring to the same underlying file description remain open.

- Q5 If more than one event occurs between epoll_wait(2) calls, are they combined or reported separately?

A5 They will be combined.

- Q9 Do I need to continuously read/write a file descriptor until EAGAIN when using the EPOLLET flag (edge-triggered behavior) ?

A9 Receiving an event from epoll_wait(2) should suggest to you that such file descriptor is ready for the requested I/O operation. You must consider it ready until the next (nonblocking) read/write yields EAGAIN. When and how you will use the file descriptor is entirely up to you.

For packet/token-oriented files (e.g., datagram socket, terminal in canonical mode), the only way to detect the end of the read/write I/O space is to continue to read/write until EAGAIN.

For stream-oriented files (e.g., pipe, FIFO, stream socket), the condition that the read/write I/O space is exhausted can also be detected by checking the amount of data read from / written to the target file descriptor. For example, if you call read(2) by asking to read a certain amount of data and read(2) returns a lower number of bytes, you can be sure of having exhausted the read I/O space for the file descriptor. The same is true when writing using write(2). (Avoid this latter technique if you cannot guarantee that the monitored file descriptor always refers to a stream-oriented file.)

2.7.5 select & epoll 比较

select的原理:

1. 使用 copy_from_user 从用户空间拷贝 fd_set 到内核空间
2. 注册回调函数 __pollwait
3. 遍历所有 fd, 调用其对应的 poll 方法 (对于socket, 这个 poll 方法是 sock_poll, sock_poll 根据情况会调用到 tcp_poll, udp_poll 或者 datagram_poll)
4. 以 tcp_poll 为例, 其核心实现就是 __pollwait, 也就是上面注册的回调函数。
5. __pollwait 的主要工作就是把 current (当前进程) 挂到设备的等待队列中, 不同的设备有不同的等待队列, 对于 tcp_poll 来说, 其等待队列是 sk->sk_sleep (注意把进程挂到等待队列中并不代表进程已经睡眠了)。在设备收到一条消息 (网络设备) 或填写完文件数据 (磁盘设备) 后, 会唤醒设备等待队列上睡眠的进程, 这时 current 便被唤醒了。
6. poll 方法返回时会返回一个描述读写操作是否就绪的 mask 掩码, 根据这个 mask 掩码给 fd_set 赋值。
7. 如果遍历完所有的 fd, 还没有返回一个可读写的 mask 掩码, 则会调用 schedule_timeout 使调用 select 的进程 (也就是 current) 进入睡眠。当设备驱动发生自身资源可读写后, 会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间 (schedule_timeout 指定), 还是没人唤醒, 则调用 select 的进程会重新被唤醒获得 CPU, 进而重新遍历 fd, 判断有没有就绪的 fd。
8. 把fd_set从内核空间拷贝到用户空间。

select的几大缺点:

1. 每次调用 select, 都需要把 fd 集合从用户态拷贝到内核态, 这个开销在 fd 很多时会很大
2. 同时每次调用 select 都需要在内核遍历传递进来的所有 fd, 这个开销在 fd 很多时也很大
3. select 支持的文件描述符数量太小了, 默认是 1024

epoll的改进:

1. 每次调用 epoll_ctl 注册时, 即将描述符拷进内核, 而不是在 epoll_wait 时重复拷贝。一个描述符只会拷贝一次。
2. epoll 不像 select 或 poll 一样每次都把当前进程轮流加入 fd 对应的设备等待队列中, 而只在 epoll_ctl 时挂一遍 (这一遍必不可少) 并为每个 fd 指定一个回调函数, 当设备就绪, 唤醒等待队列上的等待者时, 就会

调用这个回调函数，而这个回调函数会把就绪的 fd 加入一个就绪链表）。epoll_wait 的工作实际上就是轮询就绪链表中有没有 fd。

3. epoll 所支持的 fd 上限是最大可以打开文件的数目，这个数字一般远大于 2048

2.8 I/O 模型

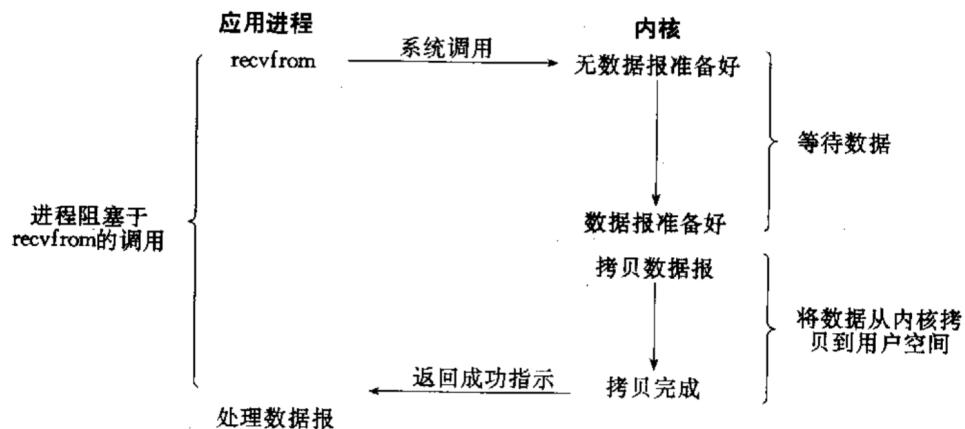


图 6.1 阻塞 I/O 模型

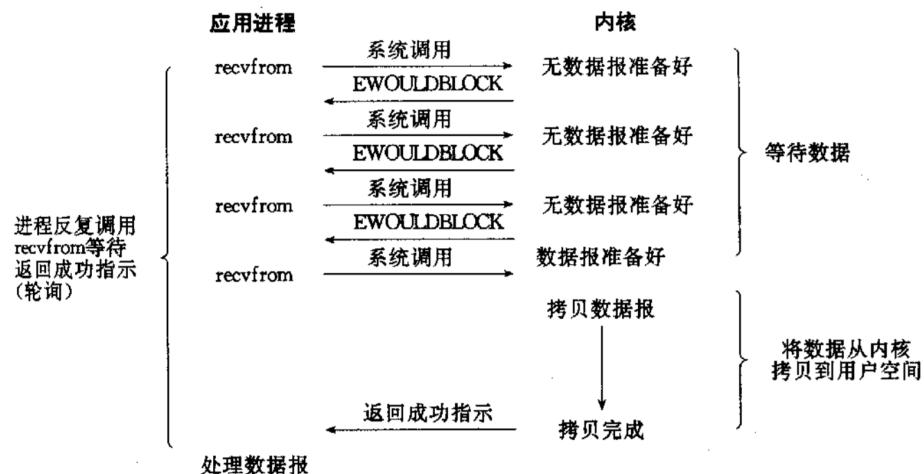


图 6.2 非阻塞 I/O 模型

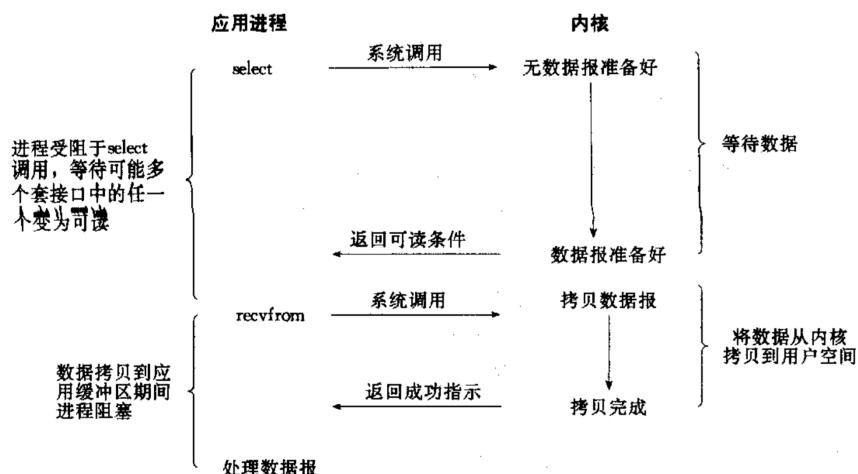


图 6.3 I/O 复用模型

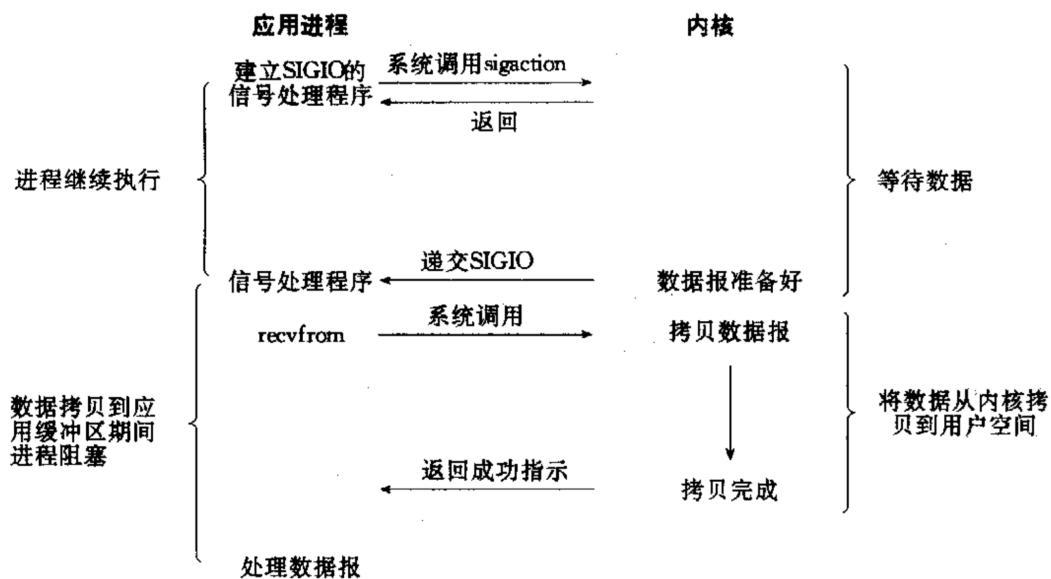


图 6.4 信号驱动 I/O 模型

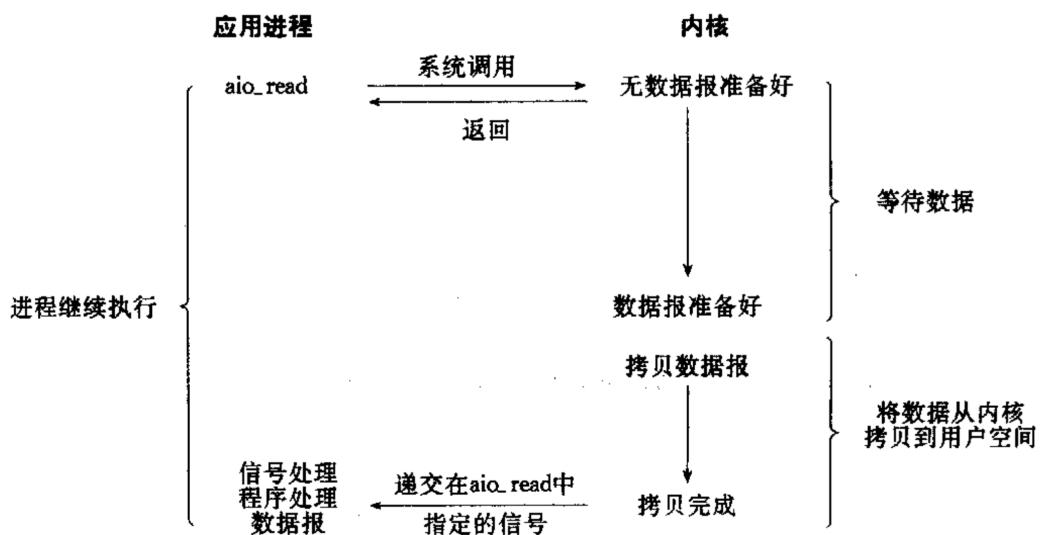


图 6.5 异步 I/O 模型

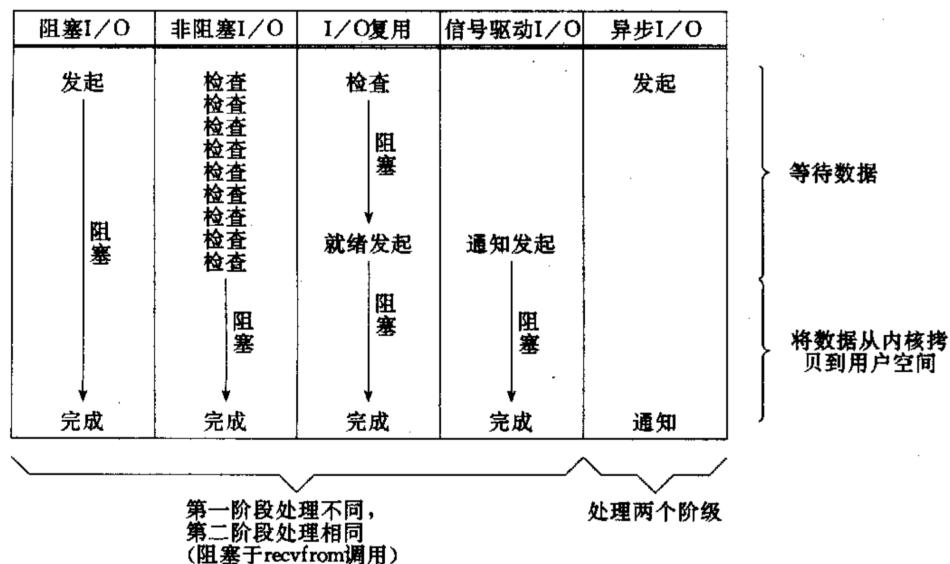


图 6.6 五个 I/O 模型的比较

3. 进程

3.1 进程环境

3.1.1 进程标识符

每个进程都有一个非负整型表示的唯一进程 ID。因为进程 ID 标识符总是唯一的，常将其用作其他标识符的一部分以保证其唯一性。

虽然是唯一的，但进程 ID 可以重用。当一个进程终止后，其进程 ID 就会被回收以供再次使用。大多数 UNIX 系统实现延迟重用算法，使得赋予新建进程的 ID 不同于最近终止进程所用的 ID，这样防止将新进程误认为是某个已终止的老进程。

ID 为 0 的进程通常是调度进程，常常被称为交换进程。该进程是内核的一部分，它并不执行磁盘上的程序，因此也被称为系统进程。

ID 为 1 的进程通常是 init 进程，在自举结束时由内核调用。init 进程绝不会终止，它是一个普通的用户进程，但以超级用户权限运行。init 进程是第一个真正意义上的进程，因此所有其他进程都是由 init 进程派生而来，也会成为所有孤儿进程的父进程。

```
#include <unistd.h>
pid_t getpid(void);           //返回调用进程的进程 ID
pid_t getppid(void);          //返回调用进程父进程 ID
```

linux 下可以通过 cat /proc/sys/kernel/pid_max 查看系统支持多少进程：

```
cat /proc/sys/kernel/pid_max
//32768
```

3.1.2 命令行参数

main(int argc, char *argv[]) 函数中的命令行参数由 exec 函数传入。argc 指明参数的个数，参数由空格分隔； argv 是字符串数组，存放各个参数。argc 最小值为 1， argv[0] 是当前程序本身的文件名。

3.1.3 环境变量

每个进程都有一张环境表，存放了该进程的环境变量。每一个表项，都是一个形如“name=value”的字符串。

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

3.1.4 资源限制

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。这些资源限制包括：

RLIMIT_AS	进程可用存储区的最大总长度
RLIMIT_CORE	core 文件的最大字节数，若值为 0 则阻止创建 core 文件
RLIMIT_CPU	CPU 时间的最大量值，当超过此软限制时，向该进程发送 SIGXCPU 信号
RLIMIT_DATA	数据段的最大字节长度，包括数据段和堆的总和
RLIMIT_FSIZE	可创建文件的最大字节长度。当超过此软限制时，向该进程发送 SIGXCPU 信号
RLIMIT_LOCKS	一个进程可持有的文件锁最大数
RLIMIT_MEMLOCK	一个进程使用 mlock 能够锁定在存储器中的最大字节长度
RLIMIT_NOFILE	每个进程能打开的最大文件数
RLIMIT_NPROC	每个实际用户 ID 可拥有的最大子进程数
RLIMIT_RSS	最大驻内存集的字节长度。若物理存储器不够，则内核从进程取回超出 RSS 部分

RLIMIT_SBSIZE	用户在任一给定时刻可以占用的套接字缓冲区的最大长度
RLIMIT_STACK	栈的最大字节长度
RLIMIT_VMEM	RLIMIT_AS 同义词

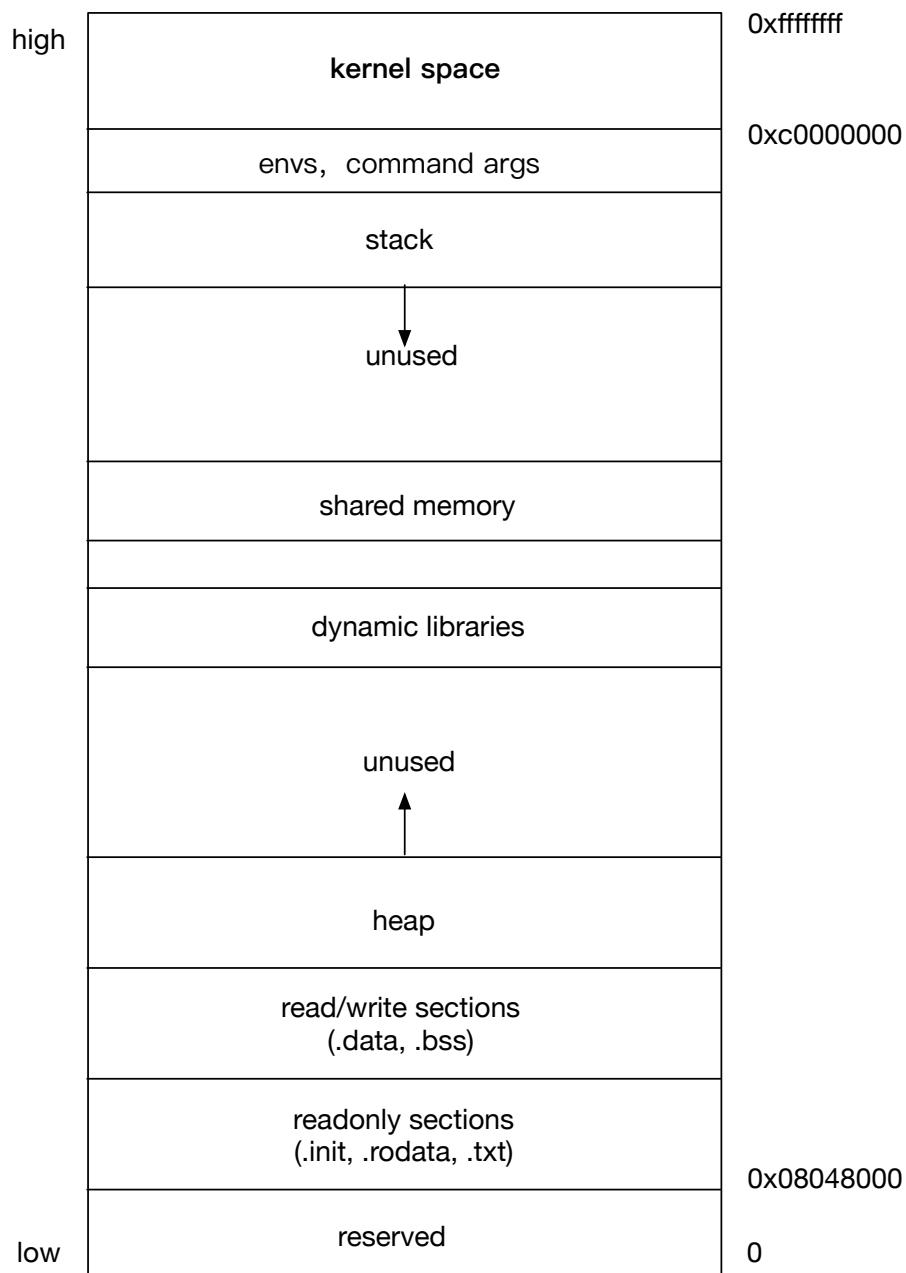
资源限制可以由下列函数查看或改变：

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

更改资源限制时，必须遵守下列三条规则：

1. 任何一个进程都可以将一个软限制值更改为不大于其硬限制值
2. 任何一个进程都可以降低其硬限制值，但它必须不小于其软限制值
3. 只有超级用户可以提高硬限制值

3.1.5 进程空间



- 正文段：这是由 CPU 执行的机器指令部分。正文段是只读的，故通常是可共享的，所以即使频繁执行程序，在存储器中也只需一个副本。
- 初始化数据段：它包含了程序中明确赋初值的全局变量或静态变量。

- 非初始化数据段：通常称此段为 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。
- 栈：
- 堆：通常在堆中进行动态存储分配。

可执行文件中还有若干其他类型的段，例如包含符号表的段、包含调试信息的段以及包含动态共享库链接表的段等等。这些部分并不装载到进程执行的程序映像中。

3.2 创建进程

不同的操作系统所提供的进程创建原语的名称和格式不尽相同，但执行创建进程原语后，操作系统所做的工作却大致相同，都包括以下几点：

1. 给新创建的进程分配一个内部标识（pcb），在内核中建立进程结构，pcb 包含：

进程描述信息：

- 进程标识符用于唯一的标识一个进程（pid, ppid）。

进程控制信息：

- 进程当前状态
- 进程优先级
- 程序开始地址
- 各种计时信息
- 通信信息

资源信息：

- 占用内存大小及管理用数据结构指针
- 交换区相关信息
- I/O 设备号、缓冲、设备相关的数结构
- 文件系统相关指针

现场保护信息（cpu 进行进程切换时）：

- 寄存器
- PC
- 程序状态字 PSW
- 栈指针

2. 复制父进程的环境，可以是写时复制
3. 为进程分配资源，包括进程映像所需要的所有元素（程序、数据、用户栈等）
4. 复制父进程地址空间的内容到该进程地址空间中
5. 置该进程的状态为就绪，插入就绪队列

```
#include <unistd.h>
pid_t fork(void);
    //子进程中返回 0，父进程返回子进程 ID，出错返回 -1
```

6. 调用 fork 函数分别在子进程和父进程返回，共返回两次
7. 父进程和子进程都从 fork 返回处开始往下执行
8. 子进程共享父进程的文件表项
9. 子进程还继承父进程其他许多属性：
 - 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

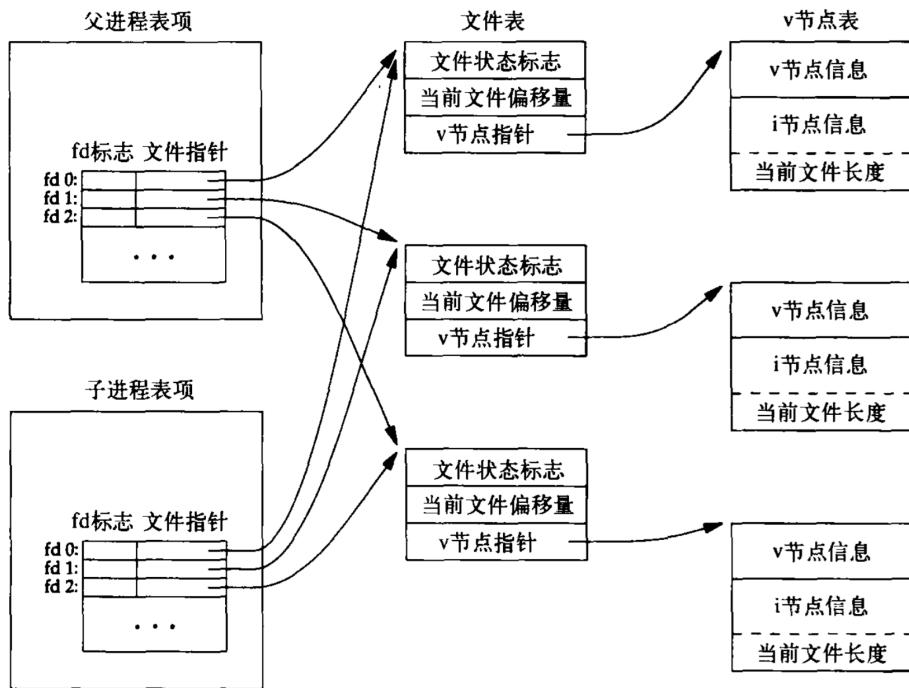


图8-1 调用fork之后父、子进程之间对打开文件的共享

- 根目录
 - 文件模式创建屏蔽字
 - 信号屏蔽和安排
 - 互斥量、条件变量、读写锁
 - 针对任一打开文件描述符的在执行时关闭标志
 - 环境
 - 连接的共享存储段
 - 存储映射
 - 资源限制
10. 父子进程之间的区别是：

- 进程 ID 不同，父进程 ID 不同
- 子进程的 tms_utime、tms_stime、tms_cutime 以及 tms ustime 均被设置为 0。
- 父进程设置的文件锁不会被子进程继承
- 子进程的未处理闹钟被清除
- 子进程的未处理信号集设置为空集

fork 常见用法：

- 一个父进程希望复制自己，使父子进程执行不同的代码段。这在网络服务进程中是常见的——父进程等待客户端的服务请求。当这种请求到达时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求到达。
- 一个进程要执行一个不同的程序。这对 shell 是常见的。这种情况下，子进程从 fork 返回后立即调用 exec。

fork 代码例程：

```
pid_t pid;

if ((pid = fork()) > 0)
{
    //parent do something
} else if (0 == pid)
{
    //child do something
}
```

```

} else
{
    //error
}

```

因此 fork 之后处理文件描述符有两种常见的情况：

- 父进程等待子进程完成
- 父子进程各自执行不同的程序段。在这种情况下，fork 之后，父子进程各自关闭它们不需要的文件描述符，这样就不会干扰对方使用文件描述符。

3.3 加载程序

进程创建后，子进程往往要调用一种 exec 函数以执行另一个程序。当调用一种 exec 函数时，该进程执行的程序完全替换为新程序，而新程序则从 main 函数开始运行。因为 exec 并不创建新的进程，所以前后进程 ID 并未改变，只是用新的程序替换了当前进程的正文、数据、堆、和栈段。

3.3.1 函数接口

```

#include <unistd.h>
int execl (const char *pathname, const char *arg0, ...);
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...);
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ...);
int execvp (const char *filename, char *const argv[]);
//All six return: -1 on error, no return on success

```

这些函数的关系如下：

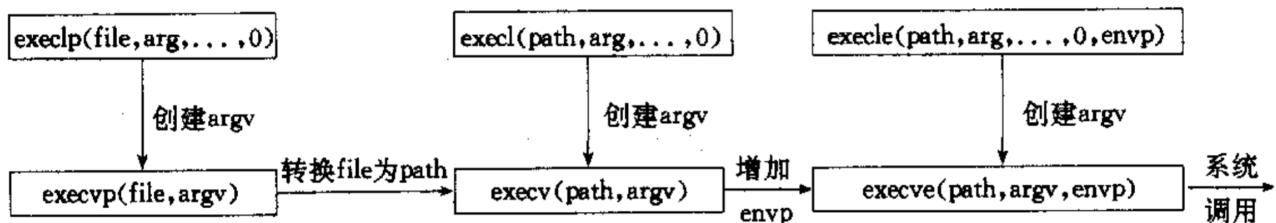


图 4.12 六个 exec 函数的关系

执行 exec 函数后，执行新程序的进程还保持原进程的下列特征：

- 进程 ID，父进程 ID
- 实际用户 ID、实际组 ID
- 附加组 ID
- 进程组 ID
- 会话 ID
- 控制终端
- 当前工作目录
- 根目录
- 文件模式创建屏蔽字
- 文件锁
- 信号屏蔽
- 未处理信号
- 资源限制
- tms_utime、tms_stime、tms_cutime 以及 tms ustime

对打开文件的处理与每个描述符的 close-on-exec 标志有关。若标志开，则执行 exec 时关闭描述符，否则保持原样。原来进程要捕捉的信号，exec 函数后都改为它们的默认动作。

3.3.2 exec 函数执行过程

1. 加载可执行文件

- 如果参数 pathname 包含 /，则将其视为路径，按照这个路径找到可执行文件
- 否则按照 PATH 环境变量指定的目录依次搜寻
- 最后搜寻进程的当前目录

```
/Library/Java/JavaVirtualMachines/jdk/Contents/Home/bin:/Library/Java/JavaVirtualMachines/jdk/Contents/Home/jre/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin:/usr/local/apache-maven/bin:/opt:/opt/apache-karaf-4.0.1/bin
```

PATH 环境变量，路径由冒号分隔。

当前目录最后搜寻，是为了避免用户程序顶替系统程序。

- 读取可执行文件头，建立进程空间与可执行文件的映射关系。同时通过文件头找到启动函数 _start 的地址，执行 _start 函数。

- 如果正常，则 exec 函数不会返回

3.4 程序运行

- 进入程序启动函数 _start
- 初始化和 OS 版本有关的全局变量
- 初始化堆
- 初始化 I/O
- 获取命令行参数和环境变量压栈
- 初始化 C 函数库的一些数据
- 初始化程序全局变量，调用全局对象构造函数
- 调用 main 函数并记录返回值
- 全局变量析构、堆销毁、关闭 I/O 等
- 系统调用 _exit/_Exit 结束进程

3.5 程序退出

有 8 种方式使进程终止，其中五种为正常终止：

- 从 main 函数返回
- 调用 exit
- 调用 _exit 或 _Exit
- 最后一个线程从其启动例程返回
- 最后一个线程调用 pthread_exit

异常终止有三种方式：

- 调用 abort
- 接到一个信号并终止
- 最后一个线程对取消请求做出响应

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
void _exit(int status);
```

exit 函数总是为相应进程冲洗标准 I/O 流，关闭所有描述符，释放它所使用的存储器等。在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止进程的子进程，如果是，则将该子进程的父进程 ID 改为 1，由 init 进程领养。

```
#include <stdlib.h>
int aexit(void (*func)(void));
```

`aexit` 函数用于注册清理函数，注册的函数会在 `exit` 函数执行时被调用。

`exit` 调用这些函数的顺序与它们登记时候的顺序相反。同一函数若登记多次，则也被调用多次。

3.6 进程终止状态

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的描述符，释放它所使用的存储器等。对上述任意一种终止情形，我们都希望终止进程能够通知其父进程它是如何终止的。对于三个终止函数，实现这一点的方法是，将其退出状态作为参数传递给函数，内核将退出状态转换为终止状态。在异常终止的情况下，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。在任意一种情况下，该终止进程的父进程都能用 `wait` 或 `waitpid` 函数取得其终止状态。

进程终止后，内核会为其保存一定量的信息，包括进程 ID、终止状态、以及使用的 CPU 时间总量。如果子进程终止而父进程不对其作善后处理，则子进程成为僵死进程，其在内核占用的空间无法被释放。如果子进程被 `init` 进程领养，因为 `init` 进程总是使用 `wait` 函数获得子进程的终止状态，所有子进程不会变为僵死进程。

```
#include <sys/wait.h>

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
    //success, 进程 ID
    //error, -1
```

子进程终止前，`wait` 使调用者阻塞，知道某个子进程终止。

`waitpid` 可以选择不阻塞，也可以等待特定的进程。

父进程可以捕捉子进程终止的信号 `SIGCLD`，从而以异步方式获取子进程终止状态，避免子进程僵死。

3.7 进程流程

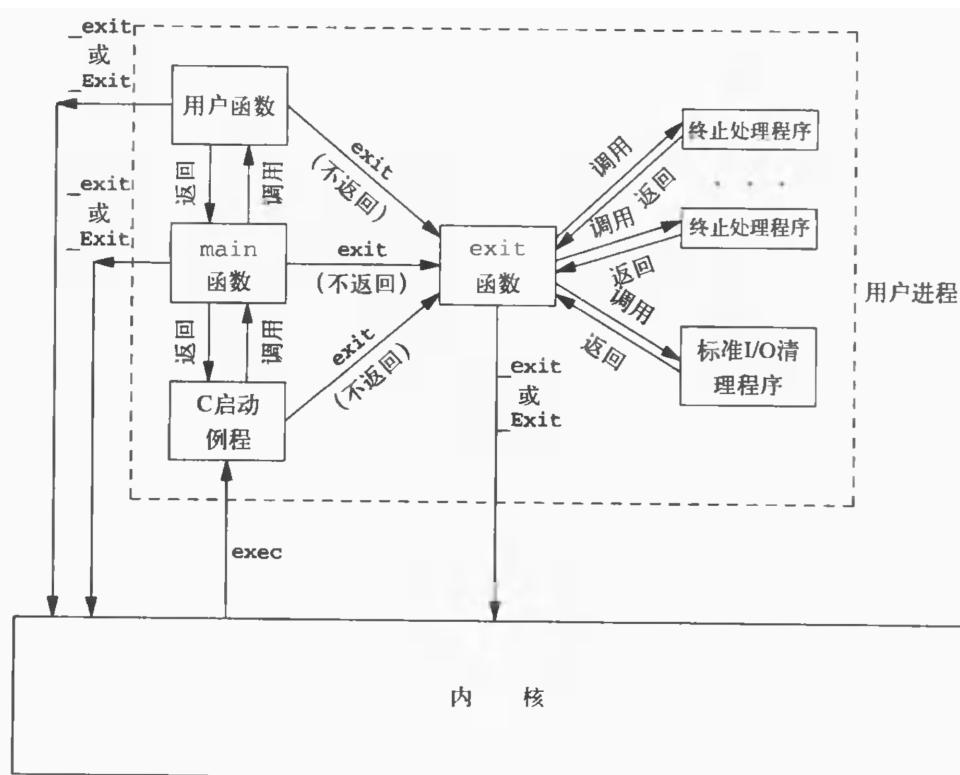


图7-1 一个C程序是如何启动和终止的

4. 进程通信

4.1 无名管道

- 原理

- 优点：管道是 UNIX 系统 IPC 的最古老形式，并且所有 UNIX 系统都提供此种通信机制，仍然是最常用的 IPC 形式。

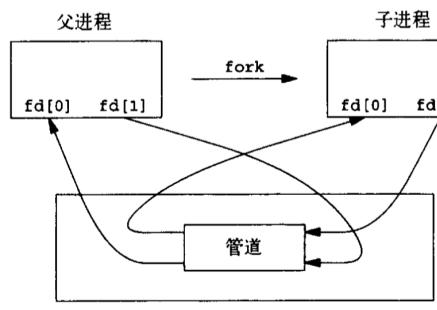


图15-2 调用 fork 之后的半双工管道

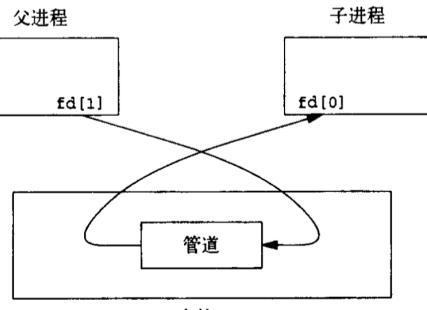


图15-3 从父进程到子进程的管道

- 缺点：

它们是半双工的，即数据只能在一个方向上流动。现在某些系统提供全双工管道，但为了可移植性，不能假定这一点。

它们只能在具有公共祖先的进程之间使用。通常，一个管道由一个进程创建，然后该进程 fork 子进程，父子进程之间就可以使用该管道。

4.2 命名管道 (FIFO)

- 原理：创建一个有名管道，管道名即为文件路径名，其放在文件系统中，本质上是一个文件，但是管道的内容还是放在内存中，所有的进程都能通过这个名字访问这个管道。
- 优点：可以在无血缘关系的进程间通信，克服无名管道的这一缺点
- 缺点：有名管道的读写，都要考虑管道是否有数据以及是否有其他进程在访问。

4.3 XSI IPC

消息队列、信号量以及共享内存，它们都叫 XSI IPC。

每个内核中的 IPC 结构（消息队列、信号量或共享存储段）在内核都用一个非负整数标识符加以引用，在用户空间需要用键与之关联。

- 缺点

- IPC 结构是在系统范围内起作用的，没有访问计数，容易造成资源无法回收。
- IPC 结构在文件系统中没有名字，我们不能用文件访问函数来修改它们的属性，而不得不额外添加系统调用。
- IPC 不是文件描述符，不能使用 I/O 多路复用。

4.4 消息队列

- 原理

消息队列是消息的链表，存放在内核中并由消息队列标识符标识。进程可通过唯一标识符对队列进行读写。通过 msgsnd() / msgrecv() 来操作队列。队列在内核中的组织形式是链表，但是有一棵键树来快速查找到表的 ID，内核中还维护了一张队列表，队列表中的每一个表项都记录了一个队列的信息，如队列的头指针，尾指针等信息。

- 优点：1 是读写操作可以使用非阻塞的形式，同步都交于内核处理；2 对于队列中的数据可以选择性读取，只读取 msg_type 数据，而不像管道那样什么数据都读取。
- 缺点：使用消息队列通信并不比管道快，对删除队列的处理不够完善，应避免使用

4.5 共享内存

- 原理

两个进程空间的存储段建立映射关系，并映射至同一块物理内存。linux 系统将共享段置于堆和栈之间。

- 优点：最快的 IPC，不需要复制；函数的接口也简单；数；不像匿名管道那样要求通信的进程有一定的父子关系。
- 缺点：共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作(如使用信号量来同步)。

4.6 信号量

- 原理：信号量是个计数器，与锁类似，用于多个进程对共享数据对象的同步访问。为了获得共享资源，需要执行以下操作：

1. 测试控制该资源的信号量
2. 若此信号量的值为正，则进程可以使用该资源，进程将信号量值减一
3. 若此信号量的值为0，则进程进入休眠状态，直至信号量值大于0。进程被唤醒后，执行第一步。
4. 进程不再使用共享资源时，将信号量加一。

尽管信号量比记录锁要快，但记录锁使用要简单，并且进程终止时系统会处理任何遗留下来的锁。

4.7 UNIX 域套接字

UNIX 域套接字用于同一台机器上运行的进程之间的通信。虽然因特网域套接字可用于同一目的，但 UNIX 域套接字的效率更高。UNIX 域套接字仅仅复制数据，并不执行协议处理，不需要添加或删除网络报文头，无需计算校验和，不产生序列号，不确认不重传。

UNIX 域套接字提供流和数据报两种接口，UNIX 域套接字是可靠的，既不会丢失消息也不会传递出错，是套接字和管道之间的混合物。

UNIX 域套接字除了在绑定地址时不是绑定 IP+port，而是绑定一个指定的路径名外，其他方面都与因特网域套接字使用相同。

5. 线程

5.1 线程环境

5.1.1 多线程的好处

- 通过为每种事件类型的处理分配单独的线程，能够简化处理异步事件的代码。每个线程在进行事件处理时可以采用同步编程模式，同步编程模式要比异步编程模式简单的多。
- 多个进程必须使用操作系统提供的复杂机制才能实现内存和文件描述符的共享，而多个线程自动地可以访问相同的存储地址空间和文件描述符。
- 有些问题可以通过将其分解从而改善整个程序的吞吐量。在只有一个线程的情况下，单个进程需要完成多个任务时，实际上需要把这些任务串行化；有了多个控制线程，相互独立的任务处理就可以交叉进行。
- 交互的程序同样可以通过使用多线程实现响应时间的改善，多线程可以把程序中处理用户输出的部分与其他部分分开。

5.1.2 线程的公共数据

进程的所有信息对该进程的所有线程都是共享的：

- 可执行的程序文本
- 程序的全局内存、堆内存、栈
- 文件描述符
- 对非屏蔽信号的处理方式

5.1.3 线程的私有数据

1. 线程 ID

每个线程都有一个线程 ID，用非负整数表示。进程 ID 在系统中是唯一的，线程 ID 在进程中是唯一的。

```
#include <pthread.h>

pthread_t pthread_self(void);
    //返回调用线程的线程 ID
int pthread_equal(pthread_t tid1, pthread_t tid2);
    //比较两个线程 ID
```

2. 一组寄存器值
3. 栈
4. 调度优先级和策略

5. 信号屏蔽字：每个线程都有自己的信号屏蔽字，线程开始时继承至调用线程

6. errno 变量

7. 其他私有数据

5.1.4 线程的属性

1. 分离状态 detachstate

线程处于分离状态时，当线程结束后，线程的底层存储资源就可以在线程结束时立即收回。

调用 pthread_join、pthread_detach 函数，都会使线程处于分离状态。如果线程已经处于分离状态，那么这两个函数调用会出错。

2. 栈大小 stacksize

3. 栈的最低地址 stackaddr

4. 栈末尾警戒缓冲区大小 guardsize

5. 并发度

6. 优先级

7. 可取消状态：线程能否被其他线程调用 pthread_cancel 取消，默认是 PTHREAD_CANCEL_ENABLE

8. 取消类型：线程默认的取消类型是延迟取消，即调用 pthread_cancel 后，线程到达取消点之前，并不会真正取消。取消类型还可以设置为异步取消，使用异步取消时，线程可以在任意时间取消，而不是非得遇到取消点才能被取消。

5.2 线程开始

```
#include <pthread.h>
int
pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
              void *(*start_routine)(void *), void *restrict arg);
```

新创建的线程可以访问进程的地址空间，并且继承调用线程的浮点环境和信号屏蔽字，但该线程的未决信号集被清除。

//注意无类型指针 arg 指向的内存调用后必须一直存在，可以在全局内存中或者由 malloc 申请在堆中，否则子线程可能会出现非法地址访问

5.3 线程终止

5.3.1 线程终止方式

如果进程中的任一线程调用了 exit, _Exit 或者 _exit，那么整个进程就会终止。与此类似，如果信号的默认动作是终止进程，那么把信号发送到线程会终止整个进程。

单个线程通过下列三种方式返回，并且不终止进程：

- 线程从其启动例程返回，返回值是线程的退出码
- 线程调用 pthread_exit
- 线程可以被同一进程中的其他线程取消

5.3.2 线程终止流程

· 调用注册的清理函数（除了正常返回时）

· 线程析构函数

· 线程终止

5.3.3 线程终止接口

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
int pthread_cancel(pthread_t tid);
    //pthread_cancel 只是提出请求，并不等待线程终止
    //成功返回 0，出错返回错误编号，比如没有相应的线程
```

线程可以安排它退出时需要调用的清理函数，则与进程调用 atexit 函数类似。线程可以建立多个清理处理函数，记录在栈中，执行顺序与注册顺序相反。

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop (int execute);
```

线程在执行以下动作时调用注册的清理函数：

- 调用 pthread_exit 时
- 相应其他线程的 pthread_cancel 请求时
- 用非零 execute 参数调用 pthread_cleanup_pop 弹出注册的清理函数时

//注意线程从启动函数返回是不会调用清理函数的。

//同 pthread_create 一样， pthread_exit 的 rval_ptr 参数指向的内存必须在线程退出后仍然存在

```
#include <pthread.h>

void pthread_join(pthread_t thread, void **rval_ptr);
    //获取指定线程的退出状态
    //调用线程会一直阻塞，直到指定线程退出
    //如果指定线程是从启动例程返回，则 rval_ptr 将包含返回码
    //如果线程被取消，则 rval_ptr 内容为 PTHREAD_CANCELED
    //如果线程是 pthread_exit 退出，则 rval_ptr 指向其传入参数 rval_ptr
    //如果不关心线程的终止状态，可以把 rval_ptr 设置为 NULL
    //如果线程已经处于分离状态，失败返回 EINVAL
```

5.4 线程同步

5.4.1 线程安全函数

5.4.2 互斥量

互斥量本质上是一把锁，访问资源前对互斥量加锁，访问完毕解锁。对互斥量加锁后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放互斥锁。如果释放互斥锁时有多个线程阻塞，那么只有第一个变为运行状态的线程才可以对互斥量加锁，其他线程继续阻塞。这种方式下，每次只有一个线程可以向前执行。

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

//success 0, failed error
```

互斥量变量用 pthread_mutex_t 类型表示，背后是一个结构体。使用互斥量前，需要对它初始化。

对于静态的互斥量，可以初始化为常量 PTHREAD_MUTEX_INITIALIZER，

对于动态分配的互斥量，则使用 pthread_mutex_init 函数初始化，也使用 pthread_mutex_destroy 释放，注意 pthread_mutex_destroy 函数一定要在释放内存之前调用，否则互斥量的所占用的资源无法收回。

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);

//success 0, failed error
```

函数 pthread_mutex_trylock 尝试加锁，如果锁被占用，不会阻塞而是立即返回。

5.4.3 读写锁

读写锁（共享-独占锁）与互斥量类似，但允许更高的并行性，一般用于读的次数远大于写的次数的情况。互斥量只有两个状态，而读写锁有三个状态。

- 未加锁状态：线程可以加读锁或写锁而不被阻塞
- 读锁状态：其他线程继续加读锁不会阻塞，但加写锁会阻塞。但如果线程试图加写锁，则后续的读锁请求都会被阻塞，以免写锁请求一直得不到满足

- 写锁状态：其他线程加任何锁都会被阻塞

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const
                      pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

5.4.4 条件变量

条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定的条件发生。

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

5.4.5 避免死锁

1. 临界区划分要小，执行时间要短，访问资源粒度要细
2. 始终以一致的顺序对共享的资源进行加锁访问
3. 使用非阻塞的 trylock 进行访问

5.5 线程和信号

1. signal actions 是 process-wide。如果一个没有处理的信号的默认动作是停止 SIGSTOP 或终止 SIGKILL (该动作是让整个进程停止或终止，而不是只针对某个线程)，那么不管这个信号是发送给哪个线程，整个进程都会停止或终止。
2. signal dispositions 信号部署是 process-wide。一个进程中的所有线程对某个信号都共享相同的信号处理函数。如果线程 A 使用 sigaction() 对某个信号，比如 SIGINT，建立了一个信号处理函数。那么当 SIGINT 发送到线程 B 时，信号处理函数也会被调用。
3. 下面几种情况，把信号发送到某个指定的线程。
 - 某个特定硬件指令执行后（在该线程内执行的），产生的信号，将会发送到该线程内。比如 SIGBUS, SIGFPE, SIGILL, SIGSEGV。
 - 当线程尝试向一个 broken pipe 写数据时，会产生一个 SIGPIPE。
 - 使用 pthread_kill() 或者 pthread_sigqueue()。这些函数允许一个线程发送信号到另一个线程（同一进程中）。
 - 其他情况都是把信号发送到整个进程（比如，kill() 和 sigqueue()）。
4. 当一个信号被发送到一个多线程的进程中（注意是发送到进程）。内核会选择该进程中的任意线程来处理该信号。这种做法是为了保持进程中信号的语意，保证不会在多线程进程中一个信号多次被执行。
5. 信号屏蔽字 (signal mask) 是线程私用的。在多线程的进程中，不存在 process-wide 的信号掩码。线程可以使用 pthread_sigmask() 来独立的屏蔽某些信号。通过这种方法，程序员可以控制那些线程响应那些信号。当线程被创建时，它将继承创建它的线程的信号掩码

6. 内核为每个线程和进程分别维护了一个未决信号的表。当使用 `sigpending()` 时，该函数返回的是整个进程未决信号表和调用该函数的线程的未决信号表的并集。当新线程被创建时，线程的 pending signals 被设置为空。当线程 A 阻塞某个信号 S 后，发送到 A 中的信号 S 将会被挂起，直到线程取消了对信号 S 的阻塞。
7. 如果一个信号处理函数打断了 `pthread_mutex_lock()`，该函数会自动的重新执行。如果信号处理函数打断了 `pthread_cond_wait()`（参见POSIX线程-条件变量），该函数要么自动重新自行（linux是这样实现的），或者返回0（这时应用要检查返回值，判断是否为假唤醒）。
8. 闹钟定时器是进程资源，并且所有的线程共享相同的 `alarm`。所以进程中的多个线程不可能互不干扰。

在编程中，为了防止信号中断线程，可以把信号加到每个线程的信号屏蔽字中，然后安排专门的线程作信号处理。这些专用线程可以进行函数调用，不需要担心在信号处理程序调用中哪些函数是安全的。

5.6 线程和fork

父进程 fork 子进程后，子进程也继承了父进程所有锁的状态。如果子进程调用 `exec` 函数执行新的程序，老的地址空间被丢弃，锁的状态无关紧要；否则子进程需要进行处理，请参看 Unix 环境编程的 12.9 节。

6. 信号

6.1 信号

每个信号都有一个名字，这些名字都以 SIG 开头，一般有 31 种信号。每个信号都是一个正整数，不存在为 0 的信号，信号的定义在 `<signal.h>` 头文件中可见，也可通过 `man signal` 查看手册。

信号是异步事件的经典实例。产生信号的事件对进程而言是随机出现的，进程不能简单地测试一个变量来判断是否产生了一个信号，而是必须告诉内核“在此信号出现时，请执行下列操作”。

6.2 产生信号的条件

很多条件可以产生信号：

- 当用户按某些终端键时，引发终端产生信号。例如按下 `ctrl+c` 通常产生终端信号 (SIGINT)。这是停止一个已经失去控制的程序的方法。
- 硬件异常产生的信号：除数为零、无效的内存引用等。这些条件通常由硬件检测到，并将其通知内核，然后内核为相应进程产生适当的信号。
- 进程调用 `kill(2)` 函数可将信号发送给另一个进程或进程组。接收信号的进程和发送信号的进程的所有者必须相同，或者发送信号的进程所有者为超级用户。
- 用户可以用 `kill(1)` 命令将信号发送给其他进程。
- 当检测到某种软条件发生，并应将其通知有关进程时也产生信号。

6.3 信号的处理

6.3.1 信号的一般处理方式

出现一个信号，一般按如下三种方式之一处理：

- 忽略此信号。大多数信号都可使用这种方式进行，但有两种信号不能被忽略，它们是 SIGKILL 和 SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供了使进程终止或停止的可靠方法。另外，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。
- 捕捉信号。为了做到这一点，要通知内核在某种信号发生时调用一个用户函数，由用户函数进行处理。不能捕捉 SIGKILL 和 SIGSTOP 信号。
- 执行系统默认动作。注意，大多数信号的系统默认动作是终止进程。

6.3.2 启动进程与加载程序

一个进程 fork 一个子进程后，子进程对信号的处理与父进程一样，只不过父进程中尚未处理完毕的信号在子进程中都直接忽略掉。子进程执行 `exec` 加载新的程序后，信号屏蔽字不变，但对于捕捉处理的信号，全部改为系统默认的处理方式，因为在新的程序中，原来的处理信号的函数不再有意义。

6.3.3 中断的系统调用

系统调用可分为两类：低速系统调用和其他系统调用。低速系统调用是可能会使进程永远阻塞的一类系统调用，包括：

- 在读写某些类型的文件（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。
- pause（按照定义，它使调用进程休眠直至捕捉到一个信号）和 wait 函数
- 某些 ioctl 函数
- 某些进程间通信函数

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，则该系统调用就被终端不在继续执行，系统调用返回出错，errno 设置为 EINTR。这样处理的理由是：因为一个信号发生了，进程捕捉到了它，这意味着已经发生了某种事情，所以应当是个唤醒阻塞的系统调用的好机会。有些实现中，被中断的系统调用还会自动重启。

6.3.3 可重入函数

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
cffgetispeed	fsync	pipe	sigaction	tcgetpgrp
cffgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cffsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cffsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_gettime
chown	getpeername	readlink	signal	timer_getoverrun
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

没有列入表中的函数大多是不可重入的，其原因为：

- 已知它们使用静态数据结构
- 它们调用 malloc 或 free
- 它们是标准 I/O 函数。标准 I/O 库的很多实现都使以不可重入的方式使用全局数据结构

在信号处理程序中调用一个不可重入函数，其结果是不可预见的。

一个线程只有一个 errno，即便上面的表格列出了可重入函数，但这些函数仍然可能改变 errno 的值，导致在信号处理阶段覆盖 errno 的值。因此，作为一个通用规则，当在信号处理程序中调用可重入函数时，应当先保存 errno 的值。6. 信号6. 信号

7. 守护进程

守护进程也称精灵进程(daemon)是生存期较长的一种进程，它们常常在系统自举时启动，仅在系统关闭时才终止。因为它们没有控制终端，所以说它们是在后台运行的。

7.1 编程规则

1. 调用 umask 将文件模式创建屏蔽字设置为 0。由继承得来的文件模式创建屏蔽字可能会拒绝某些权限。例如，若守护进程要创建一个组可读、写的文件，而继承的文件模式创建屏蔽字可能屏蔽了这两种权限，于是所要求的组可读、写就不能起作用
2. 调用 fork，然后使父进程退出。这样做实现下面几点：
 - 如果该守护进程是作为一条简单 shell 命令启动的，那么父进程终止使得 shell 认为这条命令已经执行完毕。
 - 子进程继承了父进程的进程组 ID，但具有一个新的进程 ID，这保证子进程不是一个进程组的组长进程。
 - 调用 setsid 以创建一个新会话，使调用进程：a 成为新会话的首进程，b 成为一个新进程组的组长进程，c 没有控制终端。
 - 将当前工作目录改为根目录。因为从父进程继承过来的当前工作目录可能不会长期存在。某些守护进程可能会把当前工作目录更改到某个指定位置。
 - 关闭不再需要的文件描述符。这使守护进程不再持有从其父进程继承来的某些文件描述符。
 - 某些守护进程打开 /dev/null 使其具有文件描述符 0、1、2，这样任何一个试图读标准输入、写标准输出或标准出错的例程库都不会产生任何效果。
 - 写 log 日志

7.2 守护进程惯例

UNIX 系统中，守护进程遵循下列公共惯例：

- 若守护进程使用锁文件，那么该文件通常存放在 /var/run 目录中。锁文件的名字通常是 name.pid。
- 若守护进程支持配置选项，那么配置文件通常存放在 /etc 目录中。配置文件的名字通常是 name.conf。
- 守护进程可以由命令行启动，但通常它们是由系统初始化脚本之一 (/etc/rc* 或 /etc/init.d/*) 启动的。如果守护进程终止，应当自动重新启动它，则我们可在 /etc/inittab 中为该守护进程包括 _respawn 记录项，这样 init 就将重新启动该守护进程。