第 18 章

目录与链接

作为文件相关议题的结局篇，本章将讨论目录和链接。首先是对其系统级实现进行了回顾，之后则描述了用于创建和移除目录、链接的系统调用。接下来所探讨的库函数，可允许程序扫描单个目录下的内容并遍历一个目录树（即，检查目录树中的每个文件）。

每个进程都有两个目录相关属性根目录及当前工作目录，分别用于为解释绝对路径名和相对路径名提供参照点。本章将描述修改二者的系统调用。

最后，本章讨论了相关库函数，可用来解析路径名，并将其分解为目录和文件名两部分。

18.1 目录和（硬）链接

在文件系统中，目录的存储方式类似于普通文件。目录与普通文件的区别有二。

- 在其 i-node 条目中，会将目录标记为一种不同的文件类型（参见 14.4 节）。
- 目录是经特殊组织而成的文件。本质上说就是一个表格，包含文件名和 i-node 编号。

在大多数原生 Linux 文件系统上，文件名长度可达 255 个字符。图 18-1 所示为针对示例文件（/etc/passwd）所维护的文件系统 i-node 表以及相关目录文件的部分内容，展示了目录与 i-node 之间的关系。

虽然一个进程能够打开一个目录，但却不能使用 read()去读取目录的内容。为了检索目录内容，进程必须使用本章后续讨论的系统调用和库函数。（在一些 UNIX 实现中，也可以对目录执行 read()，但这会给应用带来可移植性方面的问题。）进程同样也不能使用 write()来改变一个目录的内容，仅能借助于诸如 open()（创建一个新文件）、link()、mkdir()、symlink()、unlink()及 rmdir()之类的系统调用（4.3 节描述了 open()调用，本章稍后会介绍余下的系统调用）来间接（向内核请求）改变其内容。

i-node 表的编号始于 1，而非 0，因为若目录条目的 i-node 字段值为 0，则表明该条目尚未使用。i-node 1 用来记录文件系统的坏块。文件系统根目录(/)总是存储在 i-node 条目 2 中（如图 18-1 所示），所以内核在解析路径名时就知道该从哪里着手。

更多资源请访问稀酷客(www.ckook.com)

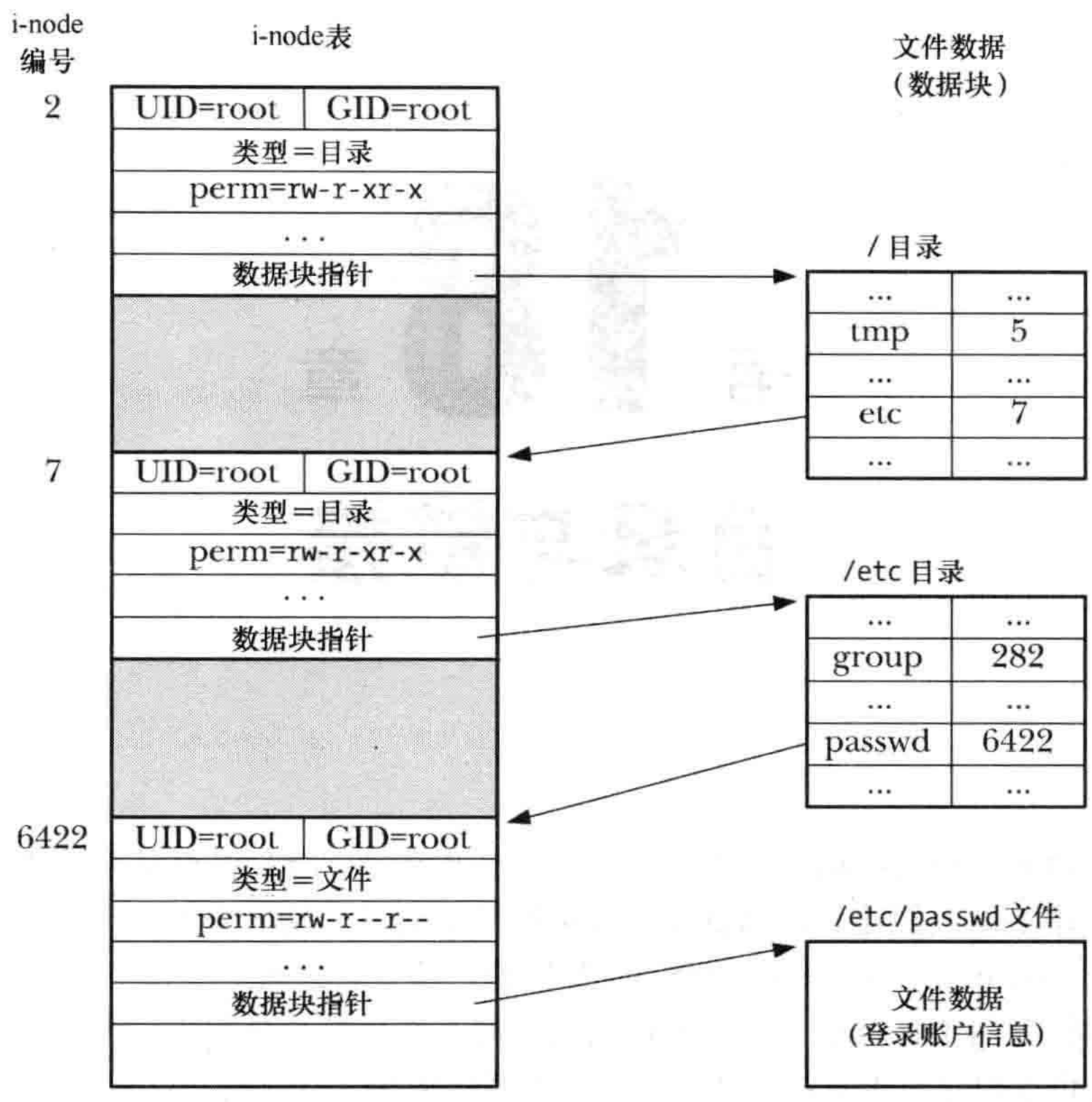

图 18-1：以文件/etc/passwd 为例，展示 i-node 和目录结构之间的关系

回顾文件 i-node（14.4 节）中存储的信息列表，会发现其中并未包含文件名，而仅通过目录列表内的一个映射来定义文件名称。其妙用在于，能够在相同或者不同目录中创建多个名称，每个均指向相同的 i-node 节点。也将这些名称称为链接，有时也称之为硬链接（稍后介绍），以示与符号链接有所区别。

> 所有的原生 Linux 和 UNIX 文件系统均支持硬链接，然而，许多非 UNIX 文件系统（比如，微软的 VFAT）则不支持。（微软的 NTFS 文件系统支持硬链接。）

可在 shell 中利用 ln 命令为一个业已存在的文件创建新的硬链接，正如下面 shell 会话日志所示。

```
$ echo -n 'It is good to collect things,' > abc
$ ls -li abc
 122232 -rw-r--r--   1 mtk      users          29 Jun 15 17:07 abc
$ ln abc xyz
$ echo ' but it is better to go on walks.' >> xyz
$ cat abc
It is good to collect things, but it is better to go on walks.
$ ls -li abc xyz
 122232 -rw-r--r--   2 mtk      users          63 Jun 15 17:07 abc
 122232 -rw-r--r--   2 mtk      users          63 Jun 15 17:07 xyz
```

Cat 命令输出中一目了然的事，经过 ls–li 命令所示 i-node 编码（即第一列）得到了进一步证实。名称 abc 和 xyz 指向相同的 i-node 条目，因此均指向相同文件。ls–li 命令所示内容的第三列为对 i-node 链接的计数。执行 ln abc xyz 命令后，abc 所指向 i-node 的链接计数升至 2，

因为现在指向该文件的有两个名字。（由于指向相同的 i-node，针对文件 xyz 输出的链接计数也是 2。）

若移除其中一个文件名，另一文件名以及文件本身将继续存在。

```
$ rm abc
$ ls -li xyz
 122232 -rw-r--r--   1 mtk      users          63 Jun 15 17:07 xyz
```

仅当 i-node 的链接计数降为 0 时，也就是移除了文件的所有名字时，才会删除（释放）文件的 i-node 记录和数据块。总结如下：rm 命令从目录列表中删除一文件名，将相应 i-node 的链接计数减一，若链接计数因此而降为 0，则还将释放该文件名所指代的 i-node 和数据块。

同一文件的所有名字（链接）地位平等——没有一个名字（比如，第一个）会优于其他名字。正如上例所示，在移除与文件相关的第一个名称后，物理文件继续存在，但只能通过另一文件名来访问其内容。

在线论坛上经常会有这样的问题出现：在程序中如何找到与文件描述符 X 相关联的文件名？简单的回答是不能，至少缺乏明确而又便于移植的手段，因为一个文件描述符指向一个 i-node，而指向这个 i-node 的文件名则可能有多个（或者甚至如 18.3 节所述，一个都没有）。

> 在 Linux 系统上，借助于 readdir()对 Linux 特有/proc/PID/fd 目录内容（内含符号链接指向进程当前打开的每个文件描述符）的扫描，可以获知一个进程当前打开了哪些文件。此外，已经移植到多个 UNIX 系统中的 lsof(1)和 fuser(1)工具也精于此道。

对硬链接的限制有二，均可用符号链接来加以规避。

- 因为目录条目（硬链接）对文件的指代采用了 i-node 编号，而 i-node 编号的唯一性仅在一个文件系统之内才能得到保障，所以硬链接必须与其指代的文件驻留在同一文件系统中。
- 不能为目录创建硬链接，从而避免出现令诸多系统程序陷于混乱的链接环路。

> 早期的 UNIX 实现一度曾允许超级用户为目录创建硬链接。这在当时是必要的，因为这些实现并未提供 mkdir()系统调用。相反，当时会使用 mknode()调用创建一个目录，然后为.和..创建链接（[Vahalia, 1996]）。虽然这一特性已是昨日黄花，但一些现代 UNIX 实现出于向后兼容的目的仍对其加以保留。
>
> 使用绑定挂载（bind mount）可以获得与为目录创建硬链接相似的效果。

18.2 符号（软）链接

符号链接，有时也称为软链接，是一种特殊的文件类型，其数据是另一文件的名称。图 18-2 展示的情况是：两个硬链接——/home/erena/this 和/home/allyn/that——指向同一个文件，而符号链接/home/kiran/other，则指代文件名/home/erena/this。

在 shell 中，符号链接是由 ln–s 命令创建的。ls–F 命令的输出结果中会在符号链接的尾部标记@。

符号链接的内容既可以是绝对路径，也可以是相对路径。解释相对符号链接时以链接本身的位置作为参照点。

符号链接的地位不如硬链接。尤其是，文件的链接计数中并未将符号链接计算在内。（因此，图 18-2 中编号为 61 的 i-node，其链接计数为 2，而不是 3。）因此，如果移除了符号链接所指向

更多资源请访问稀酷客(www.ckook.com)

的文件名，符号链接本身还将继续存在，尽管无法再对其进行解引用（下溯）操作，也将此类链接称之为悬空链接。更有甚者，还可以为并不存在的文件名创建一个符号链接。

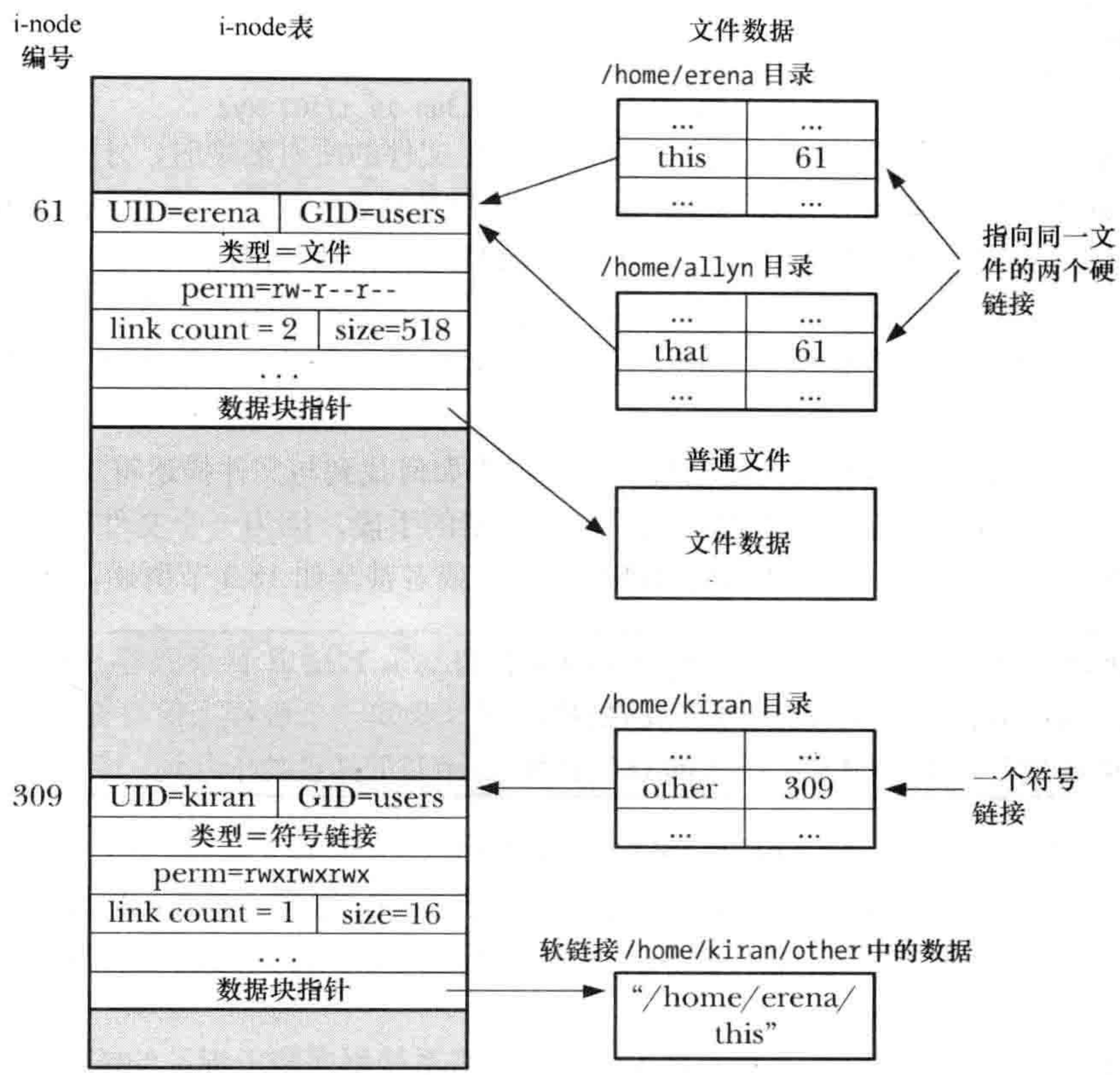

图 18-2：对硬链接和符号链接的展现

引入符号链接的是 4.2BSD。虽然未获 POSIX.1-1990 接纳，但 SUSv1 和 SUSv3 随后还是将其纳入规范。

因为符号链接指代一个文件名，而非 i-node 编号，所以可以用其来链接不同文件系统中的一个文件。对硬链接的那些制约也就不会困扰到符号链接，可以为目录创建符号链接。诸如 find 和 tar 之类的工具命令有能力识别硬链接和符号链接之间的差异，要么会在默认情况下不对符号链接进行解引用，要么会避免因使用符号链接而陷入引用环路。

符号链接之间可能会形成链路（例如，a 是指向 b 的符号链接，而 b 是指向 c 的符号链接）。当在各个文件相关的系统调用中指定了符号链接时，内核会对一系列链接层层解去引用，直抵最终文件。

SUSv3 规定，针对路径名中的每个符号链接部件，系统实现应允许对其实施至少 _POSIX_SYMLOOP_MAX 次解除引用操作。_POSIX_SYMLOOP_MAX 的规定值为 8。然而，在内核 2.6.18 之前，Linux 将解析符号链接链路时的解引用操作次数限制为 5 次。始于版本 2.6.18，Linux 内核实现了 SUSv3 所规定的最小解引用次数：8 次。Linux 还将对一个完整路径名的解引用总数限制为 40 次。施加这些限制，意在应对超长符号链接链路以及符号链接环路，从而使内核代码在解析符号链接时免于引发堆栈溢出。

某些 UNIX 文件系统的优化举措，不但没有在正文中提及，而且也未见诸于图 18-2。如果构成符号链接内容的字符串总长度很小，足以放入 i-node 中通常用于存放数据指针的位置，那么就会将字符串直接存储在那里。这省去对一个磁盘块的分配，也加速了对符号链接信息的访问，因为获取信息时仅涉及到文件的 i-node。例如，ext2、ext3 及 ext4 采用这一技术，将 i-node 中通常用于存放数据块指针的 60 个字节转而用于存放长度合适的符号字符串。实践证明，这一优化措施卓有成效。在笔者检查过的某一系统中，符号链接共计 20 700 个，其中内容长度不超过 60 个字节的占 97%。

系统调用对符号链接的解释

许多系统调用都会对符号链接进行解引用处理（即下溯 follow），从而对链接所指向的文件展开操作。还有一些系统调用对符号链接则不作处理，直接操作于链接文件本身。书中会在论及每个系统调用的同时，描述其针对符号链接的行为。表 18-1 对此作了总结。

表 18-1：各个函数对符号链接的解释

函　　数	是否对链接解引用	备　　注
access()	●	
acct()	●	
bind()	●	UNIX 域套接字带有路径名
chdir()	●	
chmod()	●	
chown()	●	
chroot()	●	
creat()	●	
exec()	●	
getxattr()	●	
lchown()		
lgetxattr()		
link()		参见 18.3 节
listxattr()	●	
llistxattr()		
lremovexattr()		
lsetxattr()		
lstat()		
lutimes()		
open()	●	除非指定了 O_NOFOLLOW 或者 O_EXCL | O_CREAT
opendir()	●	
pathconf()	●	

更多资源请访问稀酷客(www.ckook.com)

续表

函　　数	是否对链接解引用	备　　注
pivot_root()	●	
quotactl()	●	
readlink()		
removexattr()	●	
rename()		无论哪个参数中的链接，都不会对其进行解引用
rmdir()		若参数为符号链接，则调用失败，并将 errno 置为 ENOTDIR
setxattr()	●	
stat()	●	
statfs(), statvfs()	●	
swapon(), swapoff()	●	
truncate()	●	
unlink()		
uselib()	●	
utime(), utimes()	●	

少数情况下，符号链接本身及其所指向的文件会需要类似的功能，系统这时就会提供两套系统调用：一套会对链接解除引用，另一套则反之，后者在命名时会冠以字母 l。stat()和 lstat()就是一例。

有一点是约定俗成的：总是会对路径名中目录部分（即最后一个斜线字符前的所有组成部分）的符号链接进行解除引用操作。因此，在路径/somedir/somesubdir/file 中，若 somedir 和 somesubdir 属于符号链接，则一定会解除对这两个目录的引用，而针对 file 是否进行解引用与否，则取决于路径名所传入的系统调用。

18.11 节描述了一组自版本 2.6.16 加入的系统调用，对表 18-1 所展示的部分接口功能有所扩展。对于其中的某些调用而言，可以利用 flags 参数来控制是否对符号链接进行解引用操作。

符号链接的文件权限和所有权

大部分操作会无视符号链接的所有权和权限（创建符号链接时会为其赋予所有权限）。是否允许操作反而是由符号链接所指代文件的所有权和权限来决定。仅当在带有粘性权限位（15.4.5 节）的目录中对符号链接进行移除或改名操作时，才会考虑符号链接自身的所有权。

18.3　创建和移除（硬）链接：link()和 unlink()

link()和 unlink()系统调用分别创建和移除硬链接。

更多资源请访问稀酷客(www.ckook.com)

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

Returns 0 on success, or −1 on error

若 oldpath 中提供的是一个已存在文件的路径名，则系统调用 link()将以 newpath 参数所指定的路径名创建一个新链接。若 newpath 指定的路径名已然存在，则不会将其覆盖；相反，将产生一个错误（EEXIST）。

在 Linux 中，link()系统调用不会对符号链接进行解引用操作。若 oldpath 属于符号链接，则会将 newpath 创建为指向相同符号链接文件的全新硬链接。（换言之，newpath 也是符号链接，指向 oldpath 所指代的同一文件。）这一行为有悖于 SUSv3 规范。SUSv3 要求，除非另行规定（link()系统调用不在此列），否则所有执行路径名解析操作的函数都应对符号链接进行解引用。大多数其他 UNIX 实现的行事方式都与 SUSv3 相符。值得注意的是，Solaris 是个例外，默认情况下的行为与 Linux 相同。但若采用适当的编译器选项，又可提供符合 SUSv3 规范的行为。鉴于系统实现间的这种差异，应避免将 oldpath 参数指定为符号链接，以保障程序的可移植性。

SUSv4 承认现有实现间存在不一致性，同时规定 link()调用对符号链接解引用与否由实现定义。SUSv4 还将 linkat()纳入规范，在执行与 link()相同任务的同时，可利用 flag 参数来控制调用是否解析符号链接。更多细节参见 18.11 节。

```
#include <unistd.h>

int unlink(const char *pathname);
```

Returns 0 on success, or −1 on error

unlink()系统调用移除一个链接（删除一个文件名），且如果此链接是指向文件的最后一个链接，那么还将移除文件本身。若 pathname 中指定的链接不存在，则 unlink()调用失败，并将 errno 置为 ENOENT。

unlink()不能移除一个目录，完成这一任务需要使用 rmdir()或 remove()，将于 18.6 节进行介绍。

SUSv3 规定，若 pathname 中指定的是一个目录，则 unlink()调用失败，并将 errno 置为 EPERM。然而，在 Linux 中，unlink()在这种情况下会将 errno 置为 EISDIR 值。（对于与 SUSv3 间的这一差别，LSB 倒也并不讳言。）为保障可移植性，应用程序在检查这种情况时应做两手准备。

unlink()系统调用不会对符号链接进行解引用操作，若 pathname 为符号链接，则移除链接本身，而非链接指向的名称。

仅当关闭所有文件描述符时，方可删除一个已打开的文件

内核除了为每个 i-node 维护链接计数之外，还对文件的打开文件描述（参见图 5-2）计数。

更多资源请访问棒酷客(www.ckook.com)

当移除指向文件的最后一个链接时，如果仍有进程持有指代该文件的打开文件描述符，那么在关闭所有此类描述符之前，系统实际上将不会删除该文件。这一特性的妙用在于允许取消对文件的链接，而无需担心是否有其他进程已将其打开。（然而，对于链接数已降为 0 的打开文件，就无法将文件名与其重新关联起来。）此外，基于上述事实，还可以玩点小技巧：先创建并打开一个临时文件，随即取消对文件的链接（unlink），然后在程序中继续使用该文件。（这正是 5.12 节所述 tmpfile()函数的所作所为。）

程序清单 18-1 对此现象做了展示。

程序清单 18-1：使用 unlink()移除一个链接

dirs_links/t_unlink.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define CMD_SIZE 200
#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int fd, j, numBlocks;
    char shellCmd[CMD_SIZE];            /* Command to be passed to system() */
    char buf[BUF_SIZE];                 /* Random bytes to write to file */

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s temp-file [num-1kB-blocks] \n", argv[0]);

    numBlocks = (argc > 2) ? getInt(argv[2], GN_GT_0, "num-1kB-blocks")
                           : 100000;

    fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("open");

    if (unlink(argv[1]) == -1)          /* Remove filename */
        errExit("unlink");

    for (j = 0; j < numBlocks; j++)     /* Write lots of junk to file */
        if (write(fd, buf, BUF_SIZE) != BUF_SIZE)
            fatal("partial/failed write");

    snprintf(shellCmd, CMD_SIZE, "df -k `dirname %s`", argv[1]);
    system(shellCmd);                   /* View space used in file system */

    if (close(fd) == -1)                /* File is now destroyed */
        errExit("close");
    printf("********** Closed file descriptor\n");

    system(shellCmd);                   /* Review space used in file system */
    exit(EXIT_SUCCESS);
}
```

dirs_links/t_unlink.c

程序清单 18-1 中程序接受两个命令行参数。第一个参数标识程序应该创建的文件名称。

更多资源请访问稀酷客(www.ckook.com)

程序打开此文件后随即取消与文件名的链接。虽然文件名已消失，但是文件本身依然存在。然后程序向文件随机写入一些数据块，数据块数量由程序的第二个命令行参数（可选项）指定。这时，程序会利用 df(1)命令显示文件系统的空间使用情况。程序接着会关闭文件描述符，系统因之而将文件移除，程序会再次使用 df(1)命令来显示有所下降的磁盘使用情况。如下 shell 会话演示了运行程序清单 18-1 程序的情况：

```
$ ./t_unlink /tmp/tfile 1000000
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda10             5245020   3204044   2040976  62% /
********** Closed file descriptor
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda10             5245020   2201128   3043892  42% /
```

程序清单 18-1 使用 system()函数来执行 shell 命令，27.6 节将对此函数做详细描述。

18.4 更改文件名：rename()

借助于 rename()系统调用，既可以重命名文件，又可以将文件移至同一文件系统中的另一目录。

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

Returns 0 on success, or −1 on error

调用会将现有的一个路径名 oldpath 重命名为 newpath 参数所指定的路径名。

rename()调用仅操作目录条目，而不移动文件数据。改名既不影响指向该文件的其他硬链接，也不影响持有该文件打开描述符的任何进程，因为这些文件描述符指向的是打开文件描述，（在调用 open()之后）与文件名并无瓜葛。

以下规则适用与对 rename()的调用。

- 若 newpath 已经存在，则将其覆盖。
- 若 newpath 与 oldpath 指向同一文件，则不发生变化（且调用成功）。这很不合常理。顺着上一条规则的思路，通常的推断是：如果两个文件名 x 和 y 都存在，那么调用 rename("x","y")时应当把 x 移除才是。但如果 x 和 y 链接的是同一文件，事实却并非如此。

此规则源于早期的 BSD 实现，其动机可能是出于这样的考虑：要保障诸如 rename（"x", "x"）、rename（"x", "./x"）以及 rename（"x", "somedir/../x"）之类的调用不会移除文件，内核必须进行检查。而这种设计则有助于简化这一检查。

- rename()系统调用对其两个参数中的符号链接均不进行解引用。如果 oldpath 是一符号链接，那么将重命名该符号链接。如果 newpath 是一符号链接，那么会将其视为由 oldpath 重命名而成的普通路径名（即移除已有的符号链接 newpath）。
- 如果 oldpath 指代文件，而非目录，那么就不能将 newpath 指定为一个目录的路径名（否则将 errno 置为 EISDIR）。要想重命名一个文件到某一目录中（亦即将文件移到另一目录），newpath 必须包含新的文件名。如下调用既将一个文件移动到另一目录中，同时又将其改名：

更多资源请访问稀酷客(www.ckook.com)

```
rename("sub1/x", "sub2/y");
```

- 若将 oldpath 指定为目录名，则意在重命名该目录。这种情况下，必须保证 newpth 要么不存在，要么是一个空目录的名称。无论 newpath 是一个已有文件还是一个非空目录，调用都将出错（分别将 errno 置为 ENOTDIR 和 ENOTEMPTY）。
- 若 oldpath 是一目录，则 newpath 不能包含 oldpath 作为其目录前缀。例如，不能将/home/mtk 重命名为/home/mtk/bin（错误为 EINVAL）。
- oldpath 和 newpath 所指代的文件必须位于同一文件系统。之所以如此，是因为目录内容由硬链接列表组成，且硬链接所指向的 i-node 与目录位于同一文件系统。如前所述，rename()仅限于操作目录列表的内容。试图将一文件重命名至不同的文件系统将返回错误 EXDEV。（非要如此，必须从一个文件系统中将其文件内容复制到另一文件系统，然后再删除老文件。这正是 mv 命令的功能所在。）

18.5 使用符号链接：symlink()和 readlink()

现在来看看用于创建符号链接，以及检查其内容的系统调用。

symlink()系统调用会针对由 filepath 所指定的路径名创建一个新的符号链接——linkpath。（想移除符号链接，需使用 unlink()调用。）

```
#include <unistd.h>

int symlink(const char *filepath, const char *linkpath);
```

Returns 0 on success, or −1 on error

若 linkpath 中给定的路径名已然存在，则调用失败（且将 errno 置为 EEXIST）。由 filepath 指定的路径名可以是绝对路径，也可以是相对路径。

由 filepath 所命名的文件或目录在调用时无需存在。即便当时存在，也无法阻止后来将其删除。这时，linkpath 成为“悬空链接”，其他系统调用试图对其进行解引用操作都将出错（通常错误号为 ENOENT）。

如果指定一符号链接作为 open()调用的 pathname 参数，那么将打开链接指向的文件。有时，倒宁愿获取链接本身的内容，即其所指向的路径名。这正是 readlink()系统调用的本职工作，将符号链接字符串的一份副本置于 buffer 指向的字符数组中。

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buffer, size_t bufsiz);
```

Returns number of bytes placed in *buffer* on success, or −1 on error

bufsiz 是一个整型参数，用以告知 readlink()调用 buffer 中的可用字节数。

如果一切顺利，readlink()将返回实际放入 buffer 中的字节数。若链接长度超过 bufsiz，则置于 buffer 中的是经截断处理的字符串（并返回字符串大小，亦即 bufsiz）。

由于 buffer 尾部并未放置终止空字符，故而也无法分辨 readlink()所返回的字符串到底是经过截断处理，还是恰巧将 buffer 填满。验证后者的方法之一是重新分配一块更大的 buffer，并再次调用 readlink()。另外，还可以将 pathname 的长度定义为常量 PATH_MAX（参见 11.1

更多资源请访问稀酷客(www.ckook.com)

节），该常量定义了程序可拥有的最长路径名长度。

程序清单 18-4 演示了 readlink()的用法。

> 为限制符号链接中所能存储的最大字节数，SUSv3 定义了一个新限制 SYMLINK_MAX，要求系统实现对此加以定义，并规定其不得少于 255 个字节。写作本书时，Linux 尚未对此限制作出定义。而正文之所以建议使用 PATH_MAX，是因为该限制至少与 SYMLINK_MAX 大小相当。
>
> SUSv2 将 readlink()的返回类型定义为 int，而当前的许多实现（以及 Linux 中较老版本的 glibc）对此奉行不悖。SUSv3 则将 readlink()的返回值类型改为 ssize_t。

18.6 创建和移除目录：mkdir()和 rmdir()

mkdir()系统调用创建一个新目录。

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

int mkdir(const char *pathname, mode_t mode);
```

Returns 0 on success, or −1 on error

pathname 参数指定了新目录的路径名。该路径名可以是相对路径，也可以是绝对路径。若具有该路径名的文件已经存在，则调用失败并将 errno 置为 EEXIST。

对新目录所有权的设置遵循 15.3.1 节所述规则。

mode 参数指定了新目录的权限。（15.3.1、15.3.2 和 15.4.5 各节描述了目录权限位的含义。）对该位掩码值的指定方式既可以与 open()调用相同——对表 15-4 所列各常量进行或(|)操作，也可直接赋予八进制数值。既定的 mode 值还将于进程掩码相与（&）（参见 15.4.6 节）。另外，set-user-ID 位始终处于关闭状态，因为该位对于目录而言毫无意义。

如果在 mode 中设置了粘滞位（S_ISVTX），那么将对新目录设置该权限。

调用还会忽略在 mode 中设置的 set-group-ID 位（S_ISGID）。相反，若对其父目录设置了 set-group-ID 位，则同样会对新建目录设置该权限。15.3.1 节曾指出，对目录设置 set-group-ID 权限位将导致目录中新建文件的组 ID 取自目录组 ID，而非进程有效组 ID。mkdir()系统调用按此处描述的方式来传播 set-group-ID 权限位，以保证目录下所有子目录的行为均保持一致。

SUSv3 明文规定，mkdir()对 set-user-ID、set-group-ID 以及粘滞位的处理方式由实现定义。某些 UNIX 实现在新建目录时总是关闭这 3 个权限位。

新建目录包括两个条目：.（点），即指向目录自身的链接；..（点点），即指向父目录的链接。

> SUSv3 并未要求目录中包括.和..条目，只是要求当路径中出现.和..时，实现应能正确解释。若要保证应用程序的可移植性，则不应假设目录中存在这些条目。

mkdir()系统调用所创建的仅仅是路径名中的最后一部分。换言之，mkdir("aaa/bbb/ccc"，mode)仅当目录 aaa 和 aaa/bbb 已经存在的情况下才会成功。（这相当于 mkdir(1)命令的默认行为，但 mkdir(1)同时也提供-p 选项，可将不存在的中间目录一一创建。）

更多资源请访问稀酷客(www.ckook.com)

GNU C 语言库提供有 mkdtemp(template)函数，可谓 mkstemp()函数的“目录”版。该函数会创建一个唯一命名的目录，在对所有者开启读、写和执行权限的同时，不对任何其他用户赋予任何权限。不过，mkdtemp()所返回的并非一个文件描述符，而是一枚指针，指向经过修改处理的字符串，内含 template 中的实际目录名。该函数并未纳入 SUSv3 规范，也未获得所有 UNIX 实现的支持，但 SUSv4 对其作了定义。

rmdir()系统调用移除由 pathname 指定的目录，该目录可以是绝对路径名，也可以是相对路径名。

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Returns 0 on success, or −1 on error

要使 rmdir()调用成功，则要删除的目录必须为空。如果 pathname 的最后一部分为符号链接，那么 rmdir()调用将不对其进行解引用操作，并返回错误，同时将 errno 置为 ENOTDIR。

18.7 移除一个文件或目录：remove()

remove()库函数移除一个文件或一个空目录。

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns 0 on success, or −1 on error

如果 pathname 是一文件，那么 remove()去调用 unlink()；如果 pathname 为一目录，那么 remove()去调用 rmdir()。

与 unlink()、rmdir()一样，remove()不对符号链接进行解引用操作。若 pathname 是一符号链接，则 remove()会移除链接本身，而非链接所指向的文件。

如果移除一个文件只是为创建同名新文件做准备，那么编码时使用 remove()函数会更加简单，无需再去检查目录名所指是文件还是目录，然后再决定是调用 unlink()还是 rmdir()。

remove()属于 C 语言标准库函数，广为 UNIX 和非 UNIX 系统所支持。大多数非 UNIX 系统并不支持硬链接，所以将移除文件的函数命名为 unlink()也不合情理。

18.8 读目录：opendir()和 readdir()

本节所述库函数可用于打开一个目录，并逐一获取其包含文件的名称。

读取目录的库函数均以 getdents()系统调用（未纳入 SUSv3 规范）为基础，但其接口更易于使用。Linux 还提供了 readdir(2)系统调用（相对于此处描述的 readdir(3)库函数），所执行的任务类似于 getdents()，也因之而遭废止。

更多资源请访问稀酷客(www.ckook.com)

opendir()函数打开一个目录，并返回指向该目录的句柄，供后续调用使用。

```
#include <dirent.h>

DIR *opendir(const char *dirpath);
```

Returns directory stream handle, or NULL on error

opendir()函数打开由 dirpath 指定的目录，并返回指向 DIR 类型结构的指针。该结构即所谓目录流（directory stream），亦即调用者传递给下述其他函数的句柄。一旦从 opendir()返回，则将目录流指向目录列表的首条记录。

除了要创建的目录流所针对的目录由打开文件描述符指代之外，fdopendir()与 opendir()并无不同。

```
#include <dirent.h>

DIR *fdopendir(int fd);
```

Returns directory stream handle, or NULL on error

提供 fdopendir()函数，意在帮助应用程序免受 18.11 节所述各种竞态条件的困扰。

调用 fdopendir()成功后，文件描述符将处于系统的控制之下，且除了利用本节余下部分所描述的函数之外，程序不应采取任何其他方式对其进行访问。

SUSv4 定义了 fdopendir()函数（但 SUSv3 并未将其纳入规范）。

readdir()函数从一个目录流中读取连续的条目。

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

Returns pointer to a statically allocated structure describing next directory entry, or NULL on end-of-directory or error

每调用 readdir()一次，就会从 dirp 所指代的目录流中读取下一目录条目，并返回一枚指针，指向经静态分配而得的 dirent 类型结构，内含与该条目相关的如下信息：

```
struct dirent {
    ino_t d_ino;          /* File i-node number */
    char  d_name[];       /* Null-terminated name of file */
};
```

每次调用 readdir()都会覆盖该结构。

出于对程序可移植性的考虑，上述定义略去了 Linux dirent 结构中的各种非标准字段。这其中最令人感兴趣的当属 d_type，它同时获得了 BSD 流派的支持，但并未在其他 UNIX 系统中实现。该属性值用于标识命名于 d_name 之中文件的类型，诸如 DT_REG（普通文件）、DT_DIR（目录）、DT_LNK（符号链接）或 DT_FIFO（FIFO）。（这些名称类似于表 15-1 所列诸宏。）利用该属性值可省去为确定文件类型而对 lstat ()的调用。注意，写作本书时，该属性仅获得 Btrfs、ext2、ext3 以及 ext4 的全面支持。

调用 lstat()（或者 stat()，如果应对符号链接解引用时）可获得 d_name 所指向文件的更多信息，其中，路径名由之前调用 opendir()时指定的 dirpath 参数与“/”字符以及 d_name 字段的

更多资源请访问稀酷客(www.ckook.com)

返回值拼接组成。

readdir()返回时并未对文件名进行排序，而是按照文件在目录中出现的天然次序（这取决于文件系统向目录添加文件时所遵循的次序，及其在删除文件后对目录列表中空隙的填补方式）。（命令 ls–f 对文件列表的排列与调用 readdir()时一样，均未做排序处理。）

> 使用 scandir(3)函数可以获得经过排序处理的文件列表，且排列规则可由程序员定义，具体细节请参考手册页。尽管该函数未获 SUSv3 接纳，但得到了大多数 UNIX 实现的支持。SUSv4 也对 scandir()作了定义。

一旦遇到目录结尾或是出错，readdir()将返回 NULL，针对后一种情况，还会设置 errno 以示具体错误。为了区别这两种情况，可编码如下：

```
errno = 0;
direntp = readdir(dirp);
if (direntp == NULL) {
    if (errno != 0) {
        /* Handle error */
    } else {
        /* We reached end-of-directory */
    }
}
```

如果目录内容恰逢应用调用 readdir()扫描该目录时发生变化，那么应用程序可能无法观察到这些变动。SUSv3 明确指出，对于 readdir()是否会返回自上次调用 opendir()或 rewinddir()后在目录中增减的文件，规范不做要求。至于最后一次执行上述调用前就存在的文件，应确保其全部返回。

rewinddir()函数可将目录流回移到起点，以便对 readdir()的下一次调用将从目录的第一个文件开始。

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

closedir()函数将由 dirp 指代、处于打开状态的目录流关闭，同时释放流所使用的资源。

```
#include <dirent.h>

int closedir(DIR *dirp);
```

Returns 0 on success, or −1 on error

SUSv3 还定义了两个高级函数：telldir()和 seekdir()，允许随机访问目录流。有关这些函数的深入信息请参考手册页。

目录流与文件描述符

有一个目录流，就有一个文件描述符与之关联。dirfd()函数返回与 dirp 目录流相关联的文件描述符。

```
#include <dirent.h>

int dirfd(DIR *dirp);
```

Returns file descriptor on success, or −1 on error

更多资源请访问稀酷客(www.ckook.com)

例如，将 dirfd()返回的文件描述符传递给 fchdir()（参见 18.10 节），就可以把进程的当前工作目录改成相应目录。此外，还可以将其传递给 18.11 节所述各函数的 dirfd 参数。

dirfd()函数还见诸于 BSD 系统，但在其他实现中则鲜有踪迹。该函数未获 SUSv3 接纳，但 SUSv4 则对其做了规范。

这里值得一提的是，opendir()会为与目录流相关联的文件描述符自动设置 close-on-exec 标志（FD_CLOEXEC），以确保当执行 exec()时自动关闭该文件描述符。（SUSv3 要求这一行为。）close-on-exec 标志将在 27.4 节加以描述。

示例程序

程序清单 18-2 使用 opendir()、readdir()和 closedir()函数来列出由命令行参数所指定各目录的内容（若未提供参数则为当前工作目录）。以下是运行该程序的一个例子：

```
$ mkdir sub                    Create a test directory
$ touch sub/a sub/b            Make some files in the test directory
$ ./list_files sub             List contents of directory
sub/a
sub/b
```

程序清单 18-2：扫描一个目录

dirs_links/list_files.c

```
#include <dirent.h>
#include "tlpi_hdr.h"

static void             /* List all files in directory 'dirPath' */
listFiles(const char *dirpath)
{
    DIR *dirp;
    struct dirent *dp;
    Boolean isCurrent;          /* True if 'dirpath' is "." */

    isCurrent = strcmp(dirpath, ".") == 0;

    dirp = opendir(dirpath);
    if (dirp  == NULL) {
        errMsg("opendir failed on '%s'", dirpath);
        return;
    }

    /* For each entry in this directory, print directory + filename */

    for (;;) {
        errno = 0;              /* To distinguish error from end-of-directory */
        dp = readdir(dirp);
        if (dp == NULL)
            break;

        if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
            continue;           /* Skip . and .. */

        if (!isCurrent)
            printf("%s/", dirpath);
        printf("%s\n", dp->d_name);
    }
```

```
    if (errno != 0)
        errExit("readdir");

    if (closedir(dirp) == -1)
        errMsg("closedir");
}

int
main(int argc, char *argv[])
{
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [dir...]\n", argv[0]);

    if (argc == 1)              /* No arguments - use current directory */
        listFiles(".");
    else
        for (argv++; *argv; argv++)
            listFiles(*argv);

    exit(EXIT_SUCCESS);
}
```

dirs_links/list_files.c

readdir_r()函数

readdir_r()函数是 readdir()的变体。二者之间语义上的关键差异在于前者是可重入的，而后者不是。这是因为 readdir_r()对文件条目的返回利用的是由调用者分配的 entry 参数，而 readdir()则是将信息置于静态分配的结构并返回其指针。21.1.2 节和 31.1 节讨论了可重入性（reentrancy）。

```
#include <dirent.h>

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

Returns 0 on success, or a positive error number on error

针对既定 dirp，亦即之前调用 opendir()所打开的目录流，readdir_r()将下一项目录条目置于由 entry 指向的 dirent 结构中。另外，还会在 result[1]中放置指向该结构的一枚指针。如果抵达目录流尾部，那么会在 result[2]中返回 NULL（且 readdir_r()返回 0）。当出现错误时，readdir_r()不会返回-1，而是返回一个对应于 errno 的正整型值。

在 Linux 中，dirent 结构的 d_name 字段是大小为 256 字节的一个数组，足以容纳可能出现的最长文件名。虽然有几个其他的 UNIX 实现也为 d_name 定义了相同的大小，但 SUSv3 对此却并做规定，而另一些 UNIX 实现则将该字段定义为 1 字节的数组，并将正确分配结构大小的工作交给调用程序。这时，应将 d_name 字段大小设定为常量 NAME_MAX+1（考虑终止空字节）。为确保可移植性，应用程序应以如下方式分配 dirent 结构：

```
struct dirent *entryp;
size_t len;

len = offsetof(struct dirent, d_name) + NAME_MAX + 1;
entryp = malloc(len);
if (entryp == NULL)
    errExit("malloc");
```

1 译者注：kernel.org 的在线手册页则为*result，译者倾向于后者，但未与作者沟通。请读者自行验证。
2 译者注：同上，疑为*result。

更多资源请访问稀酷客(www.ckook.com)

鉴于 dirent 结构中 d_name 字段（该属性总是位于结构的最后）之前各类属性的数量和大小在不同系统中实现不一，采用 offsetof()宏（定义于<stddef.h>中）可避免程序对此产生依赖。

offsetof()宏接受两个参数——结构类型和该结构中某一字段的名称——并返回一 size_t 类型值，亦即该字段距结构起点的字节偏移量。这个宏之所以必要，是由于编译器为满足诸如 int 之类的类型的对齐要求，可能在结构中插入填充字节。这会导致结构中某一字段的偏移量可能要大于该属性之前所有字段的长度总和。

18.9 文件树遍历：nftw()

nftw()函数允许程序对整个目录子树进行递归遍历，并为子树中的每个文件执行某些操作（即，调用由程序员定义的函数）。

nftw()函数是对执行类似功能的老函数 ftw()的加强。由于提供了更多功能，对符号链接的处理也更易于把握（SUSv3 规定，ftw()的函数实现无论是否对符号链接进行解引用，均符合规范），故而新近开发的应用程序应考虑采用 nftw()（new ftw）。SUSv3 将 nftw()和 ftw()均纳入规范，但 SUSv4 将后者标记为“已废止”。

GNU C 语言函数库也提供了派生自 BSD 分支的 fts API(fts_open()、fts_read()、fts_children()、fts_set()和 fts_close())。这些函数执行的任务类似于 ftw()和 nftw()，但在遍历树方面为应用程序提供了更大的灵活性。然而，因为这些 API 目前尚未获得业界标准的接纳，也鲜有见诸于 BSD 后裔之外的其他 UNIX 实现，所以在此略而不论。

nftw()函数遍历由 dirpath 指定的目录树，并为目录树中的每个文件调用一次由程序员定义的 func 函数。

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

int nftw(const char *dirpath,
         int (*func) (const char *pathname, const struct stat *statbuf,
                      int typeflag, struct FTW *ftwbuf),
         int nopenfd, int flags);
```

Returns 0 after successful walk of entire tree, or −1 on error, or the first nonzero value returned by a call to *func*

默认情况下，nftw()会针对给定的树执行未排序的前序遍历，即对各目录的处理要先于各目录下的文件和子目录。

当 nftw()遍历目录树时，最多会为树的每一层级打开一个文件描述符。参数 nopenfd 指定了 nftw()可使用文件描述符数量的最大值。如果目录树深度超过这一最大值，那么 nftw()会在做好记录的前提下，关闭并重新打开描述符，从而避免同时持有的描述符数目突破上限 nopenfd（从而导致运行越来越慢）。在较老的 UNIX 实现中，有的系统要求每个进程可打开的文件描述符数量不得超过 20 个，这更突显出这一参数的必要性。现代 UNIX 实现允许进程打开大量的文件描述符，因此，在指定该数目时出手可以大方一些（比如，10 或者更多）。

nftw()的 flags 参数由 0 个或多个下列常量相或(|)组成，这些常量可对函数的操作做出修正。

更多资源请访问稀酷客(www.ckook.com)

FTW_CHDIR

在处理目录内容之前先调用 chdir()进入每个目录。如果打算让 func 在 pathname 参数所指定文件的驻留目录下展开某些工作，那么就应当使用这一标志。

FTW_DEPTH

对目录树执行后序遍历。这意味着，nftw()会在对目录本身执行 func 之前先对目录中的所有文件（及子目录）执行 func 调用。（这一标志名称容易引起误会——nftw()遍历目录树遵循的是深度优先原则，而非广度优先。而这一标志的作用其实就是将先序遍历改为后序遍历。）

FTW_MOUNT

不会越界进入另一文件系统。因此，如果树中某一子目录是挂载点，那么不会对其进行遍历。

FTW_PHYS

默认情况下，nftw()对符号链接进行解引用操作。而使用该标志则告知 nftw()函数不要这么做。相反，函数会将符号链接传递给 func 函数，并将 typeflag 值置为 FTW_SL，如下所述。

nftw()为每个文件调用 func 时传递 4 个参数。第一个参数 pathname 是文件的路径名。这个路径名可以是绝对路径，也可以是相对路径。如果指定 dirpath 时使用的是绝对路径，那么 pathname 就可能是绝对路径。反之，如果指定 dirpath 时使用的是相对路径名，则 pathname 中的路径可能是相对于进程调用 ntfw()时的当前工作目录而言。第二个参数 statbuf 是一枚指针，指向 stat 结构（参见 15.1 节），内含该文件的相关信息。第三个参数 typeflag 提供了有关该文件的深入信息，并具有如下特征值之一。

FTW_D

这是一个目录。

FTW_DNR

这是一个不能读取的目录（所以 nftw()不能遍历其后代）。

FTW_DP

正在对一个目录进行后序遍历，当前项是一个目录，其所包含的文件和子目录已经处理完毕。

FTW_F

该文件的类型是除目录和符号链接以外的任何类型。

FTW_NS

对该文件调用 stat()失败，可能是因为权限限制。Statbuf 中的值未定义。

FTW_SL

这是一个符号链接。仅当使用 FTW_PHYS 标志调用 nftw()函数时才返回该值。

FTW_SLN

这是一个悬空的符号链接。仅当未在 flags 参数中指定 FTW_PHYS 标志时才会出现该值。

Func 的第四个参数 ftwbuf 是一枚指针，所指向结构定义如下：

```
struct FTW {
    int base;        /* Offset to basename part of pathname */
    int level;       /* Depth of file within tree traversal */
};
```

该结构的 base 字段是指 func 函数中 pathname 参数内文件名部分（最后一个“/”字符之

更多资源请访问稀酷客(www.ckook.com)

后的部分）的整型偏移量。level 字段是指该条目相对于遍历起点（其 level 为 0）的深度。

每次调用 func 都必须返回一个整型值，由 nftw()加以解释。如果返回 0，nftw()会继续对树进行遍历，如果所有对 func 的调用均返回 0，那么 nftw()本身也将返回 0 给调用者。若返回非 0 值，则通知 nftw()立即停止对树的遍历，这时 nftw()也会返回相同的非 0 值。

由于 nftw()使用的数据结构是动态分配的，故而应用程序提前终止目录树遍历的唯一方法就是让 func 调用返回一个非 0 值。调用 longjmp()（6.8 节）从 func 退出会导致不可预期的结果——至少会引起内存泄漏。

示例程序

程序清单 18-3 展示了 nftw()的使用。

程序清单 18-3：使用 nftw()遍历目录树

dirs_links/nftw_dir_tree.c

```
#define _XOPEN_SOURCE 600       /* Get nftw() and S_IFSOCK declarations */
#include <ftw.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s\n", msg);
    fprintf(stderr, "Usage: %s [-d] [-m] [-p] [directory-path]\n", progName);
    fprintf(stderr, "\t-d Use FTW_DEPTH flag\n");
    fprintf(stderr, "\t-m Use FTW_MOUNT flag\n");
    fprintf(stderr, "\t-p Use FTW_PHYS flag\n");
    exit(EXIT_FAILURE);
}

static int                      /* Function called by nftw() */
dirTree(const char *pathname, const struct stat *sbuf, int type,
        struct FTW *ftwb)
{
    switch (sbuf->st_mode & S_IFMT) {       /* Print file type */
    case S_IFREG:  printf("-"); break;
    case S_IFDIR:  printf("d"); break;
    case S_IFCHR:  printf("c"); break;
    case S_IFBLK:  printf("b"); break;
    case S_IFLNK:  printf("l"); break;
    case S_IFIFO:  printf("p"); break;
    case S_IFSOCK: printf("s"); break;
    default:       printf("?"); break;      /* Should never happen (on Linux) */
    }

    printf(" %s  ",
            (type == FTW_D)  ? "D  " : (type == FTW_DNR) ? "DNR" :
            (type == FTW_DP) ? "DP " : (type == FTW_F)   ? "F  " :
            (type == FTW_SL) ? "SL " : (type == FTW_SLN) ? "SLN" :
            (type == FTW_NS) ? "NS " : "  ");

    if (type != FTW_NS)
        printf("%7ld ", (long) sbuf->st_ino);
    else
```

```
        printf("        ");

    printf(" %*s", 4 * ftwb->level, "");        /* Indent suitably */
    printf("%s\n",  &pathname[ftwb->base]);     /* Print basename */
    return 0;                                   /* Tell nftw() to continue */
}

int
main(int argc, char *argv[])
{
    int flags, opt;

    flags = 0;
    while ((opt = getopt(argc, argv, "dmp")) != -1) {
        switch (opt) {
        case 'd': flags |= FTW_DEPTH;   break;
        case 'm': flags |= FTW_MOUNT;   break;
        case 'p': flags |= FTW_PHYS;    break;
        default:  usageError(argv[0], NULL);
        }
    }

    if (argc > optind + 1)
        usageError(argv[0], NULL);

    if (nftw((argc > optind) ? argv[optind] : ".", dirTree, 10, flags) == -1) {
        perror("nftw");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

dirs_links/nftw_dir_tree.c

程序清单 18-3 中程序以层级缩进方式显示了一个目录树中的文件。每行显示一个文件，内容包括文件名、文件类型及 i-node 编号。可通过命令行选项来指定 nftw()调用中的 flags 参数值。下面的 shell 会话展示了运行程序的示例结果。首先创建一个新的空目录，并在其中填充各种类型的文件。

```
$ mkdir dir
$ touch dir/a dir/b                 Create some plain files
$ ln -s a dir/sl                    and a symbolic link
$ ln -s x dir/dsl                   and a dangling symbolic link
$ mkdir dir/sub                     and a subdirectory
$ touch dir/sub/x                   with a file of its own
$ mkdir dir/sub2                    and another subdirectory
$ chmod 0 dir/sub2                  that is not readable
```

然后使用该程序调用 nftw()函数，其 flags 参数为 0：

```
$ ./nftw_dir_tree dir
d D    2327983  dir
- F    2327984      a
- F    2327985      b
- F    2327984      sl              The symbolic link sl was resolved to a
l SLN  2327987      dsl
d D    2327988      sub
- F    2327989          x
d DNR  2327994      sub2
```

从以上输出可见，对符号链接 sl 进行了解析。

然后再使用该程序来调用 nftw()函数，令 flags 参数包含 FTW_PHYS 和 FTW_DEPTH 标志：

```
$ ./nftw_dir_tree -p -d dir
- F    2327984      a
- F    2327985      b
l SL   2327986      sl              The symbolic link sl was not resolved
l SL   2327987      dsl
- F    2327989          x
d DP   2327988      sub
d DNR  2327994      sub2
d DP   2327983  dir
```

从以上输出可见，未对符号链接 sl 进行解析。

nftw()的 FTW_ACTIONRETVAL 标识

始于 2.3.3 版本，glibc 允许在 ntfw()的 flags 参数中指定一个额外的非标准标志 FTW_ACTIONRETVAL。此标志改变了 nftw()函数对 func()返回值的解释方式。当指定该标识时，func()应返回下列值之一。

FTW_CONTINUE

与传统 func()函数返回 0 时一样，继续处理目录树中的条目。

FTW_SKIP_SIBLINGS.

不再进一步处理当前目录中的条目，恢复对父目录的处理。

FTW_SKIP_SUBTREE

如果 pathname 是目录（即 typeflag 为 FTW_D），那么就不对该目录下的条目调用 func()。恢复进行对该目录的下一个同级目录的处理。

FTW_STOP

与传统 func()函数返回非 0 值时一样，不再进一步处理目录树下的任何条目。nftw()将返回 FTW_STOP 给调用者。

想从<ftw.h>文件中获得对 FTW_ACTIONRETVAL 的定义，必须定义_GNU_SOURCE 特性测试宏。

18.10 进程的当前工作目录

一个进程的当前工作目录（current working directory）定义了该进程解析相对路径名的起点。新进程的当前工作目录继承自其父进程。

获取当前工作目录

进程可使用 getcwd()来获取当前工作目录。

```
#include <unistd.h>

char *getcwd(char *cwdbuf, size_t size);
```

Returns *cwdbuf* on success, or NULL on error

getcwd()函数将内含当前工作目录绝对路径的字符串（包括结尾空字符）置于 cwdbuf 指

更多资源请访问稀酷客(www.ckook.com)

向的已分配缓冲区中。调用者必须为 cwdbuf 缓冲区分配至少 sizeg 个字节的空间。（通常，cwdbuf 的大小与 PATH_MAX 常量相当。）

一旦调用成功，getcwd()将返回一枚指向 cwdbuf 的指针。如果当前工作目录的路径名长度超过 size 个字节，那么 getcwd()会返回 NULL，并将 errno 置为 ERANGE。

在 Linux/x86-32 系统中，getcwd()返回指针所指向的字符串最大长度可达 4096 个字节。如果当前工作目录（以及 cwdbuf 和 size）突破了这一限制，那么就会直接对路径名做截断处理，移去始于起点的整个目录前缀（字符串仍以空字符结尾）。换言之，当当前工作目录的绝对路径超出这一限制时，getcwd()的行为也不再可靠。

实际上，Linux 的 getcwd()系统调用为要返回的路径名在内部分配了一个虚拟内存页。x86-32 架构的页大小为 4096 字节，而在页尺寸更大的架构中（比如，Alpha 的页大小为 8192 字节），getcwd()能返回更长的路径名。

若 cwdbuf 为 NULL，且 size 为 0，则 glibc 封装函数会为 getcwd()按需分配一个缓冲区，并将指向该缓冲区的指针作为函数的返回值。为避免内存泄漏，调用者之后必须调用 free()来释放这一缓冲区。对可移植性有所要求的应用程序应当避免依赖该特性。大多数其他实现则针对 SUSv3 规范提供了一个更为简单的扩展。如果 cwdbuf 是 NULL，那么 getcwd()将分配一个大小为 size 字节的缓冲区，用于向调用者返回结果。glibc 的 getcwd()也实现了这一特性。

GNU C 函数库还为获取当前工作目录提供了另外两个函数。派生自 BSD 的 getwd(path)函数容易引起缓冲区溢出，因为该函数无法为返回的路径名长度设定上限。get_current_dir_name()函数也会返回包含当前工作目录名的一个字符串。虽然该函数易于使用，但却不具有可移植性。考虑到安全性和可移植性，getcwd()无疑是不二之选（前提是避免使用 GNU 扩展功能）。

只要具有合适的权限（大体要求是，身为进程属主或者具有 CAP_SYS_PTRACE 能力），就可通过读取（readlink()）Linux 专有符号链接/proc/PID/cwd 的内容来确定任何进程的当前工作目录。

改变当前工作目录

chdir()系统调用将调用进程的当前工作目录改变为由 pathname 指定的相对或绝对路径名（如属于符号链接，还会对其解除引用）。

```
#include <unistd.h>

int chdir(const char *pathname);
```

Returns 0 on success, or −1 on error

fchdir()系统调用与 chdir()作用相同，只是在指定目录时使用了文件描述符，而该描述符是之前调用 open()打开相应目录时获得的。

```
#define _XOPEN_SOURCE 500     /* Or: #define _BSD_SOURCE */
#include <unistd.h>

int fchdir(int fd);
```

Returns 0 on success, or −1 on error

更多资源请访问稀酷客(www.ckook.com)

以下代码片段所示为使用 fchdir()将进程的当前工作目录变为另一位置，然后再改回原始位置：

```
int fd;

fd = open(".", O_RDONLY);        /* Remember where we are */
chdir(somepath);                 /* Go somewhere else */
fchdir(fd);                      /* Return to original directory */
close(fd);
```

使用 chdir()达到同等效果的代码如下所示：

```
char buf[PATH_MAX];

getcwd(buf, PATH_MAX);           /* Remember where we are */
chdir(somepath);                 /* Go somewhere else */
chdir(buf);                      /* Return to original directory */
```

18.11 针对目录文件描述符的相关操作

始于版本 2.6.16，Linux 内核提供了一系列新的系统调用，在执行与传统系统调用相似任务的同时，还提供了一些附加功能，对某些应用程序非常有用。表 18-2 对这些调用进行了归纳。之所以在本章介绍这些系统调用，是因为它们对进程当前工作目录的传统语义做了改动。

表 18-2：系统调用使用目录文件描述来解释相对路径

新 接 口	类似的传统接口	备 注
faccessat()	access()	支持 AT_EACCESS 和 AT_SYMLINK_NOFOLLOW 标志
fchmodat()	chmod()	
fchownat()	chown()	支持 AT_SYMLINK_NOFOLLOW 标志
fstatat()	stat()	支持 AT_SYMLINK_NOFOLLOW 标志
linkat()	link()	支持（始于 Linux 2.6.18）AT_SYMLINK_FOLLOW 标志
mkdirat()	mkdir()	
mkfifoat()	mkfifo()	基于 mknodat()的库函数
mknodat()	mknod()	
openat()	open()	
readlinkat()	readlink()	
renameat()	rename()	
symlinkat()	symlink()	
unlinkat()	unlink()	支持 AT_REMOVEDIR 标志
utimensat()	utimes()	支持 AT_SYMLINK_NOFOLLOW 标志

为便于描述这些系统调用，这里就以 openat()为例。

```
#define _XOPEN_SOURCE 700      /* Or define _POSIX_C_SOURCE >= 200809 */
#include <fcntl.h>

int openat(int dirfd, const char *pathname, int flags, ... /* mode_t mode */);
```

Returns file descriptor on success, or −1 on error

更多资源请访问稀酷客(www.ckook.com)

openat()系统调用类似于传统的 open()系统调用，只是添加了一个 dirfd 参数，其作用如下。

- 如果 pathname 中为一相对路径名，那么对其解释则以打开文件描述符 dirfd 所指向的目录为参照点，而非进程的当前工作目录。
- 如果 pathname 中为一相对路径，且 dirfd 中所含为特殊值 AT_FDCWD，那么对 pathname 的解释则相对与进程当前工作目录（即与 open(2)行为一致）而言。
- 如果 pathname 中为绝对路径，那么将忽略 dirfd 参数。

openat()的 flag 参数目的与 open()相同。然而，部分表 18-2 中所列系统调用还支持 flags 参数，这是相应的传统系统调用所不具备的，其目的在于修改调用语义。出现频率最高的标志为 AT_SYMLINK_NOFOLLOW，其含义是如果 pathname 为符号链接，那么系统调用将操作于符号链接本身，而非符号链接所指向的文件。(linkat()系统调用提供了 AT_SYMLINK_FOLLOW 标志，其作用正好相反，即改变 linkat()的默认行为，当 oldpath 属于符号链接时对其进行解引用操作。)有关其他标志的详情，请参考相应手册页。

之所以要支持表 18-2 中所列的系统调用，其原因有二（此处再以 openat()为例）。

- 当调用 open()打开位于当前工作目录之外的文件时，可能会发生某些竞态条件。而使用 openat()就能够避免这一问题。在调用 open()的同时，如果 pathname 目录前缀的某些部分发生了改变，就可能导致竞争。要想避免这类竞态，可以针对目标目录打开一个文件描述符，然后将该描述符传递给 openat()。
- 如第 29 章所述，工作目录是进程的属性之一，为进程中所有线程所共享。而对某些应用程序而言，需要针对不同线程拥有不同的“虚拟”工作目录。将 openat()与应用所维护的目录文件描述符相结合，就可以模拟出这一功能。

SUSv3 并未对这些系统调用加以规范，但 SUSv4 将其包括在内。为了获得对这些系统调用的声明，必须在包含相应头文件之前（比如定义 open()的<fcntl.h>）将_XOPEN_SOURCE 特性测试宏定义为大于或等于 700 的值。另外，将_POSIX_C_SOURCE 宏的值定义为大于或等于 200809 也能收到同样效果。(在 2.10 版本之前的 glibc 中，要获得对这些系统调用的声明还需要定义_ATFILE_SOURCE 宏。)

Solaris 9 及其更高版本也提供了一些表 18-2 所列接口的版本，只是语义略有不同。

18.12 改变进程的根目录：chroot()

每个进程都有一个根目录，该目录是解释绝对路径（即那些以/开始的目录）时的起点。默认情况下，这是文件系统的真实根目录。(新进程从其父进程处继承根目录。)有些场合需要改变一个进程的根目录，而特权级（CAP_SYS_CHROOT）进程通过 chroot()系统调用能够做到这一点。

```
#define _BSD_SOURCE
#include <unistd.h>

int chroot(const char *pathname);
```

Returns 0 on success, or −1 on error

chroot()系统调用将进程的根目录改为由 pathname 指定的目录（如果 pathname 是符号链接，还将对其解引用）。自此，对所有绝对路径名的解释都将以该文件系统的这一位置作为起

更多资源请访问稀酷客(www.ckook.com)

点。鉴于这会将应用程序限定于文件系统的特定区域，有时也将此称为设立了一个 chroot 监禁区。

SUSv2 包含了对 chroot()的定义（标记为 LEGACY），但 SUSv3 又将其删去。无论如何，chroot()还是获得了大多数 UNIX 实现的支持。

> 借助于 chroot()系统调用，chroot 命令可在 chroot 监禁区中执行 shell 命令。
>
> 而通过读取（readlink()）Linux 专有/proc/PID/root 符号链接的内容，可以获取任何进程的根目录。

ftp 程序就是应用 chroot()的典型实例之一。作为一种安全措施，当用户匿名登录 ftp 时，ftp 程序将使用 chroot()为新进程设置根目录——一个专门预留给匿名登录用户的目录。调用 chroot()后，用户将受困于文件系统中新根目录下的子树中，无法在整个文件系统中信马由缰。（这里所依赖的事实是根目录是其自身的父目录。也就是说/..是/的一个链接，所以改变目录到/后再执行 cd ..命令时，用户依然会待在同一目录下。）

> 一些 UNIX 实现（不包括 Linux）允许多个硬链接指向同一目录，这时就有可能在一个子目录中创建指向其父目录（或者更高层级远祖目录）的硬链接。在这种系统实现中，如果存在指向监禁区目录树之外的硬链接，那么监禁区的安全将受到威胁。而指向监禁区之外目录的符号链接则不是问题，因为对这些符号链接的解释将在进程新根目录的框架之内进行，所以是无法染指到 chroot 监禁区之外的。

通常情况下，不是随便什么程序都可以在 chroot 监禁区中运行的，因为大多数程序与共享库之间采取的是动态链接的方式。因此，要么只能局限于运行静态链接程序，要么就在监禁区中复制一套标准的共享库系统目录（比如，包括/lib 和/usr/lib）（针对这一点，14.9.4 节描述的绑定挂载特性就派上了用场）。

chroot()系统调用从未被视为一个完全安全的监禁机制。首先，特权级程序可以在随后对 chroot()的进一步调用中利用种种手段而越狱成功。例如，特权级（CAP_MKNOD）程序能够使用 mknod()来创建一个内存设备文件（类似于/dev/mem），并通过该设备来访问 RAM 的内容，到那时，就一切皆有可能了。通常，最好不要在 chroot 监禁区文件系统内放置 set-user-ID-root 程序。

即便是对于无特权程序，也必须小心防范如下几条可能的越狱路线。

- 调用 chroot()并未改变进程的当前工作目录。因此，通常应在调用 chroot()之前或者之后调用一次 chdir()函数（例如，chroot()调用之后执行 chdir("/")）。如果没有这么做，那么进程就能够使用相对路径去访问监狱之外的文件和目录。（一些 BSD 的衍生系统杜绝了这一可能性——如果当前工作目录位于新的根目录树之外，那么 chroot()调用会将其修改为与根目录一致。）
- 如果进程针对监禁区之外的某一目录持有一打开文件描述符，那么结合 fchdir()和 chroot()即可越狱成功，如下面代码所示：

```
int fd;

fd = open("/", O_RDONLY);
chroot("/home/mtk");            /* Jailed */
fchdir(fd);
chroot(".");                    /* Out of jail */
```


更多资源请访问稀酷客(www.ckook.com)

为了防止这种可能性，必须关闭所有指向监禁区外目录的文件描述符。（其他一些 UNIX 实现提供了 fchroot()系统调用，可用于获得与上述代码片段类似的结果。）

- 即使针对上述可能性采取了防范措施，仍不足以阻止任意非特权程序（即无法控制其操作的程序）越狱成功。遭到囚禁的进程仍然能够利用 UNIX 域套接字来接受（自另一进程处）指向监禁区外目录的文件描述符。（61.13.3 节简要描述了进程间利用套接字来传递文件描述符的概念。）将这一文件描述指定为 fchdir()调用的入参，程序即可将其当前工作目录置于监禁区外，之后再通过相对路径来随意访问文件和目录。

一些 BSD 衍生系统提供的 jail()系统调用解决了包括上述问题在内的不少问题，其所创建的监禁区即使针对特权级进程也是安全的。

18.13 解析路径名：realpath()

realpath()库函数对 pathname（以空字符结尾的字符串）中的所有符号链接一一解除引用，并解析其中所有对/.和/..的引用，从而生成一个以空字符结尾的字符串，内含相应的绝对路径名。

```
#include <stdlib.h>

char *realpath(const char *pathname, char *resolved_path);
```

Returns pointer to resolved pathname on success, or NULL on error

生成的字符串将置于 resolved_path 指向的缓冲区中，该字符串应当是一个字符数组，长度至少为 PATH_MAX 个字节。一旦调用成功，realpath()将返回指向该字符串的一枚指针。

glibc 的 realpath()实现允许调用者将 resolved_path 参数指定为空。这时，realpath()会为经解析生成的路径名分配一个多达 PATH_MAX 个字节的缓冲区，并将指向该缓冲区的指针作为结果返回。（调用者必须自行调用 free()来释放该缓冲区。）SUSv3 并未将该扩展功能纳入规范，但 SUSv4 对其进行了定义。

程序清单 18-4 中的程序采用 readlink()和 realpath()来读取符号链接的内容，并将该链接解析为一个绝对路径名。下面是运行该程序的一个示例：

```
$ pwd                                   Where are we?
/home/mtk
$ touch x                               Make a file
$ ln -s x y                             and a symbolic link to it
$ ./view_symlink y
readlink: y --> x
realpath: y --> /home/mtk/x
```

程序清单 18-4：读取并解析一个符号链接

dirs_links/view_symlink.c

```
#include <sys/stat.h>
#include <limits.h>              /* For definition of PATH_MAX */
#include "tlpi_hdr.h"

#define BUF_SIZE PATH_MAX

int
main(int argc, char *argv[])
```


更多资源请访问稀酷客(www.ckook.com)

```
{
    struct stat statbuf;
    char buf[BUF_SIZE];
    ssize_t numBytes;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);
    if (lstat(argv[1], &statbuf) == -1)
        errExit("lstat");

    if (!S_ISLNK(statbuf.st_mode))
        fatal("%s is not a symbolic link", argv[1]);

    numBytes = readlink(argv[1], buf, BUF_SIZE - 1);
    if (numBytes == -1)
        errExit("readlink");
    buf[numBytes] = '\0';                       /* Add terminating null byte */
    printf("readlink: %s --> %s\n", argv[1], buf);

    if (realpath(argv[1], buf) == NULL)
        errExit("realpath");
    printf("realpath: %s --> %s\n", argv[1], buf);

    exit(EXIT_SUCCESS);
}
```

dirs_links/view_symlink.c

18.14 解析路径名字符串：dirname()和 basename()

dirname()和 basename()函数将一个路径名字符串分解成目录和文件名两部分。（这些函数执行的任务与 dirname(1)和 basename(1)命令相类似。）

```
#include <libgen.h>

char *dirname(char *pathname);
char *basename(char *pathname);
```

Both return a pointer to a null-terminated (and possibly statically allocated) string

比如，给定路径名为/home/britta/prog.c，dirname()将返回/home/britta，而 basename()将返回 prog.c。将 dirname()返回的字符串与一斜线字符（/）以及 basename()返回的字符串拼接起来，将生成一条完整的路径名。

关于 dirname()和 basename()的操作请注意以下几点。

- 将忽略 pathname 中尾部的斜线字符。
- 如果 pathname 中未包含斜线字符，那么 dirname()将返回字符串.（点），而 basename()将返回 pathname。
- 如果 pathname 仅由一个斜线字符组成，那么 dirname()和 basename()均将返回字符串/。将其应用于上述的拼接规则，所创建的路径名字符串为///。该路径名属于有效路径名。因为多个连续斜线字符相当于单个斜线字符，所以路径名///就相当于路径名/。
- 如果 pathname 为空指针或者空字符串，那么 dirname()和 basename()均将返回字符串.（点）。（拼接这些字符串将生成路径名./.，对等于.，即当前目录。）

更多资源请访问稀酷客(www.ckook.com)

表 18-3 所示为 dirname()和 basename()针对各种示例路径名所返回的字符串。

表 18-3：dirname()和 basename()返回的字符串示例

路径名字符串	dirname()	basename()
/	/	/
/usr/bin/zip	/usr/bin	zip
/etc/passwd////	/etc	passwd
/etc////passwd	/etc	passwd
etc/passwd	etc	passwd
passwd	.	passwd
passwd/	.	passwd
..	.	..
NULL	.	.

程序清单 18-5：dirname()和 basename()的应用

dirs_links/t_dirbasename.c

```
#include <libgen.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *t1, *t2;
    int j;

    for (j = 1; j < argc; j++)  {
        t1 = strdup(argv[j]);
        if (t1 == NULL)
            errExit("strdup");
        t2 = strdup(argv[j]);
        if (t2 == NULL)
            errExit("strdup");

        printf("%s ==> %s + %s\n", argv[j], dirname(t1), basename(t2));

        free(t1);
        free(t2);
    }

    exit(EXIT_SUCCESS);
}
```

dirs_links/t_dirbasename.c

dirname()和 basename()均可修改 pathname 所指向的字符串。因此，如果希望保留原有的路径名字符串，那么就必须向 dirname()和 basename()传递该字符串的副本，如程序清单 18-5 所示。该程序使用 strdup()（该函数调用了 malloc()）来制作传递给 dirname()和 basename()的字符串副本，然后再使用 free()将其释放。

最后需要指出的是，dirname()和 basename()所返回的指针均可指向经由静态分配的字符

更多资源请访问棒酷客(www.ckook.com)

串，对相同函数的后续调用可能会修改这些字符串的内容。

18.15 总结

i-node 中并不包含文件的名称。相反，对文件的命名利用的是目录条目，而目录则是列出文件名和 i-node 编号之间对应关系的一个表格。也将这些目录条目称作（硬）链接。一个文件可以有多个链接，这些链接之间的地位是平等的。可使用 link()和 unlink()来创建和移除链接，对文件的重命名则使用系统调用 rename()。

调用 symlink()可创建符号（或者软）链接。符号链接在某些方面与硬链接相类似，其差异则在于符号链接可以跨越文件系统边界，还可指代目录。符号链接只是一个内容包含了另一文件名称的文件，可通过 readlink()来获取该文件的名称。（目标）i-node 的链接计数中并未包含符号链接，如果将该链接所指向的文件移除，那么此链接将处于悬空状态。一些系统调用会自动对符号链接进行解引用（下溯），其余的则不会。有时系统会提供两种版本的系统调用，一种会解引用符号链接，另一种则不会，例如 stat()和 lstat()。

创建目录使用的是 mkdir()，移除目录则使用 rmdir()。而扫描一个目录的内容则可使用 opendir()、readdir()以及相关函数。nftw()函数允许程序遍历一棵完整的目录树，并为树中每个文件调用由程序员定义的函数。

remove()函数可以用来移除一个文件（即一个链接）或者一个空目录。

每个进程都拥有一个根目录和一个当前工作目录，分别作为解释绝对路径和相对路径的参照点。可通过 chroot()和 chdir()系统调用来修改这些属性。而 getcwd()函数则返回进程的当前工作目录。

Linux 还提供了一套新的系统调用（如：openat()），其行为与其传统同行（如：open()）相类似，不同之处则在于可利用新的系统调用来提供一个指向目录的文件描述符（而非进程的当前工作目录），用于作为解释相对路径名的参照点。这将有助于避免特定类型的竞态条件，以及为每个线程实现虚拟工作目录。

realpath()函数解析一个路径名——解引用所有的符号链接，并将所有的.和..解析为相应目录——从而生成相应的绝对路径名。dirname()和 basename()函数可用来将路径名分解为目录和文件名两部分。

18.16 练习

18-1. 4.3.2 节曾指出，如果一个文件正处于执行状态，那么要将其打开以执行写操作是不可能的（open()调用返回-1，且将 errno 置为 ETXTBSY）。然而，在 shell 中执行如下操作却是可能的：

```
$ cc -o longrunner longrunner.c
$ ./longrunner &                    Leave running in background
$ vi longrunner.c                   Make some changes to the source code
$ cc -o longrunner longrunner.c
```

最后一条命令覆盖了现有的同名可执行文件。原因何在？（提示：在每次编译后调用 ls –li 命令来查看可执行文件的 i-node 编号。）

18-2. 以下代码中对 chmod()的调用为什么会失败？

更多资源请访问稀酷客(www.ckook.com)

```
mkdir("test", S_IRUSR | S_IWUSR | S_IXUSR);
chdir("test");
fd = open("myfile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
symlink("myfile", "../mylink");
chmod("../mylink", S_IRUSR);
```

18-3. 实现 realpath()。

18-4. 修改程序清单 18-2 中的程序，用 readdir _r()来取代 readdir()。

18-5. 实现一个功能与 getcwd()相当的函数。提示：要获得当前工作目录的名称，可调用 opendir()和 readdir()来遍历其父目录（..）中的各个条目，查找其中与当前工作目录具有相同 i-node 编号及设备编号（即，分别为 stat()和 lstat()调用所返回 stat 结构中的 st_ino 和 st_dev 属性）的一项。如此这般，沿着目录树层层拾级而上（chdir("..")）并进行扫描，就能构建出完整的目录路径。当父目录与当前工作目录相同时（回忆/..与/相同的情况），就结束遍历。无论调用该函数成功与否，都应将调用者遣回其起始目录（使用 open()和 fchdir()能很方便地实现这一功能）。

18-6. 使用 FTW_DEPTH 标志来修改程序清单 18-3(nftw_dir_tree.c)中的程序。注意目录树遍历顺序的差异。

18-7. 编写一程序，使用 nftw()来遍历目录树，并打印出树中各类文件（普通文件、目录、符号链接等）的总和及百分比。

18-8. 实现 nftw()。（需要使用 opendir()、readdir()、closedir()和 stat()等系统调用。）

18-9. 18.10 节展示了两种技术（分别为 fchdir()和 chdir()），用于在将当前工作目录转到另一位置后，再返回之前的当前工作目录。假设需要反复执行这一操作，哪种方法更为高效？原因何在？请写一段程序加以验证。

更多资源请访问稀酷客(www.ckook.com)

第 19 章

监控文件事件

某些应用程序需要对文件或目录进行监控，已侦测其是否发生了特定事件。例如，当把文件加入或移出一目录时，图形化文件管理器应能判定此目录是否在其当前显示之列，而守护进程可能也想要监控自己的配置文件，以了解其是否被修改。

自内核 2.6.13 起，Linux 开始提供 inotify 机制，以允许应用程序监控文件事件。本章将介绍 inotify 的用法。

inotify 机制所取代的是 dnotify，后者一来较为陈旧，二来仅具备前者的部分功能。本章会在结尾处对 dnotify 做简要介绍，着重突出 inotify 的优势所在。

inotify 和 dnotify 都是 Linux 专有机制。（仅有少数其他系统提供类似机制。比如，BSD 所提供的 kqueue API。）

> 有几个函数库提供的（类似）API，比之 inotify 和 dnotify，既更为抽象，在可移植性方面也略胜一筹。某些应用可能更倾向于采用这样的函数库。而这其中的一些库也会调用 inotify 和 dnotify，前提是要获得操作系统的支持。FAM（文件变更监控 http:// oss.sgi.com/projects/fam/）和 Gamin (http://www.gnome.org/～veillard/gamin/)便是此类库的两个例子。

19.1 概述

使用 inotify API 有以下几个关键步骤。

1. 应用程序使用 inotify_init()来创建一 inotify 实例，该系统调用所返回的文件描述符用于在后续操作中指代该实例。
2. 应用程序使用 inotify_add_watch()向 inotify 实例（由步骤 1 创建）的监控列表添加条目，藉此告知内核哪些文件是自己的兴趣所在。每个监控项都包含一个路径名以及相关的位掩码。位掩码针对路径名指明了所要监控的事件集合。作为函数结果，inotify_add_watch()将返回一监控描述符，用于在后续操作中指代该监控项。（系统调用 inotify_rm_watch()执行其逆向操作，将之前添加入 inotify 实例的监控项移除。）
3. 为获得事件通知，应用程序需针对 inotify 文件描述符执行 read()操作。每次对 read()的成

更多资源请访问稀酷客(www.ckook.com)

功调用，都会返回一个或多个 inotify_event 结构，其中各自记录了处于 inotify 实例监控之下的某一路径名所发生的事件。

4. 应用程序在结束监控时会关闭 inotify 文件描述符。这会自动清除与 inotify 实例相关的所有监控项。

inotify 机制可用于监控文件或目录。当监控目录时，与路径自身及其所含文件相关的事件都会通知给应用程序。

inotify 监控机制为非递归。若应用程序有意监控整个目录子树内的事件，则需对该树中的每个目录发起 inotify_add_watch()调用。

可使用 select()、poll()、epoll 以及由信号驱动的 I/O（自 Linux 2.6.25 起）来监控 inotify 文件描述符。只要有事件可供读取，上述 API 便会将 inotify 文件描述符标记为可读。关乎这些编程接口的详细信息请见第 63 章。

inotify 机制属可选的 Linux 内核组件，可通过 CONFIG_INOTIFY 和 CONFIG_INOTIFY_USER 选项进行配置。

19.2 inotify API

inotify_init()系统调用可创建一新的 inotify 实例。

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

int inotify_init(void);
```

Returns file descriptor on success, or −1 on error

作为函数结果，inotify_init()会返回一个文件描述符（句柄），用于在后续操作中指代此 inotify 实例。

Linux 自内核 2.6.27 开始支持一个新的、非标准的系统调用 inotify_init1()。该系统调所执行的任务与 inotify_init()相同，但提供了一个额外的参数 flag，用于修改系统调用的行为。该参数支持的标志有二：IN_CLOEXEC 标志会使内核针对新文件描述符激活 close-on-exec 标志(FD_CLOEXEC)。引入该标志的原因正如 4.3.1 节所述 open()的 O_CLOEXEC 标志一样。IN_NONBLOCK 标志会导致内核激活底层打开文件描述的 O_NONBLOCK 标志，如此一来，未来的读操作将是非阻塞式的，省得还要额外调用 fcntl()来获得相同效果。

针对文件描述符 fd 所指代 inotify 实例的监控列表，系统调用 inotify_add_watch()既可以追加新的监控项，也可以修改现有监控项。（请参考图 19-1。）

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

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

Returns watch descriptor on success, or −1 on error

参数 pathname 标识欲创建或修改的监控项所对应的文件。调用程序必须对该文件具有读权限（调用 inotify_add_watch()时，会对文件权限做一次性检查。只要监控项继续存在，即便有

更多资源请访问稀酷客(www.ckook.com)

人更改了文件权限，使调用程序不再对文件具有读权限，调用程序依然会继续收到文件的通知消息）。

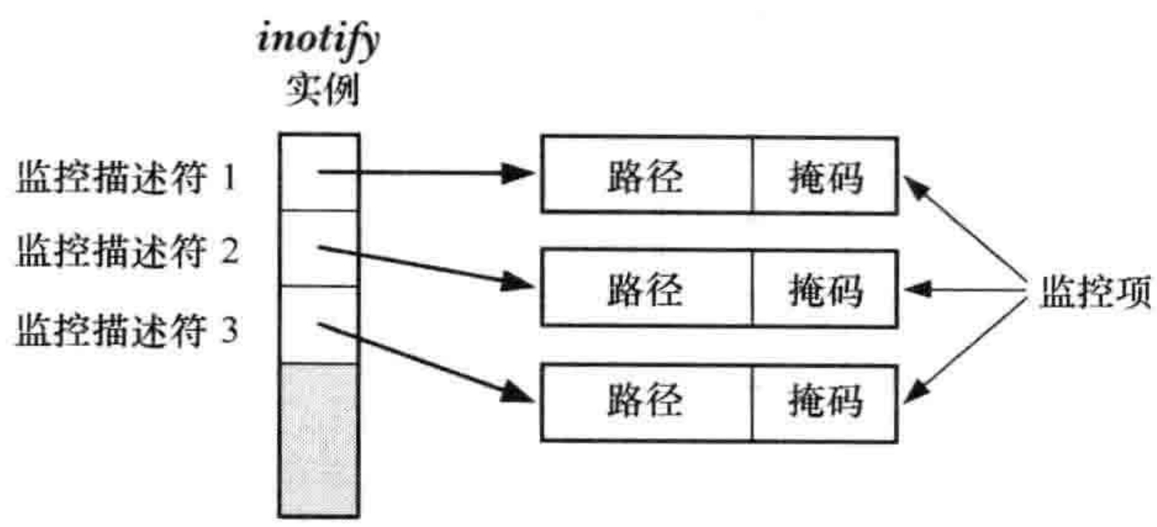

图 19-1：一个 inotify 实例及与之相关的内核数据结构

参数 mask 为一位掩码，针对 pathname 定义了意欲监控的事件。稍后会论及可在掩码中指定的各种位值。

如果先前未将 pathname 加入 fd 的监控列表，那么 inotify_add_watch()会在列表中创建一新的监控项，并返回一新的、非负监控描述符，用来在后续操作中指代此监控项。对 inotify 实例来说，该监控描述符是唯一的。

若先前已将 pathname 加入 fd 的监控列表，则 inotify_add_watch()会修改现有 pathname 监控项的掩码，并返回其监控描述符。（此描述符就是最初将 pathname 加入该监控列表的系统调用 inotify_add_watch()所返回的监控描述符。）下节在讨论 IN_MASK_ADD 标志时会就掩码的修改过程做进一步描述。

系统调用 inotify_rm_watch()会从文件描述符 fd 所指代的 inotify 实例中，删除由 wd 所定义的监控项。

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

int inotify_rm_watch(int fd, uint32_t wd);
```

Returns 0 on success, or −1 on error

参数 wd 是一监控描述符，由之前对 inotify_add_watch()的调用返回。（uint32_t 数据类型为一无符号 32 位整数。）

删除监控项会为该监控描述符生成 IN_IGNORED 事件。稍后将讨论该事件。

19.3 inotify 事件

使用 inotify_add_watch()删除或修改监控项时，位掩码参数 mask 标识了针对给定路径名（pathname）而要监控的事件。表 19-1 的“in”列列出了可在 mask 中定义的事件位。

表 19-1：inotify 事件

位　　值	In	Out	描　　述
IN_ACCESS	•	•	文件被访问（read()）
IN_ATTRIB	•	•	文件元数据改变
IN_CLOSE_WRITE	•	•	关闭为了写入而打开的文件

续表

位　　值	In	Out	描　　述
IN_CLOSE_NOWRITE	•	•	关闭以只读方式打开的文件
IN_CREATE	•	•	在受监控目录内创建了文件/目录
IN_DELETE	•	•	在受监控目录内删除文件/目录
IN_DELETE_SELF	•	•	删除受监控目录/文件本身
IN_MODIFY	•	•	文件被修改
IN_MOVE_SELF	•	•	移动受监控目录/文件本身
IN_MOVED_FROM	•	•	文件移出到受监控目录之外
IN_MOVED_TO	•	•	将文件移入受监控目录
IN_OPEN	•	•	文件被打开
IN_ALL_EVENTS	•		以上所有输出事件的统称
IN_MOVE	•		IN_MOVED_FROM | IN_MOVED_TO 事件的统称
IN_CLOSE	•		IN_CLOSE_WRITE | IN_CLOSE_NOWRITE 事件的统称
IN_DONT_FOLLOW	•		不对符号链接解引用（始于 Linux 2.6.15）
IN_MASK_ADD	•		将事件追加到 pathname 的当前监控掩码
IN_ONESHOT	•		只监控 pathname 的一个事件
IN_ONLYDIR	•		pathname 不为目录时会失败（始于 Linux 2.6.15）
IN_IGNORED		•	监控项为内核或应用程序所移除
IN_ISDIR		•	name 中所返回的文件名为路径
IN_Q_OVERFLOW		•	事件队列溢出
IN_UNMOUNT		•	包含对象的文件系统遭卸载

对于表 19-1 所列出的绝大多数位而言，顾名便可知义。以下是对一些细节的澄清。

- 当文件的元数据（比如，权限、所有权、链接计数、扩展属性、用户 ID 或组 ID 等）改变时，会发生 IN_ATTRIB 事件。
- 删除受监控对象（即，一个文件或目录）时，发生 IN_DELETE_SELF 事件。当受监控对象是一个目录，并且该目录所含文件之一遭删除时，发生 IN_DELETE 事件。
- 重命名受监控对象时，发生 IN_MOVE_SELF 事件。重命名受监控目录内的对象时，发生 IN_MOVED_FROM 和 IN_MOVED_TO 事件。其中，前一事件针对包含旧对象名的目录，后一事件则针对包含新对象名的目录。
- IN_DONT_FOLLOW、IN_MASK_ADD、IN_ONESHOT 和 IN_ONLYDIR 位并非对监控事件的定义，而是意在控制 inotify_add_watch()系统调用的行为。
- IN_DONT_FOLLOW 则规定，若 pathname 为符号链接，则不对其解引用。其作用在于令应用程序可以监控符号链接，而非符号连接所指代的文件。
- 倘若对已为同一 inotify 描述符所监控的同一路径名再次执行 inotify_add_watch()调用，那么默认情况下会用给定的 mask 掩码来替换该监控项的当前掩码。如果指定了 IN_MASK_ADD，那么则会将 mask 值与当前掩码相或。
- IN_ONESHOT 允许应用只监控 pathname 的一个事件。事件发生后，监控项会自动从

更多资源请访问稀酷客(www.ckook.com)

监控列表中消失。

- 只有当 pathname 为目录时，IN_ONLYDIR 才允许应用程序对其进行监控。如果 pathname 并非目录，那么调用 inotify_add_watch()失败，报错为 ENOTDIR。如要确保监控对象为一目录，则使用该标志可以规避竞争条件的发生。

19.4 读取 inotify 事件

将监控项在监控列表中登记后，应用程序可用 read()从 inotify 文件描述符中读取事件，以判定发生了哪些事件。若时至读取时尚未发生任何事件，read()会阻塞下去，直至有事件产生（除非对该文件描述符设置了 O_NONBLOCK 状态标志，这时若无任何事件可读，read()将立即失败，并报错 EAGAIN）。

事件发生后，每次调用 read()会返回一个缓冲区，内含一个或多个如下类型的结构（请见图 19-2）：

```
struct inotify_event {
    int      wd;          /* Watch descriptor on which event occurred */
    uint32_t mask;        /* Bits describing event that occurred */
    uint32_t cookie;      /* Cookie for related events (for rename()) */
    uint32_t len;         /* Size of 'name' field */
    char     name[];      /* Optional null-terminated filename */
};
```

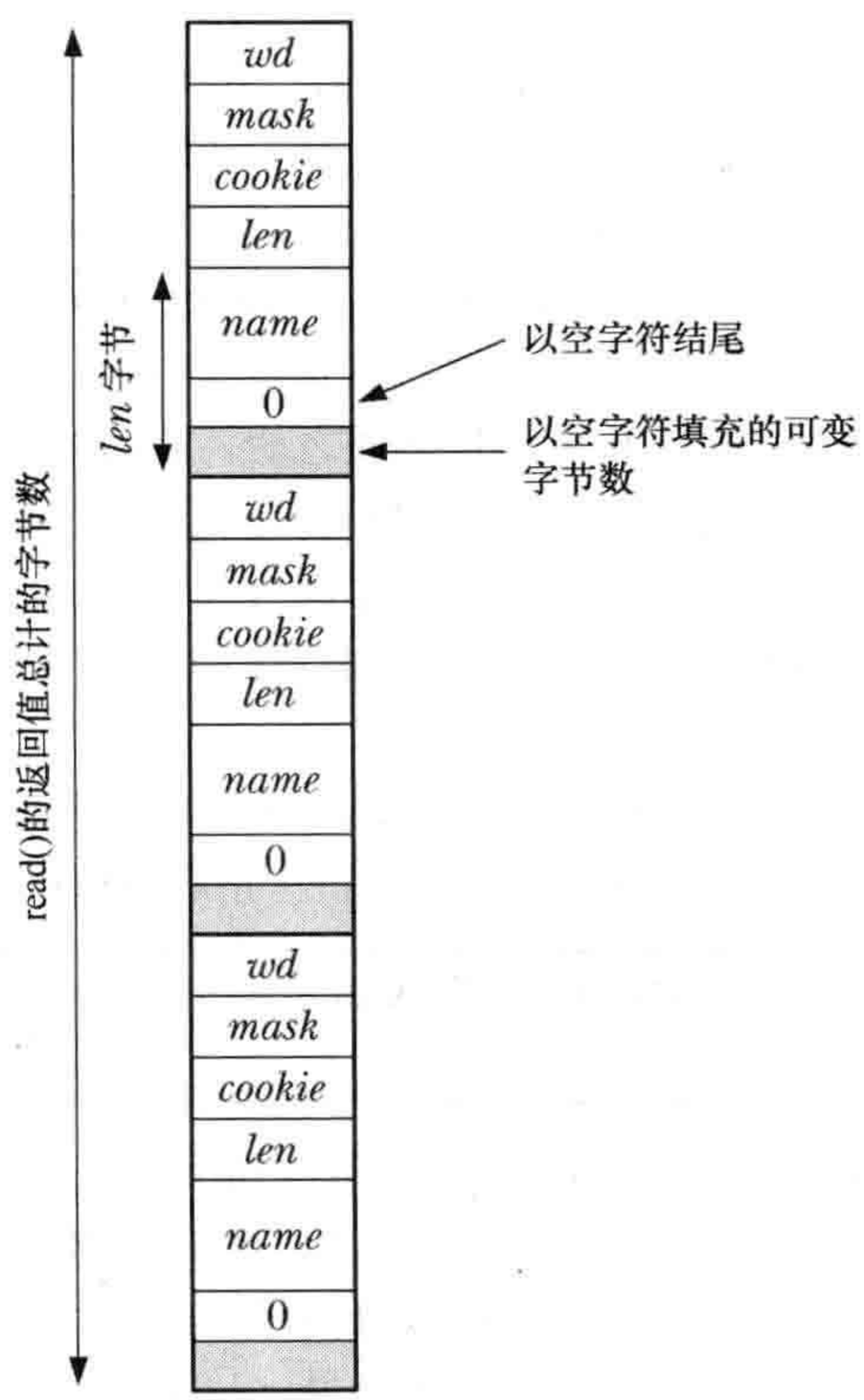

图 19-2：包含 3 个 inotify_event 结构的输入缓冲区

字段 wd 指明发生事件的是那个监控描述符。该字段值由之前对 inotify_add_watch()的调用返回。当应用程序要监控同一 inotify 文件描述符下的多个文件和目录时，字段 wd 就派上用

场。应用利用其所提供的线索来判定发生事件的特定文件或目录。（要做到这一点，应用程序必须维护专有数据结构，记录监控描述符与路径名之间的关系。）

mask 字段会返回描述该事件的位掩码。由表 19-1 所示的 Out 列展示了可出现于 mask 中的位范围。还要注意下列与特殊位相关的更多细节。

- 移除监控项时，会产生 IN_IGNORED 事件。起因可能有两个：其一，应用程序使用了 inotify_rm_watch()系统调用显式移除监控项；其二，因受监控对象被删除或其所驻留的文件系统遭卸载，致使内核隐式删除监控项。以 IN_ONESHOT 而建立的监控项因事件触发而遭自动移除时，不会产生 IN_IGNORED 事件。
- 如果事件的主体为路径，那么除去其他位以外，在 mask 中还会设置 IN_ISDIR 位。
- IN_UNMOUNT 事件会通知应用程序包含受监控对象的文件系统已遭卸载。该事件发生之后，还会产生包含 IN_IGNORED 置位的附加事件。
- 19.5 节将介绍 IN_Q_OVERFLOW，并讨论对排队 inotify 事件的限制。

使用 cookie 字段可将相关事件联系在一起。目前，只有在对文件重命名时才会用到该字段。当这种情况发生时，系统会针对待重命名文件所在目录产生 IN_MOVED_FROM 事件，然后，还会针对重命名后文件的所在目录生成 IN_MOVED_TO 事件。（若仅是在同一目录内为文件改名，系统则会针对同一目录产生上述两个事件。）两个事件的 cookie 字段值相等，故而应用程序得以将它们关联起来。

当受监控目录中有文件发生事件时，name 字段返回一个以空字符结尾的字符串，以标识该文件。若受监控对象自身有事件发生，则不使用 name 字段，将 len 字段置 0。

len 字段用于表示实际分配给 name 字段的字节数。在 read()所返回的缓冲区中，存储于 name 内的字符串结尾与下一个 inotify_event 结构的开始（请参见 19.2 节）之间，可能会有额外填充字节，故而 len 字段不可或缺。单个 inotify 事件的长度是 sizeof(struct inotify_event)+ len。

如果传递给 read()的缓冲区过小，无法容纳下一个 inotify_event 结构，那么 read()调用将以失败告终，并以 EINVAL 错误向应用程序报告这一情况。（在 2.6.21 之前版本的内核中，这种情况下 read()将返回 0。在改为报告 EINVAL 错误之后，则对编程错误的提示更为清晰。）应用程序可再次以更大的缓冲区执行 read()操作。然而，只要确保缓冲区足以容纳至少一个事件，这一问题将得以完全规避：传给 read()的缓冲区应至少为 sizeof(struct inotify_event)+ NAME_MAX + 1 字节，其中 NAME_MAX 是文件名的最大长度，此外在加上终止空字符使用的 1 个字节。

采用的缓冲区大小如大于最小值，则可自单个 read()中读取多个事件，效率极高。对 inotify 文件描述符所执行的 read()，将在已发生事件数量与缓冲区可容纳事件数量间取最小值并返回之。

针对文件描述符 fd 调用 ioctl(fd, FIONREAD, &numbytes)，会返回其所指代的 inotify 实例中的当前可读字节数。

从 inotify 文件描述符中读取的事件形成了一个有序队列。打个比方，这样一来，对文件重命名时，便可保证在 IN_MOVED_TO 事件之前能读取到 IN_MOVED_FROM 事件。

在事件队列的末尾追加一个新事件时，如果此新事件与队列当前的尾部事件拥有相同的 wd、mask、cookie 和 mask 值，那么内核会将两者合并（以避免对新事件排队）。之所以这么做，是因为很多应用程序都并不关注同一事件的反复出现，而丢弃多余的事件能降低内核维护事件队列所需的内存总量。然而，这也意味着使用 inotify 将无法可靠判定出周期性事件的发生次数或频率。

更多资源请访问稀酷客(www.ckook.com)

程序示例

虽然在前文中描述了 inotify API 的诸多细节，但实际上，该 API 使用起来却颇为简单。程序清单 19-1 展示了对 inotify 的运用。

程序清单 19-1：运用 inotify API

inotify/demo_inotify.c

```
#include <sys/inotify.h>
#include <limits.h>
#include "tlpi_hdr.h"

static void             /* Display information from inotify_event structure */
displayInotifyEvent(struct inotify_event *i)
{
    printf("    wd =%2d; ", i->wd);
    if (i->cookie > 0)
        printf("cookie =%4d; ", i->cookie);

    printf("mask = ");
    if (i->mask & IN_ACCESS)        printf("IN_ACCESS ");
    if (i->mask & IN_ATTRIB)        printf("IN_ATTRIB ");
    if (i->mask & IN_CLOSE_NOWRITE) printf("IN_CLOSE_NOWRITE ");
    if (i->mask & IN_CLOSE_WRITE)   printf("IN_CLOSE_WRITE ");
    if (i->mask & IN_CREATE)        printf("IN_CREATE ");
    if (i->mask & IN_DELETE)        printf("IN_DELETE ");
    if (i->mask & IN_DELETE_SELF)   printf("IN_DELETE_SELF ");
    if (i->mask & IN_IGNORED)       printf("IN_IGNORED ");
    if (i->mask & IN_ISDIR)         printf("IN_ISDIR ");
    if (i->mask & IN_MODIFY)        printf("IN_MODIFY ");
    if (i->mask & IN_MOVE_SELF)     printf("IN_MOVE_SELF ");
    if (i->mask & IN_MOVED_FROM)    printf("IN_MOVED_FROM ");
    if (i->mask & IN_MOVED_TO)      printf("IN_MOVED_TO ");
    if (i->mask & IN_OPEN)          printf("IN_OPEN ");
    if (i->mask & IN_Q_OVERFLOW)    printf("IN_Q_OVERFLOW ");
    if (i->mask & IN_UNMOUNT)       printf("IN_UNMOUNT ");
    printf("\n");

    if (i->len > 0)
        printf("        name = %s\n", i->name);
}
#define BUF_LEN (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

int
main(int argc, char *argv[])
{
    int inotifyFd, wd, j;
    char buf[BUF_LEN];
    ssize_t numRead;
    char *p;
    struct inotify_event *event;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname... \n", argv[0]);

①   inotifyFd = inotify_init();                /* Create inotify instance */
    if (inotifyFd == -1)
```

```
        errExit("inotify_init");

    for (j = 1; j < argc; j++) {
②       wd = inotify_add_watch(inotifyFd, argv[j], IN_ALL_EVENTS);
        if (wd == -1)
            errExit("inotify_add_watch");

        printf("Watching %s using wd %d\n", argv[j], wd);
    }

    for (;;) {                                  /* Read events forever */
③       numRead = read(inotifyFd, buf, BUF_LEN);
        if (numRead == 0)
            fatal("read() from inotify fd returned 0!");

        if (numRead == -1)
            errExit("read");

        printf("Read %ld bytes from inotify fd\n", (long) numRead);

        /* Process all of the events in buffer returned by read() */

        for (p = buf; p < buf + numRead; ) {
            event = (struct inotify_event *) p;
④           displayInotifyEvent(event);

            p += sizeof(struct inotify_event) + event->len;
        }
    }

    exit(EXIT_SUCCESS);
}
```

inotify/demo_inotify.c

程序清单 19-1 中程序将执行以下步骤。

- 使用 inotify_init()，创建 inotify 文件描述符①。
- 使用 inotify_add_watch()，将程序命令行参数中指定的每个文件加入监控项②。每个监控项都将监控所有可能发生的事件。
- 执行无限循环。
  - 从 inotify 描述符读取事件缓冲区③。
  - 调用 displayInotifyEvent()函数，以显示上述缓冲区中各 inotify_event 结构的内容④。

以下 shell 会话演示了对程序清单 19-1 所列程序的使用。首先，在后台运行该程序的实例，对两个目录进行监控。

```
$ ./demo_inotify dir1 dir2 &
[1] 5386
Watching dir1 using wd 1
Watching dir2 using wd 2
```

然后，执行某些命令，从而在两个目录中产生事件。先使用 cat(1)创建一个文件：

```
$ cat > dir1/aaa
Read 64 bytes from inotify fd
    wd = 1; mask = IN_CREATE
        name = aaa
    wd = 1; mask = IN_OPEN
        name = aaa
```


更多资源请访问稀酷客(www.ckook.com)

由后台程序所生成的上述输出表明，read()读取了包含两个事件的缓冲区。继续在该文件中执行某些输入操作，然后输入 end-of-file 字符串：

```
Hello world
Read 32 bytes from inotify fd
    wd = 1; mask = IN_MODIFY
        name = aaa
Type Control-D
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CLOSE_WRITE
        name = aaa
```

接下来，将该文件转移至另一个受监控的目录，同时对其重新命名。这会产生两个事件，一个对应于文件的源目录（监控描述符 1），另一个对应于文件的目标目录（监控描述符 2）。

```
$ mv dir1/aaa dir2/bbb
Read 64 bytes from inotify fd
    wd = 1; cookie = 548; mask = IN_MOVED_FROM
        name = aaa
    wd = 2; cookie = 548; mask = IN_MOVED_TO
        name = bbb
```

以上两个事件共享相同的 cookie 值，允许应用程序将它们联系起来。

当在其中一个受监控目录下创建子目录时，由此产生的事件掩码会置 IN_ISDIR 位，以示该事件的对象是一目录。

```
$ mkdir dir2/ddd
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CREATE IN_ISDIR
        name = ddd
```

此处，再次提醒大家，inotify 监控是非递归的。如果应用程序有意对新创建的子目录进行监控，则需进一步执行 inotify_add_watch()系统调用，并指明子目录的路径名。

最后，将其中一个受监控目录删除：

```
$ rmdir dir1
Read 32 bytes from inotify fd
    wd = 1; mask = IN_DELETE_SELF
    wd = 1; mask = IN_IGNORED
```

系统会生成最后一个事件，以通知应用程序，内核已从监控列表中删除该监控项。

19.5 队列限制和/proc 文件

对 inotify 事件做排队处理，需要消耗内核内存。正因如此，内核会对 inotify 机制的操作施以各种限制。超级用户可配置/proc/sys/fs/inotify 路径中的 3 个文件来调整这些限制：

max_queued_events

调用 inotify_init()时，使用该值来为新 inotify 实例队列中的事件数量设置上限。一旦超出这一上限，系统将生成 IN_Q_OVERFLOW 事件，并丢弃多余的事件。溢出事件的 wd 字段值为−1。

max_user_instances

对由每个真实用户 ID 创建的 inotify 实例数的限制值。

max_user_watches

对由每个真实用户 ID 创建的监控项数量的限制值。

更多资源请访问稀酷客(www.ckook.com)

这 3 个文件的典型默认值分别为 16384、128 和 8192。

19.6 监控文件的旧有系统：dnotify

Linux 还为监控文件事件提供了另一种机制。该机制名为 dnotify，问世于内核 2.4 版本，但 inotify 已令其“落伍”。相较于 inotify，dnotify 存在如下局限性。

- dnotify 机制通过向应用程序发送信号来通告事件。使用信号作为通告机制，会使应用程序的设计复杂化（请参见 22.12 节）。这也使得在函数库中使用 dnotify 变得困难，因为调用该函数的程序可能会改变对通告信号的处置（disposition）。而 inotify 机制则不使用信号。
- dnotify 的监控单元为目录。只要对该目录下的任一文件执行了任何操作，系统都会通知应用程序。相形之下，inotify 的监控对象则既可以是单个文件，也能是目录。
- 为监控目录，dnotify 需要应用程序为该目录打开文件描述符。使用文件描述符会导致两个问题。其一，由于程序处于运行中，将无法卸载包含此目录的文件系统。其二，因为每个目录都需要一个文件描述符，所以应用程序最终可能会消耗大量文件描述符。而 inotify 不使用文件描述符，故而可以避免上述问题。
- 与 inotify 相比，由 dnotify 提供的与文件事件相关的信息不够精确。当位于受监控目录下的文件发生改变时，dnotify 只会通知有事件发生，但不会说明事件具体涉及了哪个文件。因此，应用程序必须通过缓存目录内容来进行判断。此外，针对已发生事件的类型，inotify 所提供的信息也比 dnotify 更详细。
- 在某些情况下，dnotify 不支持可靠的文件事件通告机制。

在 fcntl(2)手册页中对 F_NOTIFY 操作的描述部分，以及内核源码 Documentation/dnotify.txt 中，都可找到有关 dnotify 的更多信息。

19.7 总结

当一组受监控的文件或目录有事件发生（对文件的打开、关闭、创建、删除、修改以及重命名等操作）时，Linux 专有的 inotify 机制可让应用程序获得通知。inotify 机制取代了较老的 dnotify 机制。

19.8 练习

19-1. 编写一个程序，针对其命令行参数所指定的目录，记录所有的文件创建、删除和改名操作。该程序应能够监控指定目录下所有子目录中的事件。获得所有子目录的列表需使用 nftw()（参见 18.9 节）。当在目录树下添加或删除了子目录时，受监控的子目录集合应能保持同步更新。

更多资源请访问稀酷客(www.ckook.com)

第 20 章

信号：基本概念

本章和接下来的两章将讨论信号。虽然基本概念较为简单，但因为要涵盖大量细节，所以篇幅较长。

本章包括以下主题。

- 各种不同信号及其用途。
- 内核可能为进程产生信号的环境，以及某一进程向另一进程发送信号所使用的系统调用。
- 进程在默认情况下对信号的响应方式，以及进程改变对信号响应方式的手段，特别是借助于信号处理器程序的手段，即程序收到信号时去自动调用的函数，由程序员定义。
- 使用进程信号掩码来阻塞信号，以及等待信号的相关概念。
- 如何暂停进程的执行，并等待信号的到达。

20.1 概念和概述

信号是事件发生时对进程的通知机制。有时也称之为软件中断。信号与硬件中断的相似之处在于打断了程序执行的正常流程，大多数情况下，无法预测信号到达的精确时间。

一个（具有合适权限的）进程能够向另一进程发送信号。信号的这一用法可作为一种同步技术，甚至是进程间通信（IPC）的原始形式。进程也可以向自身发送信号。然而，发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下。

- 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。硬件异常的例子包括执行一条异常的机器语言指令，诸如，被 0 除，或者引用了无法访问的内存区域。
- 用户键入了能够产生信号的终端特殊字符。其中包括中断字符（通常是 Control-C)、暂停字符（通常是 Control-Z）。
- 发生了软件事件。例如，针对文件描述符的输出变为有效，调整了终端窗口大小，定时器到期，进程执行的 CPU 时间超限，或者该进程的某个子进程退出。

针对每个信号，都定义了一个唯一的（小）整数，从 1 开始顺序展开。<signal.h>以 SIGxxxx 形式的符号名对这些整数做了定义。由于每个信号的实际编号随系统不同而不同，所以在程序中

更多资源请访问稀酷客(www.ckook.com)

总是使用这些符号名。例如，当用户键入中断字符时，将传递给进程 SIGINT 信号（信号编号为 2）。

信号分为两大类。第一组用于内核向进程通知事件，构成所谓传统或者标准信号。Linux 中标准信号的编号范围为 1～31。本章将描述这些标准信号。另一组信号由实时信号构成，其与标准信号的差异将在 22.8 节中描述。

信号因某些事件而产生。信号产生后，会于稍后被传递给某一进程，而进程也会采取某些措施来响应信号。在产生和到达期间，信号处于等待（pending）状态。

通常，一旦（内核）接下来要调度该进程运行，等待信号会马上送达，或者如果进程正在运行，则会立即传递信号（例如，进程向自身发送信号）。然而，有时需要确保一段代码不为传递来的信号所中断。为了做到这一点，可以将信号添加到进程的信号掩码中——目前会阻塞该组信号的到达。如果所产生的信号属于阻塞之列，那么信号将保持等待状态，直至稍后对其解除阻塞（从信号掩码中移除）。进程可使用各种系统调用对其信号掩码添加和移除信号。

信号到达后，进程视具体信号执行如下默认操作之一。

- 忽略信号：也就是说，内核将信号丢弃，信号对进程没有产生任何影响（进程永远都不知道曾经出现过该信号）。
- 终止（杀死）进程：这有时是指进程异常终止，而不是进程因调用 exit()而发生的正常终止。
- 产生核心转储文件，同时进程终止：核心转储文件包含对进程虚拟内存的镜像，可将其加载到调试器中以检查进程终止时的状态。
- 停止进程：暂停进程的执行。
- 于之前暂停后再度恢复进程的执行。

除了根据特定信号而采取默认行为之外，程序也能改变信号到达时的响应行为。也将此称之为对信号的处置（disposition）设置。程序可以将对信号的处置设置为如下之一。

- 采取默认行为。这适用于撤销之前对信号处置的修改、恢复其默认处置的场景。
- 忽略信号。这适用于默认行为为终止进程的信号。
- 执行信号处理器程序。

信号处理器程序是由程序员编写的函数，用于为响应传递来的信号而执行适当任务。例如，shell 为 SIGINT 信号（由中断字符串 Control-C 产生）提供了一个处理器程序，令其停止当前正在执行的工作，并将控制返回到（shell 的）主输入循环，并再次向用户呈现 shell 提示符。通知内核应当去调用某一处理器程序的行为，通常称之为安装或者建立信号处理器程序。调用信号处理器程序以响应传递来的信号，则称之为信号已处理（handled），或者已捕获（caught）。

请注意，无法将信号处置设置为终止进程或者转储核心（除非这是对信号的默认处置）。效果最为近似的是为信号安装一个处理器程序，并于其中调用 exit()或者 abort()。abort()函数（21.2.2 节）为进程产生一个 SIGABRT 信号，该信号将引发进程转储核心文件并终止。

Linux 特有的/proc/PID/status 文件包含有各种位掩码字段，通过检查这些掩码可以确定进程对信号的处理。位掩码以十六进制数形式显示，最低有效位代表信号 1，相临的左边一位代表信号 2，以此类推。这些字段分别为 SigPnd（基于线程的等待信号）、ShdPnd（进程级等待信号，始于 Linux 2.6）、SigBlk（阻塞信号）、SigIgn（忽略信号）和 SigCgt（捕获信号）。（33.2 节阐述了多线程进程对信号的处理，这将有助于澄清 SigPnd 与 ShdPnd 之间的差异。）使用 ps(1)命令的各种选项也能获得相同信息。

更多资源请访问稀酷客(www.ckook.com)

信号在 UNIX 实现中出现很早，诞生之后又历经变革。在早期实现中，信号在特定场景下有可能会丢失（即，没有传递到目标进程）。此外，尽管系统提供了执行关键代码时阻塞信号传递的机制，但阻塞有时也不大可靠。4.2BSD 利用所谓可靠信号解决了这些问题。（BSD 在创新上还更进一步，增加了额外信号来支持 shell 作业控制，请参考 34.7 节。）

System V 后来也为信号增加了可靠语义，但采用的模型与 BSD 无法兼容。这一不兼容性直到 POSIX.1-1990 标准出台后才得以解决。该标准针对可靠信号所采取的规范主要基于 BSD 模型。

22.7 节将就可靠和不可靠信号的细节展开讨论，22.13 节则简要说明了老版 BSD 和 System V 的信号 API。

20.2 信号类型和默认行为

此前曾提及，Linux 对标准信号的编号为 1～31。然而，Linux 于 signal(7)手册页中列出的信号名称却超出了 31 个。名称超出的原因有多种。有些名称只是其他名称的同义词，之所以定义是为了与其他 UNIX 实现保持源码兼容。其他名称虽然有定义，但却并未使用。以下列表介绍了各种信号。

SIGABRT

当进程调用 abort()函数（21.2.2 节）时，系统向进程发送该信号。默认情况下，该信号会终止进程，并产生核心转储文件。这实现了调用 abort()的预期目标，产生核心转储文件用于调试。

SIGALRM

经调用 alarm()或 setitimer()而设置的实时定时器一旦到期，内核将产生该信号。实时定时器是根据挂钟时间进行计时的（即人类对逝去时间的概念）。更多细节参见 23.1 节。

SIGBUS

产生该信号（总线错误，bus error）即表示发生了某种内存访问错误。如 49.4.3 节所述，当使用由 mmap()所创建的内存映射时，如果试图访问的地址超出了底层内存映射文件的结尾，那么将产生该错误。

SIGCHLD

当父进程的某一子进程终止（或者因为调用了 exit()，或者因为被信号杀死）时，（内核）将向父进程发送该信号。当父进程的某一子进程因收到信号而停止或恢复时，也可能会向父进程发送该信号。详情请参考 26.3 节。

SIGCLD

与 SIGCHLD 信号同义。

SIGCONT

将该信号发送给已停止的进程，进程将会恢复运行（即在之后某个时间点重新获得调度）。当接收信号的进程当前不处于停止状态时，默认情况下将忽略该信号。进程可以捕获该信号，以便在恢复运行时可以执行某些操作。关于该信号的更多细节请参考 22.2 节和 34.7 节。

SIGEMT

UNIX 系统通常用该信号来标识一个依赖于实现的硬件错误。Linux 系统仅在 Sun SPARC

更多资源请访问稀酷客(www.ckook.com)

实现中使用了该信号。后缀 EMT 源自仿真器陷阱（emulator trap），Digital PDP-11 的汇编程序助记符之一。

SIGFPE

该信号因特定类型的算术错误而产生，比如除以 0。后缀 FPE 是浮点异常的缩写，不过整型算术错误也能产生该信号。该信号于何时产生的精确细节取决于硬件架构和对 CPU 控制寄存器的设置。例如，在 x86-32 架构中，整数除以 0 总是产生 SIGFPE 信号，但是对浮点数除以 0 的处理则取决于是否启用了 FE_DIVBYZERO 异常。如果启用了该异常（使用 feenableexcept()），那么浮点数除以 0 也将产生 SIGFPE 信号，否则，将为操作数产生符合 IEEE 标准的结果（无穷大的浮点表示形式）。更多信息请参考 fenv(3)手册页和<fenv.h>文件。

SIGHUP

当终端断开（挂机）时，将发送该信号给终端控制进程。34.6 节将描述控制进程的概念以及产生 SIGHUP 信号的各种环境。SIGHUP 信号还可用于守护进程（比如，init、httpd 和 inetd）。许多守护进程会在收到 SIGHUP 信号时重新进行初始化并重读配置文件。借助于显式执行 kill 命令或者运行同等功效的程序或脚本，系统管理员可向守护进程手工发送 SIGHUP 信号来触发这些行为。

SIGILL

如果进程试图执行非法（即格式不正确）的机器语言指令，系统将向进程发送该信号。

SIGINFO

在 Linux 中，该信号名与 SIGPWR 信号名同义。在 BSD 系统中，键入 Control-T 可产生 SIGINFO 信号，用于获取前台进程组的状态信息。

SIGINT

当用户键入终端中断字符（通常为 Control-C）时，终端驱动程序将发送该信号给前台进程组。该信号的默认行为是终止进程。

SIGIO

利用 fcntl()系统调用，即可于特定类型（诸如终端和套接字）的打开文件描述符发生 I/O 事件时产生该信号。63.3 节将就此特性做进一步说明。

SIGIOT

在 Linux 中，该信号名与 SIGABRT 信号同义。在其他一些 UNIX 实现中，该信号表示发生了由实现定义的硬件错误。

SIGKILL

此信号为“必杀（sure kill）”信号，处理器程序无法将其阻塞、忽略或者捕获，故而“一击必杀”，总能终止进程。

SIGLOST

Linux 中存在该信号名，但并未加以使用。在其他一些 UNIX 实现中，如果远端 NFS 服务器在崩溃之后重新恢复，而 NFS 客户端却未能重新获得由本地进程所持有的锁，那么 NFS 客户端将向这些进程发送此信号。（NFS 规范并未对该特性进行标准化。）

SIGPIPE

当某一进程试图向管道、FIFO 或套接字写入信息时，如果这些设备并无相应的阅读进程，

更多资源请访问稀酷客(www.ckook.com)

那么系统将产生该信号。之所以如此，通常是因为阅读进程已经关闭了其作为 IPC 通道的文件描述符。更多细节请参考 44.2 节。

SIGPOLL

该信号从 System V 派生而来，与 Linux 中的 SIGIO 信号同义。

SIGPROF

由 setitimer()调用（参见 23.1 节）所设置的性能分析定时器刚一过期，内核就将产生该信号。性能分析定时器用于记录进程所使用的 CPU 时间。与虚拟定时器不同（参见下面的 SIGVTALRM 信号），性能分析定时器在对 CPU 时间计数时会将用户态与内核态都包含在内。

SIGPWR

这是电源故障信号。当系统配备有不间断电源（UPS）时，可以设置守护进程来监控电源发生故障时备用电池的剩余电量。如果电池电量行将耗尽（长时间停电之后），那么监控进程会将该信号发往 init 进程，而后者则将其解读为快速、有序关闭系统的一个请求。

SIGQUIT

当用户在键盘上键入退出字符（通常为 Control-\）时，该信号将发往前台进程组。默认情况下，该信号终止进程，并生成可用于调试的核心转储文件。进程如果陷入无限循环，或者不再响应时，使用 SIGQUIT 信号就很合适。键入 Control-\，再调用 gdb 调试器加载刚才生成的核心转储文件，接着用 backtrace 命令来获取堆栈跟踪信息，就能发现正在执行的是程序的哪部分代码。（[Matloff, 2008]描述了 gdb 的用法。）

SIGSEGV

这一信号非常常见，当应用程序对内存的引用无效时，就会产生该信号。引起对内存无效引用的原因很多，可能是因为要引用的页不存在（例如，该页位于堆和栈之间的未映射区域），或者进程试图更新只读内存（比如，程序文本段或者标记为只读的一块映射内存区域）中某一位置的内容，又或者进程企图在用户态（参见 2.1 节）去访问内核的部分内存。C 语言中引发这些事件的往往是解引用的指针里包含了错误地址（例如，未初始化的指针），或者传递了一个无效参数供函数调用。该信号的命名源于术语“段违例”。

SIGSTKFLT

signal(7)手册页中将其记载为“协处理器栈错误”，Linux 对该信号作了定义，但并未加以使用。

SIGSTOP

这是一个必停（sure stop）信号，处理器程序无法将其阻塞、忽略或者捕获，故而总是能停止进程。

SIGSYS

如果进程发起的系统调用有误，那么将产生该信号。这意味着系统将进程执行的指令视为一个系统调用陷阱（trap），但相关的系统调用编号却是无效的（参见 3.1 节）。

SIGTERM

这是用来终止进程的标准信号，也是 kill 和 killall 命令所发送的默认信号。用户有时会使用 kill-KILL 或者 kill-9 显式向进程发送 SIGKILL 信号。然而，这一做法通常是错误的。精心设计的应用程序应当为 SIGTERM 信号设置处理器程序，以便于其能够预先清除临时文件和释

更多资源请访问稀酷客(www.ckook.com)

放其他资源，从而全身而退。发送 SIGKILL 信号可以杀掉某个进程，从而绕开了 SIGTERM 信号的处理器程序。因此，总是应该首先尝试使用 SIGTERM 信号来终止进程，而把 SIGKILL 信号作为最后手段，去对付那些不响应 SIGTERM 信号的失控进程。

SIGTRAP

该信号用来实现断点调试功能以及 strace(1)命令（附录 A）所执行的跟踪系统调用功能。更多信息参见 ptrace(2)手册页。

SIGTSTP

这是作业控制的停止信号，当用户在键盘上输入挂起字符（通常是 Control-Z）时，将发送该信号给前台进程组，使其停止运行。第 34 章详细描述了进程组（作业）和作业控制，以及程序应在何时以及如何去处理该信号。该信号名源自“终端停止（terminal stop）”的术语。

SIGTTIN

在作业控制 shell 下运行时，若后台进程组试图对终端进行 read()操作，终端驱动程序则将向该进程组发送此信号。该信号默认将停止进程。

SIGTTOU

该信号的目的与 SIGTTIN 信号类似，但所针对的是后台作业的终端输出。在作业控制 shell 下运行时，如果对终端启用了 TOSTOP（终端输出停止）选项（可能是通过 stty tostop 命令），而某一后台进程组试图对终端进行 write()操作（参见 34.7.1 节），那么终端驱动程序将向该进程组发送 SIGTTOU 信号。该信号默认将停止进程。

SIGUNUSED

顾名思义，该信号没有使用。在 Linux 2.4 及其后续版本中，该信号名在很多架构中与 SIGSYS 信号同义。换言之，尽管信号名还保持向后兼容，但信号编号在这些架构中不再处于未使用状态。

SIGURG

系统发送该信号给一个进程，表示套接字上存在带外（也称作紧急）数据（参见 61.13.1 节）。

SIGUSR1

该信号和 SIGUSR2 信号供程序员自定义使用。内核绝不会为进程产生这些信号。进程可以使用这些信号来相互通知事件的发生，或是彼此同步。在早期的 UNIX 实现中，这是可供应用随意使用的仅有的两个信号。（实际上，进程间可以相互发送任何信号，但如果内核也为进程产生了同类信号，这两种情况就有可能产生混淆。）现代 UNIX 实现则提供了很多实时信号，也可用于程序员自定义的目的（参见 22.8 节）。

SIGUSR2

参见对 SIGUSR1 信号的描述。

SIGVTALRM

调用 setitimer()（参见 23.1 节）设置的虚拟定时器刚一到期，内核就会产生该信号。虚拟定时器计录的是进程在用户态所使用的 CPU 时间。

SIGWINCH

在窗口环境中，当终端窗口尺寸发生变化时（如 62.9 节所述，要么是由于用户手动调

更多资源请访问稀酷客(www.ckook.com)

整了大小，要么是因为程序调用 ioctl()对大小做了调整），会向前台进程组发送该信号。借助于为该信号安装的处理器程序，诸如 vi 和 less 之类的程序会在窗口尺寸调整后重新绘制输出。

SIGXCPU

当进程的 CPU 时间超出对应的资源限制时（参见 36.3 节对 RLIMIT_CPU 的描述），将发送此信号给进程。

SIGXFSZ

如果进程因试图增大文件（调用 write()或 truncate()）而突破对进程文件大小的资源限制（参见 36.3 节对 RLIMIT_FSIZE 的描述）时，那么将发送此信号给进程。

表 20-1 总结了 Linux 下与信号相关的一系列信息。关于此表，请注意以下几点。

- 信号编号列所示为在不同硬件架构下对信号的编号。除非另有说明，信号在所有架构中编号相同。信号编号在架构上的差异会在括号中予以说明，所涉及的架构包括 Sun SPARC、SPARC64 (S)、HP/Compaq/Digital Alpha (A)、MIPS (M)和 HP PA-RISC (P)。此列中的 undef 表示此符号在所示架构中未定义。
- SUSv3 列则表示 SUSv3 是否定义了该信号。
- 默认列显示了信号的默认行为。term 表示信号终止进程，core 表示进程产生核心转储文件并退出，ignore 表示忽略该信号，stop 表示信号停止了进程，cont 表示信号恢复了一个已停止的进程。

某些前面列出的信号并未见诸于表 20-1，如 SIGCLD（SIGCHLD 信号的同义词）、SIGINFO（未使用）、SIGIOT（SIGABRT 信号的同义词）、SIGLOST（未使用）和 SIGUNUSED（在许多架构中是 SIGSYS 信号的同义词）。

表 20-1：Linux 信号

名　称	信　号　值	描　述	SUSv3	默认
SIGABRT	6	中止进程	●	core
SIGALRM	14	实时定时器过期	●	term
SIGBUS	7 (SAMP=10)	内存访问错误	●	core
SIGCHLD	17 (SA=20, MP=18)	终止或者停止子进程	●	ignore
SIGCONT	18 (SA=19, M=25, P=26)	若停止则继续	●	cont
SIGEMT	undef (SAMP=7)	硬件错误		term
SIGFPE	8	算术异常	●	core
SIGHUP	1	挂起	●	term
SIGILL	4	非法指令	●	core
SIGINT	2	终端中断	●	term
SIGIO /	29 (SA=23, MP=22)	I/O 时可能产生	●	term
SIGPOLL				

更多资源请访问稀酷客(www.ckook.com)

续表

名　称	信　号　值	描　述	SUSv3	默认
SIGKILL	9	必杀（确保杀死）	●	term
SIGPIPE	13	管道断开	●	term
SIGPROF	27 (M=29, P=21)	性能分析定时器过期	●	term
SIGPWR	30 (SA=29, MP=19)	电量行将耗尽		term
SIGQUIT	3	终端退出	●	core
SIGSEGV	11	无效的内存引用	●	core
SIGSTKFLT	16 (SAM=undef, P=36)	协处理器栈错误		term
SIGSTOP	19 (SA=17, M=23, P=24)	确保停止	●	stop
SIGSYS	31 (SAMP=12)	无效的系统调用	●	core
SIGTERM	15	终止进程	●	term
SIGTRAP	5	跟踪/断点陷阱	●	core
SIGTSTP	20 (SA=18, M=24, P=25)	终端停止	●	stop
SIGTTIN	21 (M=26, P=27)	BG[1]从终端读取	●	stop
SIGTTOU	22 (M=27, P=28)	BG 向终端写	●	stop
SIGURG	23 (SA=16, M=21, P=29)	套接字上的紧急数据	●	ignore
SIGUSR1	10 (SA=30, MP=16)	用户自定义信号 1	●	term
SIGUSR2	12 (SA=31, MP=17)	用户自定义信号 2	●	term
SIGVTALRM	26 (M=28, P=20)	虚拟定时器过期	●	term
SIGWINCH	28 (M=20, P=23)	终端窗口尺寸发生变化		ignore
SIGXCPU	24 (M=30, P=33)	突破对 CPU 时间的限制	●	core
SIGXFSZ	25 (M=31, P=34)	突破对文件大小的限制	●	core

针对表 20-1 中某些信号的默认行为，要注意以下几点。

- 在 Linux 2.2 中，信号 SIGXCPU、SIGXFSZ、SIGSYS 和 SIGBUS 的默认行为是终止进程，但不会产生核心转储文件。自内核 2.4 以后，Linux 实现满足了 SUSv3 的要求，这些信号不但会引发进程终止，也将生成核心转储文件。在其他几个 UNIX 实现中，对信号 SIGXCPU 和 SIGXFSZ 的处理方式与 Linux 2.2 相同。
- 在其他的 UNIX 实现中，对 SIGPWR 信号的默认行为通常是将其忽略。
- 几个 UNIX 实现（特别是 BSD 衍生系统）默认情况下将忽略 SIGIO 信号。
- 虽然 SIGEMT 信号尚未获得任何标准的接纳，但却得到大多数 UNIX 实现的支持。然而，在其他实现中，该信号通常会导致进程终止并产生核心转储文件。
- SUSv1 将 SIGURG 信号的默认行为定义为终止进程，这也是一些较老 UNIX 实现的默认做法。而 SUSv2 则采用了现行规范（将其忽略）。

1 译者注：即后台进程组。

更多资源请访问稀酷客(www.ckook.com)

20.3 改变信号处置：signal()

UNIX 系统提供了两种方法来改变信号处置：signal()和 sigaction()。本节描述的 signal()系统调用，是设置信号处置的原始 API，所提供的接口比 sigaction()简单。另一方面，sigaction()提供了 signal()所不具备的功能。进一步而言，signal()的行为在不同 UNIX 实现间存在差异（22.7 节），这也意味着对可移植性有所追求的程序绝不能使用此调用来建立信号处理器函数。故此，sigaction()是建立信号处理器的首选 API（强力推荐）。自 20.13 节介绍了 sigaction()调用的用法之后，本书示例将一律采用该调用来建立信号处理器程序。

signal()函数虽然记录在 Linux 手册页的第 2 部分，但实际却被实现为一个基于 sigaction()系统调用的 glibc 库函数。

```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
```

Returns previous signal disposition on success, or SIG_ERR on error

这里需要对 signal()函数的原型做一些解释。第一个参数 sig，标识希望修改处置的信号编号，第二个参数 handler，则标识信号抵达时所调用函数的地址。该函数无返回值（void），并接收一个整型参数。因此，信号处理器函数一般具有以下形式：

```
void
handler(int sig)
{
    /* Code for the handler */
}
```

20.4 节将描述处理器函数中 sig 参数的目的。

signal()的返回值是之前的信号处置。像 handler 参数一样，这是一枚指针，所指向的是带有一个整型参数且无返回值的函数。换言之，编写如下代码，可以暂时为信号建立一个处理器函数，然后再将信号处置重置为其本来面目：

```
void (*oldHandler)(int);

oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
    errExit("signal");

/* Do something else here. During this time, if SIGINT is
   delivered, newHandler will be used to handle the signal. */

if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("signal");
```

使用 signal()，将无法在不改变信号处置的同时，还能获取到当前的信号处置。要想做到这一点，必须使用 sigaction()。

针对信号处理器函数指针做如下类型定义，将有助于理解 signal()的原型：

```
typedef void (*sighandler_t)(int);
```


更多资源请访问稀酷客(www.ckook.com)

signal()原型可以改写成如下形式：

```
sighandler_t signal(int sig, sighandler_t handler);
```

> 如果定义了_GNU_SOURCE 特性测试宏，那么 glibc 将在<signal.h>头文件中暴露非标准的 sighandler_t 数据类型。

在为 signal()指定 handler 参数时，可以以如下值来代替函数地址：

SIG_DFL

将信号处置重置为默认值（表 20-1）。这适用于将之前 signal()调用所改变的信号处置还原。

SIG_IGN

忽略该信号。如果信号专为此进程而生，那么内核会默默将其丢弃。进程甚至从未知道曾经产生了该信号。

调用 signal()成功将返回先前的信号处置，有可能是先前安装的处理器函数地址，也可能是常量 SIG_DFL 和 SIG_IGN 之一。如果调用失败，signal()将返回 SIG_ERR。

20.4 信号处理器简介

信号处理器程序（也称为信号捕捉器）是当指定信号传递给进程时将会调用的一个函数。本节描述了信号处理器的基本原理，而第 21 章将继续做详细介绍。

调用信号处理器程序，可能会随时打断主程序流程；内核代表进程来调用处理器程序，当处理器返回时，主程序会在处理器打断的位置恢复执行。这一工作序列可用图 20-1 来加以说明。

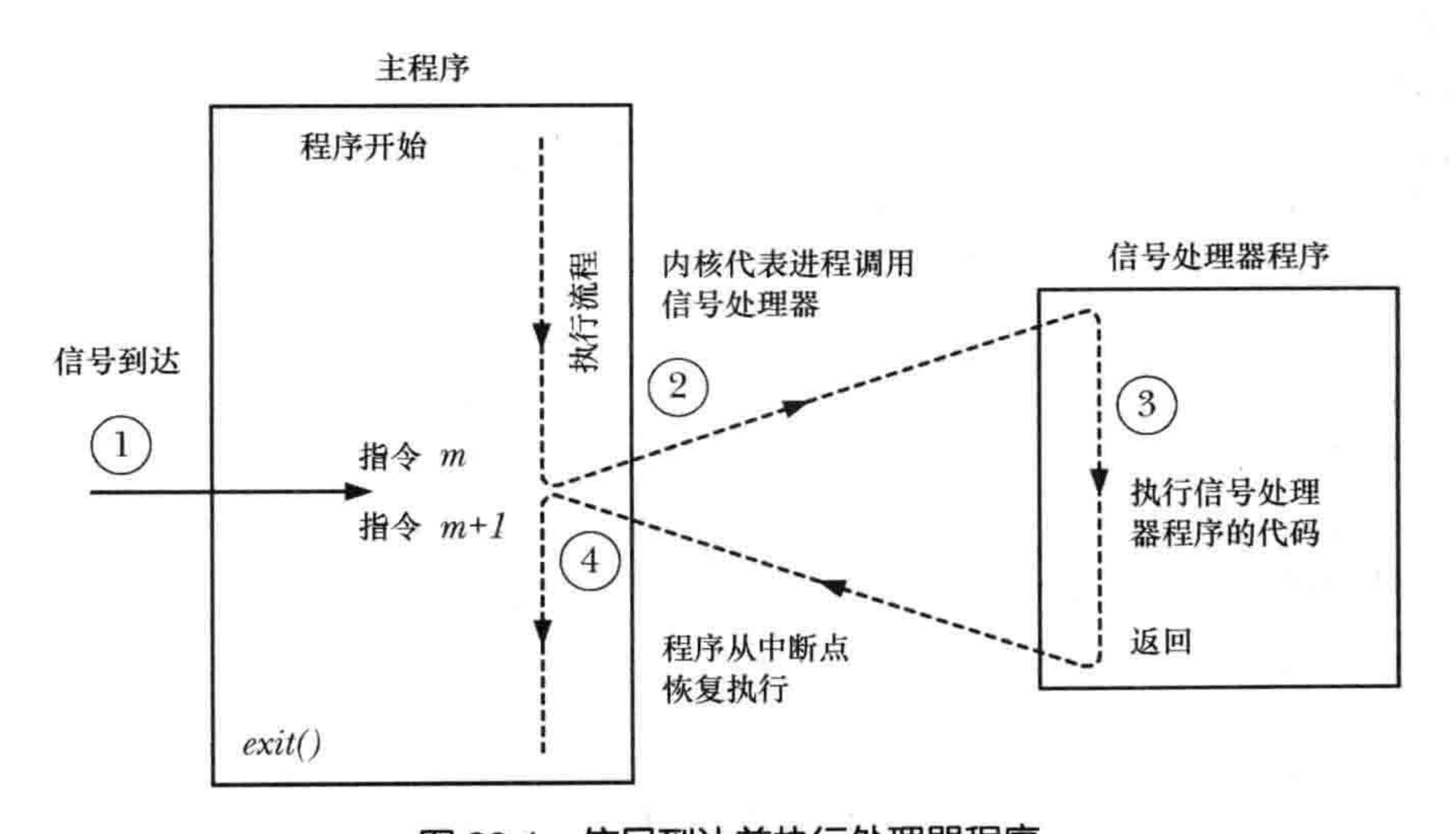

图 20-1：信号到达并执行处理器程序

虽然信号处理器程序几乎可以为所欲为，但一般而言，设计应力求简单。21.1 节将对这一点展开论述。

程序清单 20-1：为 SIGINT 信号安装一个处理器程序

signals/ouch.c

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

static void
sigHandler(int sig)
{
    printf("Ouch!\n");                  /* UNSAFE (see Section 21.1.2) */
}

int
main(int argc, char *argv[])
{
    int j;

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3);                       /* Loop slowly... */
    }
}
```

signals/ouch.c

程序清单 20-1 所示为一个简单的信号处理器函数，由主程序为 SIGINT 信号而建立。当键入中断字符（通常为 Control-C）时，终端驱动程序将产生该信号。处理器只是简单打印一条消息，随即返回。

主程序会持续循环。每次迭代，程序都将递增计数器值并将其打印出来，然后休眠几秒钟。（为了按这种方式休眠，程序使用了 sleep()函数，该函数会令调用者处于暂停状态，持续时间则由指定的秒数决定。该函数将在 23.4.1 节中进行描述。）

运行程序清单 20-1 中程序的结果如下：

```
$ ./ouch
0                       Main program loops, displaying successive integers
Type Control-C
Ouch!                   Signal handler is executed, and returns
1                       Control has returned to main program
2
Type Control-C again
Ouch!
3
Type Control-\ (the terminal quit character)
Quit (core dumped)
```

内核在调用信号处理器程序时，会将引发调用的信号编号作为一个整型参数传递给处理器函数。（就是程序清单 20-1 中处理器函数的 sig 参数）。如果信号处理器程序只捕获一种类型的信号，那么这个参数几乎无用。然而，如果安装相同的处理器来捕获不同类型的信号，那么就可以利用此参数来判定引发对处理器调用的是何种信号。

程序清单 20-2 中程序展示了这一思路，为 SIGINT 和 SIGQUIT 信号建立了同一处理器程序。（当键入终端退出字符时，通常为 Control-\，终端驱动程序将产生 SIGQUIT 信号。）处理器程序代码通过检查 sig 参数来区分这两种信号，并为每种信号采取不同措施。main()函数则

更多资源请访问稀酷客(www.ckook.com)

使用 pause()函数（参见 20.14 节的描述）来阻塞进程，直至捕获到信号。

如下 shell 会话日志演示了对该程序的使用：

```
$ ./intquit
Type Control-C
Caught SIGINT (1)
Type Control-C again
Caught SIGINT (2)
and again
Caught SIGINT (3)
Type Control-\
Caught SIGQUIT - that's all folks!
```

程序清单 20-1 和程序清单 20-2 都在信号处理器程序中使用了 printf()函数来显示消息。现实世界的应用程序一般绝不会在信号处理器程序中使用 stdio 函数，21.1.2 节将就其原因进行讨论。然而，本书各种示例仍然会在信号处理器程序中调用 printf()函数，作为观察处理器程序调用的一种简单手段。

程序清单 20-2：为两个不同信号建立同一处理器函数

signals/intquit.c

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

static void
sigHandler(int sig)
{
    static int count = 0;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), exit(); see Section 21.1.2) */

    if (sig == SIGINT) {
        count++;
        printf("Caught SIGINT (%d)\n", count);
        return;                 /* Resume execution at point of interruption */
    }

    /* Must be SIGQUIT - print a message and terminate the process */

    printf("Caught SIGQUIT - that's all folks!\n");
    exit(EXIT_SUCCESS);
}

int
main(int argc, char *argv[])
{
    /* Establish same handler for SIGINT and SIGQUIT */

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");
    if (signal(SIGQUIT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (;;)                    /* Loop forever, waiting for signals */
        pause();                /* Block until a signal is caught */
}
```

signals/intquit.c

更多资源请访问稀酷客(www.ckook.com)

20.5 发送信号：kill()

与 shell 的 kill 命令相类似，一个进程能够使用 kill()系统调用向另一进程发送信号。（之所以选择 kill 作为术语，是因为早期 UNIX 实现中大多数信号的默认行为是终止进程。）

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Returns 0 on success, or −1 on error

pid 参数标识一个或多个目标进程，而 sig 则指定了要发送的信号。如何解释 pid，要视以下 4 种情况而定。

- 如果 pid 大于 0，那么会发送信号给由 pid 指定的进程。
- 如果 pid 等于 0，那么会发送信号给与调用进程同组的每个进程，包括调用进程自身。（SUSv3 声明，除去“一组未予明确的系统进程”[1]之外，应将信号发送给同一进程组中的所有进程，且这一排除条件同样适用于余下的两种情况。）
- 如果 pid 小于−1，那么会向组 ID 等于该 pid 绝对值的进程组内所有下属进程发送信号。向一个进程组的所有进程发送信号在 shell 作业控制中有特殊用途（参见 34.7 节）。
- 如果 pid 等于−1，那么信号的发送范围是：调用进程有权将信号发往的每个目标进程，除去 init（进程 ID 为 1）和调用进程自身。如果特权级进程发起这一调用，那么会发送信号给系统中的所有进程，上述两个进程除外。显而易见，有时也将这种信号发送方式称之为广播信号。（SUSv3 并未要求将调用进程排除在信号的接收范围之外，Linux 此处所遵循的是 BSD 系统的语义。）

如果并无进程与指定的 pid 相匹配，那么 kill()调用失败，同时将 errno 置为 ESRCH（“查无此进程”）。

进程要发送信号给另一进程，还需要适当的权限，其权限规则如下。

- 特权级（CAP_KILL）进程可以向任何进程发送信号。
- 以 root 用户和组运行的 init 进程（进程号为 1），是一种特例，仅能接收已安装了处理器函数的信号。这可以防止系统管理员意外杀死 init 进程——这一系统运作的基石。
- 如图 20-2 所示，如果发送者的实际或有效用户 ID 匹配于接受者的实际用户 ID 或者保存设置用户 ID(saved set-user-id)，那么非特权进程也可以向另一进程发送信号。利用这一规则，用户可以向由他们启动的 set-user-ID 程序发送信号，而无需考虑目标进程有效用户 ID 的当前设置。将目标进程有效用户 ID 排除在检查范围之外，这一举措的辅助作用在于防止用户某甲向用户某乙的进程发送信号，而该进程正在执行的 set-user-ID 程序又属于用户某甲。（SUSv3 要求强制执行图 20-2 所示的规则，但如 kill(2)手册页所述，Linux 内核在 2.0 版本之前所遵循的规则略有不同。）
- SIGCONT 信号需要特殊处理。无论对用户 ID 的检查结果如何，非特权进程可以向同一会话中的任何其他进程发送这一信号。利用这一规则，运行作业控制的 shell 可以重启已停止的作业（进程组），即使作业进程已经修改了它们的用户 ID。（亦即，使用

1 译者注：参考 APUEv2，其实就是由系统实现决定的意思，再次鄙视 SUS 的学究笔法。

更多资源请访问稀酷客(www.ckook.com)

9.7 节所述系统调用来改变其凭据，进而成为特权级进程。）

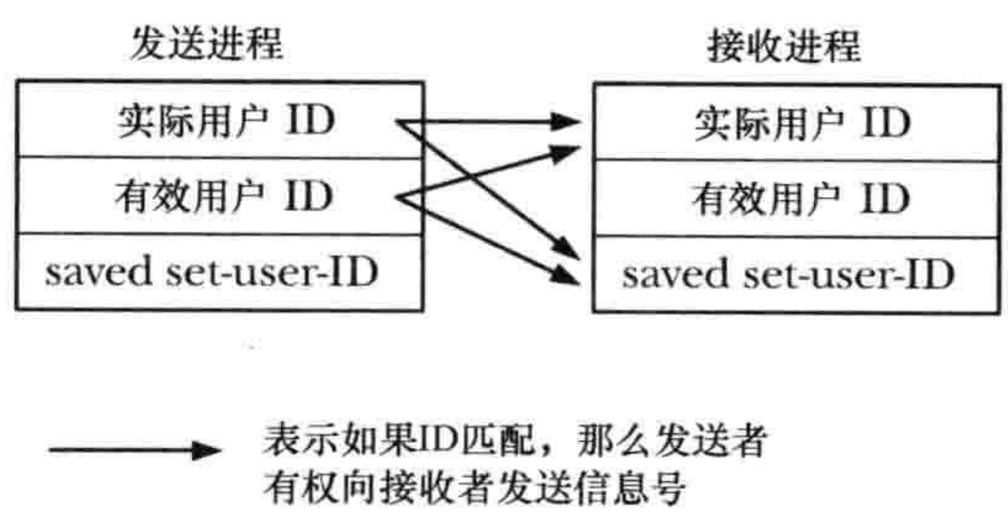

图 20-2：非特权进程发送信号所需的权限

如果进程无权发送信号给所请求的 pid，那么 kill()调用将失败，且将 errno 置为 EPERM。若 pid 所指为一系列进程（即 pid 是负值）时，只要可以向其中之一发送信号，则 kill()调用成功。

程序清单 20-3 中展示了 kill()的用法。

20.6 检查进程的存在

kill()系统调用还有另一重功用。若将参数 sig 指定为 0（即所谓空信号），则无信号发送。相反，kill()仅会去执行错误检查，查看是否可以向目标进程发送信号。从另一角度来看，这意味着，可以使用空信号来检测具有特定进程 ID 的进程是否存在。若发送空信号失败，且 errno 为 ESRCH，则表明目标进程不存在。如果调用失败，且 errno 为 EPERM（表示进程存在，但无权向目标进程发送信号）或者调用成功（有权向进程发送信号），那么就表示进程存在。

验证一个特定进程 ID 的存在并不能保证特定程序仍在运行。因为内核会随着进程的生灭而循环使用进程 ID。而一段时间之后，同一进程 ID 所指恐怕是另一进程了。此外，特定进程 ID 可能存在，但是一个僵尸（亦即，进程已死，但其父进程尚未执行 wait()来获取其终止状态，如 26.2 节所述）。

还可使用各种其他技术来检查某一特定进程是否正在运行，其中包括如下技术。

- wait()系统调用：第 26 章将描述这些调用。这些调用仅用于监控调用者的子进程。
- 信号量和排他文件锁：如果进程持续持有某一信号量或文件锁，并且一直处于被监控状态，那么如能获取到信号量或锁时，即表明该进程已经终止。第 47 章和第 53 章将描述信号量，第 55 章将描述文件锁。
- 诸如管道和 FIFO 之类的 IPC 通道：可对监控目标进程进行设置，令其在自身生命周期内持有对通道进行写操作的打开文件描述符。同时，令监控进程持有针对通道进行读操作的打开文件描述符，且当通道写入端关闭时（遭遇文件结束符），即可获知监控目标进程已经终止。监控进程对此情况的判定，既可借助于对自身文件描述符的读取，也可采用第 63 章所述的描述符监控技术之一。
- /proc/PID 接口：例如，如果进程 ID 为 12345 的进程存在，那么目录/proc/12345 将存在，可以发起诸如 stat()之类的调用来进行检查。

除去最后一项之外，循环使用进程 ID 不会影响上述所有技术。

程序清单 20-3 展示了 kill()的用法。该程序接受两个命令行参数，分别为信号编号和进程 ID，并使用 kill()将该信号发送给指定进程。如果指定了信号 0（空信号），那么程序将报告目

更多资源请访问稀酷客(www.ckook.com)

标进程是否存在。

20.7 发送信号的其他方式：raise()和 killpg()

有时，进程需要向自身发送信号（34.7.3 节就有此一例）。raise()函数就执行了这一任务。

```
#include <signal.h>

int raise(int sig);
```

Returns 0 on success, or nonzero on error

在单线程程序中，调用 raise()相当于对 kill()的如下调用：

```
kill(getpid(), sig);
```

支持线程的系统会将 raise(sig)实现为：

```
pthread_kill(pthread_self(), sig)
```

33.2.3 节描述了 pthread_kill()函数，但目前仅需了解一点就已足够，该实现意味着将信号传递给调用 raise()的特定线程。相比之下，kill(getpid(), sig)调用会发送一个信号给调用进程，并可将该信号传递给该进程的任一线程。

raise()函数起源于 C89。C 语言标准不包含诸如进程 ID 之类的操作系统细节，raise()函数之所以得以定义，是因为该函数不需要引用进程 ID。

当进程使用 raise()（或者 kill()）向自身发送信号时，信号将立即传递（即，在 raise()返回调用者之前）。

注意，raise()出错将返回非 0 值（不一定为-1）。调用 raise()唯一可能发生的错误为 EINVAL，即 sig 无效。因此，在任何指定了某一 SIGxxxx 常量的位置，都未检查该函数的返回状态。

程序清单 20-3：使用 kill()系统调用

signals/t_kill.c

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

int
main(int argc, char *argv[])
{
    int s, sig;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sig-num pid\n", argv[0]);

    sig = getInt(argv[2], 0, "sig-num");

    s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");

    } else {                    /* Null signal: process existence check */
        if (s == 0) {
```

```
        printf("Process exists and we can send it a signal\n");
    } else {
        if (errno == EPERM)
            printf("Process exists, but we don't have "
                   "permission to send it a signal\n");
        else if (errno == ESRCH)
            printf("Process does not exist\n");
        else
            errExit("kill");
    }
}

exit(EXIT_SUCCESS);
}
```

signals/t_kill.c

killpg()函数向某一进程组的所有成员发送一个信号。

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

Returns 0 on success, or −1 on error

killpg()调用相当于对 kill()的如下调用：

```
kill(-pgrp, sig);
```

如果指定 pgrp 的值为 0，那么会向调用者所属进程组的所有进程发送此信号。SUSv3 对此未作规范，但大多数 UNIX 实现对该情况的处理方式与 Linux 相同。

20.8 显示信号描述

每个信号都有一串与之相关的可打印说明。这些描述位于数组 sys_siglist 中。例如，可以用 sys_siglist[SIGPIPE]来获取对 SIGPIPE 信号（管道断开）的描述。然而，较之于直接引用 sys_siglist 数组，还是推荐使用 strsignal()函数。

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

extern const char *const sys_siglist[];

#define _GNU_SOURCE
#include <string.h>

char *strsignal(int sig);
```

Returns pointer to signal description string

strsignal()函数对 sig 参数进行边界检查，然后返回一枚指针，指向针对该信号的可打印描述字符串，或者是当信号编号无效时指向错误字符串。（在其他一些 UNIX 实现中，strsignal()函数会在 sig 无效时返回空值。）

除去边界检查之外，strsignal()函数较之于直接引用 sys_siglist 数组的另一优势是对本地

更多资源请访问稀酷客(www.ckook.com)

（locale）设置敏感（10.4 节），所以显示信号描述时会使用本地语言。

程序清单 20-4 中所示为使用 strsignal()的例子之一。

psignal()函数（在标准错误设备上）所示为 msg 参数所给定的字符串，后面跟有一个冒号，随后是对应于 sig 的信号描述。和 strsignal()一样，psignal()函数也对本地设置敏感。

```
#include <signal.h>

void psignal(int sig, const char *msg);
```

尽管 SUSv3 并未将 psignal()、strsignal()和 sys_siglist 纳入标准，但还是有许多 UNIX 实现支持它们。（SUSv4 中加入了对 psignal()和 strsignal()的规范。）

20.9 信号集

许多信号相关的系统调用都需要能表示一组不同的信号。例如，sigaction()和 sigprocmask()允许程序指定一组将由进程阻塞的信号，而 sigpending()则返回一组目前正在等待送达给一进程的信号。（稍后将描述这些系统调用。）

多个信号可使用一个称之为信号集的数据结构来表示，其系统数据类型为 sigset_t.。SUSv3 规定了一系列函数来操纵信号集，现在将描述这些函数。

像在大多数 UNIX 实现中一样，sigset_t 数据类型在 Linux 中是一个位掩码。然而，SUSv3 对此并无要求。使用其他一些类型的结构来表示信号集也是有可能的。SUSv3 仅要求可对 sigset_t 类型赋值即可。因此，必须使用某些标量类型（比如一个整数）或者一个 C 语言结构（也许包含了一个整型数组）来实现该类型。

sigemptyset()函数初始化一个未包含任何成员的信号集。sigfillset()函数则初始化一个信号集，使其包含所有信号（包括所有实时信号）。

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

Both return 0 on success, or −1 on error

必须使用 sigemptyset()或者 sigfillset()来初始化信号集。这是因为 C 语言不会对自动变量进行初始化，并且，借助于将静态变量初始化为 0 的机制来表示空信号集的作法在可移植性上存在问题，因为有可能使用位掩码之外的结构来实现信号集。（出于同一原因，为将信号集标记为空而使用 memset(3)函数将其内容清零的做法也不正确。）

信号集初始化后，可以分别使用 sigaddset()和 sigdelset()函数向一个集合中添加或者移除单个信号。

```
#include <signal.h>

int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

Both return 0 on success, or −1 on error

在 sigaddset()和 sigdelset()中，sig 参数均表示信号编号。

更多资源请访问稀酷客(www.ckook.com)

sigismember()函数用来测试信号 sig 是否是信号集 set 的成员。

```
#include <signal.h>

int sigismember(const sigset_t *set, int sig);
```

Returns 1 if *sig* is a member of *set*, otherwise 0

如果 sig 是 set 的一个成员，那么 sigismember()函数将返回 1（true），否则返回 0（false）。GNU C 库还实现了 3 个非标准函数，是对上述信号集标准函数的补充。

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

int sigandset(sigset_t *set, sigset_t *left, sigset_t *right);
int sigorset(sigset_t *dest, sigset_t *left, sigset_t *right);
```

Both return 0 on success, or −1 on error

```
int sigisemptyset(const sigset_t *set);
```

Returns 1 if *sig* is empty, otherwise 0

这些函数执行了如下任务。

- sigandset()将 left 集和 right 集的交集置于 dest 集。
- sigorset()将 left 集和 right 集的并集置于 dest 集。
- 若 set 集内未包含信号，则 sigisemptyset()返回 true。

示例程序

程序清单 20-4 所示为使用本节介绍的函数来编写的函数，供本书后续各程序调用。第一个函数 printSigset()显示了指定信号集的成员信号。该函数使用了定义于<signal.h>文件中的 NSIG 常量，其值等于信号最大编号加 1。当获取信号集成员时，会在测试所有信号编号的循环中将该值作为循环上限。

> 虽然 SUSv3 并未定义 NSIG，但是大多数 UNIX 实现都支持这一常量。只不过，要想使其可见，可能需要使用特定于实现的编译器选项。例如，在 Linux 中，就必须定义如下功能测试宏之一：BSD_SOURCE、_SVID_SOURCE 或者_GNU_SOURCE。

利用 printSigset()函数，printSigMask()和 printPendingSigs()函数分别用于显示进程的信号掩码和当前处于等待状态的信号集。这两个函数还分别使用了 sigprocmask()和 sigpending()系统调用。sigprocmask()和 sigpending()系统调用将分别在 20.10 节和 20.11 节中予以描述。

程序清单 20-4：显示信号集的函数

signals/signal_functions.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "signal_functions.h"          /* Declares functions defined here */
#include "tlpi_hdr.h"

/* NOTE: All of the following functions employ fprintf(), which
   is not async-signal-safe (see Section 21.1.2). As such, these
```


更多资源请访问稀酷客(www.ckook.com)

```
   functions are also not async-signal-safe (i.e., beware of
   indiscriminately calling them from signal handlers). */

void                        /* Print list of signals within a signal set */
printSigset(FILE *of, const char *prefix, const sigset_t *sigset)
{
    int sig, cnt;

    cnt = 0;
    for (sig = 1; sig < NSIG; sig++) {
        if (sigismember(sigset, sig)) {
            cnt++;
            fprintf(of, "%s%d (%s)\n", prefix, sig, strsignal(sig));
        }
    }

    if (cnt == 0)
        fprintf(of, "%s<empty signal set>\n", prefix);
}

int                         /* Print mask of blocked signals for this process */
printSigMask(FILE *of, const char *msg)
{
    sigset_t currMask;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigprocmask(SIG_BLOCK, NULL, &currMask) == -1)
        return -1;

    printSigset(of, "\t\t", &currMask);

    return 0;
}

int                         /* Print signals currently pending for this process */
printPendingSigs(FILE *of, const char *msg)
{
    sigset_t pendingSigs;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigpending(&pendingSigs) == -1)
        return -1;

    printSigset(of, "\t\t", &pendingSigs);

    return 0;
}
```

signals/signal_functions.c

20.10 信号掩码（阻塞信号传递）

内核会为每个进程维护一个信号掩码，即一组信号，并将阻塞其针对该进程的传递。如

更多资源请访问稀酷客(www.ckook.com)

果将遭阻塞的信号发送给某进程，那么对该信号的传递将延后，直至从进程信号掩码中移除该信号，从而解除阻塞为止。（由 33.2.1 节可知，信号掩码实际属于线程属性，在多线程进程中，每个线程都可使用 pthread_sigmask()函数来独立检查和修改其信号掩码。）

向信号掩码中添加一个信号，有如下几种方式。

- 当调用信号处理器程序时，可将引发调用的信号自动添加到信号掩码中。是否发生这一情况，要视 sigaction()函数在安装信号处理器程序时所使用的标志而定。
- 使用 sigaction()函数建立信号处理器程序时，可以指定一组额外信号，当调用该处理器程序时会将其阻塞。
- 使用 sigprocmask()系统调用，随时可以显式向信号掩码中添加或移除信号。

对前两种情况的讨论将推迟到 20.13 节对 sigaction()函数的介绍之后，现在先来讨论 sigprocmask()函数。

```
#include <signal.h>

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

Returns 0 on success, or −1 on error

使用 sigprocmask()函数既可修改进程的信号掩码，又可获取现有掩码，或者两重功效兼具。how 参数指定了 sigprocmask()函数想给信号掩码带来的变化。

SIG_BLOCK

将 set 指向信号集内的指定信号添加到信号掩码中。换言之，将信号掩码设置为其当前值和 set 的并集。

SIG_UNBLOCK

将 set 指向信号集中的信号从信号掩码中移除。即使要解除阻塞的信号当前并未处于阻塞状态，也不会返回错误。

SIG_SETMASK

将 set 指向的信号集赋给信号掩码。

上述各种情况下，若 oldset 参数不为空，则其指向一个 sigset_t 结构缓冲区，用于返回之前的信号掩码。

如果想获取信号掩码而又对其不作改动，那么可将 set 参数指定为空，这时将忽略 how 参数。

要想暂时阻止信号的传递，可以使用程序清单 20-5 中所示的一系列调用来阻塞信号，然后再将信号掩码重置为先前的状态以解除对信号的锁定。

程序清单 20-5：暂时阻塞信号传递

```
sigset_t blockSet, prevMask;

/* Initialize a signal set to contain SIGINT */

sigemptyset(&blockSet);
sigaddset(&blockSet, SIGINT);

/* Block SIGINT, save previous signal mask */

if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)
```


更多资源请访问稀酷客(www.ckook.com)

```
    errExit("sigprocmask1");

/* ... Code that should not be interrupted by SIGINT ... */

/* Restore previous signal mask, unblocking SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask2");
```

SUSv3 规定，如果有任何等待信号因对 sigprocmask()的调用而解除了锁定，那么在此调用返回前至少会传递一个信号。换言之，如果解除了对某个等待信号的锁定，那么会立刻将该信号传递给进程。

系统将忽略试图阻塞 SIGKILL 和 SIGSTOP 信号的请求。如果试图阻塞这些信号，sigprocmask()函数既不会予以关注，也不会产生错误。这意味着，可以使用如下代码来阻塞除 SIGKILL 和 SIGSTOP 之外的所有信号：

```
sigfillset(&blockSet);
if (sigprocmask(SIG_BLOCK, &blockSet, NULL) == -1)
    errExit("sigprocmask");
```

20.11 处于等待状态的信号

如果某进程接受了一个该进程正在阻塞的信号，那么会将该信号填加到进程的等待信号集中。当（且如果）之后解除了对该信号的锁定时，会随之将信号传递给此进程。为了确定进程中处于等待状态的是哪些信号，可以使用 sigpending()。

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Returns 0 on success, or −1 on error

sigpending()系统调用为调用进程返回处于等待状态的信号集，并将其置于 set 指向的 sigset_t 结构中。随后可以使用 20.9 节描述的 sigismember()函数来检查 set。

如果修改了对等待信号的处置，那么当后来解除对信号的锁定时，将根据新的处置来处理信号。这项技术虽然不经常使用，但还是存在一个应用场景，即将对信号的处置置为 SIG_IGN，或者 SIG_DFL（如果信号的默认行为是忽略），从而阻止传递处于等待状态的信号。因此，会将信号从进程的等待信号集中移除，从而不传递该信号。

20.12 不对信号进行排队处理

等待信号集只是一个掩码，仅表明一个信号是否发生，而未表明其发生的次数。换言之，如果同一信号在阻塞状态下产生多次，那么会将该信号记录在等待信号集中，并在稍后仅传递一次。（标准信号和实时信号之间的差异之一在于，如 22.8 节所述，对实时信号进行了排队处理。）

程序清单 20-6 和程序清单 20-7 显示了两个程序，可用于观察未作排队处理的信号。清单 20-6 的程序可接受多达四个命令行参数，如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
$ ./sig_sender PID num-sigs sig-num [sig-num-2]
```

第一个参数是程序发送信号的目标进程 ID。第二个参数则指定发送给目标进程的信号数量。第三个参数指定发往目标进程的信号编号。如果还提供了一个信号编号作为第四个参数，那么当程序发送完之前参数所指定的信号之后，将发送该信号的一个实例。在如下 shell 会话示例中，就使用了最后一个参数向目标进程发送一个 SIGINT 信号，发送该信号的目的将在稍后揭晓。

程序清单 20-6：发送多个信号

signals/sig_sender.c

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

int
main(int argc, char *argv[])
{
    int numSigs, sig, j;
    pid_t pid;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid num-sigs sig-num [sig-num-2]\n", argv[0]);
    pid = getLong(argv[1], 0, "PID");
    numSigs = getInt(argv[2], GN_GT_0, "num-sigs");
    sig = getInt(argv[3], 0, "sig-num");

    /* Send signals to receiver */

    printf("%s: sending signal %d to process %ld %d times\n",
            argv[0], sig, (long) pid, numSigs);

    for (j = 0; j < numSigs; j++)
        if (kill(pid, sig) == -1)
            errExit("kill");

    /* If a fourth command-line argument was specified, send that signal */

    if (argc > 4)
        if (kill(pid, getInt(argv[4], 0, "sig-num-2")) == -1)
            errExit("kill");

    printf("%s: exiting\n", argv[0]);
    exit(EXIT_SUCCESS);
}
```

signals/sig_sender.c

程序清单 20-7 中程序则被设计为去捕获程序清单 20-6 程序所发送的信号并汇总其统计数据。该程序执行了以下步骤。

- 该程序建立了单个处理器程序来捕获所有信号。（捕获 SIGKILL 和 SIGSTOP 信号是不可能的，不过将忽略在尝试为这些信号建立处理器时所发生的错误。）对于大多数类型的信号，处理器程序只是简单地使用一个数组来对信号计数。如果收到的信号为 SIGINT，那么处理器程序将对标志（gotSigint）置位，从而使程序退出主循环（下面所描述的 while 循环）。（至于 volatile 修饰符以及声明 gotSigint 变量的 sig_atomic_t 数据类型，将在 21.1.3 节中解释其用途。）
- 如果提供有一个命令行参数给程序，那么程序对所有信号的阻塞秒数将由该参数指

定，并且在解除阻塞之前会显示待处理的信号集，从而使用户在进程执行下面的步骤前向其发送信号。

- 程序执行 while 循环以消耗 CPU 时间，直至将 gotSigint 标志置位。（20.14 节和 22.9 节描述了 pause()和 sigsuspend()的用法，二者在等待信号到来期间对 CPU 的使用方式都颇为高效。）
- 退出 while 循环后，程序显示对所有接收信号的计数。

首先使用这两个程序来展示的是遭阻塞的信号无论产生了多少次，仅会传递一次。这里为接收者指定了一个睡眠间隔，并在醒来之前发送所有信号。

```
$ ./sig_receiver 15 &                    Receiver blocks signals for 15 secs
[1] 5368
./sig_receiver: PID is 5368
./sig_receiver: sleeping for 15 seconds
$ ./sig_sender 5368 1000000 10 2         Send SIGUSR1 signals, plus a SIGINT
./sig_sender: sending signal 10 to process 5368 1000000 times
./sig_sender: exiting
./sig_receiver: pending signals are:
                2 (Interrupt)
                10 (User defined signal 1)
./sig_receiver: signal 10 caught 1 time
[1]+  Done                    ./sig_receiver 15
```

发送程序的命令行参数指定了 SIGUSR1 和 SIGINT 信号，其在 Linux/x86 中的编号分别为 10 和 2。

从以上输出可知，即使一个信号发送了一百万次，但仅会传递一次给接收者。

即使进程没有阻塞信号，其所收到的信号也可能比发送给它的要少得多。如果信号发送速度如此之快，以至于在内核考虑将执行权调度给接收进程之前，这些信号就已经到达，这时就会发生上述情况，从而导致多次发送的信号在进程等待信号集中只记录了一次。如果不带任何命令行参数来执行程序清单 20-7 中程序（因此，进程没有阻塞信号，也没有睡眠），那么将看到如下情况：

```
$ ./sig_receiver &
[1] 5393
./sig_receiver: PID is 5393
$ ./sig_sender 5393 1000000 10 2
./sig_sender: sending signal 10 to process 5393 1000000 times
./sig_sender: exiting
./sig_receiver: signal 10 caught 52 times
[1]+  Done                    ./sig_receiver
```

在所发送的一百万次信号之中，接收进程仅捕获到 52 次。（捕获信号的精确数目每每不同，这取决于内核调度算法变幻莫测的决策结果。）之所以如此，原因在于，发送程序会在每次获得调度而运行时发送多个信号给接收者。然而，当接收进程得以运行时，传递来的信号只有一个，因为只会将这些信号中的一个标记为等待状态。

程序清单 20-7：捕获信号并对其计数

signals/sig_receiver.c

```
#define _GNU_SOURCE
#include <signal.h>
#include "signal_functions.h"          /* Declaration of printSigset() */
#include "tlpi_hdr.h"

static int sigCnt[NSIG];               /* Counts deliveries of each signal */
```

```
static volatile sig_atomic_t gotSigint = 0;
                                        /* Set nonzero if SIGINT is delivered */

static void
① handler(int sig)
{
    if (sig == SIGINT)
        gotSigint = 1;
    else
        sigCnt[sig]++;
}

int
main(int argc, char *argv[])
{
    int n, numSecs;
    sigset_t pendingMask, blockingMask, emptyMask;

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

②   for (n = 1; n < NSIG; n++)          /* Same handler for all signals */
        (void) signal(n, handler);      /* Ignore errors */

    /* If a sleep time was specified, temporarily block all signals,
       sleep (while another process sends us signals), and then
       display the mask of pending signals and unblock all signals */

③   if (argc > 1) {
        numSecs = getInt(argv[1], GN_GT_0, NULL);

        sigfillset(&blockingMask);
        if (sigprocmask(SIG_SETMASK, &blockingMask, NULL) == -1)
            errExit("sigprocmask");

        printf("%s: sleeping for %d seconds\n", argv[0], numSecs);
        sleep(numSecs);

        if (sigpending(&pendingMask) == -1)
            errExit("sigpending");

        printf("%s: pending signals are: \n", argv[0]);
        printSigset(stdout, "\t\t", &pendingMask);

        sigemptyset(&emptyMask);        /* Unblock all signals */
        if (sigprocmask(SIG_SETMASK, &emptyMask, NULL) == -1)
            errExit("sigprocmask");
    }

④   while (!gotSigint)                  /* Loop until SIGINT caught */
        continue;

⑤   for (n = 1; n < NSIG; n++)          /* Display number of signals received */
        if (sigCnt[n] != 0)
            printf("%s: signal %d caught %d time%s\n", argv[0], n,
                    sigCnt[n], (sigCnt[n] == 1) ? "" : "s");

    exit(EXIT_SUCCESS);
}
```

signals/sig_receiver.c

20.13 改变信号处置：sigaction ()

除去 signal()之外，sigaction()系统调用是设置信号处置的另一选择。虽然 sigaction()的用法比之 signal()更为复杂，但作为回报，也更具灵活性。尤其是，sigaction()允许在获取信号处置的同时无需将其改变，并且，还可设置各种属性对调用信号处理器程序时的行为施以更加精准的控制。此外，如 22.7 节所述，在建立信号处理器程序时，sigaction()较之 signal()函数可移植性更佳。

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Returns 0 on success, or −1 on error

sig 参数标识想要获取或改变的信号编号。该参数可以是除去 SIGKILL 和 SIGSTOP 之外的任何信号。

act 参数是一枚指针，指向描述信号新处置的数据结构。如果仅对信号的现有处置感兴趣，那么可将该参数指定为 NULL。oldact 参数是指向同一结构类型的指针，用来返回之前信号处置的相关信息。如果无意获取此类信息，那么可将该参数指定为 NULL。act 和 oldact 所指向的结构类型如下所示：

```
struct sigaction {
    void   (*sa_handler)(int);    /* Address of handler */
    sigset_t sa_mask;             /* Signals blocked during handler
                                     invocation */
    int      sa_flags;            /* Flags controlling handler invocation */
    void   (*sa_restorer)(void);  /* Not for application use */
};
```

sigaction 结构实际要比此处所展示的更为复杂，更多细节请参见 21.4 节。

sa_handler 字段对应于 signal()的 handler 参数。其所指定的值为信号处理器函数的地址，亦或是常量 SIG_IGN、SIG_DFL 之一。仅当 sa_handler 是信号处理程序的地址时，亦即 sa_handler 的取值在 SIG_IGN 和 SIG_DFL 之外，才会对 sa_mask 和 sa_flags 字段（稍后讨论）加以处理。余下的字段 sa_restorer，则不适用于应用程序（SUSv3 未予规定）。

sa_restorer 字段仅供内部使用，用以确保当信号处理器程序完成后，会去调用专用的 sigreturn()系统调用，借此来恢复进程的执行上下文，以便于进程从信号处理器中断的位置继续执行。这一用法的实例可见诸于 glibc 源文件 sysdeps/unix/sysv/linux/i386/ sigaction.c 中。

sa_mask 字段定义了一组信号，在调用由 sa_handler 所定义的处理器程序时将阻塞该组信号。当调用信号处理器程序时，会在调用信号处理器之前，将该组信号中当前未处于进程掩码之列的任何信号自动添加到进程掩码中。这些信号将保留在进程掩码中，直至信号处理器函数返回，届时将自动删除这些信号。利用 sa_mask 字段可指定一组信号，不允许它们中断此处理器程序的执行。此外，引发对处理器程序调用的信号将自动添加到进程信号掩码中。这意味着，当正在执行处理器程序时，如果同一个信号实例第二次抵达，信号处理器程序将不会递归中断自己。由于不会对遭阻塞的信号进行排队处理，如果在处理器程序执行过程中

更多资源请访问稀酷客(www.ckook.com)

重复产生这些信号中的任何信号，（稍后）对信号的传递将是一次性的。

sa_flags 字段是一个位掩码，指定用于控制信号处理过程的各种选项。该字段包含的位如下（可以相或（|））。

SA_NOCLDSTOP

若 sig 为 SIGCHLD 信号，则当因接受一信号而停止或恢复某一子进程时，将不会产生此信号。参见 26.3.2 节。

SA_NOCLDWAIT

（始于 Linux 2.6）若 sig 为 SIGCHLD 信号，则当子进程终止时不会将其转化为僵尸。更多细节参见 26.3.3 节。

SA_NODEFER

捕获该信号时，不会在执行处理器程序时将该信号自动添加到进程掩码中。SA_NOMASK 历史上曾是 SA_NODEFER 的代名词。之所以建议使用后者，是因为 SUSv3 将其纳入规范。

SA_ONSTACK

针对此信号调用处理器函数时，使用了由 sigaltstack()安装的备选栈。参见 21.3 节。

SA_RESETHAND

当捕获该信号时，会在调用处理器函数之前将信号处置重置为默认值（即 SIG_DFL）（默认情况下，信号处理器函数保持建立状态，直至进一步调用 sigaction()将其显式解除。）SA_ONESHOT 历史上曾是 SA_RESETHAND 的代名词，之所以建议使用后者，是因为 SUSv3 将其纳入规范。

SA_RESTART

自动重启由信号处理器程序中断的系统调用。参见 21.5 节。

SA_SIGINFO

调用信号处理器程序时携带了额外参数，其中提供了关于信号的深入信息。对该标志的描述参见 21.4 节。

SUSv3 定义了上述所有选项。

程序清单 21-1 展现了对 sigaction()的使用。

20.14 等待信号：pause()

调用 pause()将暂停进程的执行，直至信号处理器函数中断该调用为止（或者直至一个未处理信号终止进程为止）。

```
#include <unistd.h>

int pause(void);
```

Always returns −1 with *errno* set to EINTR

处理信号时，pause()遭到中断，并总是返回−1，并将 errno 置为 EINTR。（21.5 节描述了关于 EINTR 错误的更多信息。）

程序清单 20-2 提供了应用 pause()的一例子。

更多资源请访问稀酷客(www.ckook.com)

在 22.9 节、22.10 节及 22.11 节中，可以看到程序等待信号时暂停执行的各种其他方式。

20.15 总结

信号是发生某种事件的通知机制，可以由内核、另一进程或进程自身发送给进程。存在一系列的标准信号类型，每种都有唯一的编号和目的。

信号传递通常是异步行为，这意味着信号中断进程执行的位置是不可预测的。有时（比如，硬件产生的信号），信号也可同步传递，这意味着在程序执行的某一点可以预期并重现信号的传递。

默认情况下，要么忽略信号，要么终止进程（生成或者不生成核心转储文件），要么停止一个正在运行的进程，要么重启一个已停止的进程。特定的默认行为取决于信号类型。此外，程序可以使用 signal()或者 sigaction()来显式忽略一个信号，或者建立一个由程序员自定义的信号处理器程序，以供信号到达时调用。出于可移植性考虑，最好使用 sigaction()来建立信号处理器函数。

一个（具有适当权限的）进程可以使用 kill()向另一进程发送信号。发送空信号（0）是判定特定进程 ID 是否在用的方式之一。

每个进程都具有一个信号掩码，代表当前传递遭到阻塞的一组信号。使用 sigprocmask()可从信号掩码中添加或者移除信号。

如果接收的信号当前遭到阻塞，那么该信号将保持等待状态，直至解除对其阻塞。系统不会对标准信号进行排队处理，也就是说，将信号标记为等待状态（以及后续的传递）只会发生一次。进程能够使用 sigpending()系统调用来获取等待信号集（用以描述多个不同信号的数据结构）。

与 signal()相比，sigaction()系统调用在设置信号处置方面提供了更多控制，且更具灵活性。首先，可以指定一组调用处理器函数时将阻塞的额外信号。此外，可以使用各种标志来控制调用信号处理器时所发生的行为。例如，启用某些标志即可选择旧有的不可靠信号语义（不阻塞引发处理器调用的信号，在调用信号处理器之前就将信号处置重置为默认值）。

借助于 pause()，进程可暂停执行，直至信号到达为止。

更多信息

[Bovet & Cesati, 2005]和[Maxwell, 1999]提供了 Linux 信号实现的背景资料。[Goodheart & Cox, 1994]详细说明了 System V 版本 4 对信号的实现。GNU C 函数库手册（在线网访问：http://www.gnu.org/）包含了对信号的全面描述。

20.16 练习

20-1. 如 20.3 节所指，比之 signal()，sigaction()函数在建立信号处理器时可移植性更佳。请用 sigaction()替换程序清单 20-7 程序（sig_receiver.c）中对 signal()的调用。

20-2. 编写一程序来展示当将对等待信号的处置改为 SIG_IGN 时，程序绝不会看到（捕获）信号。

20-3. 编写一程序，以 sigaction()来建立信号处理器函数，请验证 SA_RESETHAND 和 SA_NODEFER 标志的效果。

20-4. 请用 sigaction()调用来实现 siginterrupt()。

更多资源请访问稀酷客(www.ckook.com)

第 21 章

信号：信号处理器函数

承接上一章，本章将继续介绍信号。本章的描述重点是信号处理器函数（handler），同时会拓展 20.4 节所发起的讨论。本章涵盖主题如下。

- 如何设计信号处理器函数，其中对可重入性以及异步信号安全函数的探讨必不可少。
- 从信号处理器函数中正常返回的各种途径，特别是非本地跳转技术的运用。
- 利用备选栈处理信号。
- 借助于带有 SA_SIGINFO 标志的 sigaction()函数，处理器函数能够获取引发其调用信号的更多详细信息。
- 信号处理器函数如何中断处于阻塞状态的系统调用，以及如何重启该系统调用。

21.1 设计信号处理器函数

一般而言，将信号处理器函数设计得越简单越好。其中的一个重要原因就在于，这将降低引发竞争条件的风险。下面是针对信号处理器函数的两种常见设计。

- 信号处理器函数设置全局性标志变量并退出。主程序对此标志进行周期性检查，一旦置位随即采取相应动作。（主程序若因监控一个或多个文件描述符的 I/O 状态而无法进行这种周期性检查时，则可令信号处理器函数向一专用管道写入一个字节的数据，同时将该管道的读取端置于主程序所监控的文件描述符范围之内。63.5.2 节展示了这一技术的运用。）
- 信号处理器函数执行某种类型的清理动作，接着终止进程或者使用非本地跳转（21.2.1 节）将栈解开并将控制返回到主程序中的预定位置。

后续各节将会探讨这些设计理念，以及信号处理器函数设计中的其他一些重要概念。

21.1.1 再论信号的非队列化处理

20.10 节已然提及，在执行某信号的处理器函数时会阻塞同类信号的传递（除非在调用 sigaction()时指定了 SA_NODEFER 标志）。如果在执行处理器函数时（再次）产生同类信号，那

更多资源请访问稀酷客(www.ckook.com)

么会将该信号标记为等待状态并在处理器函数返回之后再行传递。前一章还曾指出，不会对信号进行排队处理。在处理器函数执行期间，如果多次产生同类信号，那么仍然会将其标记为等待状态，但稍后只会传递一次。

信号的这种“失踪”方式无疑将影响对信号处理器函数的设计。首先，无法对信号的产生次数进行可靠计数。其次，在为信号处理器函数编码时可能需要考虑处理同类信号多次产生的情况。26.3.1 节在讨论 SIGCHLD 信号时会有相关示例。

21.1.2 可重入函数和异步信号安全函数

在信号处理器函数中，并非所有系统调用以及库函数均可予以安全调用。要了解来龙去脉，就需要解释一下以下两种概念：可重入（reentrant）函数和异步信号安全（async-signal-safe）函数。

可重入和非可重入函数

要解释可重入函数为何物，首先需要区分单线程程序和多线程程序。典型 UNIX 程序都具有一条执行线程，贯穿程序始终，CPU 围绕单条执行逻辑来处理指令。而对于多线程程序而言，同一进程却存在多条独立、并发的执行逻辑流。

第 29 章将会展示如何显式创建一个包含多条执行线程的程序。不过，多执行线程的概念与使用了信号处理器函数的程序也有关联。因为信号处理器函数可能会在任一时点异步中断程序的执行，从而在同一个进程中实际形成了两条（即主程序和信号处理器函数）独立（虽然不是并发）的执行线程。

如果同一个进程的多条线程可以同时安全地调用某一函数，那么该函数就是可重入的。此处，“安全”意味着，无论其他线程调用该函数的执行状态如何，函数均可产生预期结果。

SUSv3 对可重入函数的定义是：函数由两条或多条线程调用时，即便是交叉执行，其效果也与各线程以未定义[1]顺序依次调用时一致。

更新全局变量或静态数据结构的函数可能是不可重入的。（只用到本地变量的函数肯定是可重入的。）如果对函数的两个调用（例如：分别由两条执行线程发起）同时试图更新同一全局变量或数据类型，那么二者很可能会相互干扰并产生不正确的结果。例如，假设某线程正在为一链表数据结构添加一个新的链表项，而另一线程也正试图更新同一链表。由于为链表添加新项涉及对多枚指针的更新，一旦另一线程中断这些步骤并修改了相同的指针，结果就会产生混乱。

在 C 语言标准函数库中，这种可能性非常普遍。例如，7.1.3 节所提及的 malloc()和 free()就维护有一个针对已释放内存块的链表，用于从堆中重新分配内存。如果主程序在调用 malloc()期间为一个同样调用 malloc()的信号处理器函数所中断，那么该链表可能会遭到破坏。因此，malloc()函数族以及使用它们的其他库函数都是不可重入的。

还有一些函数库之所以不可重入，是因为它们使用了经静态分配的内存来返回信息。此类函数（本书其他地方也有论及）的例子包括 crypt()、getpwnam()、gethostbyname()以及 getservbyname()。如果信号处理器用到了这类函数，那么将会覆盖主程序中上次调用同一函数所返回的信息（反之亦然）。

将静态数据结构用于内部记账的函数也是不可重入的。其中最明显的例子就是 stdio 函数库成员（printf()、scanf()等），它们会为缓冲区 I/O 更新内部数据结构。所以，如果在信号处

1 译者注：任意。

更多资源请访问稀酷客(www.ckook.com)

理器函数中调用了 printf()，而主程序又在调用 printf()或其他 stdio 函数期间遭到了处理器函数的中断，那么有时就会看到奇怪的输出，甚至导致程序崩溃或者数据的损坏。

即使并未使用不可重入的库函数，可重入问题依然不容忽视。如果信号处理器函数和主程序都要更新由程序员自定义的全局性数据结构，那么对于主程序而言，这种信号处理器函数就是不可重入的。

如果函数是不可重入的，那么其手册页通常会或明或暗地给出提示。对于其中那些使用或返回静态分配变量的函数，需要特别留意。

示例程序

程序清单 21-1 展示了函数 crypt()（8.5 节）不可重入的本来面目。该程序接受两个字符串作为命令行参数，执行步骤如下。

1. 调用 crypt()加密第 1 个命令行参数中的字符串，并使用 strdup()将结果复制到独立缓冲区中。
2. 为 SIGINT 信号（按下 Ctrl-C 产生）创建处理器函数。处理器函数调用 crypt()加密第 2 个命令行参数所提供的字符串。
3. 进入无限 for 循环，使用 crypt()加密第 1 个命令行参数中的字符串，并检查其返回字符串与第 1 步保存的结果是否一致。

在不产生信号的情况下，第 3 步中的检查结果将总是匹配。然而，一旦收到 SIGINT 信号，而主程序又恰在 for 循环内的 crypt()调用之后，字符串的匹配检查之前遭到信号处理器函数的中断，这时就会发生字符串不匹配的情况。程序运行结果如下：

```
$ ./non_reentrant abc def
Repeatedly type Control-C to generate SIGINT
Mismatch on call 109871 (mismatch=1 handled=1)
Mismatch on call 128061 (mismatch=2 handled=2)
Many lines of output removed
Mismatch on call 727935 (mismatch=149 handled=156)
Mismatch on call 729547 (mismatch=150 handled=157)
Type Control-\ to generate SIGQUIT
Quit (core dumped)
```

由对上述输出 mismatch 和 handled 值的比较可知，在大多数情况下，处理器函数会在 main()中的 crypt()调用与字符串比较之间去覆盖静态分配的缓冲区。

程序清单 21-1：在 main()以及信号处理函数中调用不可重入的函数

signals/nonreentrant.c

```
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include "tlpi_hdr.h"

static char *str2;              /* Set from argv[2] */
static int handled = 0;         /* Counts number of calls to handler */

static void
handler(int sig)
{
    crypt(str2, "xx");
    handled++;
}
```


更多资源请访问稀酷客(www.ckook.com)

```
int
main(int argc, char *argv[])
{
    char *cr1;
    int callNum, mismatch;
    struct sigaction sa;

    if (argc != 3)
        usageErr("%s str1 str2\n", argv[0]);

    str2 = argv[2];                        /* Make argv[2] available to handler */
    cr1 = strdup(crypt(argv[1], "xx"));  /* Copy statically allocated string
                                            to another buffer */
    if (cr1 == NULL)
        errExit("strdup");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    /* Repeatedly call crypt() using argv[1]. If interrupted by a
       signal handler, then the static storage returned by crypt()
       will be overwritten by the results of encrypting argv[2], and
       strcmp() will detect a mismatch with the value in 'cr1'. */

    for (callNum = 1, mismatch = 0; ; callNum++) {
        if (strcmp(crypt(argv[1], "xx"), cr1) != 0) {
            mismatch++;
            printf("Mismatch on call %d (mismatch=%d handled=%d)\n",
                    callNum, mismatch, handled);
        }
    }
}
```

signals/nonreentrant.c

标准的异步信号安全函数

异步信号安全的函数是指当从信号处理器函数调用时，可以保证其实现是安全的。如果某一函数是可重入的，又或者信号处理器函数无法将其中断时，就称该函数是异步信号安全的。

表 21-1 所列为各种标准要求实现为异步信号安全的函数。其中，名称后未跟 v2 或 v3 字符串的函数是由 POSIX.1-1990 规定为异步信号安全的。带有 v2 标记的函数由 susv2 加入，带有 v3 标记的则由 susv3 加入。个别 UNIX 实现可能会将其他某些函数实现为异步信号安全的，但所有符合标准的 UNIX 实现都必须保证至少表中这些函数是异步信号安全的（假设由实现来提供这些函数，Linux 并未实现所有这些函数）。

SUSv4 对表 21-1 做了如下修改。

- 移除如下函数：fpathconf()、pathconf()和 sysconf()。
- 添加如下函数：execl()、execv()、faccessat()、fchmodat()、fchownat()、fexecve()、fstatat()、futimens()、linkat()、mkdirat()、mkfifoat()、mknod()、mknodat()、openat()、readlinkat()、renameat()、symlinkat()、unlinkat()、utimensat()和 utimes()。

表 21-1：POSIX.1-1990、SUSv2 和 SUSv3 规定为异步信号安全的函数

_Exit() (v3)	getpid()	sigdelset()
_exit()	getppid()	sigemptyset()
abort() (v3)	getsockname() (v3)	sigfillset()
accept() (v3)	getsockopt() (v3)	sigismember()
access()	getuid()	signal() (v2)
aio_error() (v2)	kill()	sigpause() (v2)
aio_return() (v2)	link()	sigpending()
aio_suspend() (v2)	listen() (v3)	sigprocmask()
alarm()	lseek()	sigqueue() (v2)
bind() (v3)	lstat() (v3)	sigset() (v2)
cfgetispeed()	mkdir()	sigsuspend()
cfgetospeed()	mkfifo()	sleep()
cfsetispeed()	open()	socket() (v3)
cfsetospeed()	pathconf()	sockatmark() (v3)
chdir()	pause()	socketpair() (v3)
chmod()	pipe()	stat()
chown()	poll() (v3)	symlink() (v3)
clock_gettime() (v2)	posix_trace_event() (v3)	sysconf()
close()	pselect() (v3)	tcdrain()
connect() (v3)	raise() (v2)	tcflow()
creat()	read()	tcflush()
dup()	readlink() (v3)	tcgetattr()
dup2()	recv() (v3)	tcgetpgrp()
execle()	recvfrom() (v3)	tcsendbreak()
execve()	recvmsg() (v3)	tcsetattr()
fchmod() (v3)	rename()	tcsetpgrp()
fchown() (v3)	rmdir()	time()
fcntl()	select() (v3)	timer_getoverrun() (v2)
fdatasync() (v2)	sem_post() (v2)	timer_gettime() (v2)
fork()	send() (v3)	timer_settime() (v2)
fpathconf() (v2)	sendmsg() (v3)	times()
fstat()	sendto() (v3)	umask()
fsync() (v2)	setgid()	uname()
ftruncate() (v3)	setpgid()	unlink()
getegid()	setsid()	utime()
geteuid()	setsockopt() (v3)	wait()
getgid()	setuid()	waitpid()
getgroups()	shutdown() (v3)	write()
getpeername() (v3)	sigaction()	
getpgrp()	sigaddset()	

SUSv3 强调，表 21-1 之外的所有函数对于信号而言都是不安全的，但同时指出，仅当信号处理器函数中断了不安全函数的执行，且处理器函数自身也调用了这个不安全函数时，该函数才是不安全的。换言之，编写信号处理器函数有如下两种选择。

更多资源请访问稀酷客(www.ckook.com)

- 确保信号处理器函数代码本身是可重入的，且只调用异步信号安全的函数。
- 当主程序执行不安全函数或是去操作信号处理器函数也可能更新的全局数据结构时，阻塞信号的传递。

第 2 种方法的问题是，在一个复杂程序中，要想确保主程序对不安全函数的调用不为信号处理器函数所中断，这有些困难。出于这一原因，通常就将上述规则简化为在信号处理器函数中绝不调用不安全的函数。

如果使用同一处理器函数来处理多个不同信号，或者在调用 sigaction()时设置了 SA_NODEFER 标志，那么处理器函数就有可能中断自己。因此，处理器函数如果更新了全局（或静态）变量，即便主程序不使用这些变量，那么它们依然可能是不可重入的。

信号处理器函数内部对 errno 的使用

由于可能会更新 errno，调用表 21-1 中函数依然会导致信号处理器函数不可重入，因为它们可能会覆盖之前由主程序调用函数时所设置的 errno 值。有一种变通方法，即当信号处理器函数使用了表 21-1 所列函数时，可在其入口处保存 errno 值，并在其出口处恢复 errno 的旧有值，请看下面的例子：

```
void
handler(int sig)
{
    int savedErrno;

    savedErrno = errno;

    /* Now we can execute a function that might modify errno */

    errno = savedErrno;
}
```

在本书示例程序中使用不安全函数

虽然 printf()不是异步信号安全的函数，但却频频现身于本书各种示例的信号处理器函数中。之所以如此，是因为在展示对信号处理器的调用，以及显示处理器相关变量的内容时，printf()都不失为一种简明而又便捷的方式。出于类似原因，在信号处理器函数中偶尔也会用到其他一些不安全函数，包括其他的 stdio 函数以及 strsignal()。

真正的应用程序应当避免在信号处理器函数中调用非异步信号安全的函数。为了明确这一点，每当示例的信号处理器调用这些函数时，代码注释中都会注明这一用法是不安全的。

```
printf("Some message\n");          /* UNSAFE */
```

21.1.3 全局变量和 sig_atomic_t 数据类型

尽管存在可重入问题，有时仍需要在主程序和信号处理器函数之间共享全局变量。信号处理器函数可能会随时修改全局变量——只要主程序能够正确处理这种可能性，共享全局变量就是安全的。例如，一种常见的设计是，信号处理器函数只做一件事情，设置全局标志。主程序则会周期性地检查这一标志，并采取相应动作来响应信号传递（同时清除标志）。当信号处理器函数以此方式来访问全局变量时，应该总是在声明变量时使用 volatile 关键字，从而防止编译器将其优化到寄存器中。

更多资源请访问稀酷客(www.ckook.com)

对全局变量的读写可能不止一条机器指令，而信号处理器函数就可能会在这些指令序列之间将主程序中断（也将此类变量访问称为非原子操作）。因此，C 语言标准以及 SUSv3 定义了一种整型数据类型 sig_atomic_t，意在保证读写操作的原子性。因此，所有在主程序与信号处理器函数之间共享的全局变量都应声明如下：

```
volatile sig_atomic_t flag;
```

程序清单 22-5 提供了使用 sig_atomic_t 数据类型的一个例子。

注意，C 语言的递增（++）和递减（--）操作符并不在 sig_atomic_t 所提供的保障范围之内。这些操作在某些硬件架构上可能不是原子操作（更多细节请参考 30.1 节）。在使用 sig_atomic_t 变量时唯一所能做的就是在信号处理器中进行设置，在主程序中进行检查（反之亦可）。

C99 和 SUSv3 规定，实现应当（在<stdint.h>中）定义两个常量 SIG_ATOMIC_MIN 和 SIG_ATOMIC_MAX，用于规定可赋给 sig_atomic_t 类型的值范围。标准要求，如果将 sig_atomic_t 表示为有符号值，其范围至少应该在-127～127 之间，如果作为无符号值，则应该在 0～255 之间。在 Linux 中，这两个常量分别等于有符号 32 位整型数的负、正极限值。

21.2 终止信号处理器函数的其他方法

目前为止所看到的信号处理器函数都是以返回主程序而终结。不过，只是简单地从信号处理器函数中返回并不能满足需要，有时候甚至没什么用处。（22.4 节在讨论硬件产生的信号时会举出这方面的例子。）

以下是从信号处理器函数中终止的其他一些方法。

- 使用_exit()终止进程。处理器函数事先可以做一些清理工作。注意，不要使用 exit()来终止信号处理器函数，因为它不在表 21-1 所列的安全函数中。之所以不安全，是因为如 25.1 节所述，该函数会在调用_exit()之前刷新 stdio 的缓冲区。
- 使用 kill()发送信号来杀掉进程（即，信号的默认动作是终止进程）。
- 从信号处理器函数中执行非本地跳转。
- 使用 abort()函数终止进程，并产生核心转储。

以下各节将会对最后两点做深入讨论。

21.2.1 在信号处理器函数中执行非本地跳转

6.8 节曾论及使用 setjmp()和 longjmp()来执行非本地跳转，以便从一个函数跳转至该函数的某个调用者。在信号处理器函数中也可以使用这种技术。这也是因硬件异常（例如内存访问错误）而导致信号传递之后的一条恢复途径，允许将信号捕获并把控制返回到程序中某个特定位置。例如，一旦收到 SIGINT 信号（通常由键入 Ctrl-C 产生），shell 执行一个非本地跳转，将控制返回到主输入循环中（以便读取下一条命令）。

然而，使用标准 longjmp()函数从处理器函数中退出存在一个问题。之前曾经提及，在进入信号处理器函数时，内核会自动将引发调用的信号以及由 act.sa_mask 所指定的任意信号添加到进程的信号掩码中，并在处理器函数正常返回时再将它们从掩码中清除。

如果使用 longjmp()来退出信号处理器函数，那么信号掩码会发生什么情况呢？这取决于特定 UNIX 实现的血统。在 System V 一脉中，longjmp()不会将信号掩码恢复，亦即在离开处理器函数时不会对遭阻塞的信号解除阻塞。Linux 遵循 System V 的这一特性。（这通常并非所

更多资源请访问稀酷客(www.ckook.com)

希望的行为，因为引发对信号处理器调用的信号仍将保持阻塞状态。）在源于 BSD 一脉的实现中，setjmp()将信号掩码保存在其 env 参数中，而信号掩码的保存值由 longjmp()恢复。（继承自 BSD 的实现还提供另外两个拥有 System V 语义的函数：_setjmp()和_longjmp()。）换言之，使用 longjmp()来退出信号处理器函数将有损于程序的可移植性。

> 如果编译程序时定义了_BSD_SOURCE 特性检测宏，那么（glibc 的）setjmp()将遵循 BSD 语义。

鉴于两大 UNIX 流派之间的差异，POSIX.1-1990 选择不对 setjmp()和 longjmp()的信号掩码处理进行规范，而是定义了一对新函数：sigsetjmp()和 siglongjmp()，针对执行非本地跳转时的信号掩码进行显式控制。

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
                        Returns 0 on initial call, nonzero on return via siglongjmp()
void siglongjmp(sigjmp_buf env, int val);
```

函数 sigsetjmp()和 siglongjmp()的操作与 setjmp()和 longjmp()类似。唯一的区别是参数 env 的类型不同（是 sigjmp_buf 而不是 jmp_buf），并且 sigsetjmp()多出一个参数 savesigs。如果指定 savesigs 为非 0，那么会将调用 sigsetjmp()时进程的当前信号掩码保存于 env 中，之后通过指定相同 env 参数的 siglongjmp()调用进行恢复。如果 savesigs 为 0，则不会保存和恢复进程的信号掩码。

函数 longjmp()和 siglongjmp()都不在表 21-1 所列异步信号安全函数的范围之内。因为与在信号处理器中调用这些函数一样，在执行非本地跳转之后去调用任何非异步信号安全的函数也需要冒同样的风险。此外，如果信号处理器函数中断了正在更新数据结构的主程序，那么执行非本地跳转退出处理器函数后，这种不完整的更新动作很可能会将数据结构置于不一致状态。规避这一问题的一种技术是在程序对敏感数据进行更新时，借助于 sigprocmask()临时将信号阻塞起来。

示例程序

程序清单 21-2 展示了两种类型的非本地跳转在处理信号掩码上的差异。该程序为 SIGINT 创建处理器函数，并允许选择 setjmp()+longjmp()组合或者 sigsetjmp()+siglongjmp()组合的方式来退出信号处理器函数，具体采用何种函数组合则取决于程序编译时是否对宏 USE_SIGSETJMP 进行了定义。程序会分别在进入信号处理器函数时，以及非本地跳转将控制从信号处理器交还给主程序后，显示信号掩码的当前设置。

如果利用 longjmp()来退出信号处理器函数，其结果如下：

```
$ make -s sigmask_longjmp          Default compilation causes setjmp() to be used
$ ./sigmask_longjmp
Signal mask at startup:
                <empty signal set>
Calling setjmp()
Type Control-C to generate SIGINT
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                2 (Interrupt)
(At this point, typing Control-C again has no effect, since SIGINT is blocked)
Type Control-\ to kill the program
Quit
```

由程序输出结果可知，信号处理器函数调用 longjmp()之后的信号掩码设置与进入处理器函数时保持一致。

上述 shell 会话中构建程序所使用的 makefile 由随本书发布的源码提供。选项-s 告知 make 程序不要显示正在执行的命令。使用该选项意在避免对会话日志的显示产生干扰。（[Mecklenbug, 2005]对 GNU make 程序做了说明。）

如果编译同一源文件来创建利用 siglongjmp()退出信号处理器函数的程序，则结果如下：

```
$ make -s sigmask_siglongjmp        Compiles using cc -DUSE_SIGSETJMP
$ ./sigmask_siglongjmp x
Signal mask at startup:
                <empty signal set>
Calling sigsetjmp()
Type Control-C
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                <empty signal set>
```

在这里，没有将 SIGINT 信号阻塞，因为 siglongjmp()恢复了原来的信号掩码。接着，再次按下 Ctrl-C，会再次调用该信号处理器函数。

```
Type Control-C
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                <empty signal set>
Type Control-\ to kill the program
Quit
```

由上述输出可知，siglongjmp()将信号掩码恢复到调用 sigsetjmp()时的值（即一个空信号集）。

程序清单 21-2 还展示了信号处理器函数执行非本地跳转时的一种实用技术。信号随时可能产生，所以有可能发生于 segsetjmp()（或 setjmp()）设置跳转目标之前。为杜绝这种可能（这将导致处理器函数使用未初始化的 env 缓冲区来执行非本地跳转），程序启用了守卫变量 canJump，来表征 env 缓冲区的初始化与否。如果 canJump 不为真（false），处理器函数将不执行跳转而直接返回。另一种方法是调整程序代码，在创建信号处理器函数之前去调用 sigsetjmp()（或 setjmp()）。不过对于复杂的程序而言，苛求这样的步骤执行顺序可能会有困难，而使用守卫变量也许会更简单一些。

注意，在编写程序清单 21-2 程序时使用#ifdef 是使其编码风格符合标准的最简单的手段。特别是当无法用下面的运行时检查代码来取代#ifdef 时。

```
if (useSiglongjmp)
    s = sigsetjmp(senv, 1);
else
    s = setjmp(env);
if (s == 0)
    ...
```

这一做法有违规范，因为 SUSv3 不允许在赋值语句（6.8 节）中调用 setjmp()和 sigsetjmp()。

程序清单 21-2：在信号处理器函数中执行非本地跳转

signals/sigmask_longjmp.c

```
#define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <setjmp.h>
```


更多资源请访问稀酷客(www.ckook.com)

```
#include <signal.h>
#include "signal_functions.h"          /* Declaration of printSigMask() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t canJump = 0;
                        /* Set to 1 once "env" buffer has been
                           initialized by [sig]setjmp() */
#ifdef USE_SIGSETJMP
static sigjmp_buf senv;
#else
static jmp_buf env;
#endif

static void
handler(int sig)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), strsignal(), printSigMask(); see Section 21.1.2) */

    printf("Received signal %d (%s), signal mask is:\n", sig,
            strsignal(sig));
    printSigMask(stdout, NULL);

    if (!canJump) {
        printf("'env' buffer not yet set, doing a simple return\n");
        return;
    }

#ifdef USE_SIGSETJMP
    siglongjmp(senv, 1);
#else
    longjmp(env, 1);
#endif
}
int
main(int argc, char *argv[])
{
    struct sigaction sa;

    printSigMask(stdout, "Signal mask at startup:\n");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

#ifdef USE_SIGSETJMP
    printf("Calling sigsetjmp()\n");
    if (sigsetjmp(senv, 1) == 0)
#else
    printf("Calling setjmp()\n");
    if (setjmp(env) == 0)
#endif
        canJump = 1;                    /* Executed after [sig]setjmp() */

    else                                /* Executed after [sig]longjmp() */
        printSigMask(stdout, "After jump from handler, signal mask is:\n" );
```


更多资源请访问稀酷客(www.ckook.com)

```
    for (;;)                                /* Wait for signals until killed */
        pause();
}
```

signals/sigmask_longjmp.c

21.2.2 异常终止进程：abort()

函数 abort()终止其调用进程，并生成核心转储。

```
#include <stdlib.h>

void abort(void);
```

函数 abort()通过产生 SIGABRT 信号来终止调用进程。对 SIGABRT 的默认动作是产生核心转储文件并终止进程。调试器可以利用核心转储文件来检测调用 abort()时的程序状态。

SUSv3 要求，无论阻塞或者忽略 SIGABRT 信号，abort()调用均不受影响。同时规定，除非进程捕获 SIGABRT 信号后信号处理器函数尚未返回，否则 abort()必须终止进程。后一句话值得三思。21.2 节所描述的信号处理器函数终止方法中，与此相关的就是使用非本地跳转退出处理器函数。这一做法将抵消 abort()的效果。否则，abort()将总是终止进程。在大多数实现中，终止时可确保发生如下事件：若进程在发出一次 SIGABRT 信号后仍未终止（即，处理器捕获信号并返回，以便恢复执行 abort()），则 abort()会将对 SIGABRT 信号的处理重置为 SIG_DFL，并再度发出 SIGABRT 信号，从而确保将进程杀死。

如果 abort()成功终止了进程，那么还将刷新 stdio 流并将其关闭。

程序清单 3-3 在错误处理函数中提供了使用 abort()的一个例子。

21.3 在备选栈中处理信号：sigaltstack()

在调用信号处理器函数时，内核通常会在进程栈中为其创建一帧。不过，如果进程对栈的扩展突破了对栈大小的限制时，这种做法就不大可行了。例如，栈的增长过大，以至于会触及到一片映射内存（48.5 节）或者向上增长的堆，又或者栈的大小已经直逼 RLIMIT_STACK（36.3 节）资源限制，这些都会造成这种情况的发生。

当进程对栈的扩展试图突破其上限时，内核将为该进程产生 SIGSEGV 信号。不过，因为栈空间已然耗尽，内核也就无法为进程已经安装的 SIGSEGV 处理器函数创建栈帧。结果是，处理器函数得不到调用，而进程也就终止了（SIGSEGV 的默认动作）。

如果希望在这种情况下确保对 SIGSEGV 信号处理器函数的调用，就需要做如下工作。

1. 分配一块被称为“备选信号栈”的内存区域，作为信号处理器函数的栈帧。
2. 调用 sigaltstack()，告之内核该备选信号栈的存在。
3. 在创建信号处理器函数时指定 SA_ONSTACK 标志，亦即通知内核在备选栈上为处理器函数创建栈帧。

利用系统调用 sigaltstack()，既可以创建一个备选信号栈，也可以将已创建备选信号栈的相关信息返回。

更多资源请访问稀酷客(www.ckook.com)

```
#include <signal.h>

int sigaltstack(const stack_t *sigstack, stack_t *old_sigstack);
```

Returns 0 on success, or −1 on error

参数 sigstack 所指向的数据结构描述了新备选信号栈的位置及属性。参数 old_sigstack 指向的结构则用于返回上一备选信号栈的相关信息（如果存在）。两个参数之一均可为 NULL。例如，将参数 sigstack 设为 NULL 可以发现现有备选信号栈，并且不用将其改变。不为 NULL 时，这些参数所指向的数据结构类型如下：

```
typedef struct {
    void  *ss_sp;         /* Starting address of alternate stack */
    int    ss_flags;      /* Flags: SS_ONSTACK, SS_DISABLE */
    size_t ss_size;       /* Size of alternate stack */
} stack_t;
```

字段 ss_sp 和 ss_size 分别指定了备选信号栈的位置和大小。在实际使用信号栈时，内核会将 ss_sp 值自动对齐为与硬件架构相适宜的地址边界。

备选信号栈通常既可以静态分配，也可以在堆上动态分配。SUSv3 规定将常量 SIGSTKSZ 作为划分备选栈大小的典型值，而将 MINSSIGSTKSZ 作为调用信号处理器函数所需的最小值。在 Linux/x86-32 系统上，分别将这两个值定义为 8192 和 2048。

内核不会重新划分备选栈的大小。如果栈溢出了分配给它的空间，就会产生混乱（例如，写变量超出了对栈的限制）。这通常不是一个问题，因为一般情况下会利用备选栈来处理标准栈溢出的特殊情况，常常只在这个栈上分配为数不多的几帧。SIGSEGV 处理器函数的工作不是在执行清理动作后终止进程，就是使用非本地跳转解开标准栈。

ss_flags 可以包含如下值之一：

SS_ONSTACK

如果在获取已创建备选信号栈的当前信息时该标志已然置位，就表明进程正在备选信号栈上执行。当进程已经在备选信号栈上运行时，试图调用 sigaltstack()来创建一个新的备选信号栈将会产生一个错误（EPERM）。

SS_DISABLE

在 old_sigstack 中返回，表示当前不存在已创建的备选信号栈。如果在 sigstack 中指定，则会禁用当前已创建的备选信号栈。

程序清单 21-3 演示了备选信号栈的创建和使用。在创建一个新的备选信号栈以及 SIGSEGV 的信号处理器函数之后，程序将调用一个无限递归函数，这会导致栈溢出，同时系统会向进程发送 SIGSEGV 信号。运行该程序的结果如下。

```
$ ulimit -s unlimited
$ ./t_sigaltstack
Top of standard stack is near 0xbffff6b8

Alternate stack is at          0x804a948-0x804cfff
Call    1 - top of stack near 0xbff0b3ac
Call    2 - top of stack near 0xbfe1714c
Many intervening lines of output removed
Call 2144 - top of stack near 0x4034120c

Call 2145 - top of stack near 0x4024cfac
Caught signal 11 (Segmentation fault)
Top of handler stack near      0x804c860
```

在这一 shell 会话中，命令 ulimit 负责移除 shell 之前可能设置的任何 RLIMIT_STACK 资源限制。36.3 节会解释这种资源限制。

程序清单 21-3：使用 sigaltstack()

signals/t_sigaltstack.c

```
#define _GNU_SOURCE          /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigsegvHandler(int sig)
{
    int x;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), strsignal(), fflush(); see Section 21.1.2) */

    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
    printf("Top of handler stack near     %10p\n", (void *) &x);
    fflush(NULL);

    _exit(EXIT_FAILURE);                /* Can't return after SIGSEGV */
}

static void             /* A recursive function that overflows the stack */
overflowStack(int callNum)
{
    char a[100000];                     /* Make this stack frame large */

    printf("Call %4d - top of stack near %10p\n", callNum, &a[0]);
    overflowStack(callNum+1);
}

int
main(int argc, char *argv[])
{
    stack_t sigstack;
    struct sigaction sa;
    int j;

    printf("Top of standard stack is near %10p\n", (void *) &j);

    /* Allocate alternate stack and inform kernel of its existence */

    sigstack.ss_sp = malloc(SIGSTKSZ);
    if (sigstack.ss_sp == NULL)
        errExit("malloc");
    sigstack.ss_size = SIGSTKSZ;
    sigstack.ss_flags = 0;
    if (sigaltstack(&sigstack, NULL) == -1)
        errExit("sigaltstack");
    printf("Alternate stack is at         %10p-%p\n",
            sigstack.ss_sp, (char *) sbrk(0) - 1);

    sa.sa_handler = sigsegvHandler;     /* Establish handler for SIGSEGV */
    sigemptyset(&sa.sa_mask);
```

```
    sa.sa_flags = SA_ONSTACK;           /* Handler uses alternate stack */
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        errExit("sigaction");

    overflowStack(1);
}
```

signals/t_sigaltstack.c

21.4 SA_SIGINFO 标志

如果在使用 sigaction()创建处理器函数时设置了 SA_SIGINFO 标志，那么在收到信号时处理器函数可以获取该信号的一些附加信息。为获取这一信息，需要将处理器函数声明如下：

```
void handler(int sig, siginfo_t *siginfo, void *ucontext);
```

如同标准信号处理器函数一样，第 1 个参数 sig 表示信号编号。第 2 个参数 siginfo 是用于提供信号附加信息的一个结构。该结构会与最后一个参数 ucontext 一起，在下面做详细说明。

因为上述信号处理器函数的原型不同于标准处理器函数，依照 C 语言的类型规则，将无法利用 sigaction 结构的 sa_handler 字段来指定处理器函数地址。此时需要使用另一个字段：sa_sigaction。换言之，sigaction 结构比 20.13 节所展示的要稍微复杂一些。其完整定义如下：

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

/* Following defines make the union fields look like simple fields
   in the parent structure */

#define sa_handler __sigaction_handler.sa_handler
#define sa_sigaction __sigaction_handler.sa_sigaction
```

结构 sigaction 使用联合体来合并 sa_sigaction 和 sa_handler。（大部分其他 UNIX 实现也采用相同的方式。）之所以使用联合体，是因为对 sigaction()的特定调用只会用到其中的一个字段。（不过，如果天真地认为可以彼此独立地设置 sa_handler 和 sa_sigaction，就有可能导致一些奇怪的 bug。可能的原因是在为不同的信号创建处理器函数时，多次对 sigaction()的调用复用了同一个 sigaction 结构。）

这里是使用 SA_SIGINFO 创建信号处理器函数的一个例子：

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) == -1)
    errExit("sigaction");
```

至于使用 SA_SIGINFO 标志的完整例子，请参考程序清单 22-3 和程序清单 23-5。

更多资源请访问稀酷客(www.ckook.com)

结构 siginfo_t

在以 SA_SIGINFO 标志创建的信号处理器函数中，结构 siginfo_t 是其第 2 个参数，格式如下：

```
typedef struct {
    int     si_signo;         /* Signal number */
    int     si_code;          /* Signal code */
    int     si_trapno;        /* Trap number for hardware-generated signal
                                 (unused on most architectures) */
    union sigval si_value;    /* Accompanying data from sigqueue() */
    pid_t   si_pid;           /* Process ID of sending process */
    uid_t   si_uid;           /* Real user ID of sender */
    int     si_errno;         /* Error number (generally unused) */
    void   *si_addr;          /* Address that generated signal
                                 (hardware-generated signals only) */
    int     si_overrun;       /* Overrun count (Linux 2.6, POSIX timers) */
    int     si_timerid;       /* (Kernel-internal) Timer ID
                                 (Linux 2.6, POSIX timers) */
    long    si_band;          /* Band event (SIGPOLL/SIGIO) */
    int     si_fd;            /* File descriptor (SIGPOLL/SIGIO) */
    int     si_status;        /* Exit status or signal (SIGCHLD) */
    clock_t si_utime;         /* User CPU time (SIGCHLD) */
    clock_t si_stime;         /* System CPU time (SIGCHLD) */
} siginfo_t;
```

要获取<signal.h>对 siginfo_t 的声明，必须将特性测试宏_POSIX_C_SOURCE 的值定义为大于或等于 199309。

如同大部分 UNIX 实现一样，在 Linux 系统中，siginfo_t 结构的很多字段都是联合体，因为对每个信号而言，并非所有字段都有必要。（参考<bits/siginfo.h>中的细节。）

一旦进入信号处理器函数，对结构 siginfo_t 中字段的设置如下。

si_signo

需要为所有信号设置。内含引发处理器函数调用的信号编号——与处理器函数 sig 参数的值相同。

si_code

需要为所有信号设置。如表 21-2 所示，所含代码提供了关于信号来源的深入信息。

si_value

该字段包含调用 sigqueue()发送信号时的伴随数据。22.8.1 节将讨论 sigqueue()。

si_pid

对于经由 kill()或 sigqueue()发送的信号，该字段保存了发送进程的进程 ID。

si_uid

对于经由 kill()或 sigqueue()发送的信号，该字段保存了发送进程的真实用户 ID。系统之所以提供真实用户 ID，是因为其信息量比之有效用户 ID 更为丰富。回忆 20.5 节所述关于信号发送的权限规则，如果有效用户 ID 授予发送者发送信号的权力，那么发送方的用户 ID 必须要么为 0（特权级用户），要么与接收进程的真实用户 ID 或者保存设置用户 ID（saved set-user-ID)相同。这时，接收者了解发送者的真实用户 ID 就很有用，因为它有可能不同于有效用户 ID（例如，如果发送者是一个 set-user-ID 程序）。

si_errno

如果将该字段置为非 0 值，则其所包含为一错误号（类似 errno），标志信号的产生原因。

更多资源请访问稀酷客(www.ckook.com)

Linux 通常不使用该字段。

si_addr

仅针对由硬件产生的 SIGBUG、SIGSEGV、SIGILL 和 SIGFPE 信号设置该字段。对于 SIGBUS 和 SIGSEGV 而言，该字段内含引发无效内存引用的地址。对于 SIGILL 和 SIGFPE 信号，则包含导致信号产生的程序指令地址。

以下各字段均属非标准的 Linux 扩展，仅当 POSIX 定时器（23.6 节）到期而产生信号传递时设置：

si_timerid

内含供内核内部使用的 ID，用以标识定时器。

si_overrun

设置该字段为定时器的溢出次数。

仅当收到 SIGIO 信号（63.3 节）时，才会设置下面两个字段。

si_band

该字段包含与 I/O 事件相关的“带事件”值。（直到 glibc 2.3.2，si_band 的类型都是 int 型。）

si_fd

该字段包含与 I/O 事件相关的文件描述符编号。SUSv3 并未定义这一字段，不过许多其他实现都予以了支持。

仅当收到 SIGCHLD 信号（26.3 节）时，才会对以下各字段进行设置。

si_status

该字段包含子进程的退出状态（当 si_code=CLD_EXITED 时）或者发给子进程的信号编号（即 26.1.3 节所述终止或停止子进程的信号编号）。

si_utime

该字段包含子进程使用的用户 CPU 时间。在版本 2.6 以前，以及 2.6.27 以后的内核版本中，对该字段的度量以系统时钟滴答除以 sysconf（_SC_CLK_TCK）的返回值作为基本单位。而在版本 2.6.27 之前的 2.6 内核中则存在 bug，该字段在报告时间时采用的度量单位为（可由用户配置的）jiffy（10.6 节）。SUSv3 没有定义该字段，但许多其他实现都予以支持。

si_stime

该字段包含了子进程使用的系统 CPU 时间。可参考对 si_utime 的描述。同样，SUSv3 并未定义该字段，不过许多其他实现都予以支持。

si_code 字段提供了关于信号来源的更多信息，其值如表 21-2 所示。表中第 2 列列出的信号特有值（特别是由硬件产生的 4 种信号：SIGBUS、SIGSEGV、SIGILL 和 SIGFPE）不会悉数现身于所有的 UNIX 实现以及硬件架构之上——尽管 Linux 定义了所有常量，而且 SUSv3 也定义了其中的大部分。

关于表 21-2 中所示各值，还需注意以下几点附加说明。

- 值 SI_KERNEL 和 SI_SIGIO 为 Linux 所特有，既未获 SUSv3 定义，也未获其他 UNIX 实现支持。
- SI_SIGIO 仅在 Linux2.2 中用到。自内核 2.4 起，Linux 转而采用表中的 POLL_*常量。

更多资源请访问稀酷客(www.ckook.com)

表 21-2：结构 siginfo_t 中 si_code 字段返回值一览表

信　　号	si_code 的值	信 号 来 源
任意（所有）	SI_ASYNCIO	异步 I/O（AIO）操作已经完成
	SI_KERNEL	从内核发送（例如，来自于终端驱动程序的信号）
	SI_MESGQ	消息到达 POSIX 消息队列（自 Linux 2.6.6）
	SI_QUEUE	利用 sigqueue()从用户进程发出的实时信号
	SI_SIGIO	SIGIO 信号（仅 Linux 2.2 支持）
	SI_TIMER	POSIX（实时）定时器到期
	SI_TKILL	调用 tkill()或 tgkill()的用户进程（自 Linux 2.4.19）
	SI_USER	调用 kill()或 raise()的用户进程
SIGBUS	BUS_ADRALN	无效的地址对齐
	BUS_ADRERR	不存在的物理地址
	BUS_MCEERR_AO	硬件内存错误，动作为可选（自 Linux 2.6.32）
	BUS_MCEERR_AR	硬件内存错误，动作为必需（自 Linux 2.6.32）
	BUS_OBJERR	对象特有的硬件错误
SIGCHLD	CLD_CONTINUED	因 SIGCONT 信号，子进程得以继续执行（自 Linux 2.6.9）
	CLD_DUMPED	子进程异常终止，并产生核心转储
	CLD_EXITED	子进程退出
	CLD_KILLED	子进程异常终止，且不产生核心转储
	CLD_STOPPED	子进程停止
	CLD_TRAPPED	受到跟踪的子进程停止
SIGFPE	FPE_FLTDIV	浮点除 0
	FPE_FLTINV	无效的浮点操作
	FPE_FLTOVF	浮点溢出
	FPE_FLTRES	浮点结果不精确
	FPE_FLTUND	浮点下溢
	FPE_INTDIV	整型除 0
	FPE_INTOVF	整型溢出
	FPE_SUB	下标超出范围
SIGILL	ILL_BADSTK	内部栈错误
	ILL_COPROC	协处理器错误
	ILL_ILLADR	非法地址模式
	ILL_ILLOPC	非法操作码
	ILL_ILLOPN	非法操作数
	ILL_ILLTRP	非法陷入
	ILL_PRVOPC	特权级操作码
	ILL_PRVREG	特权级寄存器

更多资源请访问稀酷客(www.ckook.com)

续表

信　　号	si_code 的值	信 号 来 源
SIGPOLL/SIGIO	POLL_ERR	I/O 错误
	POLL_HUP	设备断开
	POLL_IN	输入数据有效
	POLL_MSG	输入消息有效
	POLL_OUT	输出缓冲区有效
	POLL_PRI	高优先级输入有效
SIGSEGV	SEGV_ACCERR	映射对象的无效权限
	SEGV_MAPERR	未映射为对象的地址
SIGTRAP	TRAP_BRANCH	进程分支陷入
	TRAP_BRKPT	进程断点
	TRAP_HWBKPT	硬件断点/监测点
	TRAP_TRACE	进程跟踪陷入

SUSv4 定义了功用与 psignal()（20.8 节）相仿的 psiginfo()函数。函数 psiginfo()带有两个参数，分别是指向 siginfo_t 结构的指针和一个消息字符串。该函数在标准错误设备上输出字符串消息，接着显示描述于 siginfo_t 结构中的信号信息。glibc 自 2.10 版开始提供 psiginfo()函数。glibc 实现会显示信号的描述信息及来源（根据 si_code 字段所示），对于某些信号，还会列出 siginfo_t 结构中的其他字段。函数 psiginfo()是 SUSv4 中的新丁，并非所有系统都予以支持。

参数 ucontext

以 SA_SIGINFO 标志所创建的信号处理器函数，其最后一个参数是 ucontext，一个指向 ucontext_t 类型结构（定义于<ucontext.h>）的指针。（因为 SUSv3 并未规定该参数的任何细节，所以将其定义为 void 类型指针。）该结构提供了所谓的用户上下文信息，用于描述调用信号处理器函数前的进程状态，其中包括上一个进程信号掩码以及寄存器的保存值，例如程序计数器（cp）和栈指针寄存器（sp）。信号处理器函数很少用到此类信息，所以此处也略而不论。

使用结构 ucontext_t 的其他函数有 getcontext()、makecontext()、setcontext()和 swapcontext()，分别对应的功能是允许进程去接收、创建、改变以及交换执行上下文。（这些操作有点类似于 setjmp()和 longjmp()，但更为通用。）可以使用这些函数来实现协程（coroutines），令进程的执行线程在两个（或多个）函数之间交替。SUSv3 规定了这些函数，但将它们标记为已废止。SUSv4 则将其删去，并建议使用 POSIX 线程来重写旧有的应用程序。glibc 手册页提供了关于这些函数的深入信息。

更多资源请访问稀酷客(www.ckook.com)

21.5 系统调用的中断和重启

考虑如下场景。

1. 为某信号创建处理器函数。
2. 发起一个阻塞的系统调用（blocking system call），例如，从终端设备调用的 read()就会阻塞到有数据输入为止。
3. 当系统调用遭到阻塞时，之前创建了处理器函数的信号传递了过来，随即引发对处理器函数的调用。

信号处理器返回后又会发生什么？默认情况下，系统调用失败，并将 errno 置为 EINTR。这是一种有用的特性。23.3 节将会描述如何使用定时器（会产生 SIGALRM 信号）来设置像 read()之类阻塞系统调用的超时。

不过，更为常见的情况是希望遭到中断的系统调用得以继续运行。为此，可在系统调用遭信号处理器中断的事件中，利用如下代码来手动重启系统调用。

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue;                   /* Do nothing loop body */

if (cnt == -1)                  /* read() failed with other than EINTR */
    errExit("read");
```

如果需要频繁使用上述代码，那么定义成如下宏会很方便：

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);
```

使用该宏，可以将早先对 read()的调用改写如下 ：

```
NO_EINTR(cnt = read(fd, buf, BUF_SIZE));

if (cnt == -1)                  /* read() failed with other than EINTR */
    errExit("read");
```

GNU C 库提供了一个（非标准）宏，其作用与定义于<unistd.h>中的 NO_EINTR()相同。该宏名为 TEMP_FAILURE_RETRY()，定义特性测试宏_GNU_SOURCE 后即可使用。

即使采用了类似 NO_EINTR()这样的宏，让信号处理器来中断系统调用还是颇为不便，因为只要有意重启阻塞的调用，就需要为每个阻塞的系统调用添加代码。反之，可以调用指定了 SA_RESTART 标志的 sigaction()来创建信号处理器函数，从而令内核代表进程自动重启系统调用，还无需处理系统调用可能返回的 EINTR 错误。

标志 SA_RESTART 是针对每个信号的设置。换言之，允许某些信号的处理器函数中断阻塞的系统调用，而其他系统调用则可以自动重启。

SA_RESTART 标志对哪些系统调用（和库函数）有效

不幸的是，并非所有的系统调用都可以通过指定 SA_RESTART 来达到自动重启的目的。究其原因，有部分历史因素。

- 4.2BSD 引入了重启系统调用的概念，包括中断对 wait()和 waitpid()的调用，以及如下 I/O 系统调用：read()、readv()、write()和阻塞的 ioctl()操作。I/O 系统调用都是可中断的，所以只有在操作“慢速（slow）”设备时，才可以利用 SA_RESTART 标志来自动重启调用。慢速设备包括终端（terminal）、管道（pipe）、FIFO 以及套接字（socket）。对于这

更多资源请访问稀酷客(www.ckook.com)

些文件类型，各种 I/O 操作都有可能堵塞。（相比之下，磁盘文件并未沦入慢速设备之列，因为借助于缓冲区高速缓存，磁盘 I/O 请求一般都可以立即得到满足。当出现磁盘 I/O 请求时，内核会令该进程休眠，直至完成 I/O 动作为止。）

- 其他大量阻塞的系统调用则继承自 System V，在其初始设计中并未提供重启系统调用的功能。

在 Linux 中，如果采用 SA_RESTART 标志来创建系统处理器函数，则如下阻塞的系统调用（以及构建于其上的库函数）在遭到中断时是可以自动重启的。

- 用来等待子进程（26.1 节）的系统调用：wait()、waitpid()、wait3()、wait4()和 waitid()。
- 访问慢速设备时的 I/O 系统调用：read()、readv()、write()、writev()和 ioctl()。如果在收到信号时已经传递了部分数据，那么还是会中断输入输出系统调用，但会返回成功状态：一个整型值，表示已成功传递数据的字节数。
- 系统调用 open()，在可能阻塞的情况下（例如，如 44.7 节所述，在打开 FIFO 时）。
- 用于套接字的各种系统调用：accept()、accept4()、connect()、send()、sendmsg()、sendto()、recv()、recvfrom()和 recvmsg()。（在 Linux 中，如果使用 setsockopt()来设置超时，这些系统调用就不会自动重启。更多细节请参考 signal(7)手册页。）
- 对 POSIX 消息队列进行 I/O 操作的系统调用：mq_receive()、mq_timedreceive()、mq_send()和 mq_timedsend()。
- 用于设置文件锁的系统调用和库函数：flock()、fcntl()和 lockf()。
- Linux 特有系统调用 futex()的 FUTEX_WAIT 操作。
- 用于递减 POSIX 信号量的 sem_wait()和 sem_timedwait()函数。（在一些 UNIX 实现上，如果设置了 SA_RESTART 标志，sem_wait()就会重启。）
- 用于同步 POSIX 线程的函数：pthread_mutex_lock()、pthread_mutex_trylock()、pthread_mutex_timedlock()、pthread_cond_wait()和 pthread_cond_timedwait()。

内核 2.6.22 之前，不管是否设置了 SA_RESTART 标志，futex()、sem_wait()和 sem_timedwait()遭到中断时总是产生 EINTR 错误。

以下阻塞的系统调用（以及构建于其上的库函数）则绝不会自动重启（即便指定了 SA_RESTART）。

- poll()、ppoll()、select()和 pselect()这些 I/O 多路复用调用。（SUSv3 明文规定，无论设置 SA_RESTART 标志与否，都不对 select()和 pselect()遭处理器函数中断时的行为进行定义。）
- Linux 特有的 epoll_wait()和 epoll_pwait()系统调用。
- Linux 特有的 io_getevents()系统调用。
- 操作 System V 消息队列和信号量的阻塞系统调用：semop()、semtimedop()、msgrcv()和 msgsnd()。（虽然 System V 原本并未提供自动重启系统调用的功能，但在某些 UNIX 实现上，如果设置了 SA_RESTART 标志，这些系统调用还是会自动重启。）
- 对 inotify 文件描述符发起的 read()调用。
- 用于将进程挂起指定时间的系统调用和库函数：sleep()、nanosleep()和 clock_nanosleep()。
- 特意设计用来等待某一信号到达的系统调用：pause()、sigsuspend()、sigtimedwait()和 sigwaitinfo()。

为信号修改 SA_RESTART 标志

函数 siginterrupt()用于改变信号的 SA_RESTART 设置。

更多资源请访问稀酷客(www.ckook.com)

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

Returns 0 on success, or −1 on error

若参数 flag 为真（1），则针对信号 sig 的处理器函数将会中断阻塞的系统调用的执行。如果 flag 为假（0），那么在执行了 sig 的处理器函数之后，会自动重启阻塞的系统调用。

函数 siginterrupt()的工作原理是：调用 sigaction()获取信号当前处置的副本，调整自结构 oldact 中返回的 SA_RESTART 标志，接着再次调用 sigaction()来更新信号处置。

SUSv4 标记 sigterrupt()为已废止，并推荐使用 sigaction()加以替代。

对于某些 Linux 系统调用，未处理的停止信号会产生 EINTR 错误

在 Linux 上，即使没有信号处理器函数，某些阻塞的系统调用也会产生 EINTR 错误。如果系统调用遭到阻塞，并且进程因信号（SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU）而停止，之后又因收到 SIGCONT 信号而恢复执行时，就会发生这种情况。

以下系统调用和函数具有这一行为：epoll_pwait()、epoll_wait()、对 inotify 文件描述符执行的 read()调用、semop()、semtimedop()、sigtimedwait()和 sigwaitinfo()。

内核 2.6.24 之前，poll()也曾存在这种行为，2.6.22 之前的 sem_wait()、sem_timedwait()、futex(FUTEX_WAIT)，2.6.9 之前的 msgrcv()和 msgsnd()，以及 Linux 2.4 及其之前的 nanosleep()也同样如此。

在 Linux 2.4 及其之前的版本中，也可以以这种方式来中断 sleep()，但是不会返回错误值，而是返回休眠所剩余的秒数。

这种行为的结果是，如果程序可能因信号而停止和重启，那么就需要添加代码来重新启动这些系统调用，即便该程序并未为停止信号设置处理器函数。

21.6 总结

本章讨论了影响信号处理器函数操作与设计的一系列因素。

由于没有对信号排队，故而在为处理器编码时，有时必须要考虑特定类型信号多次发生的可能性，即使之前信号只产生过一次。可重入问题会影响到对全局变量的修改方式，还限制了可从信号处理器函数中安全调用的函数范围。

除了返回之外，信号处理器函数的终止还存在多种其他方法，其中包括：调用_exit()，发送信号来终止进程（kill()、raise()或 abort()），或者执行非本地跳转。借助于 sigsetjmp()和 siglongjmp()，可以在执行非本地跳转时为程序提供处理信号掩码的显式控制手段。

可以使用 sigaltstack()来为进程定义备选信号栈。这是调用信号处理器函数时，用来替代标准进程栈的一块内存。当标准栈因增长过大（内核会在此时向进程发送 SIGSEGV 信号）而消耗殆尽时，备选栈就特别有用。

如果在调用 sigaction()时设置了 SA_SIGINFO 标志，那么所创建的信号处理器函数就能接收信号的附加信息。siginfo_t 结构提供了这些信息，其地址则传递给信号处理器作为参数。

如果信号处理器函数中断了阻塞的系统调用，系统调用会产生 EINTR 错误。利用这种特性，就可以为阻塞的系统调用设置一个定时器。如果有意，可以手动重启遭到中断的系统调

更多资源请访问稀酷客(www.ckook.com)

用。另外，在调用 sigaction()创建信号处理器函数时，如果设置了 SA_RESTART 标志，那么大部分（但非全部）系统调用都将会自动重启。

更多信息

参考 20.15 节所列信息来源。

21.7 练习

21.1. 实现 abort()。

更多资源请访问稀酷客(www.ckook.com)

第 22 章

信号：高级特性

本章是“信号”主题系列讨论（始于第 20 章）的完结篇，涵盖了一些更为高级的议题，如下所示。

- 核心转储文件。
- 与信号的传递、处置及处理相关的特殊情况。
- 信号的同步产生和异步产生。
- 信号的传递时机及传递顺序。
- 信号处理器函数对系统调用的中断，以及如何自动重启遭到中断的系统调用。
- 实时信号。
- 用 sigsuspend()来设置进程信号掩码并等待信号到达。
- 用 sigwaitinfo()（和 sigtimedwait()）同步等待信号到达。
- 用 signalfd()从一个文件描述符中接收信号。
- 较老的 BSD 版信号 API 和 System V 版信号 API。

22.1 核心转储文件

特定信号会引发进程创建一个核心转储文件并终止运行（参考表 20-1）。所谓核心转储是内含进程终止时内存映像的一个文件。（术语 core 源于一种老迈的内存技术。）将该内存映像加载到调试器中，即可查明信号到达时程序代码和数据的状态。

引发程序生成核心转储文件的方式之一是键入退出字符（通常为 Control-\），从而生成 SIGQUIT 信号。

```
$ ulimit -c unlimited                    Explained in main text
$ sleep 30
Type Control-\
Quit (core dumped)
$ ls -l core                             Shows core dump file for sleep(1)
-rw-------   1 mtk    users     57344 Nov 30 13:39 core
```

本例中，当检测出子进程（运行 sleep 命令的进程）为 SIGQUIT 信号所杀，并生成核心

更多资源请访问稀酷客(www.ckook.com)

转储文件时，shell 会显示“Quit（core dump）”消息。

核心转储文件创建于进程的工作目录中，名为 core。这是核心转储文件的默认位置和名称。稍后，将解释如何改变这些默认值。

借助于许多实现所提供的工具（例如 FreeBSD 和 Solaris 中的 gcore），可获取某一正在运行进程的核心转储文件。Linux 系统也有类似功能，使用 gdb 去连接（attach）一个正在运行的进程，然后运行 gcore 命令。

不产生核心转储文件的情况

以下情况不会产生核心转储文件。

- 进程对于核心转储文件没有写权限。造成这种情况的原因有进程对将要创建核心转储文件的所在目录可能没有写权限，或者是因为存在同名（且不可写，亦或非常规类型，例如，目录或符号链接）的文件。
- 存在一个同名、可写的普通文件，但指向该文件的（硬）链接数超过一个。
- 将要创建核心转储文件的所在目录并不存在。
- 把进程“核心转储文件大小”这一资源限制置为 0。36.3 节将就这一限制（RLIMIT_CORE）进行详细讨论。上例就使用了 ulimit 命令（C shell 中为 limit 命令）来取消对核心转储文件大小的任何限制。
- 将进程“可创建文件的大小”这一资源限制设置为 0。36.3 节将描述这一限制（RLIMIT_FSIZE）。
- 对进程正在执行的二进制可执行文件没有读权限。这样[1]就防止了用户借助于核心转储文件来获取本无法读取的程序代码。
- 以只读方式挂载当前工作目录所在的文件系统，或者文件系统空间已满，又或者 i-node 资源耗尽。还有一种情况，即用户已经达到其在该文件系统上的配额限制。
- Set-user-ID（set-group-ID）程序在由非文件属主（或属组）执行时，不会产生核心转储文件。这可以防止恶意用户将一个安全程序的内存转储出来，再针对诸如密码之类的敏感信息进行刺探。

借助于 Linux 专有系统调用 prctl()的 PR_SET_DUMPABLE 操作，可以为进程设置 dumpable 标志。当非文件属主（或属组）运行 set-user-ID（set-group-ID）程序时，如设置该标志即可生成核心转储文件。PR_SET_DUMPABLE 操作始见于 Linux 2.4，更多详细信息参见 prctl(2)手册页。另外，始于内核版本 2.6.13，针对 set-user-ID 和 set-group-ID 进程是否产生核心转储文件，/proc/sys/fs/suid_dumpable 文件开始提供系统级控制。详情参见 proc(5)手册页。

始于内核版本 2.6.23，利用 Linux 特有的/proc/PID/coredump_filter，可以对写入核心转储文件的内存映射类型（第 49 章将解释内存映射）施以进程级控制。该文件中的值是一个 4 位掩码，分别对应于 4 种类型的内存映射：私有匿名映射、私有文件映射、共享匿名映射以及共享文件映射。文件默认值提供了传统的 Linux 行为：仅对私有匿名映射和共享匿名映射进行转储。详情参见 core(5)手册页。

1 译者注：指不生成核心转储文件。

更多资源请访问稀酷客(www.ckook.com)

为核心转储文件命名：/proc/sys/kernel/core_pattern

从 Linux 版本 2.6 开始，可以根据 Linux 特有的/proc/sys/kernel/core_pattern 文件所包含的格式化字符串来控制对系统上生成的所有核心转储文件的命名。默认情况下，该文件所含字符串为 core。特权级用户可以将该文件内容定义为包含表 22-1 所列的任一格式说明符，待实际命名时再以表中右列所示相应值加以替换。此外，允许字符串中包含斜线（/）。换言之，处在控制范围之内的，不仅包括核心文件的名称，还包括核心文件的所在（绝对或相对）目录。替换所有格式说明符后，由此生成的路径名字符串长度至多可达 128 个字符（Linux 2.6.19 之前为 64 个字符），超出部分将予以截断。

Linux 从内核版本 2.6.19 开始支持 core_pattern 文件的另一种语法。如果该文件包含一个以管道符（|）为首的字符串，那么会将该文件的剩余字符串视为一个程序，其可选参数可包含表 22-1 所示的%说明符——当进程转储核心文件时，将执行该程序。并且会将核心转储至该程序的标准输入，而非一个文件。详情请参考 core(5)手册页。

其他一些 UNIX 实现也提供了类似于 core_pattern 的机制。例如，在 BSD 一派中，会将程序名追加到文件名尾部，形如 core.progname。Solaris 提供了一个工具（coreadm），允许由用户来选择核心转储文件的名称和存放目录。

表 22-1：服务于/proc/sys/kernel/core_pattern 的文件说明符

说 明 符	替 代 为
%c	对核心文件大小的资源软限制（字节数；始于 Linux 2.6.24）
%e	可执行文件名（不含路径前缀）
%g	遭转储进程的实际组 ID
%h	主机系统的名称
%p	遭转储进程的进程 ID
%s	导致进程终止的信号编号
%t	转储时间，始于 Epoch，以秒为单位
%u	遭转储进程的实际用户 ID
%%	单个%字符

22.2 传递、处置及处理的特殊情况

本节讨论了针对特定信号，适用于其传递、处置以及处理方面的特殊规则。

SIGKILL 和 SIGSTOP

SIGKILL 信号的默认行为是终止一个进程，SIGSTOP 信号的默认行为是停止一个进程，二者的默认行为均无法改变。当试图用 signal()和 sigaction()来改变对这些信号的处置时，将总是返回错误。同样，也不能将这两个信号阻塞。这是一个深思熟虑的设计决定。不允许修改这些信号的默认行为，这也意味着总是可以利用这些信号来杀死或者停止一个

更多资源请访问稀酷客(www.ckook.com)

失控进程。

SIGCONT 和停止信号

如前所述，可使用 SIGCONT 信号来使某些（因接收 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 信号而）处于停止状态的进程得以继续运行。由于这些停止信号具有独特目的，所以在某些情况下内核对它们的处理方式将有别于其他信号。

如果一个进程处于停止状态，那么一个 SIGCONT 信号的到来总是会促使其恢复运行，即使该进程正在阻塞或者忽略 SIGCONT 信号。该特性之所以必要，是因为如果要恢复这些处于停止状态的进程，舍此之外别无他法。（如果处于停止状态的进程正在阻塞 SIGCONT 信号，并且已经为 SIGCONT 信号建立了处理器函数，那么在进程恢复运行后，只有当取消了对 SIGCONT 的阻塞时，进程才会去调用相应的处理器函数。）

> 如果有任一其他信号发送给了一个已经停止的进程，那么在进程收到 SIGCONT 信号而恢复运行之前，信号实际上并未传递。SIGKILL 信号则属于例外，因为该信号总是会杀死进程，即使进程目前处于停止状态。

每当进程收到 SIGCONT 信号时，会将处于等待状态的停止信号丢弃（即进程根本不知道这些信号）。相反，如果任何停止信号传递给了进程，那么进程将自动丢弃任何处于等待状态的 SIGCONT 信号。之所以采取这些步骤，意在防止之前发送的一个停止信号会在随后撤销 SIGCONT 信号的行为，反之亦然。

由终端产生的信号若已被忽略，则不应改变其信号处置

如果程序在执行时发现，已将对由终端产生信号的处置置为了 SIG_IGN（忽略），那么程序通常不应试图去改变信号处置。这并非系统的硬性规定，而是编写应用程序时所应遵循的惯例，34.7.3 节将解释其理由。与之相关的信号有：SIGHUP、SIGINT、SIGQUIT、SIGTTIN、SIGTTOU 和 SIGTSTP。

22.3 可中断和不可中断的进程睡眠状态

前文曾指出，SIGKILL 和 SIGSTOP 信号对进程的作用是立竿见影的。对于这一论断，此处要加入一条限制。内核经常需要令进程进入休眠，而休眠状态又分为两种。

- TASK_INTERRUPTIBLE：进程正在等待某一事件。例如，正在等待终端输入，等待数据写入当前的空管道，或者等待 System V 信号量值的增加。进程在该状态下所耗费的时间可长可短。如果为这种状态下的进程产生一个信号，那么操作将中断，而传递来的信号将唤醒进程。ps(1)命令在显示处于 TASK_INTERRUPTIBLE 状态的进程时，会将其 STAT（进程状态）字段标记为字母 S。
- TASK_UNINTERRUPTIBLE：进程正在等待某些特定类型的事件，比如磁盘 I/O 的完成。如果为这种状态下的进程产生一个信号，那么在进程摆脱这种状态之前，系统将不会把信号传递给进程。ps(1)命令在显示处于 TASK_UNINTERRUPTIBLE 状态的进程时，会将其 STAT 字段标记为字母 D。

因为进程处于 TASK_UNINTERRUPTIBLE 状态的时间通常转瞬即逝，所以系统在进程脱离该状态时传递信号的现象也不易于被发现。然而，在极少数情况下，进程可能会因硬件故

更多资源请访问稀酷客(www.ckook.com)

障、NFS 问题或者内核缺陷而在该状态下保持挂起。这时，SIGKILL 将不会终止挂起进程。如果问题诱因无法得到解决，那么就只能通过重启系统来消灭该进程。

大多数UNIX系统实现都支持TASK_INTERRUPTIBLE和TASK_UNINTERRUPTIBLE状态。从内核 2.6.25 开始，Linux 加入第三种状态来解决上述挂起进程的问题。

- TASK_KILLABLE：该状态类似于 TASK_UNINTERRUPTIBLE，但是会在进程收到一个致命信号（即一个杀死进程的信号）时将其唤醒。在对内核代码的相关部分进行改造后，就可使用该状态来避免各种因进程挂起而重启系统的情况。这时，向进程发送一个致命信号就能杀死进程。为使用 TASK_KILLABLE 而进行代码改造的首个内核模块是 NFS。

22.4 硬件产生的信号

硬件异常可以产生 SIGBUS、SIGFPE、SIGILL，和 SIGSEGV 信号，调用 kill()函数来发送此类信号是另一种途径，但较为少见。SUSv3 规定，在硬件异常的情况下，如果进程从此类信号的处理器函数中返回，亦或进程忽略或阻塞了此类信号，那么进程的行为未定义。原因如下。

- 从信号处理器中返回：假设机器语言指令产生了上述信号之一，并因此而调用了信号处理器函数。当从处理器函数正常返回后，程序会尝试从其中断处恢复执行。可当初引发信号产生的恰恰正是这条指令，所以信号会再次“光临”。故事的结局通常是，程序进入无限循环，重复调用信号处理器函数。
- 忽略信号：忽略因硬件而产生的信号于情理不合，试想算术异常之后，程序应当如何继续执行呢？无法明确。当由于硬件异常而产生上述信号之一时，Linux 会强制传递信号，即使程序已经请求忽略此类信号。
- 阻塞信号。与上一种情况一样，阻塞因硬件而产生的信号也不合情理：不清楚程序随后应当如何继续执行。在 2.4 以及更早的版本中，Linux 内核仅会将阻塞硬件产生信号的种种企图一一忽略，信号无论如何都会传递给进程，随后要么进程终止，要么信号处理器会捕获信号——在程序安装有信号处理器的情况下。始于 Linux 2.6，如果信号遭到阻塞，那么该信号总是会立刻杀死进程，即使进程已经为此信号安装了处理器函数。（对于因硬件而产生的信号，Linux 2.6 之所以会改变对其处于阻塞状态下的处理方式，是由于 Linux 2.4 的行为中隐藏有缺陷，并可能在多线程程序中引起死锁。）

随本书发布源码中的 signals/demo_SIGFPE.c 程序就展示了忽略或者阻塞 SIGFPE 信号的后果，或者可正常返回的处理器将其捕获的结果。

正确处理硬件产生信号的方法有二：要么接受信号的默认行为（进程终止）；要么为其编写不会正常返回的处理器函数。除了正常返回之外，终结处理器执行的手段还包括调用_exit()以终止进程，或者调用 siglongjmp()（21.2.1 节），确保将控制传递回程序中（产生信号的指令位置之外）的某一位置。

22.5 信号的同步生成和异步生成

前文已然论及，进程一般无法预测其接收信号的时间。要证实这一点，需要对信号的同

更多资源请访问稀酷客(www.ckook.com)

步生成和异步生成加以区分。

截止目前所探讨的均属于信号的异步生成，即引发信号产生（无论信号发送者是内核还是另一进程）的事件，其发生与进程的执行无关。（例如，用户输入中断字符，或者子进程终止。）对于异步产生的信号，本节起始处的论断并非虚言。

然而，有时候信号的产生是由进程本身的执行造成的，前面就曾提及两个这样的例子。

- 执行特定的机器语言指令，可导致硬件异常，并因此而产生 22.4 节所述的硬件产生信号（SIGBUS、SIGFPE、SIGILL、SIGSEGV 和 SIGEMT）。
- 进程可以使用 raise()、kill()或者 killpg()向自身发送信号。

在这些情况下，信号的产生就是同步的——会立即传递信号（除非该信号遭到阻塞，但还要参考 22.4 节就阻塞硬件产生信号而展开的讨论）。换言之，本节开始处的论断则并不成立。对于同步产生的信号而言，其传递不但可以预测，而且可以重现。

注意，同步是对信号产生方式的描述，并不针对信号本身。所有的信号既可以同步产生（例如，进程使用 kill()向自身发送信号），也可以异步产生（例如，由另一进程使用 kill()来发送信号）。

22.6 信号传递的时机与顺序

本节的主题有二。其一，具体于何时去传递一个处于等待状态的信号；其二，对于多个遭到阻塞，且处于等待状态的信号一旦同时解除阻塞，将会发生什么情况？

何时传递一个信号？

如 22.5 节所述，同步产生的信号会立即传递。例如，硬件异常会触发一个即时信号，而当进程使用 raise()向自身发送信号时，信号会在 raise()调用返回前就已经发出。

当异步产生一个信号时，即使并未将其阻塞，在信号产生和实际传递之间仍可能会存在一个（瞬时）延迟。在此期间，信号处于等待状态。这是因为内核将等待信号传递给进程的时机是，该进程正在执行，且发生由内核态到用户态的下一次切换时。实际上，这意味着在以下时刻才会传递信号。

- 进程在前度超时后，再度获得调度时（即，在一个时间片的开始处）。
- 系统调用完成时（信号的传递可能引起正在阻塞的系统调用过早完成）。

解除对多个信号的阻塞时，信号的传递顺序

如果进程使用 sigprocmask()解除了对多个等待信号的阻塞，那么所有这些信号会立即传递给该进程。

就目前的 Linux 实现而言，Linux 内核按照信号编号的升序来传递信号。例如，如果对处于等待状态的信号 SIGINT（信号编号为 2）和 SIGQUIT（信号编号为 3）同时解除阻塞，那么无论这两个信号的产生次序如何，SIGINT 都将先于 SIGQUIT 而传递。

然而，也不能对传递（标准）信号的特定顺序产生任何依赖，因为 SUSv3 规定，多个信号的传递顺序由系统实现决定。（该条款仅适用于标准信号。如 22.8 节所述，实时信号的相关标准规定，对于解除阻塞的实时信号而言，其传递顺序必须得到保障。）

当多个解除了阻塞的信号正在等待传递时，如果在信号处理器函数执行期间发生了内核态和用户态之间的切换，那么将中断此处理器函数的执行，转而去调用第二个信号处理器函数（如此递进），如图 22-1 所示。

更多资源请访问棒酷客(www.ckook.com)

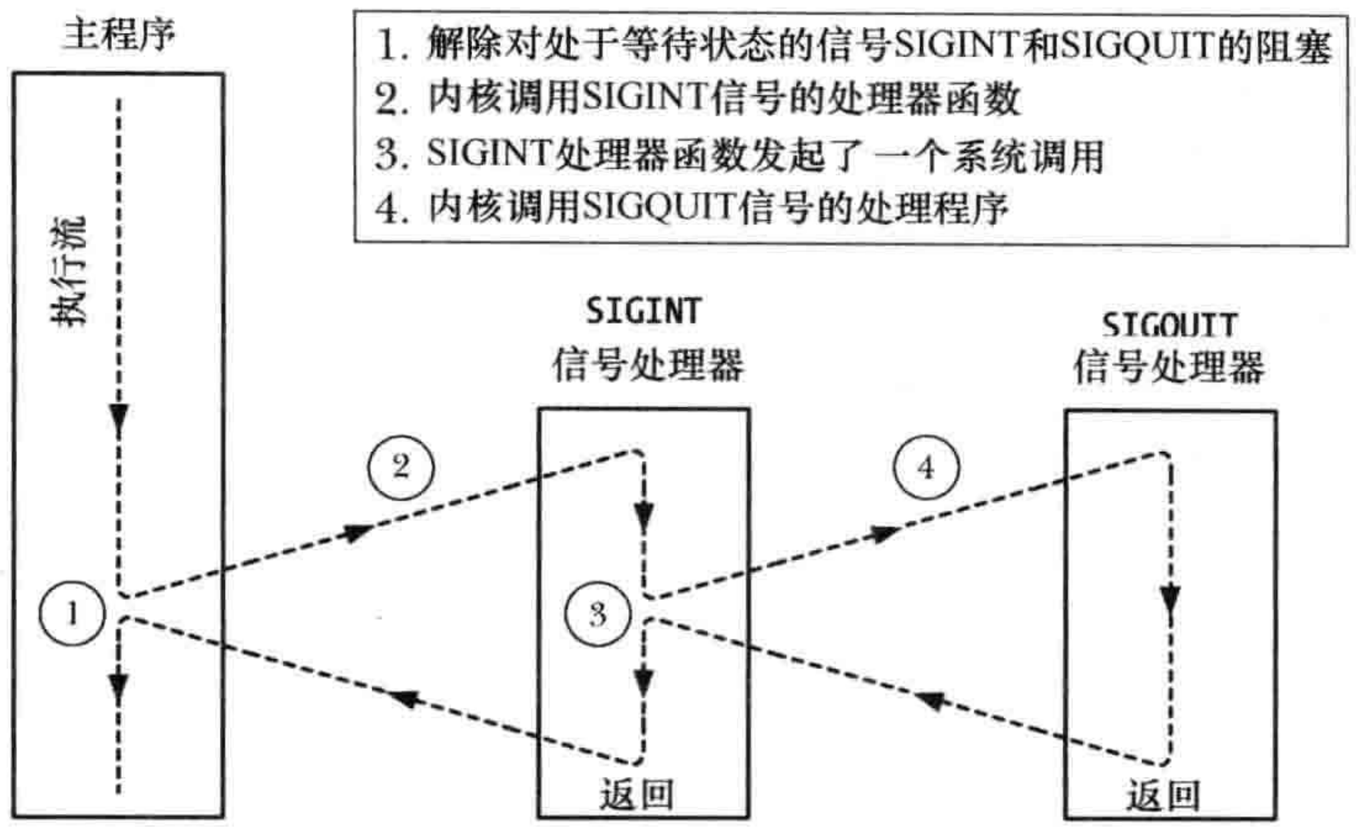

图 22-1：对多个解除阻塞信号的传递

22.7 signal()的实现及可移植性

本节展示了如何使用 sigaction()来实现 signal()。实现虽然简单明了，但还需要顾及这一事实，由于历史沿革和 UNIX 实现之间的差异，signal()曾具有各种不同的语义。尤其是，信号的早期实现并不可靠，这意味着：

- 刚一进入信号处理器，会将信号处置重置为其默认行为。（这对应于 20.13 节描述的 SA_RESETHAND 标志。）要想在同一信号“再度光临”时再次调用该信号处理器函数，程序员必须在信号处理器内部调用 signal()，以显式重建处理器函数。这种情况存在一个问题：在进入信号处理器和重建处理器之间存在一个短暂的窗口期，而如果同一信号在此期间再度来袭，那么将只能按照其默认处置来进行处理。
- 在信号处理器执行期间，不会对新产生的信号进行阻塞。（这对应于 20.12 节描述的 SA_NODEFER 标志。）这意味着，如果在某一信号处理器函数执行期间，同类信号再度光顾，那么将对该处理器函数进行递归调用。假定一串信号中彼此的时间间隔足够短，那么对处理器函数的递归调用将可能导致堆栈溢出。

除了不可靠之外，早期的 UNIX 实现并未提供系统调用的自动重启功能（即，21.5 节所述 SA_RESTART 标志的相关行为）。

4.2BSD 针对可靠信号的实现纠正了这些限制，其他一些 UNIX 实现也纷纷效仿。然而，时至今日，这些早期语义依然存在于 System V 的 signal()实现之中。更有甚者，诸如 SUSv3 和 C99 之类的当代标准对 signal()的这些方面也有意不予规范。

整合上述信息，对 signal()的实现如程序清单 22-1 所示。该实现默认将提供信号的现代语义。如果编译时带有–DOLD_SIGNAL 选项，那么将提供早期的不可靠信号语义，且不能启用系统调用的自动重启功能。

程序清单 22-1：signal()的实现之一

signals/signal.c

```
#include <signal.h>

typedef void (*sighandler_t)(int);
```

```
sighandler_t
signal(int sig, sighandler_t handler)
{
    struct sigaction newDisp, prevDisp;

    newDisp.sa_handler = handler;
    sigemptyset(&newDisp.sa_mask);
#ifdef OLD_SIGNAL
    newDisp.sa_flags = SA_RESETHAND | SA_NODEFER;
#else
    newDisp.sa_flags = SA_RESTART;
#endif

    if (sigaction(sig, &newDisp, &prevDisp) == -1)
        return SIG_ERR;
    else
        return prevDisp.sa_handler;
}
```

signals/signal.c

glibc 的一些细节

随着时间推移，glibc 对 signal()库函数的实现也历经变化。较新版本（glibc 2 及更高版本）的函数库默认提供现代语义。而老版本则提供早期的不可靠（System V-兼容）语义。

> Linux 内核将 signal()实现为系统调用，并提供较老的、不可靠语义。然而，glibc 库则利用 sigaction()实现了 signal()库函数，从而将 signal()系统调用旁路。

如果执意在现代 glibc 版本中使用不可靠信号语义，那么可以显式以（非标准的）sysv_signal()函数来替代对 signal()的调用。

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

void ( *sysv_signal(int sig, void (*handler)(int)) ) (int);
```

Returns previous signal disposition on success, or SIG_ERR on error

sysv_signal()函数的参数与 signal()函数相同。

若编译程序时并未定义_BSD_SOURCE 特性测试宏，则 glibc 会隐式将所有 signal()调用重新定义为 sysv_signal()调用，亦即启用 signal()的不可靠语义。默认情况下会定义_BSD_SOURCE，但是（除非显式定义了_BSD_SOURCE）如果编译程序时定义了诸如_SVID_SOURCE 或_XOPEN_SOURCE 之类的其他特性测试宏，那么对_BSD_SOURCE 的默认定义将会失效。

sigaction()是建立信号处理器的首选 API

鉴于上述 System V 与 BSD 之间（以及 glibc 新老版本之间）的可移植性问题，应当坚持使用 sigaction()而非 signal()来建立信号处理器，这不失为一种稳妥之举。本书剩下部分都将遵循这一做法。（另一种选择是，编写类似于程序清单 22-1 的 signal()版本，精确设定所需要的标志，供应用程序内部使用。）不过，还应注意，使用 signal()将信号处置设置为 SIG_IGN 或者 SIG_DFL 的手法具有良好的可移植性（程序也更为简短），所以也很常用。

更多资源请访问稀酷客(www.ckook.com)

22.8 实时信号

定义于 POSIX.1b 中的实时信号，意在弥补对标准信号的诸多限制。较之于标准信号，其优势如下所示。

- 实时信号的信号范围有所扩大，可应用于应用程序自定义的目的。而标准信号中可供应用随意使用的信号仅有两个：SIGUSR1 和 SIGUSR2。
- 对实时信号所采取的是队列化管理。如果将某一实时信号的多个实例发送给一进程，那么将会多次传递信号。相反，如果某一标准信号已经在等待某一进程，而此时即使再次向该进程发送此信号的实例，信号也只会传递一次。
- 当发送一个实时信号时，可为信号指定伴随数据（一整型数或者指针值），供接收进程的信号处理器获取。
- 不同实时信号的传递顺序得到保障。如果有多个不同的实时信号处于等待状态，那么将率先传递具有最小编号的信号。换言之，信号的编号越小，其优先级越高。如果是同一类型的多个信号在排队，那么信号（以及伴随数据）的传递顺序与信号发送来时的顺序保持一致。

SUSv3 要求，实现所提供的各种实时信号不得少于_POSIX_RTSIG_MAX（定义为 8）个。Linux 内核则定义了 32 个不同的实时信号，编号范围为 32～63。<signal.h>头文件所定义的 RTSIG_MAX 常量则表征实时信号的可用数量，而此外所定义的常量 SIGRTMIN 和 SIGRTMAX 则分别表示可用实时信号编号的最小值和最大值。

> 采用 LinuxThreads 线程实现的系统将 SIGRTMIN 定义为 35（而非 32），这是因为 LinuxThreads 内部使用了前三个实时信号。而采用 NPTL 线程实现的系统则将 SIGRTMIN 定义为 34，因为 NPTL 内部使用了前两个实时信号。

对实时信号的区分方式有别于标准信号，不再依赖于所定义常量的不同。然而，程序员不应将实时信号编号的整型值在应用程序代码中写死，因为实时信号的范围因 UNIX 实现的不同而各异。与之相反，指代实时信号编号则可以采用 SIGRTMIN+x 的形式。例如，表达式（SIGRTMIN + 1）就表示第二个实时信号。

注意，SUSv3 并未要求 SIGRTMAX 和 SIGRTMIN 是简单的整数值。可以将其定义为函数（就像 Linux 中那样）。这也意味着，不能编写如下代码以供预处理器处理：

```
#if SIGRTMIN+100 > SIGRTMAX              /* WRONG! */
#error "Not enough realtime signals"
#endif
```

相反，必须在运行时执行等效检查。

对排队实时信号的数量限制

排队的实时信号（及其相关数据）需要内核维护相应的数据结构，用于罗列每个进程的排队信号。由于这些数据结构会消耗内核内存，故而内核对排队实时信号的数量设置了限制。

SUSv3 允许实现为每个进程中可排队的（各类）实时信号数量设置上限，并要求其不得少于_POSIX_SIGQUEUE_MAX（定义为 32）。实现可借助于对 SIGQUEUE_MAX 常量的定义来表示其所允许的排队实时信号数量。发起如下调用也能获得这一信息：

更多资源请访问稀酷客(www.ckook.com)

```
lim = sysconf(_SC_SIGQUEUE_MAX);
```

若系统使用的 glibc 库版本在 2.4 之前，则该调用返回-1。从 glibc 2.4 开始，其返回值由内核版本决定。在 Linux 2.6.8 之前，调用将返回 Linux 专有文件/proc/sys/kernel/rtsig-max 中的值。该文件所定义为针对所有进程中可能排队的实时信号总数的系统级限制。默认值为 1024，不过特权级进程可以对其进行修改。至于当前的排队实时信号总数，可以从 Linux 专有的/proc/sys/kernel/rtsig-nr 文件中读取。

从版本 2.6.8 开始，Linux 取消了这些/proc 文件。取而代之的是资源限制 RLIMIT_SIGPENDING（36.3 节）。针对某个特定实际用户 ID 下辖的所有进程，该限制限定了其可排队的信号总数。sysconf()调用从 glibc2.10 版本开始返回 RLIMIT_SIGPENDING 限制。（至于正在等待某一进程的实时信号数量，可以从 Linux 专有文件/proc/PID/status 中的 SigQ 字段读取。）

使用实时信号

为了能让一对进程收发实时信号，SUSv3 提出以下几点要求。

- 发送进程使用 sigqueue()系统调用来发送信号及其伴随数据。

> 使用 kill()、killpg()和 raise()调用也能发送实时信号。然而，至于系统是否会对利用此类接口所发送的信号进行排队处理，SUSv3 规定，由具体实现决定。这些接口在 Linux 中会对实时信号进行排队，但在其他许多 UNIX 实现中，情况则不然。

- 要为该信号建立了一个处理器函数，接收进程应以 SA_SIGINFO 标志发起对 sigaction()的调用。因此，调用信号处理器时就会附带额外参数，其中之一是实时信号的伴随数据。

> 在 Linux 中，即使接收进程在建立信号处理器时并未指定 SA_SIGINFO 标志，也能对实时信号进行队列化管理（但在这种情况下，将不可能获得信号的伴随数据）。然而，SUSv3 也不要求实现确保这一行为，所以依赖这一点将有损于应用的可移植性。

22.8.1 发送实时信号

系统调用 sigqueue()将由 sig 指定的实时信号发送给由 pid 指定的进程。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);
```

Returns 0 on success, or -1 on error

使用 sigqueue()发送信号所需要的权限与 kill()（参见 20.5 节）的要求一致。也可以发送空信号（即信号 0），其语义与 kill()中的含义相同。（不同于 kill()，sigqueue()不能通过将 pid 指定为负值而向整个进程组发送信号。）

程序清单 22-2：使用 sigqueue()发送实时信号

signals/t_sigqueue.c

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include "tlpi_hdr.h"
```


更多资源请访问稀酷客(www.ckook.com)

```
int
main(int argc, char *argv[])
{
    int sig, numSigs, j, sigData;
    union sigval sv;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid sig-num data [num-sigs]\n", argv[0]);

    /* Display our PID and UID, so that they can be compared with the
       corresponding fields of the siginfo_t argument supplied to the
       handler in the receiving process */

    printf("%s: PID is %ld, UID is %ld\n", argv[0],
            (long) getpid(), (long) getuid());

    sig = getInt(argv[2], 0, "sig-num");
    sigData = getInt(argv[3], GN_ANY_BASE, "data");
    numSigs = (argc > 4) ? getInt(argv[4], GN_GT_0, "num-sigs") : 1;

    for (j = 0; j < numSigs; j++) {
        sv.sival_int = sigData + j;
        if (sigqueue(getLong(argv[1], 0, "pid"), sig, sv) == -1)
            errExit("sigqueue %d", j);
    }

    exit(EXIT_SUCCESS);
}
```

signals/t_sigqueue.c

参数 value 指定了信号的伴随数据，具有以下形式：

```
union sigval {
    int   sival_int;      /* Integer value for accompanying data */
    void *sival_ptr;      /* Pointer value for accompanying data */
};
```

对该参数的解释则取决于应用程序，由其选择对联合体（union)中的 sival_int 属性还是 sival_ptr 属性进行设置。sigqueue()中很少使用 sival_ptr，因为指针的作用范围在进程内部，对于另一进程几乎没有意义。该字段得以一展身手之处，应该是在使用 sigval 联合体的其他函数中，诸如 23.6 节的 POSIX 计时器和 52.6 节的 POSIX 消息队列通知。

> 包括 Linux 在内的几个 UNIX 实现定义了与 union sigval 同义的数据类型 sigval_t。然而，该类型既未获得 SUSv3 接纳，也没有得到其他实现的支持。对可移植性有所要求的应用程序应当避免使用。

一旦触及对排队信号的数量限制，sigqueue()调用将会失败。同时将 errno 置为 EAGAIN，以示需要再次发送该信号（在当前队列中某些信号传递之后的某一时间点）。

程序清单 22-2 提供了 sigqueue()的应用示例。该程序最多接受 4 个参数，其中前 3 项为必填项：目标进程 ID、信号编号以及伴随实时信号的整型值。如果需要为指定信号发送多个实例，那么可以用可选的第 4 个参数来指定实例数量。在这种情况下，会为每个信号的伴随整型值依次加 1。22.8.2 节将展示该程序的用法。

22.8.2 处理实时信号

可以像标准信号一样，使用常规（单参数）信号处理器来处理实时信号。此外，也可

更多资源请访问稀酷客(www.ckook.com)

以用带有 3 个参数的信号处理器函数来处理实时信号，其建立则会用到 SA_SIGINFO 标志（参见 21.4 节）。以下为使用 SA_SIGINFO 标志为第六个实时信号建立处理器函数的代码示例：

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_RESTART | SA_SIGINFO;

if (sigaction(SIGRTMIN + 5, &act, NULL) == -1)
    errExit("sigaction");
```

一旦采用了 SA_SIGINFO 标志，传递给信号处理器函数的第二个参数将是一个 siginfo_t 结构，内含实时信号的附加信息。21.4 节详细描述了这一数据结构。对于一个实时信号而言，会在 siginfo_t 结构中设置如下字段。

- si_signo 字段，其值与传递给信号处理器函数的第一个参数相同。
- si_code 字段表示信号来源，内容为表 21-2 中所示各值之一。对于通过 sigqueue()发送的实时信号来说，该字段值总是为 SI_QUEUE。
- si_value 字段所含数据，由进程于使用 sigqueue()发送信号时在 value 参数（sigval union）中指定。正如前文指出，对该数据的解释由应用程序决定。（若信号由 kill()发送，则 si_value 字段所含信息无效。）
- si_pid 和 si_uid 字段分别包含信号发送进程的进程 ID 和实际用户 ID。

程序清单 22-3 提供了处理实时信号的一个例子。该程序捕获信号，并针对传递给信号处理器函数的 siginfo_t 结构，一一显示其中的各个字段值。该程序可接收两个整型命令行参数，均为可选项。如果提供了第一个参数，那么主程序将阻塞所有信号并进入休眠，休眠秒数由该参数指定。在此期间，将对进程的实时信号进行排队处理，并可观察解除对信号阻塞时所发生的情况。第二个参数指定了信号处理器函数在返回前所应休眠的秒数。指定一个非 0 值（默认为 1 秒）将有助于放缓程序的执行，便于看清处理多个信号时所发生的情况。

可以将程序清单 22-3 中程序与程序清单 22-2 中程序（t_sigqueue.c）结合起来探索实时信号的行为，正如以下 shell 会话日志所示：

```
$ ./catch_rtsigs 60 &
[1] 12842
$ ./catch_rtsigs: PID is 12842          Shell prompt mixed with program output
./catch_rtsigs: signals blocked - sleeping 60 seconds
Press Enter to see next shell prompt
$ ./t_sigqueue 12842 54 100 3           Send signal three times
./t_sigqueue: PID is 12843, UID is 1000
$ ./t_sigqueue 12842 43 200
./t_sigqueue: PID is 12844, UID is 1000
$ ./t_sigqueue 12842 40 300
./t_sigqueue: PID is 12845, UID is 1000
```

最终，catch_rtsigs 程序结束休眠，随着信号处理器捕获到各种信号而一一显示消息。（之所以看到 shell 提示符和程序的下一行输出混杂在一起，是因为 catch_rtsigs 程序正在后台输出信息。）可以看出，实时信号在传递时遵循低编号优先的原则，并且在传递给处理器函数的 siginfo_t 结构中包含了发送进程的进程 ID 和用户 ID。

更多资源请访问稀酷客(www.ckook.com)

```
$ ./catch_rtsigs: sleep complete
caught signal 40
    si_signo=40, si_code=-1 (SI_QUEUE), si_value=300
    si_pid=12845, si_uid=1000
caught signal 43
    si_signo=43, si_code=-1 (SI_QUEUE), si_value=200
    si_pid=12844, si_uid=1000
```

接下来的输出由同一实时信号的 3 个实例产生。由 si_value 值可知，这些信号的传递顺序与发送顺序相同。

```
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=101
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=102
    si_pid=12843, si_uid=1000
```

继续使用 shell 的 kill 命令向程序 catch_rtsigs 发送信号。一如既往，处理器函数接收到的 siginfo_t 结构中包含了发送进程的进程 ID 和用户 ID，但此时的 si_code 值为 SI_USER。

```
Press Enter to see next shell prompt
$ echo $$                                   Display PID of shell
12780
$ kill -40 12842                            Uses kill(2) to send a signal
$ caught signal 40
    si_signo=40, si_code=0 (SI_USER), si_value=0
    si_pid=12780, si_uid=1000               PID is that of the shell
Press Enter to see next shell prompt
$ kill 12842                                Kill catch_rtsigs by sending SIGTERM
Caught 6 signals
Press Enter to see notification from shell about terminated background job
[1]+  Done                    ./catch_rtsigs 60
```

程序清单 22-3：处理实时信号

signals/catch_rtsigs.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static volatile int handlerSleepTime;
static volatile int sigCnt = 0;         /* Number of signals received */
static volatile int allDone = 0;

static void             /* Handler for signals established using SA_SIGINFO */
siginfoHandler(int sig, siginfo_t *si, void *ucontext)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf()); see Section 21.1.2) */

    /* SIGINT or SIGTERM can be used to terminate program */

    if (sig == SIGINT || sig == SIGTERM) {
        allDone = 1;
```


更多资源请访问稀酷客(www.ckook.com)

```
    return;
}

sigCnt++;
printf("caught signal %d\n", sig);

printf("    si_signo=%d, si_code=%d (%s), ", si->si_signo, si->si_code,
        (si->si_code == SI_USER) ? "SI_USER" :
        (si->si_code == SI_QUEUE) ? "SI_QUEUE" : "other");
printf("si_value=%d\n", si->si_value.sival_int);
printf("    si_pid=%ld, si_uid=%ld\n", (long) si->si_pid, (long) si->si_uid);

sleep(handlerSleepTime);
}
int
main(int argc, char *argv[])
{
    struct sigaction sa;
    int sig;
    sigset_t prevMask, blockMask;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [block-time [handler-sleep-time]]\n", argv[0]);

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    handlerSleepTime = (argc > 2) ?
                getInt(argv[2], GN_NONNEG, "handler-sleep-time") : 1;

    /* Establish handler for most signals. During execution of the handler,
       mask all other signals to prevent handlers recursively interrupting
       each other (which would make the output hard to read). */

    sa.sa_sigaction = siginfoHandler;
    sa.sa_flags = SA_SIGINFO;
    sigfillset(&sa.sa_mask);

    for (sig = 1; sig < NSIG; sig++)
        if (sig != SIGTSTP && sig != SIGQUIT)
            sigaction(sig, &sa, NULL);

    /* Optionally block signals and sleep, allowing signals to be
       sent to us before they are unblocked and handled */

    if (argc > 1) {
        sigfillset(&blockMask);
        sigdelset(&blockMask, SIGINT);
        sigdelset(&blockMask, SIGTERM);

        if (sigprocmask(SIG_SETMASK, &blockMask, &prevMask) == -1)
            errExit("sigprocmask");

        printf("%s: signals blocked - sleeping %s seconds\n", argv[0], argv[1]);
        sleep(getInt(argv[1], GN_GT_0, "block-time"));
        printf("%s: sleep complete\n", argv[0]);

        if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
            errExit("sigprocmask");
    }
```

```
    while (!allDone)                    /* Wait for incoming signals */
        pause();
}
```

signals/catch_rtsigs.c

22.9 使用掩码来等待信号：sigsuspend()

在解释 sigsuspend()的功用之前，先介绍一下它的一种使用场景。在对信号编程时偶尔会遇到如下情况。

1. 临时阻塞一个信号，以防止其信号处理器不会将某些关键代码片段的执行中断。
2. 解除对信号的阻塞，然后暂停执行，直至有信号到达。

为达到这一目的，可能会尝试使用程序清单 22-4 中代码所示方法。

程序清单 22-4：解除阻塞并等待信号的错误做法

```
sigset_t prevMask, intMask;
struct sigaction sa;

sigemptyset(&intMask);
sigaddset(&intMask, SIGINT);

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");

/* Block SIGINT prior to executing critical section. (At this
   point we assume that SIGINT is not already blocked.) */

if (sigprocmask(SIG_BLOCK, &intMask, &prevMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

/* Critical section: do some work here that must not be
   interrupted by the SIGINT handler */

/* End of critical section - restore old mask to unblock SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

/* BUG: what if SIGINT arrives now... */

pause();                            /* Wait for SIGINT */
```

程序清单 22-4 中代码存在一个问题。假设 SIGINT 信号的传递发生在第二次调用 sigprocmask()之后，调用 pause()之前。（实际上，该信号可能产生于执行关键片段期间的任一时刻，仅当解除对信号的阻塞后才会随之而传递。）SIGINT 信号的传递将导致对处理器函数的调用，而当处理器返回后，主程序恢复执行，pause()调用将陷入阻塞，直到 SIGINT 信号的第二个实例到达为止。这有违代码的本意：解除对 SIGINT 阻塞并等待其第一次出现。

更多资源请访问稀酷客(www.ckook.com)

即使在关键片段的起始点（即首次调用 sigprocmask()）和 pause()调用之间产生 SIGINT 信号的可能性不大，但这确实是上述代码的一处缺陷。这种取决于时间的缺陷是竞态条件（5.1 节）的例子之一。通常，竞态条件发生于两个进程或线程共享资源时。然而，此处的竞态条件却发生在主程序和其自身的信号处理器之间。

要避免这一问题，需要将解除信号阻塞和挂起进程这两个动作封装成一个原子操作。这正是 sigsuspend()系统调用的目的所在。

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

(Normally) returns -1 with *errno* set to EINTR

sigsuspend()系统调用将以 mask 所指向的信号集来替换进程的信号掩码，然后挂起进程的执行，直到其捕获到信号，并从信号处理器中返回。一旦处理器返回，sigsuspend()会将进程信号掩码恢复为调用前的值。

调用 sigsuspend()，相当于以不可中断方式执行如下操作：

```
sigprocmask(SIG_SETMASK, &mask, &prevMask);      /* Assign new mask */
pause();
sigprocmask(SIG_SETMASK, &prevMask, NULL);       /* Restore old mask */
```

虽然恢复老的信号掩码乍看起来似乎麻烦，但为了在需要反复等待信号的情况下避免竞态条件，这一做法就至关重要。在这种情况下，除非是在 sigsuspend()调用期间，否则信号必须保持阻塞状态。如果稍后需要对在调用 sigsuspend()之前遭到阻塞的信号解除阻塞，可以进一步调用 sigprocmask()。

若 sigsuspend()因信号的传递而中断，则将返回-1，并将 errno 置为 EINTR。如果 mask 指向的地址无效，则 sigsuspend()调用失败，并将 errno 置为 EFAULT。

示例程序

程序清单 22-5 展示了对 sigsuspend()的使用。该程序执行如下步骤。

- 调用 printSigMask()函数（程序清单 20-4）来显示进程信号掩码的初始值。
- 阻塞 SIGINT 和 SIGQUIT 信号，并保存原始的进程信号掩码。
- 为 SIGINT 和 SIGQUIT 信号建立相同的处理器函数。该处理器显示一条消息，且若对其调用因 SIGQUIT 信号的传递而引起，则设置全局变量 gotSigquit。
- 循环执行，直至对 gotSigquit 进行了设置。每次循环都执行如下步骤。
  - 使用 printSigMask()函数显示信号掩码的当前值。
  - 令 CPU 忙于循环并持续数秒钟，以此来模拟对一个关键片段的执行。
  - 使用 printPendingSigs()函数来显示等待信号的掩码（程序清单 20-4）。
  - 使用 sigsuspend()来解除对 SIGINT 和 SIGQUIT 信号的阻塞，并等待信号（如果尚未有信号处于等待状态）。
- 使用 sigprocmask()将进程信号掩码恢复为原始状态，然后再使用 printSigMask()来显示信号掩码。

程序清单 22-5：使用 sigsuspend()

signals/t_sigsuspend.c

```
#define _GNU_SOURCE      /* Get strsignal() declaration from <string.h> */
```

```
#include <string.h>
#include <signal.h>
#include <time.h>
#include "signal_functions.h"          /* Declarations of printSigMask()
                                          and printPendingSigs() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigquit = 0;

static void
handler(int sig)
{
    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
                                       /* UNSAFE (see Section 21.1.2) */
    if (sig == SIGQUIT)
        gotSigquit = 1;
}

int
main(int argc, char *argv[])
{
    int loopNum;
    time_t startTime;
    sigset_t origMask, blockMask;
    struct sigaction sa;

①   printSigMask(stdout, "Initial signal mask is:\n");

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SIGINT);
    sigaddset(&blockMask, SIGQUIT);
②   if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
        errExit("sigprocmask - SIG_BLOCK");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
③   if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGQUIT, &sa, NULL) == -1)
        errExit("sigaction");

④   for (loopNum = 1; !gotSigquit; loopNum++) {
        printf("=== LOOP %d\n", loopNum);

        /* Simulate a critical section by delaying a few seconds */

        printSigMask(stdout, "Starting critical section, signal mask is:\n");
        for (startTime = time(NULL); time(NULL) < startTime + 4; )
            continue;                  /* Run for a few seconds elapsed time */

        printPendingSigs(stdout,
                "Before sigsuspend() - pending signals:\n");
        if (sigsuspend(&origMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
    }

⑤   if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
        errExit("sigprocmask - SIG_SETMASK");
```


更多资源请访问稀酷客(www.ckook.com)

```
⑥     printSigMask(stdout, "=== Exited loop\nRestored signal mask to:\n");

      /* Do other processing... */

      exit(EXIT_SUCCESS);
  }
```

signals/t_sigsuspend.c

以下 shell 会话日志所示为程序清单 22-5 中程序的运行结果示例：

```
$ ./t_sigsuspend
Initial signal mask is:
                <empty signal set>
=== LOOP 1
Starting critical section, signal mask is:
                2 (Interrupt)
                3 (Quit)
Type Control-C; SIGINT is generated, but remains pending because it is blocked
Before sigsuspend() - pending signals:
                2 (Interrupt)
Caught signal 2 (Interrupt)          sigsuspend() is called, signals are unblocked
```

程序调用 sigsuspend()解除了对 SIGINT 信号的阻塞，还显示了最后一行输出。正是在那一点，调用了信号处理器，并显示了那一行输出。

主程序会继续循环。

```
=== LOOP 2
Starting critical section, signal mask is:
                2 (Interrupt)
                3 (Quit)
Type Control-\ to generate SIGQUIT
Before sigsuspend() - pending signals:
                3 (Quit)
Caught signal 3 (Quit)               sigsuspend() is called, signals are unblocked
=== Exited loop                      Signal handler set gotSigquit
Restored signal mask to:
                <empty signal set>
```

此时按下 Control-\，将导致信号处理器去设置 gotSigquit 标志，并转而引发主程序终止循环。

22.10 以同步方式等待信号

22.9 节描述了如何结合信号处理器和 sigsuspend()来挂起一个进程的执行，直至传来一个信号。然而，这需要编写信号处理器函数，还需要应对信号异步传递所带来的复杂性。对于某些应用而言，这种方法过于繁复。作为替代方案，可以利用 sigwaitinfo()系统调用来同步接收信号。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

Returns number of delivered signal on success, or −1 on error

sigwaitinfo()系统调用挂起进程的执行，直至 set 指向信号集中的某一信号抵达。如果调用 sigwaitinfo()时，set 中的某一信号已经处于等待状态，那么 sigwaitinfo()将立即返回。传递来

更多资源请访问稀酷客(www.ckook.com)

的信号就此从进程的等待信号队列中移除，并且将返回信号编号作为函数结果。info 参数如果不为空，则会指向经过初始化处理的 siginfo_t 结构，其中所含信息与提供给信号处理器函数的 siginfo_t 参数（21.4 节）相同。

sigwaitinfo()所接受信号的传递顺序和排队特性与信号处理器所捕获的信号相同，就是说，不对标准信号进行排队处理，对实时信号进行排队处理，并且对实时信号的传递遵循低编号优先的原则。

除了卸去编写信号处理器的负担之外，使用 sigwaitinfo()来等待信号也要比信号处理器外加 sigsuspend()的组合要稍快一些（见练习 22-3）。

将对 set 中信号集的阻塞与调用 sigwaitinfo()结合起来，这当属明智之举。（即便某一信号遭到阻塞，仍然可以使用 sigwaitinfo()来获取等待信号。）如果没有这么做，而信号在首次调用 sigwaitinfo()之前，或者两次连续调用 sigwaitinfo()之间到达，那么对信号的处理将只能依照其当前处置。

SUSv3 规定，调用 sigwaitinfo()而不阻塞 set 中的信号将导致不可预知的行为（其行为未定义）。

程序清单 22-6 所示为使用 sigwaitinfo()的例子之一。程序首先阻塞所有信号，然后延迟数秒时间，具体秒数由可选命令行参数来指定，从而允许在调用 sigwaitinfo()之前向程序发送信号。程序随即持续循环调用 sigwaitinfo()来接收输入信号，直至收到 SIGINT 或 SIGTERM 信号。

如下 shell 会话日志展示了程序清单 22-6 中程序的运行情况。程序在后台运行，并指定在执行 sigwaitinfo()前需延迟 60 秒，随后再向进程发送两个信号：

```
$ ./t_sigwaitinfo 60 &
./t_sigwaitinfo: PID is 3837
./t_sigwaitinfo: signals blocked
./t_sigwaitinfo: about to delay 60 seconds
[1] 3837
$ ./t_sigqueue 3837 43 100                    Send signal 43
./t_sigqueue: PID is 3839, UID is 1000
$ ./t_sigqueue 3837 42 200                    Send signal 42
./t_sigqueue: PID is 3840, UID is 1000
```

最终，程序完成睡眠，sigwaitinfo()调用循环接收排队信号。（由于 t_sigwaitinfo 程序正在后台输出信息，故而可以观察到 shell 提示符和程序的下一行输出混杂在一起。）至于处理器所捕获到的实时信号，可以看出，编号低的信号率先传递，而且，借助于传递给信号处理器函数的 siginfo_t 结构，还可以获得发送进程的进程 ID 和用户 ID。

```
$ ./t_sigwaitinfo: finished delay
got signal: 42
    si_signo=42, si_code=-1 (SI_QUEUE), si_value=200
    si_pid=3840, si_uid=1000
got signal: 43
    si_signo=43, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=3839, si_uid=1000
```

继续使用 shell 的 kill 命令向进程发送信号。可以观察到，这次将 si_code 字段置为 SI_USER（而非 SI_QUEUE）。

```
Press Enter to see next shell prompt
$ echo $$                                     Display PID of shell
3744
$ kill -USR1 3837                             Shell sends SIGUSR1 using kill()
$ got signal: 10                              Delivery of SIGUSR1
    si_signo=10, si_code=0 (SI_USER), si_value=100
```

```
    si_pid=3744, si_uid=1000                    3744 is PID of shell
Press Enter to see next shell prompt
$ kill %1                                       Terminate program with SIGTERM
$
Press Enter to see notification of background job termination
[1]+  Done                    ./t_sigwaitinfo 60
```

收到 SIGUSR1 信号，由其输出可知，si_value 字段值为 100。该值是由 sigqueue()发送的前一信号初始化而成。前文曾指出，仅对由 sigqueue()所发送的信号，si_value 字段所包含的信息才是可靠的。

程序清单 22-6：使用 sigwaitinfo()来同步等待信号

signals/t_sigwaitinfo.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include <time.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int sig;
    siginfo_t si;
    sigset_t allSigs;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [delay-secs]\n", argv[0]);

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    /* Block all signals (except SIGKILL and SIGSTOP) */

    sigfillset(&allSigs);
    if (sigprocmask(SIG_SETMASK, &allSigs, NULL) == -1)
        errExit("sigprocmask");
    printf("%s: signals blocked\n", argv[0]);

    if (argc > 1) {             /* Delay so that signals can be sent to us */
        printf("%s: about to delay %s seconds\n", argv[0], argv[1]);
        sleep(getInt(argv[1], GN_GT_0, "delay-secs"));
        printf("%s: finished delay\n", argv[0]);
    }

    for (;;) {                  /* Fetch signals until SIGINT (^C) or SIGTERM */
        sig = sigwaitinfo(&allSigs, &si);
        if (sig == -1)
            errExit("sigwaitinfo");

        if (sig == SIGINT || sig == SIGTERM)
            exit(EXIT_SUCCESS);

        printf("got signal: %d (%s)\n", sig, strsignal(sig));
        printf("    si_signo=%d, si_code=%d (%s), si_value=%d\n",
                si.si_signo, si.si_code,
                (si.si_code == SI_USER) ? "SI_USER" :
```

```
                    (si.si_code == SI_QUEUE) ? "SI_QUEUE" : "other",
                si.si_value.sival_int);
        printf("    si_pid=%ld, si_uid=%ld\n",
                (long) si.si_pid, (long) si.si_uid);
    }
}
```

signals/t_sigwaitinfo.c

sigtimedwait()系统调用是 sigwaitinfo()调用的变体。唯一的区别是 sigtimedwait()允许指定等待时限。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
```

Returns number of delivered signal on success, or −1 on error or timeout (EAGAIN)

timeout 参数指定了允许 sigtimedwait()等待一个信号的最大时长，是指向如下类型结构的一枚指针：

```
struct timespec {
    time_t tv_sec;      /* Seconds ('time_t' is an integer type) */
    long   tv_nsec;     /* Nanoseconds */
};
```

填写 timespec 结构的所属字段，也就指定了允许 sigtimedwait()等待的最大秒数和纳秒数。如果将这两个字段均指定为 0，那么函数将立刻超时，就是说，会去轮询检查是否有指定信号集中的任一信号处于等待状态。[1]如果调用超时而又没有收到信号，sigtimedwait()将调用失败，并将 errno 置为 EAGAIN。

如果将 timeout 参数指定为 NULL，那么 sigtimedwait()将完全等同于 sigwaitinfo()。SUSv3 对于 timeout 的 NULL 值含义也语焉不详，而某些 UNIX 实现则将该值视为轮询请求并立即将其返回。

22.11 通过文件描述符来获取信号

始于内核 2.6.22，Linux 提供了（非标准的）signalfd()系统调用；利用该调用可以创建一个特殊文件描述符，发往调用者的信号都可从该描述符中读取。signalfd 机制为同步接受信号提供了 sigwaitinfo()之外的另一种选择。

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

int signalfd(int fd, const sigset_t *mask, int flags);
```

Returns file descriptor on success, or −1 on error

mask 参数是一个信号集，指定了有意通过 signalfd 文件描述符来读取的信号。如同 sigwaitinfo()一样，通常也应该使用 sigprocmask()阻塞 mask 中的所有信号，以确保在有机会读

1 译者注：有，则将该信号的信息返回。作者的书似乎没有 man 手册页写得明了，请读者自行比较。

更多资源请访问稀酷客(www.ckook.com)

取这些信号之前，不会按照默认处置对它们进行处理。

如果指定 fd 为-1，那么 signalfd()会创建一个新的文件描述符，用于读取 mask 中的信号；否则，将修改与 fd 相关的 mask 值，且该 fd 一定是由之前对 signalfd()的一次调用创建而成。

早期实现将 flag 参数保留下来供将来使用，且必须将其指定为 0。然而，Linux 从版本 2.6.27 开始支持下面两个标志。

SFD_CLOEXEC

为新的文件描述符设置 close-on-exec（FD_CLOEXEC）标志。该标志之所以必要，与 4.3.1 节中描述的 open()O_CLOEXEC 标志的设置理由相同。

SFD_NONBLOCK

为底层的打开文件描述设置 O_NONBLOCK 标志，以确保不会阻塞未来的读操作。既省去了一个额外的 fcntl()调用，又获得了相同的结果。

创建文件描述符之后，可以使用 read()调用从中读取信号。提供给 read()的缓冲区必须足够大，至少应能够容纳一个 signalfd_siginfo 结构。<sys/signalfd.h>文件定义了该结构，如下所示：

```
struct signalfd_siginfo {
    uint32_t  ssi_signo;    /* Signal number */
    int32_t   ssi_errno;    /* Error number (generally unused) */
    int32_t   ssi_code;     /* Signal code */
    uint32_t  ssi_pid;      /* Process ID of sending process */
    uint32_t  ssi_uid;      /* Real user ID of sender */
    int32_t   ssi_fd;       /* File descriptor (SIGPOLL/SIGIO) */
    uint32_t  ssi_tid;      /* Kernel timer ID (POSIX timers) */
    uint32_t  ssi_band;     /* Band event (SIGPOLL/SIGIO) */
    uint32_t  ssi_tid;      /* (Kernel-internal) timer ID (POSIX timers) */
    uint32_t  ssi_overrun;  /* Overrun count (POSIX timers) */
    uint32_t  ssi_trapno;   /* Trap number */
    int32_t   ssi_status;   /* Exit status or signal (SIGCHLD) */
    int32_t   ssi_int;      /* Integer sent by sigqueue() */
    uint64_t  ssi_ptr;      /* Pointer sent by sigqueue() */
    uint64_t  ssi_utime;    /* User CPU time (SIGCHLD) */
    uint64_t  ssi_stime;    /* System CPU time (SIGCHLD) */
    uint64_t  ssi_addr;     /* Address that generated signal
                               (hardware-generated signals only) */
};
```

该结构中字段所返回的信息与传统 siginfo_t 结构（21.4 节）中类似命名的字段信息相同。

read()每次调用都将返回与等待信号数目相等的 signalfd_siginfo 结构，并填充到已提供的缓冲区中。如果调用时并无信号正在等待，那么 read()将阻塞，直到有信号到达。也可以使用 fcntl()的 F_SETFL 操作（5.3 节）来为文件描述符设置 O_NONBLOCK 标志，使得读操作不再阻塞，且若无信号等待，则调用失败，errno 为 EAGAIN。

当从 signalfd 文件描述符中读取到一信号时，该信号获得接纳，且不再为该进程而等待。

程序清单 22-7：使用 signalfd()来读取信号

signals/signalfd_sigval.c

```
#include <sys/signalfd.h>
#include <signal.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    sigset_t mask;
    int sfd, j;
    struct signalfd_siginfo fdsi;
    ssize_t s;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sig-num...\n", argv[0]);

    printf("%s: PID = %ld\n", argv[0], (long) getpid());

    sigemptyset(&mask);
    for (j = 1; j < argc; j++)
        sigaddset(&mask, atoi(argv[j]));

    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        errExit("sigprocmask");

    sfd = signalfd(-1, &mask, 0);
    if (sfd == -1)
        errExit("signalfd");

    for (;;) {
        s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
        if (s != sizeof(struct signalfd_siginfo))
            errExit("read");

        printf("%s: got signal %d", argv[0], fdsi.ssi_signo);
        if (fdsi.ssi_code == SI_QUEUE) {
            printf("; ssi_pid = %d; ", fdsi.ssi_pid);
            printf("ssi_int = %d", fdsi.ssi_int);
        }
        printf("\n");
    }
}
```

signals/signalfd_sigval.c

select()、poll()和 epoll（参见第 63 章）可以将 signalfd 描述符和其他描述符混合起来进行监控。撇开其他用途不提，该特性可成为 63.5.2 节所述 self-pipe 技巧之外的另一选择。如果有信号正在等待，那么这些技术将文件描述符指示为可读取。

当不再需要 signalfd 文件描述符时，应当关闭 signalfd 以释放相关内核资源。

程序清单 22-7 展示了 signalfd()的用法。程序为在命令行参数中指定的信号创建掩码，阻塞这些信号，然后创建用来读取这些信号的 signalfd 文件描述符，之后循环从该文件描述符中读取信号，并显示返回的 signalfd_siginfo 结构中的部分信息。如下 shell 会话在后台运行了程序清单 22-7 中程序，并使用程序清单 22-2 中程序（t_sigqueue.c）向该进程发送实时信号及伴随数据：

```
$ ./signalfd_sigval 44 &
./signalfd_sigval: PID = 6267
[1] 6267
$ ./t_sigqueue 6267 44 123          Send signal 44 with data 123 to PID 6267
./t_sigqueue: PID is 6269, UID is 1000
./signalfd_sigval: got signal 44; ssi_pid=6269; ssi_int=123
$ kill %1                           Kill program running in background
```


更多资源请访问稀酷客(www.ckook.com)

22.12 利用信号进行进程间通信

从某种角度，可将信号视为进程间通信（IPC）的方式之一。然而，信号作为一种 IPC 机制却也饱受限制。首先，与后续各章描述的其他 IPC 方法相比，对信号编程既繁且难，具体原因如下。

- 信号的异步本质就意味着需要面对各种问题，包括可重入性需求、竞态条件及在信号处理器中正确处理全局变量。（如果用 sigwaitinfo()或者 signalfd()来同步获取信号，这些问题中的大部分都不会遇到。）
- 没有对标准信号进行排队处理。即使是对于实时信号，也存在对信号排队数量的限制。这意味着，为了避免丢失信息，接收信号的进程必须想方设法通知发送者，自己为接受另一个信号做好了准备。要做到这一点，最显而易见的方法是由接收者向发送者发送信号。

还有一个更深层次的问题，信号所携带的信息量有限：信号编号以及实时信号情况下一字之长的附加数据（一个整数或者一枚指针值）。与诸如管道之类的其他 IPC 方法相比，过低的带宽使得信号传输极为缓慢。

由于上述种种限制，很少将信号用于 IPC。

22.13 早期的信号 API（System V 和 BSD）

之前对信号的讨论一直着眼于 POSIX 信号 API。本节将简要回顾一下 System V 和 BSD 提供的历史 API。虽然所有的新应用程序都应当使用 POSIX API，但是在从其他 UNIX 实现移植（通常较为老迈的）应用时，可能还是会碰到这些过时的 API。当移植这些使用老旧 API 的程序时，因为 Linux（像许多其他 UNIX 实现一样）提供了与 System V 和 BSD 兼容的 API，所以通常所要做的全部工作不过是在 Linux 平台上重新进行编译而已。

System V 信号 API

如前所述，System V 中的信号 API 存在一个重要差异：当使用 signal()建立处理器函数时，得到的是老版、不可靠的信号语义。这意味着不会将信号添加到进程的信号掩码中，调用信号处理器时会将信号处置重置为默认行为，以及不会自动重启系统调用。

下面，简单介绍一些 System V 信号 API 中的函数。手册页提供有全部的细节。SUSv3 定义了所有这些函数，但指出应优先使用现代版的 POSIX 等价函数。SUSv4 将这些函数标记为已废止。

```
#define _XOPEN_SOURCE 500
#include <signal.h>

void (*sigset(int sig, void (*handler)(int)))(int);
```

On success: returns the previous disposition of *sig*, or SIG_HOLD if *sig* was previously blocked; on error −1 is returned

为了建立一个具有可靠语义的信号处理器，System V 提供了 sigset()调用（原型类似于

更多资源请访问稀酷客(www.ckook.com)

signal()）。与 signal()一样，可以将 sigset()的 handler 参数指定为 SIG_IGN、SIG_DFL 或者信号处理器函数的地址。此外，还可以将其指定为 SIG_HOLD，在将信号添加到进程信号掩码的同时保持信号处置不变。

如果指定 handler 参数为 SIG_HOLD 之外的其他值，那么会将 sig 从进程信号掩码中移除（即，如果 sig 遭到阻塞，那么将解除对其阻塞）。

```
#define _XOPEN_SOURCE 500
#include <signal.h>

int sighold(int sig);
int sigrelse(int sig);
int sigignore(int sig);
```

All return 0 on success, or −1 on error

```
int sigpause(int sig);
```

Always returns −1 with *errno* set to EINTR

sighold()函数将一个信号添加到进程信号掩码中。sigrelse()函数则是从信号掩码中移除一个信号。sigignore()函数设定对某信号的处置为“忽略（ignore）”。sigpause()函数类似于 sigsuspend()函数，但仅从进程信号掩码中移除一个信号，随后将暂停进程，直到有信号到达。

BSD 信号 API

POSIX 信号 API 从 4.2BSD API 中汲取了很多灵感，所以 BSD 函数与 POSIX 函数大体相仿。

如同前文对 System V 信号 API 中函数的描述一样，首先给出 BSD 信号 API 中各函数的原型，随后简单解释一下每个函数的操作。再啰嗦一句，手册页提供有全部细节。

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

int sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);
```

Returns 0 on success, or −1 on error

sigvec()类似于 sigaction()。vec 和 ovec 参数是指向如下类型结构的指针：

```
struct sigvec {
    void (*sv_handler)();
    int  sv_mask;
    int  sv_flags;
};
```

sigvec 结构中的字段与 sigaction 结构中的那些字段紧密对应。第一个显著差异是 sv_mask（类似与 sa_mask）字段是一个整型，而非 sigset_t 类型。这意味着，在 32 位架构中，最多支持 31 个不同信号。另一个不同之处则在于在 sv_flags（类似与 a_flags）字段中使用了 SV_INTERRUPT 标志。因为重启系统调用是 4.2BSD 的默认行为，该标志是用来指定应使用信号处理器来中断慢速系统调用。（这点与 POSIX API 截然相反，在使用 sigaction()建立信号处理器时，如果希望启用系统调用重启功能，就必须显式指定 SA_RESTART 标志。）

更多资源请访问稀酷客(www.ckook.com)

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

int sigblock(int mask);
int sigsetmask(int mask);
```

Both return previous signal mask

```
int sigpause(int sigmask);
```

Always returns −1 with *errno* set to EINTR

```
int sigmask(sig);
```

Returns signal mask value with bit *sig* set

sigblock()函数向进程信号掩码中添加一组信号。这类似于 sigprocmask()的 SIG_BLOCK 操作。sigsetmask()调用则为信号掩码指定了一个绝对值。这类似于 sigprocmask()的 SIG_SETMASK 操作。

sigpause()类似于 sigsuspend()。注意，对该函数的定义在 System V 和 BSD API 中具有不同的调用签名。GNU C 函数库默认提供 System V 版本，除非在编译程序时指定了特性测试宏 _BSD_SOURCE。

sigmask()宏将信号编号转换成相应的 32 位掩码值。此类位掩码可以彼此相或，一起创建一组信号，如下所示：

```
sigblock(sigmask(SIGINT) | sigmask(SIGQUIT));
```

22.14 总结

某些信号会引发进程创建一个核心转储文件，并终止进程。核心转储所包含的信息可供调试器检查进程终止时的状态。默认情况下，对核心转储文件的命名为 core，但 Linux 提供了 /proc/sys/kernel/core_pattern 文件来控制对核心转储文件的命名。

信号的产生方式既可以是异步的，也可以是同步的。当由内核或者另一进程发送信号给进程时，信号可能是异步产生的。进程无法精确预测异步产生信号的传递时间。（文中曾指出，异步信号通常会在接收进程第二次从内核态切换到用户态时进行传递。）因进程自身执行代码而直接产生的信号则属于是同步产生的，例如，执行了一个引发硬件异常的指令，或者去调用 raise()。同步生成的信号，其传递可以精确预测（立即传递）。

实时信号是 POSIX 对原始信号模型的扩展，不同之处包括对实时信号进行队列化管理，具有特定的传递顺序，并且还可以伴随少量数据一同发送。设计实时信号，意在供应用程序自定义使用。实时信号的发送使用 sigqueue()系统调用，并且还向信号处理器函数提供了一个附加参数（siginfo_t 结构），以便其获得信号的伴随数据，以及发送进程的进程 ID 和实际用户 ID。

sigsuspend()系统调用在自动修改进程信号掩码的同时，还将挂起进程的执行直到信号到达，且二者属于同一原子操作。为了避免执行上述功能时出现竞态条件，确保 sigsuspend()的原子性至关重要。

可以使用 sigwaitinfo()和 sigtimedwait()来同步等待一个信号。这省去了对信号处理器的设计和编码工作。对于以等待信号的传递为唯一目的的程序而言，使用信号处理器纯属多此一举。

更多资源请访问稀酷客(www.ckook.com)

像 sigwaitinfo()和 sigtimedwait()一样，可以使用 Linux 特有的 signalfd()系统调用来同步等待一个信号。这一接口的独特之处在于可以通过文件描述符来读取信号。还可以使用 select()、poll()和 epoll 来对其进行监控。

尽管可以将信号视为 IPC 的方式之一，但诸多制约因素令其常常无法胜任这一目的，其中包括信号的异步本质、不对信号进行排队处理的事实，以及较低的传递带宽。信号更为常见的应用场景是用于进程同步，或是各种其他目的（比如，事件通知、作业控制以及定时器到期）。

信号的基本概念虽然简单，但因为涉及的细节很多，所以对其讨论用去了 3 章的篇幅。信号在系统调用 API 的各部分中都扮演着重要角色，后面几章还将重温对信号的使用。此外，还有各种信号相关的函数是针对线程的（比如，pthread_kill()和 pthread_sigmask()），将延后至 33.2 节进行讨论。

更多信息

参见 20.15 节所列的信息来源。

22.15 练习

22-1. 22.2 节曾指出，假设进程为 SIGCONT 信号建立了处理器函数并将其阻塞，如果该进程已停止（stopped）后因收到一个 SIGCONT 信号而恢复执行，那么仅当解除了对 SIGCONT 信号的阻塞时才会去调用信号处理器函数。编写一个程序来验证这一点。回忆一下，按下终端暂停字符（通常为 Control-Z）可以停止进程，使用 kill-CONT 命令（或者隐蔽一点，使用 shell 的 fg 命令）可以发送 SIGCONT 信号。

22-2. 如果实时信号和标准信号在同时等待一个进程，那么 SUSv3 对信号的传递顺序未予定义。编写一程序来展示 Linux 是如何处理这一情况的。（令程序为所有信号设置处理器函数，阻塞这些信号并持续一段时间，以便于向其发送各种信号，最后解除对所有信号的阻塞。）

22-3. 22.10 节指出，接收信号时，利用 sigwaitinfo()调用要比信号处理器外加 sigsuspend()调用的方法来得快。随本书发布的源码中提供的 signals/sig_speed_ sigsuspend.c 程序使用 sigsuspend()在父、子进程之间交替发送信号。请对两进程间交换一百万次信号所花费的时间进行计时。（信号交换次数可通过程序命令行参数来提供。）使用 sigwaitinfo()作为替代技术来对程序进行修改，并度量该版本的耗时。两个程序间的速度差异在哪里？

22-4. 使用 POSIX 信号 API 来实现 System V 函数 sigset()、sighold()、sigrelse()、sigignore()和 sigpause()。

更多资源请访问稀酷客(www.ckook.com)

第 23 章

定时器与休眠

定时器是进程规划自己在未来某一时刻接获通知的一种机制。休眠则能使进程（或线程）暂停执行一段时间。本章讨论了定时器设置以及休眠的接口，涵盖主题如下。

- 针对间隔式定时器设置的传统 UNIX API（setitimer()和 alarm()），一经设定，会在特定的一段时间后通知进程。
- 允许进程休眠特定时间的 API 接口。
- POSIX.1b 时钟和定时器 API 接口。
- Linux 特有的 timerfd 功能，允许所创建定时器的到期信息可从文件描述符中读取。

23.1 间隔定时器

系统调用 setitimer()创建一个间隔式定时器（interval timer），这种定时器会在未来某个时间点到期，并于此后（可选择地）每隔一段时间到期一次。

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

int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

Returns 0 on success, or −1 on error

通过在调用 setitimer()时为 which 指定以下值，进程可以创建 3 种不同类型的定时器。

ITIMER_REAL

创建以真实时间倒计时的定时器。到期时会产生 SIGALARM 信号并发送给进程。

ITIMER_VIRTUAL

创建以进程虚拟时间（用户模式下的 CPU 时间）倒计时的定时器。到期时会产生信号 SIGVTALRM。

ITIMER_PROF

创建一个 profiling 定时器，以进程时间（用户态与内核态 CPU 时间的总和）倒计时。到

更多资源请访问稀酷客(www.ckook.com)

期时，则会产生 SIGPROF 信号。

对所有这些信号的默认处置（disposition）均会终止进程。除非真地期望如此，否则就需要针对这些定时器信号创建处理器函数。

参数 new_value 和 old_value 均为指向结构 itimerval 的指针，结构的定义如下：

```
struct itimerval {
    struct timeval it_interval;     /* Interval for periodic timer */
    struct timeval it_value;        /* Current value (time until
                                       next expiration) */
};
```

结构 itimerval 中的字段类型均为 timeval 结构，timeval 又由秒和微秒两部分组成：

```
struct timeval {
    time_t      tv_sec;             /* Seconds */
    suseconds_t tv_usec;            /* Microseconds (long int) */
};
```

参数 new_value 的下属结构 it_value 指定了距离定时器到期的延迟时间。另一下属结构 it_interval 则说明该定时器是否为周期性定时器。如果 it_interval 的两个字段值均为 0，那么该定时器就属于在 it_value 所指定的时间间隔后到期的一次性定时器。只要 it_interval 中的任一字段非 0，那么在每次定时器到期之后，都会将定时器重置为在指定间隔后再次到期。

进程只能拥有上述 3 种定时器中的一种。当第 2 次调用 setitimer()时，修改已有定时器的属性要符合参数 which 中的类型。如果调用 setitimer()时将 new_value.it_value 的两个字段均置为 0，那么会屏蔽任何已有的定时器。

若参数 old_value 不为 NULL，则以其所指向的 itimerval 结构来返回定时器的前一设置。如果 old_value.it_value 的两个字段值均为 0，那么该定时器之前处于屏蔽状态。如果 old_value.it_interval 的两个字段值均为 0，那么该定时器之前被设置为历经 old_value.it_value 指定时间而到期的一次性定时器。对于需要在新定时器到期后将其还原的情况而言，获取定时器的前一设置就很重要。如果不关心定时器的前一设置，可以将 old_value 置为 NULL。

定时器会从初始值（it_value）倒计时一直到 0 为止。递减为 0 时，会将相应信号发送给进程，随后，如果时间间隔值（it_interval）非 0，那么会再次将 it_value 加载至定时器，重新开始向 0 倒计时。

可以在任何时刻调用 getitimer()，以了解定时器的当前状态、距离下次到期的剩余时间。

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

int getitimer(int which, struct itimerval *curr_value);
```

Returns 0 on success, or −1 on error

系统调用 getitimer()返回由 which 指定定时器的当前状态，并置于由 curr_value 所指向的缓冲区中。这与 setitimer()借参数 old_value 所返回的信息完全相同，区别则在于 getitimer()无需为了获取这些信息而改变定时器的设置。子结构 curr_value.it_value 返回距离下一次到期所剩余的总时间。该值会随定时器倒计时而变化，如果设置定时器时将 it_interval 置为非 0 值，那么会在定时器到期时将其重置。子结构 curr_value.it_interval 返回定时器的间隔时间，除非再次调用 setitimer()，否则该值一直保持不变。

使用 setitimer()（和 alam()，稍后讨论）创建的定时器可以跨越 exec()调用而得以保存，但由 fork()创建的子进程并不继承该定时器。

更多资源请访问稀酷客(www.ckook.com)

SUSv4 废止了 getitimer()和 setitimer()，同时推荐使用 POSIX 定时器 API（23.6 节）。

示例程序

程序清单 23-1 演示了 setitimer()和 getitimer()的使用，所执行的步骤如下。

- 为 SIGALRM 信号创建处理器函数③。
- 利用命令行参数为实时（ITIMER_REAL）定时器设置到期值及间隔时间④。若未提供命令行参数，程序默认创建一个两秒到期的一次性定时器。
- 进入一个循环⑤，消耗 CPU 时间并周期性地调用函数 displayTimes()①。该函数会显示自程序启动以来逝去的真实时间，以及 ITIMER_REAL 定时器的当前状态。

每当定时器到期时，都会调用 SIGALRM 处理器函数，其中会去设置全局标志 gotAlarm②。一旦设置了这一标志，主程序循环就会调用 displayTimers()来显示处理器函数的调用时点以及定时器状态⑥。（之所以采用这一方式来设计信号处理器函数，意在避免从处理器函数内部去调用非异步信号安全的函数，究其原因可参考 21.1.2 节。）如果定时器的时间间隔为 0，那么程序会在第 1 次收到信号时即行退出。否则，程序会在捕获到 3 个信号后终止。⑦

运行程序清单 23-1 中的程序，可以看到如下结果：

```
$ ./real_timer 1 800000 1 0          Initial value 1.8 seconds, interval 1 second
       Elapsed   Value  Interval
START:    0.00
Main:     0.50    1.30    1.00       Timer counts down until expiration
Main:     1.00    0.80    1.00
Main:     1.50    0.30    1.00
ALARM:    1.80    1.00    1.00       On expiration, timer is reloaded from interval
Main:     2.00    0.80    1.00
Main:     2.50    0.30    1.00
ALARM:    2.80    1.00    1.00
Main:     3.00    0.80    1.00
Main:     3.50    0.30    1.00
ALARM:    3.80    1.00    1.00
That's all folks
```

程序清单 23-1：实时定时器的使用

timers/real_timer.c

```
#include <signal.h>
#include <sys/time.h>
#include <time.h>
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotAlarm = 0;
                        /* Set nonzero on receipt of SIGALRM */

/* Retrieve and display the real time, and (if 'includeTimer' is
   TRUE) the current value and interval for the ITIMER_REAL timer */

static void
① displayTimes(const char *msg, Boolean includeTimer)
{
    struct itimerval itv;
    static struct timeval start;
    struct timeval curr;
    static int callNum = 0;             /* Number of calls to this function */
```

```
    if (callNum == 0)                    /* Initialize elapsed time meter */
        if (gettimeofday(&start, NULL) == -1)
            errExit("gettimeofday");

    if (callNum % 20 == 0)               /* Print header every 20 lines */
        printf("       Elapsed   Value Interval\n");
    if (gettimeofday(&curr, NULL) == -1)
        errExit("gettimeofday");
    printf("%-7s %6.2f", msg, curr.tv_sec - start.tv_sec +
                      (curr.tv_usec - start.tv_usec) / 1000000.0);

    if (includeTimer) {
        if (getitimer(ITIMER_REAL, &itv) == -1)
            errExit("getitimer");
        printf("  %6.2f  %6.2f",
                itv.it_value.tv_sec + itv.it_value.tv_usec / 1000000.0,
                itv.it_interval.tv_sec + itv.it_interval.tv_usec / 1000000.0);
    }

    printf("\n");
    callNum++;
}

static void
sigalrmHandler(int sig)
{
②   gotAlarm = 1;
}

int
main(int argc, char *argv[])
{
    struct itimerval itv;
    clock_t prevClock;
    int maxSigs;                /* Number of signals to catch before exiting */
    int sigCnt;                 /* Number of signals so far caught */
    struct sigaction sa;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [secs [usecs [int-secs [int-usecs]]]]\n", argv[0]);

    sigCnt = 0;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigalrmHandler;
③   if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");

    /* Exit after 3 signals, or on first signal if interval is 0 */

    maxSigs = (itv.it_interval.tv_sec == 0 &&
               itv.it_interval.tv_usec == 0) ? 1 : 3;

    displayTimes("START:", FALSE);

    /* Set timer from the command-line arguments */

    itv.it_value.tv_sec = (argc > 1) ? getLong(argv[1], 0, "secs") : 2;
    itv.it_value.tv_usec = (argc > 2) ? getLong(argv[2], 0, "usecs") : 0;
```


更多资源请访问稀酷客(www.ckook.com)

```
    itv.it_interval.tv_sec = (argc > 3) ? getLong(argv[3], 0, "int-secs") : 0;
    itv.it_interval.tv_usec = (argc > 4) ? getLong(argv[4], 0, "int-usecs") : 0;

④   if (setitimer(ITIMER_REAL, &itv, 0) == -1)
        errExit("setitimer");

    prevClock = clock();
    sigCnt = 0;

⑤   for (;;) {

        /* Inner loop consumes at least 0.5 seconds CPU time */

        while (((clock() - prevClock) * 10 / CLOCKS_PER_SEC) < 5) {
⑥           if (gotAlarm) {                    /* Did we get a signal? */
                gotAlarm = 0;
                displayTimes("ALARM:", TRUE);

                sigCnt++;
⑦               if (sigCnt >= maxSigs) {
                    printf("That's all folks\n");
                    exit(EXIT_SUCCESS);
                }
            }
        }

        prevClock = clock();
        displayTimes("Main: ", TRUE);
    }
}
```

timers/real_timer.c

更为简单的定时器接口：alarm()

系统调用 alarm()为创建一次性实时定时器提供了一个简单接口。（历史上，alarm()曾是设置定时器的原始 UNIX API。）

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Always succeeds, returning number of seconds remaining on any previously set timer, or 0 if no timer previously was set

参数 seconds 表示定时器到期的秒数。到期时，会向调用进程发送 SIGALRM 信号。

调用 alarm()会覆盖对定时器的前一个设置。调用 alarm(0)可屏蔽现有定时器。

alarm()的返回值是定时器前一设置距离到期的剩余秒数，如未设置定时器则返回 0。

23.3 节提供了一个使用 alarm()的例子。

> 本书后面的一些示例会使用 alarm()来启动定时器，同时不为 SIGALRM 信号设置处理器函数。采用该技术可以确保，即便进程没有终止，也能将其杀死。

setitimer()和 alarm()之间的交互

Linux 中，alarm()和 setitimer()针对同一进程（per-process）共享同一实时定时器，这也意

更多资源请访问稀酷客(www.ckook.com)

味着，无论调用两者之中的哪个完成了对定时器的前一设置，同样可以调用二者中的任一函数来改变这一设置。其他 UNIX 系统的情况可能会有所不同（也就是说，这两个函数可能分别控制着不同的定时器）。对于 setitimer()与 alarm()之间的交互，以及二者与 sleep()函数（23.4.1 节）之间的交互，SUSv3 均未加以规范。为了确保应用程序可移植性的最大化，程序设置实时定时器的函数只能在二者中选择其一。

23.2 定时器的调度及精度

取决于当前负载和对进程的调度，系统可能会在定时器到期的瞬间（通常是几分之一秒）之后才去调度其所属进程。尽管如此，由 setitimer()或本章后续介绍的其他接口所创建的周期性定时器，在到期后依然会恪守其规律性。例如，假设设置一个实时定时器每两秒到期一次，虽然上述延迟可能会影响每个定时器事件的送达，但系统对后续定时器到期的调度依然会严格遵循两秒的时间间隔。换言之，间隔式定时器不受潜在错误左右。

虽然 setitimer()使用的 timeval 结构提供有微秒级精度，但是传统意义上定时器精度还是受制于软件时钟（10.6 节）频率。如果定时器值未能与软件时钟间隔的倍数严格匹配，那么定时器值则会向上取整。也就是说，假如有一个间隔为 19100 微秒（刚刚超过 19 毫秒）的定时器，如果 jiffy（软件时钟周期）为 4 毫秒，那么定时器实际上会每隔 20 毫秒过期一次。

高分辨率定时器

对于现代 Linux 内核而言，适才关于定时器分辨率受限于软件时钟频率的论断已经不再成立。自版本 2.6.21 开始，Linux 内核可选择是否支持高分辨率定时器。如果选择支持（通过内核配置选项 CONFIG_HIGH_RES_TIMERS），那么本章各种定时器以及休眠接口的的精度则不再受内核 jiffy（软件时钟周期）的影响，可以达到底层硬件所支持的精度。在现代硬件平台上，精度达到微秒级是司空见惯的事情。

23.5.1 节将介绍函数 clock_getres()，可以用其返回值来判断系统是否支持高分辨率定时器。

23.3 为阻塞操作设置超时

实时定时器的用途之一是为某个阻塞系统调用设置其处于阻塞状态的时间上限。例如，当用户在一段时间内没有输入整行命令时，可能希望取消对终端的 read()操作。处理如下。

1. 调用 sigaction()为 SIGALRM 信号创建处理器函数，排除 SA_RESTART 标志以确保系统调用不会重新启动（参考 21.5 节）。
2. 调用 alarm()或 setitimer()来创建一个定时器，同时设定希望系统调用阻塞的时间上限。
3. 执行阻塞式系统调用。
4. 系统调用返回后，再次调用 alarm()或 setitimer()以屏蔽定时器（以防止系统调用在定时器到期之前就已完成的情况）。
5. 检查系统调用失败时是否将 errno 置为 EINTR（系统调用遭到中断）。

程序清单 23-2 针对 read()调用展示了这一技术，创建定时器时使用的是 alarm()。

更多资源请访问稀酷客(www.ckook.com)

程序清单 23-2：运行设置了超时的 read()

timers/timed_read.c

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

#define BUF_SIZE 200

static void     /* SIGALRM handler: interrupts blocked system call */
handler(int sig)
{
    printf("Caught signal\n");          /* UNSAFE (see Section 21.1.2) */
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    char buf[BUF_SIZE];
    ssize_t numRead;
    int savedErrno;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-secs [restart-flag]]\n", argv[0]);

    /* Set up handler for SIGALRM. Allow system calls to be interrupted,
       unless second command-line argument was supplied. */

    sa.sa_flags = (argc > 2) ? SA_RESTART : 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = handler;
    if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");

    alarm((argc > 1) ? getInt(argv[1], GN_NONNEG, "num-secs") : 10);

    numRead = read(STDIN_FILENO, buf, BUF_SIZE - 1);

    savedErrno = errno;                 /* In case alarm() changes it */
    alarm(0);                           /* Ensure timer is turned off */
    errno = savedErrno;

    /* Determine result of read() */

    if (numRead == -1) {
        if (errno == EINTR)
            printf("Read timed out\n");
        else
            errMsg("read");
    } else {
        printf("Successful read (%ld bytes): %.*s",
                (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

timers/timed_read.c

注意，程序清单 23-2 中程序理论上存在导致竞争条件的可能性。如果定时器到期时处于 alarm()调用之后，read()调用之前，那么信号处理器函数将不会中断 read()。由于在这种场景下设定的超时值一般相对较大（至少几秒），故而发生上述情况的概率极低，因此这种技术实际上是可行的。[Stevens & Rago, 2005]推荐了另一种方法，使用的是 longjmp()。在处理 I/O 系统调用时，还有另一种备选方案，利用了系统调用 select()或 poll()（第 63 章）的超时特性，锦上添花的是还能同时等待多路描述符的 I/O。

23.4 暂停运行（休眠）一段固定时间

有时需要将进程挂起（固定的）一段时间。将前述定时器函数与 sigsuspend()相结合固然可以达到这一目的，但使用休眠函数会更为简单。

23.4.1 低分辨率休眠：sleep()

函数 sleep()可以暂停调用进程的执行达数秒之久（由参数 seconds 设置），或者在捕获到信号（从而中断调用）后恢复进程的运行。

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns 0 on normal completion, or number of unslept seconds if prematurely terminated

如果休眠正常结束，sleep()返回 0。如果因信号而中断休眠，sleep()将返回剩余（未休眠）的秒数。与 alarm()和 setitimer()所设置的定时器相同，由于系统负载的原因，内核可能会在完成 sleep()的一段（通常很短）时间后才对进程重新加以调度。

对于 sleep()和 alarm()以及 setitimer()之间的交互方式，SUSv3 并未加以规范。Linux 将 sleep()实现为对 nanosleep()（23.4.2 节）的调用，其结果是 sleep()与定时器函数之间并无交互。不过，许多其他的实现，尤其是一些老系统，会使用 alarm()以及 SIGALRM 信号处理器函数来实现 sleep()。考虑到可移植性，应避免将 sleep()和 alarm()以及 setitimer()混用。

23.4.2 高分辨率休眠：nanosleep()

函数 nanosleep()的功用与 sleep()类似，但更具优势，其中包括能以更高分辨率来设定休眠间隔时间。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep(const struct timespec *request, struct timespec *remain);
```

Returns 0 on successfully completed sleep, or −1 on error or interrupted sleep

参数 request 指定了休眠的持续时间，是一个指向如下结构的指针：

```
struct timespec {
    time_t tv_sec;          /* Seconds */
    long   tv_nsec;         /* Nanoseconds */
};
```


更多资源请访问稀酷客(www.ckook.com)

tv_nsec 字段为纳秒值，取值范围在 0～999999999 之间。

nanosleep()的更大优势在于，SUSv3 明文规定不得使用信号来实现该函数。这意味着，与 sleep()不同，即使将 nanosleep()与 alarm()或 setitimer()混用，也不会危及程序的可移植性。

尽管 nanosleep()的实现并未使用信号，但还是可以通过信号处理器函数来将其中断。这时，nanosleep()将返回-1，并将 errno 置为 EINTR。同时，若参数 remain 不为 NULL，则该指针所指向的缓冲区将返回剩余的休眠时间。可利用这一返回值重启该系统调用以完成休眠。程序清单 23-3 演示了这一用途。程序从命令行参数中获取传入 nanosleep()的秒和纳秒值，并反复循环执行 nanosleep()，直至耗尽全部的休眠间隔时间。如果信号 SIGINT（按下 Ctrl-C 产生）的处理器函数将 nanosleep()中断，那么会以参数 remain 中的返回值重新调用 nanosleep()。其运行结果如下：

```
$ ./t_nanosleep 10 0                    Sleep for 10 seconds
Type Control-C
Slept for:  1.853428 secs
Remaining:  8.146617000
Type Control-C
Slept for:  4.370860 secs
Remaining:  5.629800000
Type Control-C
Slept for:  6.193325 secs
Remaining:  3.807758000
Slept for: 10.008150 secs
Sleep complete
```

虽然 nanosleep()允许设定纳秒级精度的休眠间隔值，但其精度依然受制于软件时钟的间隔大小（10.6 节）。如果指定的间隔值并非软件时钟间隔的整数倍，那么会对其向上取整。

前文曾提及，在支持高精度定时器的系统中，休眠时间间隔的精度要比软件时钟间隔精细许多。

当以高频率接收信号时，这一取整行为会给程序清单 23-3 中程序所采用的编程手法带来问题。由于返回的 remain 时间未必是软件时钟间隔的整数倍，故而 nanosleep()的每次重启都会遭遇取整错误。其结果是，nanosleep()每次重启后的休眠时间都要长于前一调用返回的 remain 值。在信号接收频率极高的情况下（与软件时钟间隔的频率一致或更高），进程的休眠可能永远也完成不了。Linux 2.6 中，使用带有 TIMER_ABSTIME 选项的 clock_nanosleep()可以避免这一问题。23.5.4 节将对 clock_nanosleep()加以讨论。

在 2.4 以及更早期的 Linux 内核版本中，nanosleep()的实现存在着一种奇怪的特性。假设正在执行 nanosleep()的进程因信号而停止，当进程于稍后截获 SIGCONT 而继续运行时，nanosleep()会如期调用失败并返回 EINTR 错误。不过，如果进程接着重启 nanosleep()调用，那么进程处于停止状态所消耗的时间将不会计入休眠间隔时间，进程的休眠时间也就比预期的要久。Linux 2.6 中去除了这一怪异特性，nanosleep()在收到 SIGCONT 信号时将自动恢复，进程处于停止状态所消耗的时间也会计入休眠间隔时间。

程序清单 23-3：使用 nanosleep()

timers/t_nanosleep.c

```
#define _POSIX_C_SOURCE 199309
#include <sys/time.h>
```

```
#include <time.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigintHandler(int sig)
{
    return;                     /* Just interrupt nanosleep() */
}

int
main(int argc, char *argv[])
{
    struct timeval start, finish;
    struct timespec request, remain;
    struct sigaction sa;
    int s;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs nanosecs\n", argv[0]);

    request.tv_sec = getLong(argv[1], 0, "secs");
    request.tv_nsec = getLong(argv[2], 0, "nanosecs");

    /* Allow SIGINT handler to interrupt nanosleep() */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigintHandler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    if (gettimeofday(&start, NULL) == -1)
        errExit("gettimeofday");

    for (;;) {
        s = nanosleep(&request, &remain);
        if (s == -1 && errno != EINTR)
            errExit("nanosleep");

        if (gettimeofday(&finish, NULL) == -1)
            errExit("gettimeofday");
        printf("Slept for: %9.6f secs\n", finish.tv_sec - start.tv_sec +
                        (finish.tv_usec - start.tv_usec) / 1000000.0);

        if (s == 0)
            break;                  /* nanosleep() completed */

        printf("Remaining: %2ld.%09ld\n", (long) remain.tv_sec,
                remain.tv_nsec);
        request = remain;           /* Next sleep is with remaining time */
    }

    printf("Sleep complete\n");
    exit(EXIT_SUCCESS);
}
```

timers/t_nanosleep.c

更多资源请访问稀酷客(www.ckook.com)

23.5 POSIX 时钟

POSIX 时钟（原定义于 POSIX.1b）所提供的时钟访问 API 可以支持纳秒级的时间精度，其中表示纳秒级时间值的 timespec 结构同样也用于 nanosleep()（23.4.2 节）调用。

Linux 中，调用此 API 的程序必须以-lrt 选项进行编译，从而与 librt（realtime，实时）函数库相链接。

POSIX 时钟 API 的主要系统调用包括获取时钟当前值的 clock_gettime()、返回时钟分辨率的 clock_getres()，以及更新时钟的 clock_settime()。

23.5.1 获取时钟的值：clock_gettime()

系统调用 clock_gettime()针对参数 clockid 所指定的时钟返回时间。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_getres(clockid_t clockid, struct timespec *res);
```

Both return 0 on success, or −1 on error

返回的时间值置于 tp 指针所指向的 timespec 结构中。虽然 timespec 结构提供了纳秒级精度，但 clock_gettime()返回的时间值粒度可能还是要更大一点。系统调用 clock_getres()在参数 res 中返回指向 timespec 结构的指针，机构中包含了由 clockid 所指定时钟的分辨率。

clockid_t 是一种由 SUSv3 定义的数据类型，用于表示时钟标识符。表 23-1 中第 1 列值即可用于设定 clockid。

表 23-1：POSIX.1b 时钟类型

时钟 ID	描　　述
CLOCK_REALTIME	可设定的系统级实时时钟
CLOCK_MONOTONIC	不可设定的恒定态时钟
CLOCK_PROCESS_CPUTIME_ID	每进程 CPU 时间的时钟（自 Linux 2.6.12）
CLOCK_THREAD_CPUTIME_ID	每线程 CPU 时间的时钟（自 Linux 2.6.12）

CLOCK_REALTIME 时钟是一种系统级时钟，用于度量真实时间。与 CLOCK_MONOTONIC 时钟不同，它的设置是可以变更的。

SUSv3 规定，CLOCK_MONOTONIC 时钟对时间的度量始于“未予规范的过去某一时点”，系统启动后就不会发生改变。该时钟适用于那些无法容忍系统时钟发生跳跃性变化（例如：手工改变了系统时间）的应用程序。Linux 上，这种时钟对时间的测量始于系统启动。

CLOCK_PROCESS_CPUTIME_ID 时钟测量调用进程所消耗的用户和系统 CPU 时间。CLOCK_THREAD_CPUTIME_ID 时钟的功用与之相类似，不过测量对象是进程中的单条线程。

SUSv3 规范了表 23-1 中的所有时钟，但强制要求实现的仅有 CLOCK_REALTIME 一种，这同时也是受到 UNIX 实现广泛支持的时钟类型。

更多资源请访问稀酷客(www.ckook.com)

Linux 2.6.28 增加了一种新的时钟类型：CLOCK_MONOTONIC_RAW。类似于 CLOCK_MONOTONIC，这也是一种无法设置的时钟，但是提供了对纯基于硬件时间的访问，且不受 NTP 时间调整的影响。这种非标准时钟适用于专业时钟同步应用程序。

Linux 2.6.35 又提供了两种新时钟：CLOCK_REALTIME_COARSE 和 CLOCK_MONTIC_COARSE。这些时钟类似于 CLOCK_REALTIME 和 CLOCK_MONTONIC，适用于那些希望以最小代价获取较低分辨率时间戳的程序。这些非标准时钟不会引发对硬件时钟的任何访问（访问某些硬件时钟源的代价高昂），其返回值的分辨率为 jiffy（软件时钟周期，见 10.6 节）。

23.5.2 设置时钟的值：clock_settime()

系统调用 clock_settime()利用参数 tp 所指向缓冲区中的时间来设置由 clockid 指定的时钟。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_settime(clockid_t clockid, const struct timespec *tp);
```

Returns 0 on success, or −1 on error

如果由 tp 指定的时间并非由 clock_getres()所返回时钟分辨率的整数倍，时间会向下取整。

特权级（CAP_SYS_TIME）进程可以设置 CLOCK_REALTIME 时钟。该时钟的初始值通常是自 Epoch（1970 年 1 月 1 日 0 点 0 分 0 秒）以来的时间。表 23-1 中的其他时钟均不可更改。

根据 SUSv3，系统实现可允许设置 CLOCK_PROCESS_CPUTIME_ID 和 CLOCK_THREAD_CPUTIME_ID 型时钟。撰写本书之际，这些时钟在 Linux 上依然是只读属性。

23.5.3 获取特定进程或线程的时钟 ID

要测量特定进程或线程所消耗的 CPU 时间，首先可借助本节所描述的函数来获取其时钟 ID。接着再以此返回 id 去调用 clock_gettime()，从而获得进程或线程耗费的 CPU 时间。

函数 clock_getcpuclockid()会将隶属于 pid 进程的 CPU 时间时钟的标识符置于 clockid 指针所指向的缓冲区中。

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clockid);
```

Returns 0 on success, or a positive error number on error

参数 pid 为 0 时，clock_getcpuclockid()返回调用进程的 CPU 时间时钟 ID。

函数 pthread_getcpuclockid()是 clock_getcpuclockid()的 POSIX 线程版，返回的标识符所标识的时钟用于度量调用进程中指定线程消耗的 CPU 时间。

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread, clockid_t *clockid);
```

Returns 0 on success, or a positive error number on error

更多资源请访问棒酷客(www.ckook.com)

参数 thread 是 POSIX 线程 ID，用于指定希望获取的 CPU 时钟 ID 所从属的线程。返回的时钟 ID 存放于 clockid 指针所指向的缓冲区中。

23.5.4 高分辨率休眠的改进版：clock_nanosleep()

类似于 nanosleep()，Linux 特有的 clock_nanosleep()系统调用也可以暂停调用进程，直到历经一段指定的时间间隔后，亦或是收到信号才恢复运行。本节将讨论两者间的差异。

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_nanosleep(clockid_t clockid, int flags,
       const struct timespec *request, struct timespec *remain);
```

Returns 0 on successfully completed sleep,
or a positive error number on error or interrupted sleep

参数 request 及 remain 同 nanosleep()中的对应参数目的相似。

默认情况下（即 flags 为 0），由 request 指定的休眠间隔时间是相对时间（类似于 nanosleep()）。不过，如果在 flags（参考程序清单 23-4）中设定了 TIMER_ABSTIME，request 则表示 clockid 时钟所测量的绝对时间。这一特性对于那些需要精确休眠一段指定时间的应用程序至关重要。如果只是先获取当前时间，计算与目标时间的差距，再以相对时间进行休眠，进程可能执行到一半就被占先了[1]，结果休眠时间比预期的要久。

如 23.4.2 节所述，对于那些被信号处理器函数中断并使用循环重启休眠的进程来说，“嗜睡（oversleeping）”问题尤其明显。如果以高频率接收信号，那么按相对时间休眠（nanosleep()所执行的类型）的进程在休眠时间上会有较大误差。但可以通过如下方式来避免嗜睡问题：先调用 clock_gettime()获取时间，加上期望休眠的时间量，再以 TIMER_ABSTIME 标志调用 clock_nanosleep()函数（并且，如果被信号处理器中断，则会重启系统调用）。

指定 TIMER_ABSTIME 时，不再（且不需要）使用参数 remain。如果信号处理器程序中断了 clock_nanosleep()调用，再次调用该函数来重启休眠时，request 参数不变。

将 clock_nanosleep()与 nanosleep()区分开来的另一特性在于，可以选择不同的时钟来测量休眠间隔时间。可在 clockid 中指定所期望的时钟 CLOCK_REALTIME、CLOCK_ MONOTONIC 或 CLOCK_PROCESS_CPUTIME_ID。请参考表 23-1 对这些时钟的描述。

程序清单 23-4 演示了 clock_nanosleep()的用法：针对 CLOCK_REALTIME 时钟，以绝对时间休眠 20 秒。

程序清单 23-4：使用 clock_nanosleep()

```
struct timespec request;

/* Retrieve current value of CLOCK_REALTIME clock */

if (clock_gettime(CLOCK_REALTIME, &request) == -1)
    errExit("clock_gettime");

request.tv_sec += 20;                /* Sleep for 20 seconds from now */
```

1 译者注：内核调度所为。

```
s = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &request, NULL);
if (s != 0) {
    if (s == EINTR)
        printf("Interrupted by signal handler\n");
    else
        errExitEN(s, "clock_nanosleep");
}
```

23.6 POSIX 间隔式定时器

使用 setitimer()来设置经典 UNIX 间隔式定时器，会受到如下制约。

- 针对 ITIMER_REAL、ITIMER_VIRTUAL 和 ITIMER_PROF 这 3 类定时器，每种只能设置一个。
- 只能通过发送信号的方式来通知定时器到期。另外，也不能改变到期时产生的信号。
- 如果一个间隔式定时器到期多次，且相应信号遭到阻塞时，那么会只调用一次信号处理器函数。换言之，无从知晓是否出现过定时器溢出（timer overrun）的情况。
- 定时器的分辨率只能达到微秒级。不过，一些系统的硬件时钟提供了更为精细的时钟分辨率，软件此时应采用这一较高分辨率。

POSIX.1b 定义了一套 API 来突破这些限制，Linux 2.6 实现了这一 API。

在较老的 Linux 系统上，glibc 通过基于线程的实现提供了这一 API 的不完整版。不过，这种用户空间内的实现是无法提供此处描述的所有特性的。

POSIX 定时器 API 将定时器生命周期划分为如下几个阶段。

- 以系统调用 timer_create()创建一个新定时器，并定义其到期时对进程的通知方法。
- 以系统调用 timer_settime()来启动或停止一个定时器。
- 以系统调用 timer_delete()删除不再需要的定时器。

由 fork()创建的子进程不会继承 POSIX 定时器。调用 exec()期间亦或进程终止时将停止并删除定时器。

Linux 上，调用 POSIX 定时器 API 的程序编译时应使用-lrt 选项，从而与 librt（实时）函数库相链接。

23.6.1 创建定时器：timer_create()

函数 timer_create()创建一个新定时器，并以由 clockid 指定的时钟来进行时间度量。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);
```

Returns 0 on success, or −1 on error

设置参数 clockid，可以使用表 23-1 中的任意值，也可以采用 clock_getcpuclocid()或 pthread_getcpuclockid()返回的 clockid 值。函数返回时会在参数 timerid 所指向的缓冲区中放置定时器句柄（handle），供后续调用中指代该定时器之用。这一缓冲区的类型为 timer_t，是一

更多资源请访问稀酷客(www.ckook.com)

种由 SUSv3 定义的数据类型，用于标识定时器。

参数 evp 可决定定时器到期时对应用程序的通知方式，指向类型为 sigevent 的数据结构，具体定义如下：

```
union sigval {
    int   sival_int;              /* Integer value for accompanying data */
    void *sival_ptr;              /* Pointer value for accompanying data */
};

struct sigevent {
    int          sigev_notify;    /* Notification method */
    int          sigev_signo;     /* Timer expiration signal */
    union sigval sigev_value;     /* Value accompanying signal or
                                     passed to thread function */
    union {
        pid_t      _tid;          /* ID of thread to be signaled /
        struct {
            void (*_function) (union sigval);
                                  /* Thread notification function */
            void  *_attribute;    /* Really 'pthread_attr_t *' */
        } _sigev_thread;
    } _sigev_un;
};

#define sigev_notify_function    _sigev_un._sigev_thread._function
#define sigev_notify_attributes  _sigev_un._sigev_thread._attribute
#define sigev_notify_thread_id   _sigev_un._tid
```

可以表 23-2 所示值之一来设置结构中的 sigev_notify 字段。

表 23-2：sigevent 结构中 sigev_notify 字段的值

sigev_notify 的值	通 知 方 法	SUSv3
SIGEV_NONE	不通知；使用 timer_gettime()监测定时器	●
SIGEV_SIGNAL	发送 sigev_signo 信号给进程	●
SIGEV_THREAD	调用 sigev_notify_function 作为新线程的启动函数	●
SIGEV_THREAD_ID	发送 sigev_signo 信号给 sigev_notify_thread_id 所标识的线程	

关于 sigev_notify 常量值的更多细节，以及 sigval 结构中与每个常量值相关的字段，特做如下说明。

SIGEV_NONE

不提供定时器到期通知。进程可以使用 timer_gettime()来监控定时器的运转情况。

SIGEV_SIGNAL

定时器到期时，为进程生成指定于 sigev_signo 中的信号。如果 sigev_signal 为实时信号，那么 sigev_value 字段则指定了信号的伴随数据（整型或指针）（22.8.1 节）。通过 siginfo_t 结构的 si_value 可获取这一数据，至于 siginfo_t 结构，既可以直接传递给该信号的处理器函数，也可以由调用 sigwaitinfo()或 sigtimerdwait()返回。

SIGEV_THREAD

定时器到期时，会调用由 sigev_notify_function 字段指定的函数。调用该函数类似于调用新线程的启动函数。上述措词摘自 SUSv3，即允许系统实现以如下两种方式为周期性定时器

更多资源请访问稀酷客(www.ckook.com)

产生通知：要么将每个通知分别传递给一个唯一的新线程，要么将通知成系列发送给单个新线程。可将 sigev_notify_attribytes 字段置为 NULL，或是指向 pthread_attr_t 结构的指针，并在结构中定义线程属性。在 sigev_value 中设定的联合体 sigval 值是传递给函数的唯一参数。

SIGEV_THREAD_ID

这与 SIGEV_SIGNAL 相类似，只是发送信号的目标线程 ID 要与 sigev_notify_thread_id 相匹配。该线程应与调用线程同属一个进程。（伴随 SIGEV_SIGNAL 通知，会将信号置于针对整个进程的一个队列中排队，并且，如果进程包含多条线程，那么可将信号传递给进程中的任意线程。）可用 clone()或 gettid()的返回值对 sigev_notify_thread_id 赋值。设计 SIGEV_THREAD_ID 标志，意在供线程库使用。（要求线程实现使用 28.2.1 节描述的 CLONE_THREAD 选项。现代 NPTL 线程实现采用了 CLONE_THREAD，但较老的 LinuxThreads 线程则没有。）

除去 Linux 系统特有的 SIGEV_THREAD_ID 之外，SUSv3 定义了上述所有常量。

将参数 evp 置为 NULL，这相当于将 sigev_notify 置为 SIGEV_SIGNAL，同时将 sigev_signo 置为 SIGALRM（这与其他系统可能会有出入，因为 SUSv3 的措词是：一个缺省的信号值），并将 sigev_value.sival_int 置为定时器 ID。

在当前实现中，内核会为每个用 timer_create()创建的 POSIX 定时器在队列中预分配一个实时信号结构。之所以要采取预分配，旨在确保当定时器到期时，至少有一个有效结构可服务于所产生的队列化信号。这也意味着可以创建的 POSIX 定时器数量受制于排队实时信号的数量（参考 22.8 节）。

23.6.2 配备和解除定时器：timer_settime()

一旦创建了定时器，就可以使用 timer_settime()对其进行配备（启动）或解除（停止）。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
                  struct itimerspec *old_value);
```

Returns 0 on success, or −1 on error

函数 timer_settime()的参数 timerid 是一个定时器句柄（handle），由之前对 timer_create()的调用返回。

参数 value 和 old_value 则类似于函数 setitimer()的同名参数：value 中包含定时器的新设置，old_value 则用于返回定时器的前一设置（参考稍后对 timer_gettime()的说明）。如果对定时器的前一设置不感兴趣，可将 old_value 设为 NULL。参数 value 和 old_value 都是指向结构 itimerspec 的指针，该结构定义如下：

```
struct itimerspec {
    struct timespec it_interval;    /* Interval for periodic timer */
    struct timespec it_value;       /* First expiration */
};
```

结构 itimerspec 中的所有字段都是 timespec 类型的结构，用秒和纳秒来指定时间：

```
struct timespec {
    time_t tv_sec;                  /* Seconds */
    long   tv_nsec;                 /* Nanoseconds */
};
```


更多资源请访问稀酷客(www.ckook.com)

it_value 指定了定时器首次到期的时间。如果 it_interval 的任一子字段非 0，那么这就是一个周期性定时器，在经历了由 it_value 指定的初次到期后，会按这些子字段指定的频率周期性到期。如果 it_interval 的下属字段均为 0，那么这个定时器将只到期一次。

若将 flags 置为 0，则会将 value.it_value 视为始于 timer_settime()（与 setitimer()类似）调用时间点的相对值。如果将 flags 设为 TIMER_ABSTIME，那么 value.it_value 则是一个绝对时间（从时钟值 0 开始）。一旦时钟过了这一时间，定时器会立即到期。

为了启动定时器，需要调用函数 timer_settime()，并将 value.it_value 的一个或全部下属字段设为非 0 值。如果之前曾经配备过定时器，timer_settime()会将之前的设置替换掉。

如果定时器的值和间隔时间并非对应时钟分辨率（由 clock_getres()返回）的整数倍，那么会对这些值做向上取整处理。

定时器每次到期时，都会按特定方式通知进程，这种方式由创建定时器的 timer_create()定义。如果结构 it_interval 包含非 0 值，那么会用这些值来重新加载 it_value 结构。

要解除定时器，需要调用 timer_settime()，并将 value.it_value 的所有字段指定为 0。

23.6.3 获取定时器的当前值：timer_gettime()

系统调用 timer_gettime()返回由 timerid 指定 POSIX 定时器的间隔以及剩余时间。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
```

Returns 0 on success, or −1 on error

curr_value 指针所指向的 itimerspec 结构中返回的是时间间隔以及距离下次定时器到期的时间。即使是以 TIMER_ABSTIME 标志创建的绝对时间定时器，在 curr_value.it_value 字段中返回的也是距离定时器下次到期的时间值。

如果返回结构 curr_value.it_value 的两个字段均为 0，那么定时器当前处于停止状态。如果返回结构 curr_value.it_interval 的两个字段都是 0，那么该定时器仅在 curr_value.it_value 给定的时间到期过一次。

23.6.4 删除定时器：timer_delete()

每个 POSIX 定时器都会消耗少量系统资源。所以，一旦使用完毕，应当用 timer_delete()来移除定时器并释放这些资源。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_delete(timer_t timerid);
```

Returns 0 on success, or −1 on error

参数 timerid 是之前调用 timer_create()时返回的句柄。对于已启动的定时器，会在移除前自动将其停止。如果因定时器到期而已经存在待定（pending）信号，那么信号会保持这一状态。（SUSv3 对此并未加以规范，所以其他的一些 UNIX 实现可能会有不同行为。）当进程终止时，会自动删除所有定时器。

更多资源请访问稀酷客(www.ckook.com)

23.6.5 通过信号发出通知

如果选择通过信号来接收定时器通知，那么处理这些信号时既可以采用信号处理器函数，也可以调用 sigwaitinfo()或是 sigtimerdwait()。接收进程借助于这两种方法可以获得一个 siginfo_t 结构（21.4 节），其中包含与信号相关的深入信息。（要在信号处理器函数中使用这种特性，创建信号处理器函数时需设置 SA_SIGINFO 标志。）在结构 siginfo_t 中设置如下字段。

- si_signo：包含由定时器产生的信号。
- si_code：置为 SI_TIMER，表示这是因 POSIX 定时器到期而产生的信号。
- si_value：将该字段置为以 timer_create()创建定时器时在 evp.sigev_value 中提供的值。

为 evp.sigev_value 指定不同的值，可以将到期时发送同类信号的不同定时器区分开来。

调用 timer_create()时，通常将 evp.sigev_value.sival_ptr 赋值为当前调用中参数 timerid 的地址（见程序清单 23-5）。从而允许信号处理器函数（或 sigwaitinfo()调用）获得产生信号的定时器 ID。（另外，也可以将调用函数 timer_create()时给定的 timerid 参数置于一结构中，并将结构地址赋予 evp.sigev_value.sival_ptr。）

Linux 还为 siginfo_t 结构提供了如下非标准字段。

- si_overrun：包含了定时器溢出个数（在 23.6.6 节中说明）。

Linux 还支持另一个非标准字段 si_timerid，其中包含一个标识符，供系统内部识别定时器之用（与 timer_create()返回的 ID 不同）。对于应用程序来说没什么用处。

程序清单 23-5 所演示的是使用信号作为 POSIX 定时器的通知机制。

程序清单 23-5：使用信号进行 POSIX 定时器通知

timers/ptmr_sigev_signal.c

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>
#include "curr_time.h"                  /* Declares currTime() */
#include "itimerspec_from_str.h"        /* Declares itimerspecFromStr() */
#include "tlpi_hdr.h"

#define TIMER_SIG SIGRTMAX              /* Our timer notification signal */

static void
① handler(int sig, siginfo_t *si, void *uc)
{
    timer_t *tidptr;

    tidptr = si->si_value.sival_ptr;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(); see Section 21.1.2) */
    printf("[%s] Got signal %d\n", currTime("%T"), sig);
    printf("    *sival_ptr         = %ld\n", (long) *tidptr);
    printf("    timer_getoverrun() = %d\n", timer_getoverrun(*tidptr));
}

int
main(int argc, char *argv[])
{
```

```
    struct itimerspec ts;
    struct sigaction  sa;
    struct sigevent   sev;
    timer_t *tidlist;
    int j;

    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);

    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

    /* Establish handler for notification signal */

    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
②   if (sigaction(TIMER_SIG, &sa, NULL) == -1)
        errExit("sigaction");

    /* Create and start one timer for each command-line argument */

    sev.sigev_notify = SIGEV_SIGNAL;    /* Notify via signal */
    sev.sigev_signo = TIMER_SIG;        /* Notify using this signal */

    for (j = 0; j < argc - 1; j++) {
③       itimerspecFromStr(argv[j + 1], &ts);

        sev.sigev_value.sival_ptr = &tidlist[j];
                    /* Allows handler to get ID of this timer */

④       if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);

⑤       if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }

⑥   for (;;)                            /* Wait for incoming timer signals */
        pause();
}
```

timers/ptmr_sigev_signal.c

程序清单 23-5 程序的每个命令行参数都为定时器指定了初始值及间隔时间。程序的“用法”输出中描述了这些参数的语法，并在后面的 shell 会话中做了演示。程序执行的步骤如下。

- 为用于定时器通知的信号创建处理器函数②。
- 为每一个命令行参数，创建④并配备⑤一个使用 SIGEV_SIGNAL 通知机制的 POSIX 定时器。至于将命令行参数转换③为 itimerspec 结构的函数 itimerspecFromStr()，请参考程序清单 23-6。
- 每当一个定时器到期时，都将发送由 sev.sigev_signo 指定的信号给进程。信号处理器函数会将 sev.sigev_value.sival_ptr 中提供的值（定时器 ID，tidlist[j]）以及定时器溢出值①显示出来。
- 创建并配备定时器之后，在循环中反复调用 pause()，以等待定时器到期⑥。

更多资源请访问稀酷客(www.ckook.com)

程序清单 23-6 中函数可将程序 23-5 的命令行参数转化为相应的 itimerspec 结构。函数可识别的字符串参数格式在源码文件开始的注释中做了说明（并在下面的 shell 会话中做了演示）。

程序清单 23-6：将“时间+间隔”的字符串转换为 itimerspec 的值

timers/itimerspec_from_str.c

```
#define_POSIX_C_SOURCE 199309
#include <string.h>
#include <stdlib.h>
#include "itimerspec_from_str.h"        /* Declares function defined here */

/* Convert a string of the following form to an itimerspec structure:
   "value.sec[/value.nanosec][:interval.sec[/interval.nanosec]]".
   Optional components that are omitted cause 0 to be assigned to the
   corresponding structure fields. */

void
itimerspecFromStr(char *str, struct itimerspec *tsp)
{
    char *cptr, *sptr;

    cptr = strchr(str, ':');
    if (cptr != NULL)
        *cptr = '\0';

    sptr = strchr(str, '/');
    if (sptr != NULL)
        *sptr = '\0';

    tsp->it_value.tv_sec = atoi(str);
    tsp->it_value.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
    if (cptr == NULL) {
        tsp->it_interval.tv_sec = 0;
        tsp->it_interval.tv_nsec = 0;
    } else {
        sptr = strchr(cptr + 1, '/');
        if (sptr != NULL)
            *sptr = '\0';
        tsp->it_interval.tv_sec = atoi(cptr + 1);
        tsp->it_interval.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
    }
}
```

timers/itimerspec_from_str.c

如下 shell 会话演示了对程序清单 23-5 中程序的调用，创建了一个初始到期值为 2 秒，间隔时间为 5 秒的定时器。

```
$ ./ptmr_sigev_signal 2:5
Timer ID: 134524952 (2:5)
[15:54:56] Got signal 64                    SIGRTMAX is signal 64 on this system
    *sival_ptr         = 134524952          sival_ptr points to the variable tid
    timer_getoverrun() = 0
[15:55:01] Got signal 64
    *sival_ptr         = 134524952
    timer_getoverrun() = 0
Type Control-Z to suspend the process
[1]+  Stopped        ./ptmr_sigev_signal 2:5
```

挂起程序后，暂停几秒钟，在恢复程序运行之前会有多个定时器到期。

```
$ fg
./ptmr_sigev_signal 2:5
[15:55:34] Got signal 64
    *sival_ptr         = 134524952
    timer_getoverrun() = 5
Type Control-C to kill the program
```

程序输出的最后一行表明发生了 5 次定时器溢出，亦即在捕获上一信号之后定时器到期了 6 次。

23.6.6 定时器溢出

假设已经选择通过信号（即 sigev_notify 为 SIGEV_SIGNAL）传递的方式来接收定时器到期通知。进一步假设，在捕获或接收相关信号之前，定时器到期多次。这可能是因为进程再次获得调度前的延时所致。另外，不论是直接调用 sigprocmask()，还是在信号处理器函数里暗中处理，也都有可能堵塞相关信号的发送。如何知道发生了这些定时器溢出呢？

也许会认为使用实时信号有助于解决这个问题，因为可以对实时信号的多个实例进行排队。不过，由于对排队实时信号有数量上的限制，结果证明这种方法也无法奏效。所以 POSIX.1b 委员会选用了另一种方法：一旦选择通过信号来接收定时器通知，那么即便用了实时信号，也绝不会对该信号的多个实例进行排队。相反，在接收信号后（无论是通过信号处理器函数还是调用 sigwaitinfo()），可以获取定时器溢出计数，即在信号生成与接收之间发生的定时器到期额外次数。如果上次收到信号后定时器发生了 3 次到期，那么溢出计数是 2。

接收到定时器信号之后，有两种方法可以获取定时器溢出值。

- 调用 timer_getoverrun()，稍后将会讨论。这是由 SUSv3 指定去获取溢出计数的方法。
- 使用随信号一同返回的结构 siginfo_t 中的 si_overrun 字段值。这种方法可以避免 timer_getoverrun()的系统调用开销，但同时也是一种 Linux 扩展方法，无法移植。

每次收到定时器信号后，都会重置定时器溢出计数。若自处理或接收定时器信号之后，定时器仅到期一次，则溢出计数为 0（即无溢出）。

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_getoverrun(timer_t timerid);
```

Returns timer overrun count on success, or −1 on error

函数 timer_getoverrun()返回由参数 timerid 指定定时器的溢出值。

根据 SUSv3 规定（表 21-1），函数 timer_getoverrun()是异步信号安全的函数之一，故而在信号处理器函数内部调用也是安全的。

23.6.7 通过线程来通知

SIGEV_THREAD 标志允许程序从一个独立的线程中调用函数来获取定时器到期通知。要理解这一标志的含义，需要具备第 29 章和第 30 章中关于 POSIX 线程的知识。如果不了解 POSIX 线程，那么在查看本节示例程序前，可能需要预先阅读一下这些章节。

程序清单 23-7 演示了 SIGEV_THREAD 的使用。该程序的命令行参数与程序清单 23-5 相

更多资源请访问稀酷客(www.ckook.com)

同。所执行的步骤如下。

- 针对每个命令行参数，程序都创建⑥并配备⑦一个使用了 SIGEV_THREAD 通知机制③的 POSIX 定时器。
- 每当定时器到期时，会在一条独立线程中调用由 sev.sigev_notify_function 指定的函数。调用函数时，使用由 sev.sigev_value.sival_ptr 指定的值作为参数。程序中会将定时器 ID（tidlist[j]）的地址赋给该字段⑤，以便在调用通知函数时可以获得定时器 ID。
- 创建和配备所有定时器之后，主程序进入循环并等待定时器到期⑧。每次循环，程序都会调用 pthread_cond_wait()，等待处理定时器通知的线程就条件变量（cond）发出信号。
- 每次定时器到期都会调用函数 threadFunc()①。在打印消息后，增加全局变量 expireCnt 的值。考虑到定时器可能溢出，会将 timer_getoverrun()的返回值也加入 expireCnt 变量中。（23.6.6 节解释了定时器溢出与 SIGEV_SIGNAL 通知机制之间的关系。定时器溢出还可以与 SIGEV_THREAD 机制协作使用，因为在调用通知函数前，定时器可能会多次到期。）通知函数就条件变量（cond）发出信号，告知主程序定时器到期。

下面的 shell 会话日志展示了对程序清单 23-7 中程序的调用。在本例中，程序创建了两个定时器：一个定时器首次到期时间为 5 秒，并设置了 5 秒的时间间隔；另一个初次到期时间为 10 秒，并设置了 10 秒的时间间隔。

```
$ ./ptmr_sigev_thread 5:5 10:10
Timer ID: 134525024 (5:5)
Timer ID: 134525080 (10:10)
[13:06:22] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 1
[13:06:27] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 2
[13:06:27] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 3
Type Control-Z to suspend the program
[1]+  Stopped        ./ptmr_sigev_thread 5:5 10:10
$ fg                                          Resume execution
./ptmr_sigev_thread 5:5 10:10
[13:06:45] Thread notify
    timer ID=134525024
    timer_getoverrun()=2                      There were timer overruns
main(): count = 6
[13:06:45] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 7
Type Control-C to kill the program
```

程序清单 23-7：使用线程函数发送 POSIX 定时器通知

timers/ptmr_sigev_thread.c

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include "curr_time.h"              /* Declaration of currTime() */
```

```
#include "tlpi_hdr.h"
#include "itimerspec_from_str.h"    /* Declares itimerspecFromStr() */

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int expireCnt = 0;           /* Number of expirations of all timers */

static void                         /* Thread notification function */
① threadFunc(union sigval sv)
{
    timer_t *tidptr;
    int s;

    tidptr = sv.sival_ptr;

    printf("[%s] Thread notify\n", currTime("%T"));
    printf("    timer ID=%ld\n", (long) *tidptr);
    printf("    timer_getoverrun()=%d\n", timer_getoverrun(*tidptr));

    /* Increment counter variable shared with main thread and signal
       condition variable to notify main thread of the change. */

    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    expireCnt += 1 + timer_getoverrun(*tidptr);

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

②   s = pthread_cond_signal(&cond);
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    struct itimerspec ts;
    timer_t *tidlist;
    int s, j;
    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);

    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

③   sev.sigev_notify = SIGEV_THREAD;            /* Notify via thread */
④   sev.sigev_notify_function = threadFunc;     /* Thread start function */
    sev.sigev_notify_attributes = NULL;
            /* Could be pointer to pthread_attr_t structure */

    /* Create and start one timer for each command-line argument */
```

```
    for (j = 0; j < argc - 1; j++) {
        itimerspecFromStr(argv[j + 1], &ts);

⑤       sev.sigev_value.sival_ptr = &tidlist[j];
                /* Passed as argument to threadFunc() */

⑥       if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);

⑦       if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }

    /* The main thread waits on a condition variable that is signaled
       on each invocation of the thread notification function. We
       print a message so that the user can see that this occurred. */

    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

⑧   for (;;) {
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
        printf("main(): expireCnt = %d\n", expireCnt);
    }
}
```

timers/ptmr_sigev_thread.c

23.7 利用文件描述符进行通知的定时器：timerfd API

始于版本 2.6.25，Linux 内核提供了另一种创建定时器的 API。Linux 特有的 timerfd API，可从文件描述符中读取其所创建定时器的到期通知。因为可以使用 select()、poll()和 epoll()（将在第 63 章进行讨论）将这种文件描述符会同其他描述符一同进行监控，所以非常实用。（至于说本章讨论的其他定时器 API，想要把一个或多个定时器与一组文件描述符放在一起同时监测，可不是件容易的事。）

这组 API 中的 3 个新系统调用，其操作与 23.6 节所述的 timer_create()、timer_settime()和 timer_gettime()相类似。

新加入的第 1 个系统调用是 timerfd_create()，它会创建一个新的定时器对象，并返回一个指代该对象的文件描述符。

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

int timerfd_create(int clockid, int flags);
```

Returns file descriptor on success, or −1 on error

参数 clockid 的值可以设置为 CLOCK_REALTIME 或 CLOCK_MONOTONIC（参考表 23-1）。

timerfd_create()的最初实现将参数 flags 预留供未来使用，必须设置为 0。不过，Linux 内核从 2.6.27 版本开始支持下面两种 flags 标志。

TFD_CLOEXEC

为新的文件描述符设置运行时关闭标志（FD_CLOEXEC）。与 4.3.1 节介绍的 open()标志 O_CLOEXEC 适用于相同情况。

TFD_NONBLOCK

为底层的打开文件描述设置 O_NONBLOCK 标志，随后的读操作将是非阻塞式的。这样设置省却了对 fcntl()的额外调用，却能达到相同效果。

timerfd_create()创建的定时器使用完毕后，应调用 close()关闭相应的文件描述符，以便于内核能够释放与定时器相关的资源。

系统调用 timerfd_settime()可以配备（启动）或解除（停止）由文件描述符 fd 所指代的定时器。

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

int timerfd_settime(int fd, int flags, const struct itimerspec *new_value,
                    struct itimerspec *old_value);
```

Returns 0 on success, or −1 on error

参数 new_value 为定时器指定新设置。参数 old_value 可用来返回定时器的前一设置（细节请参考随后对 timerfd_gettime()的说明）。如果不关心定时器的前一设置，可将 old_value 置为 NULL。两个参数均指向 itimerspec 结构，用法与 timer_settime()（参考 23.6.2 节）相同。

参数 flags 与 timer_settime()中的对应参数类似。可以是 0，此时将 new_value.it_value 的值视为相对于调用 timerfd_settime()时间点的相对时间，也可以设为 TFD_TIMER_ABSTIME，将其视为一个绝对时间（从时钟的 0 点开始测量）。

系统调用 timerfd_gettime()返回文件描述符 fd 所标识定时器的间隔及剩余时间。

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

int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

Returns 0 on success, or −1 on error

同 timer_gettime()一样，间隔以及距离下次到期的时间均返回 curr_value 指向的结构 itimerspec 中。即使是以 TFD_TIMER_ABSTIME 标志创建的绝对时间定时器，curr_vallue.it_value 字段中返回值的意义也会保持不变。如果返回的结构 curr_value.it_value 中所有字段值均为 0，那么该定时器已经被解除。如果返回的结构 curr_value.it_interval 中两字段值均为 0，那么定时器只会到期一次，到期时间在 curr_value.it_value 中给出。

timerfd 与 fork()及 exec()之间的交互

调用 fork()期间，子进程会继承 timerfd_create()所创建文件描述符的拷贝。这些描述符与父进程的对应描述符均指代相同的定时器对象，任一进程都可读取定时器的到期信息。

timerfd_create()创建的文件描述符能跨越 exec()得以保存（除非将描述符置为运行时关闭，如 27.4 节所述），已配备的定时器在 exec()之后会继续生成到期通知。

从 timerfd 文件描述符读取

一旦以 timerfd_settime()启动了定时器，就可以从相应文件描述符中调用 read()来读取定时

更多资源请访问稀酷客(www.ckook.com)

器的到期信息。出于这一目的，传给 read()的缓冲区必须足以容纳一个无符号 8 字节整型（uint64_t）数。

在上次使用 timerfd_settime()修改设置以后，或是最后一次执行 read()后，如果发生了一起到多起定时器到期事件，那么 read()会立即返回，且返回的缓冲区中包含了已发生的到期次数。如果并无定时器到期，read()会一直阻塞直至产生下一个到期。也可以执行 fcntl()的 F_SETFL 操作（5.3 节）为文件描述符设置 O_NONBLOCK 标志，这时的读动作是非阻塞式的，且如果没有定时器到期，则返回错误，并将 errno 值置为 EAGAIN。

如前所述，可以利用 select()、poll()和 epoll()对 timerfd 文件描述符进行监控。如果定时器到期，会将对应的文件描述符标记为可读。

示例程序

程序清单 23-8 演示了 timerfd API 的使用。该程序从命令行取得两个参数。第 1 个参数为必填项，用以标识定时器的初始和间隔时间。（程序清单 23-6 中的函数 itimerspecFromStr()可以用来解析这一参数。）第 2 个参数是可选项，表示程序退出之前应等待的定时器过期最大次数，其默认值为 1。

程序调用 timerfd_create()来创建一个定时器，并通过 timerfd_settime()将其启动。接着进入循环，从文件描述符中读取定时器到期通知，直至达到指定的定时器到期次数。每次 read()之后，程序都会显示定时器启动以来的逝去时间、读取到的到期次数以及至今为止的到期总数。

下面的 shell 会话日志中，通过命令行参数创建了一个初始时间为 1 秒，间隔为 1 秒，最大到期次数为 100 次的定时器。

```
$ ./demo_timerfd 1:1 100
1.000: expirations read: 1; total=1
2.000: expirations read: 1; total=2
3.000: expirations read: 1; total=3
Type Control-Z to suspend program in background for a few seconds
[1]+  Stopped                 ./demo_timerfd 1:1 100
$ fg                                          Resume program in foreground
./demo_timerfd 1:1 100
14.205: expirations read: 11; total=14        Multiple expirations since last read()
15.000: expirations read: 1; total=15
16.000: expirations read: 1; total=16
Type Control-C to terminate the program
```

从以上结果可以看出，程序在后台暂停时定时器出现了多次到期。在程序恢复运行之后，第 1 次 read()调用就返回了所有这些到期。

程序清单 23-8：使用 timerfd API

timers/demo_timerfd.c

```
#include <sys/timerfd.h>
#include <time.h>
#include <stdint.h>                    /* Definition of uint64_t */
#include "itimerspec_from_str.h"       /* Declares itimerspecFromStr() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct itimerspec ts;
```

```
    struct timespec start, now;
    int maxExp, fd, secs, nanosecs;
    uint64_t numExp, totalExp;
    ssize_t s;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]] [max-exp]\n", argv[0]);

    itimerspecFromStr(argv[1], &ts);
    maxExp = (argc > 2) ? getInt(argv[2], GN_GT_0, "max-exp") : 1;
    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        errExit("timerfd_create");

    if (timerfd_settime(fd, 0, &ts, NULL) == -1)
        errExit("timerfd_settime");

    if (clock_gettime(CLOCK_MONOTONIC, &start) == -1)
        errExit("clock_gettime");

    for (totalExp = 0; totalExp < maxExp;) {

        /* Read number of expirations on the timer, and then display
           time elapsed since timer was started, followed by number
           of expirations read and total expirations so far. */

        s = read(fd, &numExp, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            errExit("read");

        totalExp += numExp;

        if (clock_gettime(CLOCK_MONOTONIC, &now) == -1)
            errExit("clock_gettime");

        secs = now.tv_sec - start.tv_sec;
        nanosecs = now.tv_nsec - start.tv_nsec;
        if (nanosecs < 0) {
            secs--;
            nanosecs += 1000000000;
        }

        printf("%d.%03d: expirations read: %llu; total=%llu\n",
                secs, (nanosecs + 500000) / 1000000,
                (unsigned long long) numExp, (unsigned long long) totalExp);
    }

    exit(EXIT_SUCCESS);
}
```

timers/demo_timerfd.c

23.8 总结

进程可以使用 setitimer()或 alarm()来设置定时器，以便于在经历指定的一段实际（或进程）时间后收到信号通知。定时器的用途之一是为系统调用的阻塞设定时间上限。

应用程序如需暂停执行一段特定间隔的实际时间，可以使用各种合适的休眠函数。

更多资源请访问稀酷客(www.ckook.com)

Linux 2.6 所实现的 POSIX.1b 扩展为高精度时钟和定时器定义了一套 API。POSIX.1b 定时器比传统（settimer()）UNIX 定时器更具优势，可以：创建多个定时器；选择定时器到期时的通知信号；获取定时器溢出计数，以便判断自上次到期通知后定时器是否又发生了多次到期；选择通过执行线程函数而非递送信号来获取定时器通知。

Linux 特有的 timerfd API 提供了一组创建定时器的接口，与 POSIX 定时器 API 相类似，但允许从文件描述符中读取定时器通知。还可使用 select()、poll()和 epoll()来监控这些描述符。

更多信息

在每个函数的原理（rationael）部分，SUSv3 就本章所述（标准）定时器和休眠接口一一指出其要点所在。[Callmeister, 1995]则探讨了 POSIX.1b 时钟和定时器。

23.9 练习

23-1. 尽管 Linux 将 alarm()实现为系统调用，但这当属蛇足。请用 setitimer()实现 alarm()。

23-2. 试着将程序清单 23-3（t_nanosleep.c）程序置于后台运行，并设置 60 秒的休眠间隔，同时使用如下命令发送尽可能多的 SIGINT 信号给后台进程：

```
$ while true; do kill -INT pid; done
```

应该能注意到程序休眠时间要长于预期。将 nanosleep()用 clock_gettime()（使用 CLOCK_REALTIME 时钟）和设置 TIMER_ABSTIME 标志的 clock_nanosleep()来替换。（此练习需要 Linux 2.6 版本。）反复测试修改后的程序，并解释新老程序间的差别。

23-3. 编写一个程序验证：如果调用 timer_create()时将参数 evp 置为 NULL，那么这就等同于将 evp 设为指向 sigevent 结构的指针，并将该结构中的 sigev_notify 置为 SIGEV_SIGNAL，将 sigev_signo 置为 SIGALRM，将 si_value.sival_int 置为定时器 ID。

23-4. 修改程序清单 23-5（ptmr_sigev_signal.c）中程序，并用 sigwaitinfo()替换信号处理器函数。

更多资源请访问稀酷客(www.ckook.com)

第24章

进程的创建

本章以及随后的3章将探讨进程的创建和终止，以及进程执行新程序的过程。本章主要讨论进程的创建，不过，在切入正题之前，将首先概括一下这4章所涵盖的主要系统调用。

24.1 fork()、exit()、wait()以及execve()的简介

本章以及随后几章的议题会集中在fork()、exit()、wait()以及execve()这几个系统调用上。上述每种系统调用都各有变体，后续会一一论及。此处将首先对这4个系统调用及其典型用法简单加以介绍。

- 系统调用fork()允许一进程（父进程）创建一新进程（子进程）。具体做法是，新的子进程几近于对父进程的翻版：子进程获得父进程的栈、数据段、堆和执行文本段（6.3节）的拷贝。可将此视为把父进程一分为二，术语fork也由此得名。
- 库函数exit（status）终止一进程，将进程占用的所有资源（内存、文件描述符等）归还内核，交其进行再次分配。参数status为一整型变量，表示进程的退出状态。父进程可使用系统调用wait()来获取该状态。

> 库函数exit()位于系统调用_exit()之上。第25章将解释二者之间的差异。这里只是强调，在调用fork()之后，父、子进程中一般只有一个会通过调用exit()退出，而另一进程则应使用_exit()终止。

- 系统调用wait（&status）的目的有二：其一，如果子进程尚未调用exit()终止，那么wait()会挂起父进程直至子进程终止；其二，子进程的终止状态通过wait()的status参数返回。
- 系统调用execve(pathname，argv，envp)加载一个新程序（路径名为pathname，参数列表为argv，环境变量列表为envp）到当前进程的内存。这将丢弃现存的程序文本段，并为新程序重新创建栈、数据段以及堆。通常将这一动作称为执行（execing）一个新程序。稍后会介绍构建于execve()之上的多个库函数，每种都为编程接口提供了实用的变体。在彼此差异无关宏旨的场合，循例会将此类函数统称为exec()，尽管实际上并没有以之命名的系统调用或者库函数。

更多资源请访问稀酷客(www.ckook.com)

其他一些操作系统则将 fork()和 exec()的功能合二为一，形成单一的 spawn 操作——创建一个新进程并执行指定程序。比较而言，UNIX 的方案通常更为简单和优雅。两步走的策略使得 API 更为简单（系统调用 fork()无需参数），程序也得以在这两步之间执行一些其他操作，因而更具弹性。另外，只执行 fork()而不执行 exec()的场景也颇为常见。

SUSv3 所详细规定的 posix_spawn()函数，就将 fork()和 exec()的功能结合起来，但规范并未对实现此函数做强制要求。Linux 的 glibc 函数库实现了该函数以及 SUSv3 中的其他几个相关 API。将 posix_spawn()纳入 SUSv3，意在为缺乏交换（swap）设施或内存管理单元（memory-management units）的硬件架构（嵌入式系统大多如此）编写具备可移植性的应用程序。在此类架构上实现传统意义的 fork()，即便存在可能性，难度也很大。

图 24-1 对 fork()、exit()、wait()以及 execve()之间的相互协同作了总结。（此图勾勒了 shell 执行一条命令所历经的步骤：shell 读取命令，进行各种处理，随之创建子进程以执行该命令，如此循环不已。）

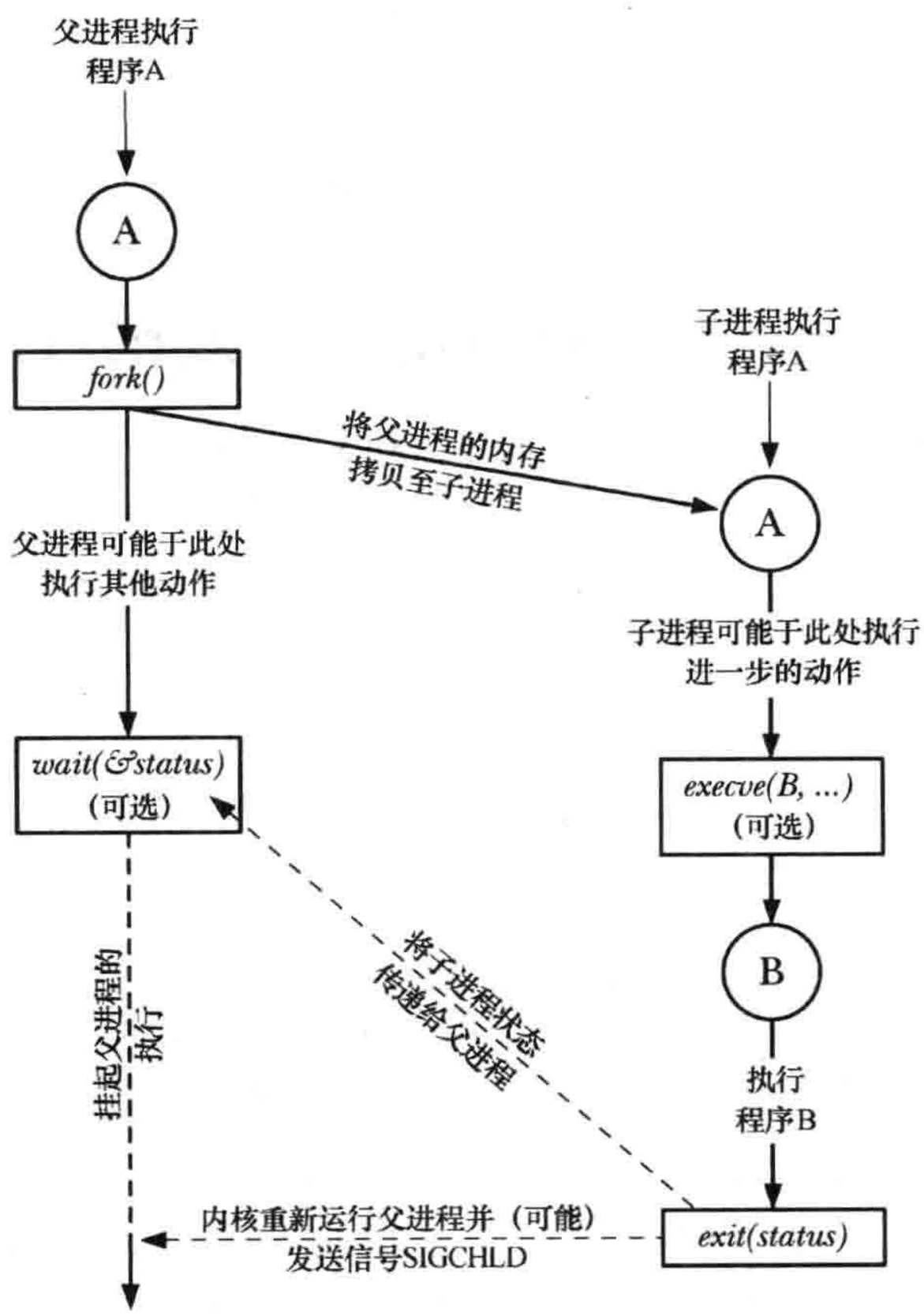

图 24-1：概述函数 fork()、exit()、wait()和 execve()的协同使用

图中对 execve()的调用并非必须。有时，让子进程继续执行与父进程相同的程序反而会有妙用。最终，两种情况殊途同归：总是要通过调用 exit()（或接收一个信号）来终止子进程，而父进程可调用 wait()来获取其终止状态。

同样，对 wait()的调用也属于可选项。父进程可以对子进程不闻不问，继续我行我素。不

过，由后续内容可知，对 wait()的使用通常也是不可或缺的，每每在 SIGCHLD 信号的处理程序中使用。当子进程终止时，内核会为其父进程产生此类信号（默认的处理是忽略 SIGCHLD 信号，下图将此标记为可选，原因正在于此）。

24.2 创建新进程：fork()

在诸多应用中，创建多个进程是任务分解时行之有效的方法。例如，某一网络服务器进程可在侦听客户端请求的同时，为处理每一请求而创建一新的子进程，与此同时，服务器进程会继续侦听更多的客户端连接请求。以此类手法分解任务，通常会简化应用程序的设计，同时提高了系统的并发性。（即，可同时处理更多的任务或请求。）

系统调用 fork()创建一新进程（child），几近于对调用进程（parent）的翻版。

```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or -1 on error;
in successfully created child: always returns 0

理解 fork()的诀窍是，要意识到，完成对其调用后将存在两个进程，且每个进程都会从 fork()的返回处继续执行。

这两个进程将执行相同的程序文本段，但却各自拥有不同的栈段、数据段以及堆段拷贝。子进程的栈、数据以及栈段开始时是对父进程内存相应各部分的完全复制。执行 fork()之后，每个进程均可修改各自的栈数据、以及堆段中的变量，而并不影响另一进程。

程序代码则可通过 fork()的返回值来区分父、子进程。在父进程中，fork()将返回新创建子进程的进程 ID。鉴于父进程可能需要创建，进而追踪多个子进程（通过 wait()或类似方法），这种安排还是很实用的。而 fork()在子进程中则返回 0。如有必要，子进程可调用 getpid()以获取自身的进程 ID，调用 getppid()以获取父进程 ID。

当无法创建子进程时，fork()将返回-1。失败的原因可能在于，进程数量要么超出了系统针对此真实用户（real user ID）在进程数量上所施加的限制（RLIMIT_NPROC，36.3 节将对此加以描述），要么是触及允许该系统创建的最大进程数这一系统级上限。

调用 fork()时，有时会采用如下习惯用语：

```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

调用 fork()之后，系统将率先“垂青”于哪个进程（即调度其使用 CPU），是无法确定的，意识到这一点极为重要。在设计拙劣的程序中，这种不确定性可能会导致所谓“竞争条件（race condition）”的错误，24.2 节会对此做进一步说明。

更多资源请访问稀酷客(www.ckook.com)

程序清单 24-1 展示了 fork()的用法。该程序创建一子进程，并对继承自 fork()的全局及自动变量拷贝进行修改。

使用 sleep()（存在于由父进程所执行的代码中），意在允许子进程先于父进程获得系统调度并使用 CPU，以便在父进程继续运行之前完成自身任务并退出。要想确保这一结果，sleep()的这种用法并非万无一失，24.5 节中的方法更胜一筹。

运行程序清单 24-1 中程序，其输出如下。

```
$ ./t_fork
PID=28557 (child)  idata=333 istack=666
PID=28556 (parent) idata=111 istack=222
```

以上输出表明，子进程在 fork()时拥有了自己的栈和数据段拷贝，且其对这些段中变量的修改将不会影响父进程。

程序清单 24-1：调用 fork()

procexec/t_fork.c

```
#include "tlpi_hdr.h"

static int idata = 111;             /* Allocated in data segment */

int
main(int argc, char *argv[])
{
    int istack = 222;               /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);                   /* Give child a chance to execute */
        break;
    }

    /* Both parent and child come here */

    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);

    exit(EXIT_SUCCESS);
}
```

procexec/t_fork.c

24.2.1 父、子进程间的文件共享

执行 fork()时，子进程会获得父进程所有文件描述符的副本。这些副本的创建方式类似于 dup()，这也意味着父、子进程中对应的描述符均指向相同的打开文件句柄（即 open file description，详见 5.4 节译注）。正如 5.4 节所述，打开文件句柄包含有当前文件偏移量（由 read()、write()和 lseek()修改）以及文件状态标志（由 open()设置，通过 fcntl()的 F_SETFL 操作改变）。

更多资源请访问稀酷客(www.ckook.com)

一个打开文件的这些属性因之而在父子进程间实现了共享。举例来说，如果子进程更新了文件偏移量，那么这种改变也会影响到父进程中相应的描述符。

程序清单 24-2 所展示的正是这样一个事实：fork()之后，这些属性将在父子进程之间共享。该程序使用 mkstemp()打开一个临时文件，接着调用 fork()以创建子进程。子进程改变文件偏移量以及文件状态标志，最后退出。父进程随即获取文件偏移量和标志，以验证其可以观察到由子进程所造成的变化。此程序运行结果如下：

```
$ ./fork_file_sharing
File offset before fork(): 0
O_APPEND flag before fork() is: off
Child has exited
File offset in parent: 1000
O_APPEND flag in parent is: on
```

关于程序清单 24-2 为何要将 lseek()的返回值强制转换为 long long，参见 5.10 节。

程序清单 24-2：在父子进程间共享文件偏移量和打开文件状态标志

procexec/fork_file_sharing.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd, flags;
    char template[] = "/tmp/testXXXXXX";

    setbuf(stdout, NULL);                   /* Disable buffering of stdout */

    fd = mkstemp(template);
    if (fd == -1)
        errExit("mkstemp");

    printf("File offset before fork(): %lld\n",
            (long long) lseek(fd, 0, SEEK_CUR));

    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");
    printf("O_APPEND flag before fork() is: %s\n",
            (flags & O_APPEND) ? "on" : "off");
    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:     /* Child: change file offset and status flags */
        if (lseek(fd, 1000, SEEK_SET) == -1)
            errExit("lseek");

        flags = fcntl(fd, F_GETFL);         /* Fetch current flags */
        if (flags == -1)
            errExit("fcntl - F_GETFL");
        flags |= O_APPEND;                  /* Turn O_APPEND on */
        if (fcntl(fd, F_SETFL, flags) == -1)
```

```
        errExit("fcntl - F_SETFL");
    _exit(EXIT_SUCCESS);

default:    /* Parent: can see file changes made by child */
    if (wait(NULL) == -1)
        errExit("wait");                /* Wait for child exit */
    printf("Child has exited\n");

    printf("File offset in parent: %lld\n",
            (long long) lseek(fd, 0, SEEK_CUR));

    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");
    printf("O_APPEND flag in parent is: %s\n",
            (flags & O_APPEND) ? "on" : "off");
    exit(EXIT_SUCCESS);
    }
}
```

procexec/fork_file_sharing.c

父子进程间共享打开文件属性的妙用屡见不鲜。例如，假设父子进程同时写入一文件，共享文件偏移量会确保二者不会覆盖彼此的输出内容。不过，这并不能阻止父子进程的输出随意混杂在一起。要想规避这一现象，需要进行进程间同步。比如，父进程可以使用系统调用 wait()来暂停运行并等待子进程退出。shell 就是这么做的：只有当执行命令的子进程退出后，shell 才会打印出提示符（除非用户在命令行最后加上&符以显式在后台运行命令）。

如果不需要这种对文件描述符的共享方式，那么在设计应用程序时，应于 fork()调用后注意两点：其一，令父、子进程使用不同的文件描述符；其二，各自立即关闭不再使用的描述符（亦即那些经由其他进程使用的描述符）。如果进程之一执行了 exec()，那么 27.4 节所描述的执行时关闭功能（close-on-exec）也会很有用处。图 24-2 展示了这些步骤。

24.2.2 fork()的内存语义

从概念上说来，可以将 fork()认作对父进程程序段、数据段、堆段以及栈段创建拷贝。的确，在一些早期的 UNIX 实现中，此类复制确实是原汁原味：将父进程内存拷贝至交换空间，以此创建新进程映像（image），而在父进程保持自身内存的同时，将换出映像置为子进程。不过，真要是简单地将父进程虚拟内存页拷贝到新的子进程，那就太浪费了。原因有很多，其中之一是：fork()之后常常伴随着 exec()，这会用新程序替换进程的代码段，并重新初始化其数据段、堆段和栈段。大部分现代 UNIX 实现（包括 Linux）采用两种技术来避免这种浪费。

- 内核（Kernel）将每一进程的代码段标记为只读，从而使进程无法修改自身代码。这样，父、子进程可共享同一代码段。系统调用 fork()在为子进程创建代码段时，其所构建的一系列进程级页表项（page-table entries）均指向与父进程相同的物理内存页帧。
- 对于父进程数据段、堆段和栈段中的各页，内核采用写时复制（copy-on-write）技术来处理。（[Bach, 1986]和[Bovert & Cersati, 2005]描述了写时复制的实现。）最初，内核做了一些设置，令这些段的页表项指向与父进程相同的物理内存页，并将这些页面自身标记为只读。调用 fork()之后，内核会捕获所有父进程或子进程针对这些页面的修改企图，并为将要修改的（about-to-be-modified）页面创建拷贝。系统将新的页面拷贝分配给遭内核捕获的进程，还会对子进程的相应页表项做适当调整。从这一刻起，

更多资源请访问稀酷客(www.ckook.com)

父、子进程可以分别修改各自的页拷贝，不再相互影响。图 24-3 展示了写时复制技术。

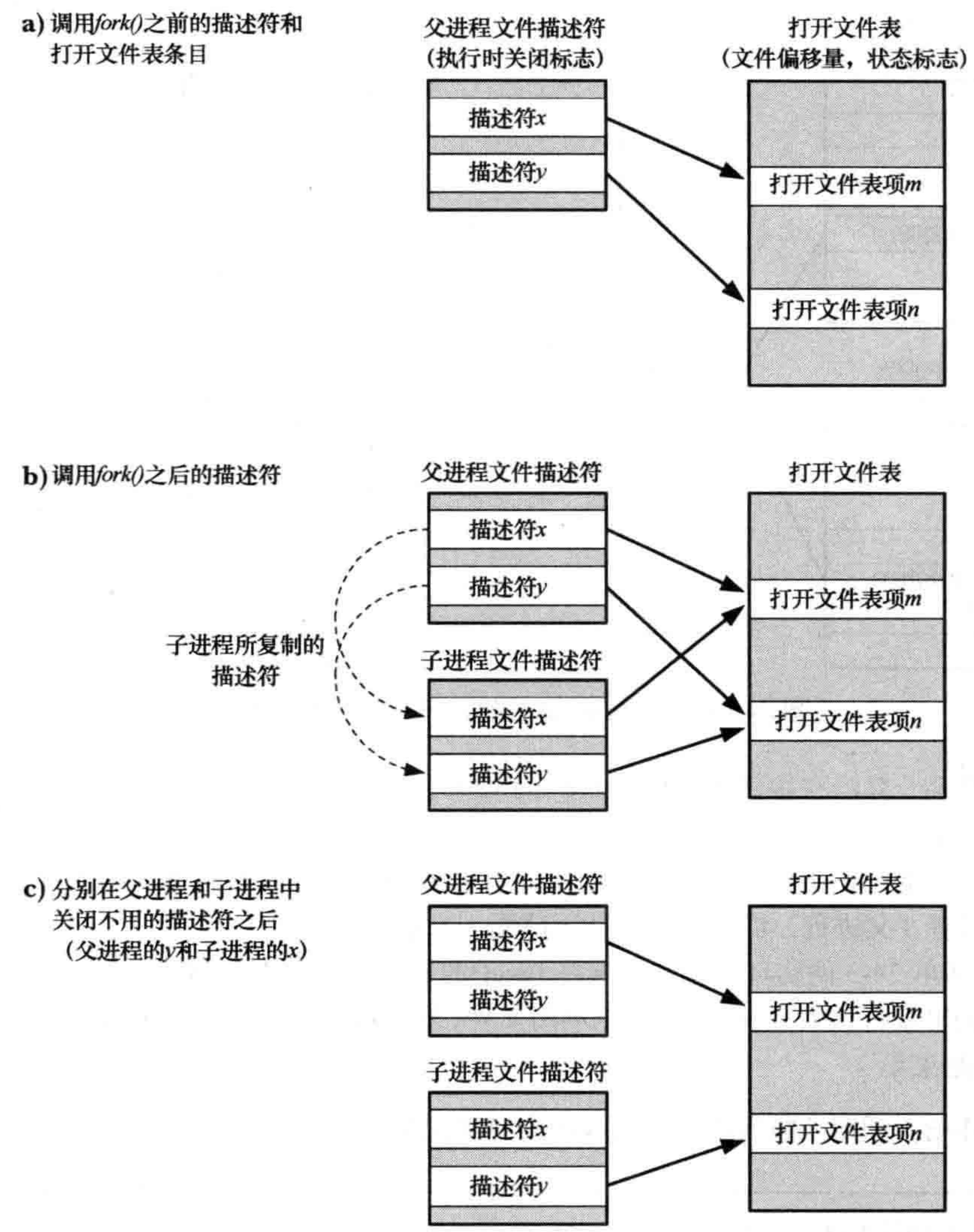

图 24-2：执行 fork()期间对文件描述符的复制，以及关闭不再使用的描述符

控制进程的内存需求

通过将 fork()与 wait()组合使用，可以控制一个进程的内存需求。进程的内存需求量，亦即进程所使用的虚拟内存页范围，受到多种因素的影响，例如，调用函数，或从函数返回时栈的变化情况，对 exec()的调用，以及因调用 malloc()和 free()而对堆所做的修改——这点对这里的讨论有着特殊意义。

假设以程序清单 24-3 所示方式调用 fork()和 wait()，且将对某函数 func()的调用置于括号之中。由执行程序可知，由于所有可能的变化都发生于子进程，故而从对 func()的调用之前开始，父进程的内存使用量将保持不变。这一用法的实用性则归于如下理由。

- 若已知 func()导致内存泄露，或是引发堆内存的过度碎片化，该技术则可以避免这些问题。（要是无法访问 func()的源码，想要处理这些问题也就无从谈起。）

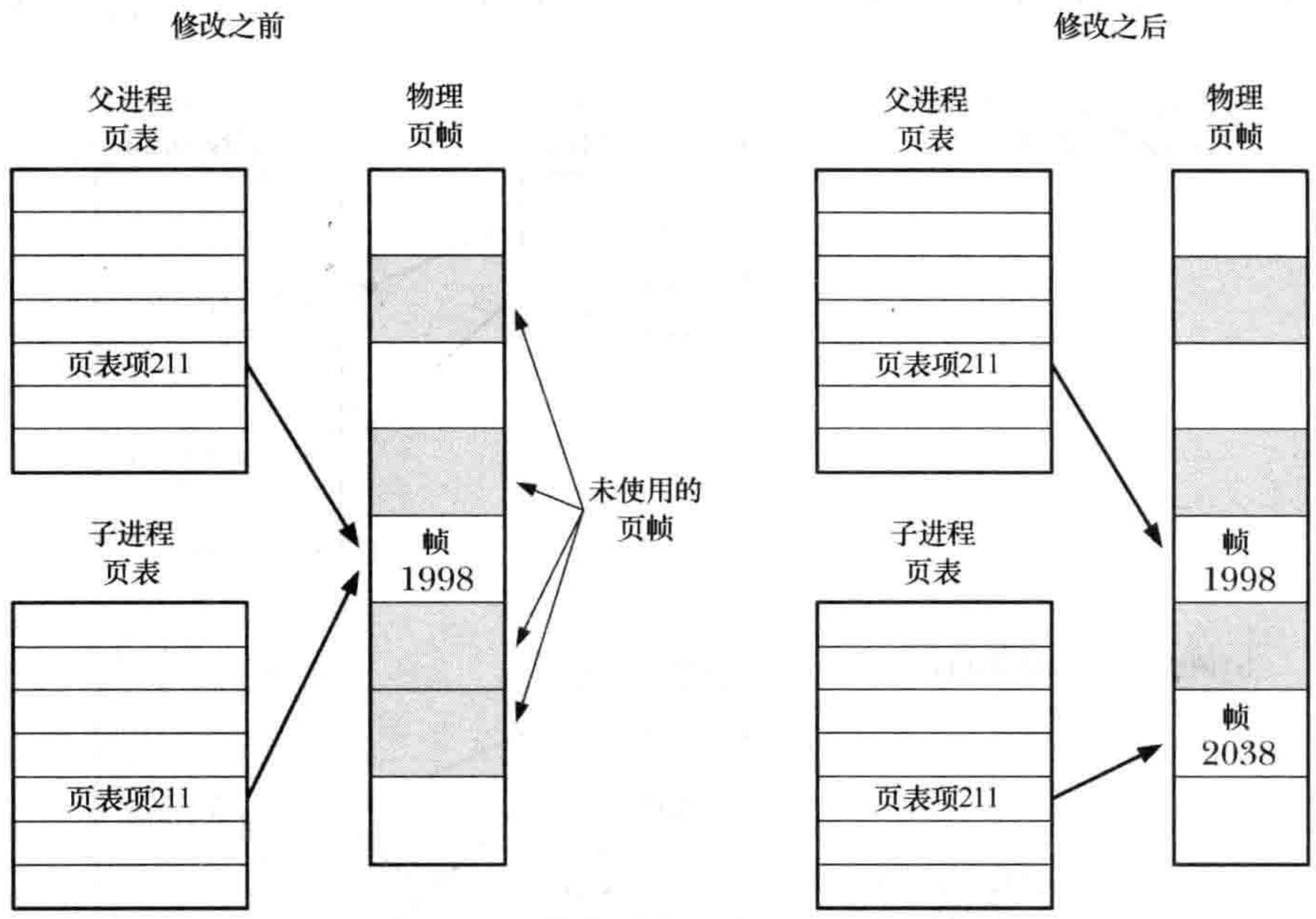

图 24-3：对一共享写时复制页进行修改前后的页表

- 假设某一算法在做树状分析（tree analysis）的同时需要进行内存分配（例如，游戏程序需要分析一系列可能的招法以及对方的应手）。本可以调用 free()来释放所有已分配的内存，不过在某些情况下，使用此处所描述的技术会更为简单，返回（父进程），且调用者（父进程）的内存需求并无改变。

如程序清单 24-3 的实现所示，必须将 func()的返回结果置于 exit()的 8 位传出值中，父进程调用 wait()可获得该值。不过，也可以利用文件、管道或其他一些进程间通信技术，使 func()返回更大的结果集[1]。

程序清单 24-3：调用函数而不改变进程的内存需求量

from **procexec/footprint.c**

```
pid_t childPid;
int status;

childPid = fork();
if (childPid == -1)
    errExit("fork");

if (childPid == 0)              /* Child calls func() and */
    exit(func(arg));            /* uses return value as exit status */

/* Parent waits for child to terminate. It can determine the
   result of func() by inspecting 'status'. */

if (wait(&status) == -1)
    errExit("wait");
```

from **procexec/footprint.c**

1 译者注：针对游戏程序的分析结果而言。

24.3 系统调用 vfork()

在早期的 BSD 实现中，fork()会对父进程的数据段、堆和栈施行严格的复制。如前所述，这是一种浪费，尤其是在调用 fork()后立即执行 exec()的情况下。出于这一原因，BSD 的后期版本引入了 vfork()系统调用，尽管其运作含义稍微有些不同（实则有些怪异），但效率要远高于 BSD fork()。现代 UNIX 采用写时复制技术来实现 fork()，其效率较之于早期的 fork()实现要高出许多，进而将对 vfork()的需求剔除殆尽。虽然如此，Linux（如同许多其他的 UNIX 实现一样）还是提供了具有 BSD 语义的 vfork()系统调用，以期为程序提供尽可能快的 fork 功能。不过，鉴于 vfork()的怪异语义可能会导致一些难以察觉的程序缺陷（bug），除非能给性能带来重大提升（这种情况发生的概率极小），否则应当尽量避免使用这一调用。

类似于 fork()，vfork()可以为调用进程创建一个新的子进程。然而，vfork()是为子进程立即执行 exec()的程序而专门设计的。

```
#include <unistd.h>

pid_t vfork(void);
```

In parent: returns process ID of child on success, or -1 on error;
in successfully created child: always returns 0

vfork()因为如下两个特性而更具效率，这也是其与 fork()的区别所在。

- 无需为子进程复制虚拟内存页或页表。相反，子进程共享父进程的内存，直至其成功执行了 exec()或是调用_exit()退出。
- 在子进程调用 exec()或_exit()之前，将暂停执行父进程。

这两点还另有深意：由于子进程使用父进程的内存，因此子进程对数据段、堆或栈的任何改变将在父进程恢复执行时为其所见。此外，如果子进程在 vfork()与后续的 exec()或_exit()之间执行了函数返回，这同样会影响到父进程。这与 6.8 节所描述的例子（试图以 longjmp()进入一个已经执行了返回的函数中）相类似。同样相似的还有这一乱局的收场——以典型的段错误（SIGSEGV）而告终。

在不影响父进程的前提下，子进程能在 vfork()与 exec()之间所做的操作屈指可数。其中包括对打开文件描述符进行操作（但不能施之于 stdio 文件流）。因为系统是在内核空间为每个进程维护文件描述符表（5.4 节），且在 vfork()调用期间将复制该表，所以子进程对文件描述符的操作不会影响到父进程。

SUSv3 指出，在如下情况下程序行为未定义：a）修改了除用于存储 vfork()返回值的 pid_t 型变量之外的任何数据；b）从调用 vfork()的函数中返回；c）在成功地调用_exit()或执行 exec()之前，调用了任何其他函数。

28.2 节在介绍系统调用 clone()时将会提及，由 fork()或 vfork()创建的子进程还具有少量其他进程属性的自有拷贝。

---

vfork()的语义在于执行该调用后，系统将保证子进程先于父进程获得调度以使用 CPU。24.2 节曾经提及 fork()是无法保证这一点的，父、子进程均有可能率先获得调度。

程序清单 24-4 展示了 vfork()的用法，将其区分于 fork()的两种语义特性显露无遗：子进程共享父进程的内存，父进程会一直挂起直至子进程终止或调用 exec()。运行该程序，其输出结果如下：

更多资源请访问稀酷客(www.ckook.com)

```
$ ./t_vfork
Child executing          Even though child slept, parent was not scheduled
Parent executing
istack=666
```

由输出的最后一行可知，子进程对变量 istack 的修改影响了父进程的对应变量。

程序清单 24-4：使用 vfork()

procexec/t_vfork.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int istack = 222;

    switch (vfork()) {
    case -1:
        errExit("vfork");

    case 0:             /* Child executes first, in parent's memory space */
        sleep(3);                   /* Even if we sleep for a while,
                                       parent still is not scheduled */
        write(STDOUT_FILENO, "Child executing\n", 16);
        istack *= 3;                /* This change will be seen by parent */
        _exit(EXIT_SUCCESS);

    default:            /* Parent is blocked until child exits */
        write(STDOUT_FILENO, "Parent executing\n", 17);
        printf("istack=%d\n", istack);
        exit(EXIT_SUCCESS);
    }
}
```

procexec/t_vfork.c

除非速度绝对重要的场合，新程序应当舍 vfork()而取 fork()。原因在于，当使用写时复制语义实现 fork()（大部分现代 UNIX 实现皆是如此）时，在速度几近于 vfork()的同时，又避免了 vfork()的上述怪异行止。（28.3 节会给出 fork()与 vfork()在速度方面的某些比较。）

SUSv3 将 vfork()标记为已过时，SUSv4 则进一步将其从规范中删除。对于 vfork()运作的诸多细节，SUSv3 颇有些语焉不详，因而可能将其实现为对 fork()的调用。如此一来，那么 vfork()的 BSD 语义将不复存在。一些 UNIX 系统还真就把 vfork()实现为对 fork()的调用，Linux 系统在内核 2.0 及其之前的版本中也是如此。

在使用时，一般应立即在 vfork()之后调用 exec()。如果 exec()执行失败，子进程应调用_exit()退出。（vfork()产生的子进程不应调用 exit()退出，因为这会导致对父进程 stdio 缓冲区的刷新和关闭。25.4 节将会详述这一点。）

vfork()的其他用法，尤其当其依赖于内存共享以及进程调度方面的独特语义时，将可能破坏程序的可移植性，其中尤以将 vfork()实现为简单调用 fork()的情况为甚。

24.4 fork()之后的竞争条件（Race Condition）

调用 fork()后，无法确定父、子进程间谁将率先访问 CPU。（在多处理器系统中，它们可能会同时各自访问一个 CPU。）就应用程序而言，如果为了产生正确的结果而或明或暗

更多资源请访问稀酷客(www.ckook.com)

（implicitly or explicitly）地依赖于特定的执行序列，那么将可能因竞争条件（5.1 节曾论及）而导致失败。由于此类问题的发生取决于内核根据系统当时的负载而做出的调度决定，故而往往难以发现。

可以用程序清单 24-5 中程序来验证这种不确定性。该程序循环使用 fork()来创建多个子进程。在每个 fork()调用后，父、子进程都会打印一条信息，其中包含循环计数器值以及标识父/子进程身份的字符串。例如，如果要求程序只产生一个子进程，其结果可能如下：

```
$ ./fork_whos_on_first 1
0 parent
0 child
```

可以使用该程序来生成大量子进程，并且分析其输出，观察父、子进程间每次到底由谁率先输出了结果。在某一 Linux/x86-32 2.2.19 系统上令此程序生成一百万个子进程，其分析结果表明，除去 332 次之外，都是由父进程先行输出结果（占总数的 99.97%）。

对程序清单 24-5 运行结果进行分析的脚本为 procexec/fork_whos_on_first.count.swk，在随本书发布的源代码中提供。

依据这一结果可以推测，在 Linux 2.2.19 中，fork()之后总是继续执行父进程。而子进程之所以在 0.03%的情况中首先输出结果，是因为父进程在有机会输出消息之前，其 CPU 时间片（CPU time slice）就到期了。换言之，如果该程序所代表的情况总是依赖于如下假设，即 fork()之后总是调度父进程，那么程序通常可以正常运行，不过每 3000 次将会出现一次错误。当然，如果希望父进程能在调度子进程前执行大量工作，那么出错的可能性将会大增。在一个复杂程序中调试这样的错误会很困难。

程序清单 24-5：fork()之后，父、子进程竞争输出信息

procexec/fork_whos_on_first.c

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

int
main(int argc, char *argv[])
{
    int numChildren, j;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-children]\n", argv[0]);

    numChildren = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-children") : 1;

    setbuf(stdout, NULL);                /* Make stdout unbuffered */

    for (j = 0; j < numChildren; j++) {
        switch (childPid = fork()) {
        case -1:
            errExit("fork");

        case 0:
            printf("%d child\n", j);
            _exit(EXIT_SUCCESS);

        default:
```


更多资源请访问稀酷客(www.ckook.com)

```
            printf("%d parent\n", j);
            wait(NULL);                    /* Wait for child to terminate */
            break;
        }
    }

    exit(EXIT_SUCCESS);
}
```

procexec/fork_whos_on_first.c

虽然 Linux 2.2.19 总是在 fork()之后继续运行父进程，但在其他 UNIX 实现上，甚至不同版本的 Linux 内核之间，却不能视其为理所当然。在内核稳定版 2.4 系列中，一度曾试验性地推出了一个“fork()之后由子进程先运行”的补丁，其调度结果与内核 2.2.19 完全相反。虽然这一改变之后又为 2.4 系列内核所舍弃，不过后来还是在 Linux 2.6 中采用，因此，程序假定于 2.2.19 内核的行为会在内核 2.6 中遭到推翻。

fork()之后对父、子进程的调度谁先谁后？其结果孰优孰劣？最近的一些实验又推翻了内核开发者关于这一问题的评估。从 Linux 2.6.32 开始，父进程再度成为 fork()之后，默认情况下率先调度的对象。将 Linux 专有文件/proc/sys/kernel/sched_child_runs_first 设为非 0 值可以改变该默认设置。

要了解支持“fork()之后先调度子进程”行为的理由，可考虑当 fork()产生的子进程立即执行 exec()时“写时复制”所发生的情况。此时，一方面父进程在 fork()之后继续修改数据页和栈页，另一方面内核要为子进程复制那些“将要修改”的页。由于子进程一旦获得调度会立即执行 exec()，故而这一页复制动作纯属浪费。基于这一论点，先调度子进程的决策更佳。如此一来，等到下次调度到父进程时，就无需复制内存页了。在一个繁忙的 Linux/X86-32 系统上（内核版本为 2.6.30），利用程序清单 24-5 中程序创建一百万个子进程，结果表明子进程率先输出的情况占总数的 99.98%。（这一百分比的精确值取决于诸如系统负载之类的因素。）在其他 UNIX 实现中测试该程序的结果则表明，对于由哪一进程在 fork()之后率先获得调度的问题，各系统的处理规则差异巨大。

Linux 2.6.32 改回“fork()之后先调度父进程”，其论据则基于如下发现：fork()之后，父进程在 CPU 中正处于活跃状态，并且其内存管理信息也被置于硬件内存管理单元的转译后备缓冲器（TLB, translation look-aside buffer）中。所以，先运行父进程将提高性能。在非正式场合下，针对分别采取上述两种行为的内核构建版本进行了时间度量，其结果也证实了这一点。

总之，值得强调的是：两种行为间的性能差异很小，对于大部分应用程序并无影响。

上述讨论清楚地阐明，不应对 fork()之后执行父、子进程的特定顺序做任何假设。若确需保证某一特定执行顺序，则必须采用某种同步技术。后续各章将会介绍多种同步技术，其中包括信号量（semaphore）、文件锁（file lock）以及进程间经由管道（pipe）的消息发送。接下来会描述另一种方法，那就是使用信号（signal）。

24.5 同步信号以规避竞争条件

调用 fork()之后，如果进程某甲需等待进程某乙完成某一动作，那么某乙（即活动进程）可在动作完成后向某甲发送信号；某甲则等待即可。

程序清单 24-6 演示了这一技术。该程序假设父进程必须等待子进程完成某些动作。如果

更多资源请访问稀酷客(www.ckook.com)

是子进程反过来要等待父进程，那么将父、子进程中与信号相关的调用对掉即可。父、子进程甚至可能多次互发信号以协调彼此行为，尽管实际上更有可能采用信号量、文件锁或消息传递等技术来进行此类协调。

> [Stevens & Rago, 2005] 建议将此类同步方法（阻塞信号，发送信号，捕获信号）封装为一组标准的进程同步函数。这一做法的优点在于，如果有意，后续可以其他进程间通信（IPC）机制替换信号的使用。

需要注意：程序清单 24-6 在 fork()之前就阻塞了同步信号（SIGUSR1）。若父进程试图在 fork()之后阻塞该信号，则避之唯恐不及的竞争条件恐怕将不期而遇。（此程序假设与子进程的信号掩码状态无关；如有必要，可以在 fork()之后的子进程中解除对 SIGUSR1 的阻塞。）

如下 shell 会话日志（log）则展示了程序清单 24-6 的运行情况：

```
$ ./fork_sig_sync
[17:59:02 5173] Child started - doing some work
[17:59:02 5172] Parent about to wait for signal
[17:59:04 5173] Child about to signal parent
[17:59:04 5172] Parent got signal
```

程序清单 24-6：利用信号来同步进程间动作

procexec/fork_sig_sync.c

```
#include <signal.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

#define SYNC_SIG SIGUSR1                /* Synchronization signal */

static void             /* Signal handler - does nothing but return */
handler(int sig)
{
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    sigset_t blockMask, origMask, emptyMask;
    struct sigaction sa;

    setbuf(stdout, NULL);               /* Disable buffering of stdout */

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SYNC_SIG);    /* Block signal */
    if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
        errExit("sigprocmask");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = handler;
    if (sigaction(SYNC_SIG, &sa, NULL) == -1)
        errExit("sigaction");

    switch (childPid = fork()) {
    case -1:
```

```
        errExit("fork");

    case 0: /* Child */

        /* Child does some required action here... */
        printf("[%s %ld] Child started - doing some work\n",
                currTime("%T"), (long) getpid());
        sleep(2);               /* Simulate time spent doing some work */

        /* And then signals parent that it's done */

        printf("[%s %ld] Child about to signal parent\n",
                currTime("%T"), (long) getpid());
        if (kill(getppid(), SYNC_SIG) == -1)
            errExit("kill");

        /* Now child can do other things... */

        _exit(EXIT_SUCCESS);

    default: /* Parent */

        /* Parent may do some work here, and then waits for child to
           complete the required action */

        printf("[%s %ld] Parent about to wait for signal\n",
                currTime("%T"), (long) getpid());
        sigemptyset(&emptyMask);
        if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        printf("[%s %ld] Parent got signal\n", currTime("%T"), (long) getpid());

        /* If required, return signal mask to its original state */

        if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
            errExit("sigprocmask");

        /* Parent carries on to do other things... */

        exit(EXIT_SUCCESS);
    }
}
```

procexec/fork_sig_sync.c

24.6 总结

系统调用 fork()通过复制一个与调用进程（父进程）几乎完全一致的拷贝来创建一个新进程（子进程）。系统调用 vfork()是一种更为高效的 fork()版本，不过因为其语义独特——vfork()产生的子进程将使用父进程内存，直至其调用 exec()或退出；于此同时，将会挂起（suspended）父进程，所以应尽量避免使用。

调用 fork()之后，不应对父、子进程获得调度以使用 CPU 的先后顺序有所依赖。对执行顺序做出假设的程序易于产生所谓“竞争条件”的错误。由于此类错误的产生依赖于诸如系统负载之类的外部因素，故而其发现和调试将非常困难。

更多资源请访问稀酷客(www.ckook.com)

更多信息

[Bach, 1986] 和 [Goodheart & Cox, 1994] 论述了 UNIX 系统中 fork()、execve()、wait()以及 exit()的实现细节。[Bovert & Cesati, 2005]和[Love, 2010]则就进程的创建和终止提供了专属于 Linux 系统的实现细节。

24.7 练习

24-1. 程序在执行完如下一系列 fork()调用后会产生多少新进程（假定没有调用失败）？

```
fork();
fork();
fork();
```

24-2. 编写一个程序以便验证调用 vfork()之后，子进程可以关闭一文件描述符（例如：描述符 0）而不影响对应父进程中的文件描述符。

24-3. 假设可以修改程序源代码，如何在某一特定时刻生成一核心转储（core dump）文件，而同时进程得以继续执行？

24-4. 在其他 UNIX 实现上实验程序清单 24-5（fork_whos_on_first.c）中的程序，并判断在执行 fork()后这些系统是如何调度父子进程的。

24-5. 假定在程序清单 24-6 的程序中，子进程也需要等待父进程完成某些操作。为确保达成这一目的，应如何修改程序？

更多资源请访问稀酷客(www.ckook.com)

第25章

进程的终止

本章所述为进程的退出过程。首先说明如何调用 exit()和_exit()以终止一个进程。接着讨论运用退出处理程序（exit handler），在进程调用 exit()时自动执行清理动作。最后，将探讨 fork()、stdio 缓冲区以及 exit()之间的某些交互。

25.1 进程的终止：_exit()和 exit()

通常，进程有两种终止方式。其一为异常（abnormal）终止，如 20.1 节所述，由对一信号的接收而引发，该信号的默认动作为终止当前进程，可能产生核心转储（core dump）。此外，进程可使用_exit()系统调用正常（normally）终止。

```
#include <unistd.h>

void _exit(int status);
```

_exit()的 status 参数定义了进程的终止状态（termination status），父进程可调用 wait()以获取该状态。虽然将其定义为 int 类型，但仅有低 8 位可为父进程所用。按照惯例，终止状态为 0 表示进程“功成身退”，而非 0 值则表示进程因异常而退出。对非 0 返回值的解释则并无定例；不同的应用程序自成一派，并会在文档中加以描述。SUSv3 规定有两个常量：EXIT_SUCCESS(0)和 EXIT_FAILURE(1)，本书中大部分程序就采用了这一约定。

调用_exit()的程序总会成功终止（即，_exit()从不返回）。

> 虽然可将 0～255 之间的任意值赋给_exit()的 status 参数，并传递给父进程，不过如取值大于 128 将在 shell 脚本中引发混乱。原因在于，当以信号（signal）终止一命令时，shell 会将变量$?置为 128 与信号值之和，以表征这一事实。如果这与进程调用_exit()时所使用的相同 status 值混杂起来，将令 shell 无法区分。

程序一般不会直接调用_exit()，而是调用库函数 exit()，它会在调用_exit()前执行各种动作。

```
#include <stdlib.h>
```


更多资源请访问稀酷客(www.ckook.com)

```
void _exit(int status);
```

exit()会执行的动作如下。

- 调用退出处理程序（通过 atexit()和 on_exit()注册的函数），其执行顺序与注册顺序相反（见 25.3 节）。
- 刷新 stdio 流缓冲区。
- 使用由 status 提供的值执行_exit()系统调用。

与专属于 UNIX 的_exit()不同，exit()则属于标准 C 语言函数库，也就是说，所有的 C 语言实现都支持 exit()。

程序的另一种终止方法是从 main()函数中返回（return），或者或明或暗地一直执行到 main()函数的结尾处[1]。执行 return n 等同于执行对 exit(n)的调用，因为调用 main()的运行时函数会将 main()的返回值作为 exit()的参数。

存在一种情况，从 main()函数中返回与调用 exit()并不相同。如果在退出的处理过程中所执行的任何步骤需要访问 main()函数的本地变量，那么从 main()函数中返回会导致未定义的行为。例如，在调用 setvbuf()或 setbuff()（见 13.2 节）时引用了 main 函数的本地变量，就会发生这种情况。

执行未指定返回值的 return，或是无声无息地执行到 main()函数结尾，同样会导致 main()的调用者执行 exit()函数，不过，视所支持的不同 C 语言标准版本，以及所使用的不同编译器选项，其结果也有所不同。

- C89 标准未就上述情况下的行为进行定义，程序可以返回任意的 status 值。Linux gcc 的默认行为就是如此，程序的退出状态是取自于栈或特定 CPU 寄存器中的随机值。应避免以这一方式终止程序。
- C99 标准则要求，执行至 main 函数结尾处的情况应等同于调用 exit(0)。如果使用 gcc–std=c99 在 Linux 中编译程序，将会获得这种效果。

25.2 进程终止的细节

无论进程是否正常终止，都会发生如下动作。

- 关闭所有打开文件描述符、目录流（18.8 节）、信息目录描述符（参考手册页 catopen(3)和 catgets(3)），以及（字符集）转换描述符（见 iconv_open(3)手册页）。
- 作为文件描述符关闭的后果之一，将释放该进程所持有的任何文件锁（第 55 章）。
- 分离（detach）任何已连接的 System V 共享内存段，且对应于各段的 shm_nattch 计数器值将减一。（参考 48.8 节。）
- 进程为每个 System V 信号量所设置的 semadj 值将会被加到信号量值中（参考 47.8 节）。
- 如果该进程是一个管理终端（terminal）的管理进程，那么系统会向该终端前台（foreground）进程组中的每个进程发送 SIGHUP 信号，接着终端会与会话（session）脱离。34.6 节将就此进行深入讨论。
- 将关闭该进程打开的任何 POSIX 有名信号量，类似于调用 sem_close()。

1 译者注：即 main()函数尾部无 return 语句。

更多资源请访问稀酷客(www.ckook.com)

- 将关闭该进程打开的任何 POSIX 消息队列，类似于调用 mq_close()。
- 作为进程退出的后果之一，如果某进程组成为孤儿，且该组中存在任何已停止进程（stopped processes），则组中所有进程都将收到 SIGHUP 信号，随之为 SIGCONT 信号。34.7.4 节将深入讨论这一点。
- 移除该进程通过 mlock()或 mlockall()（50.2 节）所建立的任何内存锁。
- 取消该进程调用 mmap()所创建的任何内存映射（mapping）。

25.3 退出处理程序

有时，应用程序需要在进程终止时自动执行一些操作。试以一个应用程序库为例，如果进程使用了该程序库，那么在进程终止前该库需要自动执行一些清理动作。因为库本身对于进程何时以及如何退出并无控制权，也无法要求主程序在退出前调用库中特定的清理函数，故而也不能保证一定会执行清理动作。解决这一问题的方法之一是使用退出处理程序（exit handler）。老版 System V 手册则使用术语“程序终止过程”（program termination routine）。

退出处理程序是一个由程序设计者提供的函数，可于进程生命周期的任意时点注册，并在该进程调用 exit()正常终止时自动执行。如果程序直接调用_exit()或因信号而异常终止，则不会调用退出处理程序。

当进程收到信号而终止时，将不会调用退出处理程序。这一事实一定程度上限制了它们的效用。此时最佳的应对方式莫若为可能发送给进程的信号建立信号处理程序，并于其中设置标志位，令主程序据此来调用 exit()。因为 exit()不属于表 21-1 所列的异步信号安全（async-signal-safe）函数，所以通常不能在信号处理程序中对其发起调用。即便如此，还是无法处理 SIGKILL 信号，因为无法改变 SIGKILL 的默认行为。这也是应该避免使用 SIGKILL 来终止进程的另一原因（如 20.2 节所述）。建议使用信号 SIGTERM，这也是 kill 命令默认发送的信号。

注册退出处理程序

GNU C 语言函数库提供两种方式来注册退出处理程序。第一种方法是使用由 SUSv3 定义的 atexit()函数。

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns 0 on success, or nonzero on error

函数 atexit()将 func 加到一个函数列表中，进程终止时会调用该函数列表的所有函数。应将函数 func 定义为不接受任何参数，也无返回值，一般格式如下：

```
void
func(void)
{
    /* Perform some actions */
}
```

注意 atexit()在出错时返回非 0 值（不一定是-1）。

可以注册多个退出处理程序（甚至可以将同一函数注册多次）。当应用程序调用 exit()时，

更多资源请访问稀酷客(www.ckook.com)

这些函数的执行顺序与注册顺序相反。这一设计很符合逻辑，因为，一般情况下较早注册的函数所执行的是更为基本的清理动作，可能需要在调用后续注册的函数后再执行。

本质上，可以在退出处理程序中执行任何希望的动作，包括注册附加的退出处理程序，并将其置于留待调用的剩余函数列表的头部。不过，一旦有任一退出处理程序无法返回——无论因为调用了_exit()还是进程因收到信号而终止（例如，退出处理程序调用函数 raise()），那么就不会再调用剩余的处理程序。此外，调用 exit()时通常需要执行的剩余动作也将不再执行。

SUSv3 规定，若退出处理程序自身调用 exit()，其结果未定义。在 Linux 上，会照常调用剩余的退出处理程序。不过，在某些系统上，这将导致对所有退出处理程序的再次调用，并引发无限循环调用（直至栈溢出将该进程杀死）。为保障可移植性，应用程序应避免在退出处理程序内部调用 exit()。

SUSv3 要求系统实现应允许一个进程能够注册至少 32 个退出处理程序。使用系统调用 sysconf(_SC_ATEXIT_MAX)，应用程序即可确定由实现所定义的可注册退出处理程序的数量上限。（但是，并无方法获知有多少已注册的处理程序。）通过运用动态分配链表将已注册的处理程序串接起来，glibc 允许注册的退出处理程序数量近乎于无限。对于 Linux，sysonf(_SC_ATEXIT_MAX)返回 2147482647（即，32 位有符号整型数的最大值）。换言之，在触及可注册函数数量的这一上限前，总会有其他原因（例如，内存不足）导致程序崩溃。

通过 fork()创建的子进程会继承父进程注册的退出处理函数。而进程调用 exec()时，会移除所有已注册的退出处理程序。（这是结果势所必然，因为 exec()会替换包括退出处理程序在内的所有原程序代码段。）

无法取消经由 atexit()或 on_exit()（见稍后的描述）注册的退出处理程序。不过，可以令退出处理程序在执行动作之前检查全局执行标志是否置位，或者清除该标志来屏蔽退出处理程序。

经由 atexit()注册的退出处理程序会受到两种限制。其一，退出处理程序在执行时无法获知传递给 exit()的状态。有时候，知道状态是必要的；例如，退出处理程序会视进程退出成功与否而执行不同的动作。其二，无法给退出处理程序指定参数。如果拥有这一特性，退出处理程序能根据传入参数的不同而执行不同动作，或使用不同参数多次注册同一个函数。

为摆脱这些限制，glibc 提供了一个（非标准的）替代方法：on_exit()。

```
#define _BSD_SOURCE             /* Or: #define _SVID_SOURCE */
#include <stdlib.h>

int on_exit(void (*func)(int, void *), void *arg);
```

Returns 0 on success, or nonzero on error

函数 on_exit()的参数 func 是一个指针，指向如下类型的函数:

```
void
func(int status, void *arg)
{
    /* Perform cleanup actions */
}
```

调用时，会传递两个参数给 func()：提供给 exit()的 status 参数和注册时供给 on_exit()的一份 arg 参数拷贝。虽然定义为指针类型，参数 arg 的意义仍然可由设计者支配。可将其用作指向结构的指针，同样，通过审慎地强制转换，也可将其作为整型或其他标量类型使用。

更多资源请访问稀酷客(www.ckook.com)

类似于 atexit()，on_exit()出错时返回非 0 值（不一定是-1）。

如同 atexit()一样，通过 on_exit()可以注册多个退出处理程序。使用 atexit()和 on_exit()注册的函数位于同一函数列表。如果在程序中同时用到了这两种方式，则会按照使用这两个方法注册的相反顺序来执行相应的退出处理程序。

虽然比 atexit()更灵活，但对于要保障可移植性的程序来说，还是应避免使用 on_exit()。因为并无标准涵盖到它，并且几乎也没有其他 UNIX 实现支持这一用法。

程序范例

程序清单 25-1 展示了如何利用 atexit()和 on_exit()注册退出处理程序的例子。运行程序会得到如下输出结果：

```
$ ./exit_handlers
on_exit function called: status=2, arg=20
atexit function 2 called
atexit function 1 called
on_exit function called: status=2, arg=10
```

程序清单 25-1：使用退出处理程序

procexec/exit_handlers.c

```
#define _BSD_SOURCE     /* Get on_exit() declaration from <stdlib.h> */
#include <stdlib.h>
#include "tlpi_hdr.h"

static void
atexitFunc1(void)
{
    printf("atexit function 1 called\n");
}

static void
atexitFunc2(void)
{
    printf("atexit function 2 called\n");
}

static void
onexitFunc(int exitStatus, void *arg)
{
    printf("on_exit function called: status=%d, arg=%ld\n",
                exitStatus, (long) arg);
}

int
main(int argc, char *argv[])
{
    if (on_exit(onexitFunc, (void *) 10) != 0)
        fatal("on_exit 1");
    if (atexit(atexitFunc1) != 0)
        fatal("atexit 1");
    if (atexit(atexitFunc2) != 0)
        fatal("atexit 2");
    if (on_exit(onexitFunc, (void *) 20) != 0)
        fatal("on_exit 2");
```


更多资源请访问稀酷客(www.ckook.com)

```
    exit(2);
}
```

procexec/exit_handlers.c

25.4 fork()、stdio 缓冲区以及_exit()之间的交互

程序清单 25-2 生成的输出结果乍看颇令人费解。当程序标准输出定向到终端时，会看到预期的结果：

```
$ ./fork_stdio_buf
Hello world
Ciao
```

不过，当重定向标准输出到一个文件时，结果如下：

```
$ ./fork_stdio_buf > a
$ cat a
Ciao
Hello world
Hello world
```

以上输出中有两件怪事：printf()的输出行出现了两次，且 write()的输出先于 printf()。

程序清单 25-2：fork()与 stdio 缓冲区的交互

procexec/fork_stdio_buf.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    /* Both child and parent continue execution here */

    exit(EXIT_SUCCESS);
}
```

procexec/fork_stdio_buf.c

要理解为什么 printf()的输出消息出现了两次，首先要记住，是在进程的用户空间内存中（参考 13.2 节）维护 stdio 缓冲区的。因此，通过 fork()创建子进程时会复制这些缓冲区。当标准输出定向到终端时，因为缺省为行缓冲，所以会立即显示函数 printf()输出的包含换行符的字符串。不过，当标准输出重定向到文件时，由于缺省为块缓冲，所以在本例中，当调用 fork()时，printf()输出的字符串仍在父进程的 stdio 缓冲区中，并随子进程的创建而产生一份副本。父、子进程调用 exit()时会刷新各自的 stdio 缓冲区，从而导致重复的输出结果。

可以采用以下任一方法来避免重复的输出结果。

- 作为针对 stdio 缓冲区问题的特定解决方案，可以在调用 fork()之前使用函数 fflush()来刷新 stdio 缓冲区。作为另一种选择，也可以使用 setvbuf()和 setbuf()来关闭 stdio 流的缓冲功能。

更多资源请访问稀酷客(www.ckook.com)

- 子进程可以调用_exit()而非 exit()，以便不再刷新 stdio 缓冲区。这一技术例证了一个更为通用的原则：在创建子进程的应用中，典型情况下仅有一个进程（一般为父进程）应通过调用 exit()终止，而其他进程应调用_exit()终止，从而确保只有一个进程调用退出处理程序并刷新 stdio 缓冲区，这也算是众望所归吧。

还存在其他方法，可以（有时很有必要）允许父子进程都调用 exit()。例如，可以设计这样的退出处理程序，即使是从多个进程中调用，它们也能够正确地处理，或者令应用程序仅在调用 fork()之后才去安装退出处理程序。此外，有时可能确实希望所有的应用程序都在 fork()之后刷新 stdio 缓冲区。这时，可以见机行事，要么选择使用 exit()来终止进程，要么在每个进程中均显式调用 fflush()。

程序清单 25-2 中 write()的输出并未出现两次，这是因为 write()会将数据直接传给内核缓冲区，fork()不会复制这一缓冲区。

程序输出重定向到文件时出的第二件怪事，原因现在也清楚了。write()的输出结果先于 printf()而出现，是因为 write()会将数据立即传给内核高速缓存，而 printf()的输出则需要等到调用 exit ()刷新 stdio 缓冲区时。（如 13.7 节所述，通常，在混合使用 stdio 函数和系统调用对同一文件进行 I/O 处理时，需要特别谨慎。）

25.5 总结

进程的终止分为正常和异常两种。异常终止可能是由于某些信号引起，其中的一些信号还可能导致进程产生一个核心转储文件。

正常的终止可以通过调用_exit()完成，更多的情况下，则是使用_exit()的上层函数 exit()完成。_exit()和 exit()都需要一个整型参数，其低 8 位定义了进程的终止状态。依照惯例，状态 0 用来表示进程成功完成，非 0 则表示异常退出。

不管进程正常终止与否，内核都会执行多个清理步骤。调用 exit()正常终止一个进程，将会引发执行经由 atexit()和 on_exit()注册的退出处理程序（执行顺序与注册顺序相反），同时刷新 stdio 缓冲区。

更多信息

请参考 24.6 节所列的深入信息来源。

25.6 练习

如果子进程调用 exit(-1)，父进程将会看到何种退出状态（由 WEXITSTATUS()返回）？

更多资源请访问稀酷客(www.ckook.com)

第 26 章

监控子进程

在很多应用程序的设计中，父进程需要知道其某个子进程于何时改变了状态——子进程终止或因收到信号而停止。本章描述两种用于监控子进程的技术：系统调用 wait()（及其变体）以及信号 SIGCHLD。

26.1 等待子进程

对于许多需要创建子进程的应用来说，父进程能够监测子进程的终止时间和过程是很有必要的。wait()以及若干相关的系统调用提供了这一功能。

26.1.1 系统调用 wait()

系统调用 wait()等待调用进程的任一子进程终止，同时在参数 status 所指向的缓冲区中返回该子进程的终止状态。

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

pid_t wait(int *status);
```

Returns process ID of terminated child, or -1 on error

系统调用 wait()执行如下动作。

1. 如果调用进程并无之前未被等待的子进程终止[1]，调用将一直阻塞，直至某个子进程终止。如果调用时已有子进程终止，wait()则立即返回。
2. 如果 status 非空，那么关于子进程如何终止的信息则会通过 status 指向的整型变量返回。26.1.3 节将讨论自 status 返回的信息。
3. 内核将会为父进程下所有子进程的运行总量追加进程 CPU 时间（10.7 节）以及资源使用数据。

1 译者注：原文为“if on（previously unwaited-for）child of the calling process has yet terminated”。

更多资源请访问稀酷客(www.ckook.com)

4. 将终止子进程的 ID 作为 wait()的结果返回。

出错时，wait()返回-1。可能的错误原因之一是调用进程并无之前未被等待的[1]子进程，此时会将 errno 置为 ECHILD。换言之，可使用如下代码中的循环来等待调用进程的所有子进程退出。

```
while ((childPid = wait(NULL)) != -1)
    continue;
if (errno != ECHILD)                /* An unexpected error... */
    errExit("wait");
```

程序清单 26-1 演示了 wait()的用法。该程序创建多个子进程，每个子进程对应于一个（整型）命令行参数。每个子进程休眠若干秒后退出，休眠时间分别由相应各命令行参数指定。与此同时，在创建所有的子进程之后，父进程循环调用 wait()来监控这些子进程的终止。而直到 wait()返回-1 时才会退出循环。（这并非唯一的手段，另一种退出循环的方法是当记录终止子进程数量的变量 numDead 与创建的子进程数目相同时，也会退出循环。）以下 shell 会话日志显示了使用该程序创建 3 个子进程时的情况。

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

如果于同一时点存在多个子进程退出，SUSv3 并未对 wait()处理这些子进程的顺序加以规定，换言之，该顺序取决于具体实现。即使不同的 Linux 内核版本之间，行为也有所不同。

程序清单 26-1：创建并等待多个子进程

procexec/multi_wait.c

```
#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"              /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;        /* Number of children so far waited for */
    pid_t childPid;     /* PID of waited for child */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);           /* Disable buffering of stdout */

    for (j = 1; j < argc; j++) {    /* Create one child for each argument */
        switch (fork()) {
```

1 译者注：此处原文为 previously unwaited-for。

```
        case -1:
            errExit("fork");

        case 0:                         /* Child sleeps for a while then exits */
            printf("[%s] child %d started with PID %ld, sleeping %s "
                    "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:                        /* Parent just continues around loop */
            break;
        }
    }

    numDead = 0;
    for (;;) {                          /* Parent waits for each child to exit */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {                    /* Some other (unexpected) error */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
                currTime("%T"), (long) childPid, numDead);
    }
}
```

procexec/multi_wait.c

26.1.2 系统调用 waitpid()

系统调用 wait()存在诸多限制，而设计 waitpid()则意在突破这些限制。

- 如果父进程已经创建了多个子进程，使用 wait()将无法等待某个特定子进程的完成，只能按顺序等待下一个子进程的终止。
- 如果没有子进程退出，wait()总是保持阻塞。有时候会希望执行非阻塞的等待：是否有子进程退出，立判可知。
- 使用 wait()只能发现那些已经终止的子进程。对于子进程因某个信号（如 SIGSTOP 或 SIG TTIN）而停止，或是已停止子进程收到 SIGCONT 信号后恢复执行的情况就无能为力了。

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

pid_t waitpid(pid_t pid, int *status, int options);
```

Returns process ID of child, 0 (see text), or −1 on error

waitpid()与 wait()的返回值以及参数 status 的意义相同。（对 status 中返回值的解释请参考 26.1.3 节。）参数 pid 用来表示需要等待的具体子进程，意义如下：

- 如果 pid 大于 0，表示等待进程 ID 为 pid 的子进程。

更多资源请访问稀酷客(www.ckook.com)

- 如果 pid 等于 0，则等待与调用进程（父进程）同一个进程组（process group）的所有子进程。34.2 节将描述进程组的概念。
- 如果 pid 小于-1，则会等待进程组标识符与 pid 绝对值相等的所有子进程。
- 如果 pid 等于-1，则等待任意子进程。wait(&status)的调用与 waitpid(-1, &status, 0)等价。

参数 options 是一个位掩码（bit mask），可以包含（按位或操作）0 个或多个如下标志（均在 SUSv3 中加以规范）。

WUNTRACED

除了返回终止子进程的信息外，还返回因信号而停止的子进程信息。

WCONTINUED (自 Linux2.6.10 以来)

返回那些因收到 SIGCONT 信号而恢复执行的已停止子进程的状态信息。

WNOHANG

如果参数 pid 所指定的子进程并未发生状态改变，则立即返回，而不会阻塞，亦即 poll（轮询）。在这种情况下，waitpid()返回 0。如果调用进程并无与 pid 匹配的子进程，则 waitpid()报错，将错误号置为 ECHILD。

程序清单 26-3 演示了 waitpid()的使用。

SUSv3 在其对 waitpid()的原理阐述中特别指出，WUNTRACED 的名称是源于 BSD 的历史产物。BSD 有两种停止进程的方法：作为系统调用 ptrace()追踪的结果，或者因收到一个信号而停止。当通过 ptrace()追踪一个子进程时，那么（除 SIGKILL 之外的）任何信号都会造成子进程停止，接着会将信号 SIGCHLD 发给父进程。即使子进程忽略这些信号，这一行为仍会发生。不过，如果子进程阻塞了这些信号（除非是无法阻塞的 SIGSTOP 信号），子进程就不会停止。

26.1.3 等待状态值

由 wait()和 waitpid()返回的 status 的值，可用来区分以下子进程事件。

- 子进程调用_exit()（或 exit()）而终止，并指定一个整型值作为退出状态。
- 子进程收到未处理信号而终止。
- 子进程因为信号而停止，并以 WUNTRACED 标志调用 waitpid()。
- 子进程因收到信号 SIGCONT 而恢复，并以 WCONTINUED 标志调用 waitpid()。

此处用术语“等待状态”（wait status）来涵盖上述所有情况，而使用“终止状态”（termination status）的称谓来指代前两种情况。（在 shell 中，可通过读取$?变量值来获取上次执行命令的终止状态。）

虽然将变量 status 定义为整型（int），但实际上仅使用了其最低的 2 个字节。对这 2 个字节的填充方式取决于子进程所发生的具体事件，如图 26-1 所示。

图 26-1 所示为 Linux/x86-32 下等待状态值的格式。不同的实现版本细节会有所不同。SUSv3 并未对信息格式做出具体规定，也未规定只能使用 status 变量的最低 2 个字节。要保证应用程序的可移植性，应总是使用本节介绍的宏（macro）来获取相应的值，而不应直接按位读取其内容。

更多资源请访问稀酷客(www.ckook.com)

图 26-1：自 wait()和 waitpid()的 status 参数所返回的值

头文件<sys/wait.h>定义了用于解析等待状态值的一组标准宏。对自 wait()或 waitpid()返回的 status 值进行处理时，以下列表中各宏只有一个会返回真（true）值。如列表所示，另有其他宏可对 status 值做进一步分析。

WIFEXITED (status)

若子进程正常结束则返回真（true）。此时，宏 WEXITSTATUS(status)返回子进程的退出状态。（如 25.1 节所述，父进程仅关注子进程退出状态的最低 8 位。）

WIFSIGNALED (status)

若通过信号杀掉子进程则返回真（true）。此时，宏 WTERMSIG(status)返回导致子进程终止的信号编号。若子进程产生内核转储文件，则宏 WCOREDUMP(status)返回真值（true）。SUSv3 并未规范宏 WCOREDUMP()，不过大部分 UNIX 实现均支持该宏。

WIFSTOPPED (status)

若子进程因信号而停止，则此宏返回为真值（true）。此时，宏 WSTOPSIG(status)返回导致子进程停止的信号编号。

WIFCONTINUED (status)

若子进程收到 SIGCONT 而恢复执行，则此宏返回真值（true)。自 Linux 2.6.10 之后开始支持该宏。

注意：尽管上述宏的参数也以 status 命名，不过此处所指只是简单的整型变量，而非像 wait()和 waitpid()所要求的那样是指向整型的指针。

示例程序

程序清单 26-2 中的函数 printWaitStatus()使用了上述所有宏。此函数分析并输出了等待状态值的内容。

程序清单 26-2：输出 wait()及相关调用返回的状态值

procexec/print_wait_status.c

```
#define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <sys/wait.h>
```

```
#include "print_wait_status.h"  /* Declaration of printWaitStatus() */
#include "tlpi_hdr.h"

/* NOTE: The following function employs printf(), which is not
   async-signal-safe (see Section 21.1.2). As such, this function is
   also not async-signal-safe (i.e., beware of calling it from a
   SIGCHLD handler). */

void                        /* Examine a wait() status using the W* macros */
printWaitStatus(const char *msg, int status)
{
    if (msg != NULL)
        printf("%s", msg);

    if (WIFEXITED(status)) {
        printf("child exited, status=%d\n", WEXITSTATUS(status));

    } else if (WIFSIGNALED(status)) {
        printf("child killed by signal %d (%s)",
                WTERMSIG(status), strsignal(WTERMSIG(status)));
#ifdef WCOREDUMP        /* Not in SUSv3, may be absent on some systems */
        if (WCOREDUMP(status))
            printf(" (core dumped)");
#endif
        printf("\n");

    } else if (WIFSTOPPED(status)) {
        printf("child stopped by signal %d (%s)\n",
                WSTOPSIG(status), strsignal(WSTOPSIG(status)));

#ifdef WIFCONTINUED     /* SUSv3 has this, but older Linux versions and
                           some other UNIX implementations don't */
    } else if (WIFCONTINUED(status)) {
        printf("child continued\n");
#endif

    } else {            /* Should never happen */
        printf("what happened to this child? (status=%x)\n",
                (unsigned int) status);
    }
}
```

procexec/print_wait_status.c

程序清单 26-3 使用了 printWaitStatus()函数。该程序创建了一个子进程，该子进程会循环调用 pause()（在此期间可以向子进程发送信号），但如果在命令行中指定了整型参数，则子进程会立即退出，并以该整型值作为退出状态。同时，父进程通过 waitpid()监控子进程，打印子进程返回的状态值并将其作为参数传递给 printWaitStatus()。一旦发现子进程已正常退出，亦或因某一信号而终止，父进程会随即退出。

如下 shell 会话展示了执行程序清单 26-3 程序的几个例子。首先，创建一子进程并立即退出，且其状态值为 23：

```
$ ./child_status 23
Child started with PID = 15807
waitpid() returned: PID=15807; status=0x1700 (23,0)
child exited, status=23
```

接下来，在后台运行该程序，并向子进程发送 SIGSTOP 和 SIGCONT 信号。

```
$ ./child_status &
[1] 15870
$ Child started with PID = 15871
kill -STOP 15871
$ waitpid() returned: PID=15871; status=0x137f (19,127)
child stopped by signal 19 (Stopped (signal))
kill -CONT 15871
$ waitpid() returned: PID=15871; status=0xffff (255,255)
child continued
```

输出的最后两行只会在 Linux 2.6.10 及其之后的内核版本中出现，因为早期内核并不支持 waitpid()的 WCONTINUED 选项。（由于后台运行程序的输出有时会与 shell 提示符混在一起，故而该 shell 会话稍微有些难以阅读。）

接着，再发送 SIGABRT 信号来终止子进程：

```
kill -ABRT 15871
$ waitpid() returned: PID=15871; status=0x0006 (0,6)
child killed by signal 6 (Aborted)
Press Enter, in order to see shell notification that background job has terminated
[1]+  Done                    ./child_status
$ ls -l core
ls: core: No such file or directory
$ ulimit -c                                        Display RLIMIT_CORE limit
0
```

虽然 SIGABRT 的默认行为是产生一个内核转储文件并终止进程，但这里并未产生转储文件。这是由于屏蔽内核转储所致，如以上命令 ulimit 的输出所示，将 RLIMIT_CORE 资源软限制（见 36.3 节）置为 0，该限制规定了转储文件大小的最大值。

再次重复同一实验，不过这次在发送信号 SIGABRT 给子进程之前，放开了对转储文件大小的限制。

```
$ ulimit -c unlimited                              Allow core dumps
$ ./child_status &
[1] 15902
$ Child started with PID = 15903
kill -ABRT 15903                                   Send SIGABRT to child
$ waitpid() returned: PID=15903; status=0x0086 (0,134)
child killed by signal 6 (Aborted) (core dumped)
Press Enter, in order to see shell notification that background job has terminated
[1]+  Done                    ./child_status
$ ls -l core                                       This time we get a core dump
-rw-------   1 mtk      users       65536 May  6 21:01 core
```

程序清单 26-3：使用 waitpid()获取子进程状态

procexec/child_status.c

```
#include <sys/wait.h>
#include "print_wait_status.h"          /* Declares printWaitStatus() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int status;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);

    switch (fork()) {
```

```
    case -1: errExit("fork");

    case 0:             /* Child: either exits immediately with given
                           status or loops waiting for signals */
        printf("Child started with PID = %ld\n", (long) getpid());
        if (argc > 1)                   /* Status supplied on command line? */
            exit(getInt(argv[1], 0, "exit-status"));
        else                            /* Otherwise, wait for signals */
            for (;;)
                pause();
        exit(EXIT_FAILURE);             /* Not reached, but good practice */

    default:            /* Parent: repeatedly wait on child until it
                           either exits or is terminated by a signal */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED
#ifdef WCONTINUED       /* Not present on older versions of Linux */
                                                | WCONTINUED
#endif
                    );
            if (childPid == -1)
                errExit("waitpid");

            /* Print status in hex, and as separate decimal bytes */

            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                    (long) childPid,
                    (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);

            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

procexec/child_status.c

26.1.4 从信号处理程序中终止进程

如表 20-1 所列，默认情况下某些信号会终止进程。有时，可能希望在进程终止之前执行一些清理步骤。为此，可以设置一个处理程序（handler）来捕获这些信号，随即执行清理步骤，再终止进程。如果这么做，需要牢记的是：通过 wait()和 waitpid()调用，父进程依然可以获取子进程的终止状态。例如，如果在信号处理程序中调用_exit(EXIT_SUCCESS)，父进程会认为子进程是正常终止。

如果需要通知父进程自己因某个信号而终止，那么子进程的信号处理程序应首先将自己废除，然后再次发出相同信号，该信号这次将终止这一子进程。信号处理程序需包含如下代码：

```
void
handler(int sig)
{
    /* Perform cleanup steps */

    signal(sig, SIG_DFL);           /* Disestablish handler */
```


更多资源请访问稀酷客(www.ckook.com)

```
    raise(sig);                         /* Raise signal again */
}
```

26.1.5 系统调用 waitid()

与 waitpid()类似，waitid()返回子进程的状态。不过，waitid()提供了 waitpid()所没有的扩展功能。该系统调用源于系统 V（System V），不过现在已获 SUSv3 采用，并从版本 2.6.9 开始，将其加入 Linux 内核。

在 Linux 2.6.9 之前，通过 glibc 实现提供了一版 waitid()。然而，由于完全实现该接口需要内核的支持，因此 glibc 版实现并未提供比 waitpid()更多的功能。

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns 0 on success or if WNOHANG was specified and there were no children to wait for, or −1 on error

参数 idtype 和 id 指定需要等待哪些子进程，如下所示。

- 如果 idtype 为 P_ALL，则等待任何子进程，同时忽略 id 值。
- 如果 idtype 为 P_PID，则等待进程 ID 为 id 进程的子进程。
- 如果 idtype 为 P_PGID，则等待进程组 ID 为 id 各进程的所有子进程。

请注意，与 waitpid()不同，不能靠指定 id 为 0 来表示与调用者属于同一进程组的所有进程。相反，必须以 getpgrp()的返回值来显式指定调用者的进程组 ID。

waitpid()与 waitid()最显著的区别在于，对于应该等待的子进程事件，waitid()可以更为精确地控制。可通过在 options 中指定一个或多个如下标识（按位或运算）来实现这种控制。

WEXITED

等待已终止的子进程，而无论其是否正常返回。

WSTOPPED

等待已通过信号而停止的子进程。

WCONTINUED

等待经由信号 SIGCONT 而恢复的子进程。

以下附加标识也可以通过按位或运算加入 options 中。

WNOHANG

与其在 waitpid()中的意义相同。如果匹配 id 值的子进程中并无状态信息需要返回，则立即返回（一个轮询）。此时，waitid()返回 0。如果调用进程并无子进程与 id 的值相匹配，则 waitid 调用失败，且错误号为 ECHILD。

WNOWAIT

通常，一旦通过 waitid()来等待子进程，那么必然会去处理所谓“状态事件”。不过，如

更多资源请访问稀酷客(www.ckook.com)

果指定了 WNOWAIT，则会返回子进程状态，但子进程依然处于可等待的（waitable）状态，稍后可再次等待并获取相同信息。

执行成功，waitid()返回 0，且会更新指针 infop 所指向的 siginfo_t 结构，以包含子进程的相关信息。以下是结构 siginfo_t 的字段情况。

si_code

该字段包含以下值之一：CLD_EXITED，表示子进程已通过调用_exit()而终止；CLD_KILLED，表示子进程为某个信号所杀；CLD_STOPPED，表示子进程因某个信号而停止；CLD_CONTINUED，表示（之前停止的）子进程因接收到（SIGCONT）信号而恢复执行。

si_pid

该字段包含状态发生变化子进程的进程 ID。

si_signo

总是将该字段置为 SIGCHLD。

si_status

该字段要么包含传递给_exit()的子进程退出状态，要么包含导致子进程停止、继续或终止的信号值。可以通过读取 si_code 值来判定具体包含的是哪一种类型的信息。

si_uid

该字段包含子进程的真正用户 ID。大部分其他 UNIX 实现不会设置该字段。

在 Solaris 系统中，此结构还包含两个附加字段：si_stime 和 si_utime，分别包含子进程使用的系统和用户 CPU 时间。SUSv3 并不要求 waitid()处理这两个字段。

waitid()操作的一处细节需要进一步澄清。如果在 options 中指定了 WNOHANG，那么 waitid()返回 0 意味着以下两种情况之一：在调用时子进程的状态已经改变（关于子进程的相关信息保存在 infop 指针所指向的结构 siginfo_t 中），或者没有任何子进程的状态有所改变。对于没有任何子进程改变状态的情况，一些 UNIX 实现（包括 Linux）会将 siginfo_t 结构内容清 0。这也是区分两种情况的方法之一：检查 si_pid 的值是否为 0。不幸的是，SUSv3 并未规范这一行为，一些 UNIX 实现此时会保持结构 siginfo_t 原封不动。（未来针对 SUSv4 的勘误表可能会增加在这种情况下将 si_pid 和 si_signo 置 0 的要求。）区分这两种情况唯一可移植的方法是：在调用 waitid()之前就将结构 siginfo_t 的内容置为 0，正如以下代码所示：

```
siginfo_t info;
...
memset(&info, 0, sizeof(siginfo_t));
if (waitid(idtype, id, &info, options | WNOHANG) == -1)
    errExit("waitid");
if (info.si_pid == 0) {
    /* No children changed state */
} else {
    /* A child changed state; details are provided in 'info' */
}
```

26.1.6 系统调用 wait3()和 wait4()

系统调用 wait3()和 wait4()执行与 waitpid()类似的工作。主要的语义差别在于，wait3()和 wait4()

更多资源请访问稀酷客(www.ckook.com)

在参数 rusage 所指向的结构中返回终止子进程的资源使用情况。其中包括进程使用的 CPU 时间总量以及内存管理的统计数据。36.1 节将在介绍系统调用 getrusage()时详细讨论 rusage 结构。

```
#define _BSD_SOURCE        /* Or #define _XOPEN_SOURCE 500 for wait3() */
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Both return process ID of child, or −1 on error

除了对参数 rusage 的使用之外，调用 wait3()等同于以如下方式调用 waitpid()：

```
waitpid(-1, &status, options);
```

与之相类似，对 wait4()的调用等同于对 waitpid()的如下调用：

```
waitpid(pid, &status, options);
```

换言之，wait3()等待的是任意子进程，而 wait4()则可以用于等待选定的一个或多个子进程。

在一些 UNIX 实现中，wait3()和 wait4()仅返回已终止子进程的资源使用情况。而对于 Linux 系统，如果在 options 中指定了 WUNTRACED 选项，则还可以获取到停止子进程的资源使用信息。

这两个系统调用的名称来自于它们所使用参数的个数。虽然源自 BSD 系统，不过现在大部分的 UNIX 实现都支持它们。这两个系统调用均未获得 SUSv3 标准的接纳。（SUSv2 标准纳入了 wait3()，但将其标记为“已过时”。）

本书一般会避免使用 wait3()和 wait4()。通常情况下，此类调用所返回的额外信息没有什么价值。此外，未获业界标准的接纳也会限制其可移植性。

26.2 孤儿进程与僵尸进程

父进程与子进程的生命周期一般都不相同，父、子进程间互有长短。这就引出了下面两个问题。

- 谁会是孤儿（orphan）子进程的父进程？进程 ID 为 1 的众进程之祖——init 会接管孤儿进程。换言之，某一子进程的父进程终止后，对 getppid()的调用将返回 1。这是判定某一子进程之“生父”是否“在世”的方法之一（前提是假设该子进程由 init 之外的进程创建）。

使用参数 PR_SET_PDEATHSIG 调用 Linux 特有的系统调用 prctl()，将有可能导致某一进程在成为孤儿时收到特定信号。

- 在父进程执行 wait()之前，其子进程就已经终止，这将会发生什么？此处的要点在于，即使子进程已经结束，系统仍然允许其父进程在之后的某一时刻去执行 wait()，以确定该子进程是如何终止的。内核通过将子进程转为僵尸进程（zombie）来处理这种情况。这也意味着将释放子进程所把持的大部分资源，以便供其他进程重新使用。该进程所唯一保留的是内核进程表中的一条记录，其中包含了子进程 ID、终止状态、资源使用数据（36.1 节）等信息。

至于僵尸进程名称的由来，则源于 UNIX 系统对电影情节的效仿——无法通过信号来杀死僵尸进程，即便是（银弹）SIGKILL。这就确保了父进程总是可以执行 wait()方法。

当父进程执行 wait()后，由于不再需要子进程所剩余的最后信息，故而内核将删除僵尸进程。另一方面，如果父进程未执行 wait()随即退出，那么 init 进程将接管子进程并自动调用 wait()，

更多资源请访问稀酷客(www.ckook.com)

从而从系统中移除僵尸进程。

如果父进程创建了某一子进程，但并未执行 wait()，那么在内核的进程表中将为该子进程永久保留一条记录。如果存在大量此类僵尸进程，它们势必将填满内核进程表，从而阻碍新进程的创建。既然无法用信号杀死僵尸进程，那么从系统中将其移除的唯一方法就是杀掉它们的父进程（或等待其父进程终止），此时 init 进程将接管和等待这些僵尸进程，从而从系统中将它们清理掉。

在设计长生命周期的父进程（例如：会创建众多子进程的网络服务器和 Shell）时，这些语义具有重要意义。换句话说，在此类应用中，父进程应执行 wait()方法，以确保系统总是能够清理那些死去的子进程，避免使其成为长寿僵尸。如 26.3.1 节所述，父进程在处理 SIGCHLD 信号时，对 wait()的调用既可同步，也可异步。

程序清单 26-4 展示了一个僵尸进程的创建，以及发送 SIGKILL 信号无法杀死僵尸进程的例子。运行这一程序的输出如下:

```
$ ./make_zombie
Parent PID=1013
Child (PID=1014) exiting
 1013 pts/4    00:00:00 make_zombie                    Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
After sending SIGKILL to make_zombie (PID=1014):
 1013 pts/4    00:00:00 make_zombie                    Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
```

以上输出中，ps(1)所输出的字符串<defunct>表示进程处于僵尸状态。

程序清单 26-4 使用 system()函数来执行通过字符串参数传入的 shell 命令。27.6 节将会详细描述 system()函数。

程序清单 26-4：创建一个僵尸子进程

procexec/make_zombie.c

```
#include <signal.h>
#include <libgen.h>             /* For basename() declaration */
#include "tlpi_hdr.h"

#define CMD_SIZE 200

int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    pid_t childPid;

    setbuf(stdout, NULL);       /* Disable buffering of stdout */

    printf("Parent PID=%ld\n", (long) getpid());

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:     /* Child: immediately exits to become zombie */
        printf("Child (PID=%ld) exiting\n", (long) getpid());
        _exit(EXIT_SUCCESS);
    default:    /* Parent */
        sleep(3);               /* Give child a chance to start and exit */
        snprintf(cmd, CMD_SIZE, "ps | grep %s", basename(argv[0]));
```

```
    cmd[CMD_SIZE - 1] = '\0';       /* Ensure string is null-terminated */
    system(cmd);            /* View zombie child */

    /* Now send the "sure kill" signal to the zombie */

    if (kill(childPid, SIGKILL) == -1)
        errMsg("kill");
    sleep(3);               /* Give child a chance to react to signal */
    printf("After sending SIGKILL to zombie (PID=%ld):\n", (long) childPid);
    system(cmd);            /* View zombie child again */

    exit(EXIT_SUCCESS);
}
}
```

procexec/make_zombie.c

26.3 SIGCHLD 信号

子进程的终止属异步事件。父进程无法预知其子进程何时终止。（即使父进程向子进程发送 SIGKILL 信号，子进程终止的确切时间还依赖于系统的调度：子进程下一次在何时使用 CPU。）之前已经论及，父进程应使用 wait()（或类似调用）来防止僵尸子进程的累积，以及采用如下两种方法来避免这一问题。

- 父进程调用不带 WNOHANG 标志的 wait()，或 waitpid()方法，此时如果尚无已经终止的子进程，那么调用将会阻塞。
- 父进程周期性地调用带有 WNOHANG 标志的 waitpid()，执行针对已终止子进程的非阻塞式检查（轮询）。

这两种方法使用起来都有所不便。一方面，可能并不希望父进程以阻塞的方式来等待子进程的终止。另一方面，反复调用非阻塞的 waitpid()会造成 CPU 资源的浪费，并增加应用程序设计的复杂度。为了规避这些问题，可以采用针对 SIGCHLD 信号的处理程序。

26.3.1 为 SIGCHLD 建立信号处理程序

无论一个子进程于何时终止，系统都会向其父进程发送 SIGCHLD 信号。对该信号的默认处理是将其忽略，不过也可以安装信号处理程序来捕获它。在处理程序中，可以使用 wait()（或类似方法）来收拾僵尸进程。不过，使用这一方法时需要掌握一些窍门。

由 20.10 节和 20.12 节可知，当调用信号处理程序时，会暂时将引发调用的信号阻塞起来（除非为 sigaction()指定了 SA_NODEFER 标志），且不会对 SIGCHLD 之流的标准信号进行排队处理。这样一来，当 SIGCHILD 信号处理程序正在为一个终止的子进程运行时，如果相继有两个子进程终止，即使产生了两次 SIGCHLD 信号，父进程也只能捕获到一个。结果是，如果父进程的 SIGCHLD 信号处理程序每次只调用一次 wait()，那么一些僵尸子进程可能会成为“漏网之鱼”。

解决方案是：在 SIGCHLD 处理程序内部循环以 WNOHANG 标志来调用 waitpid()，直至再无其他终止的子进程需要处理为止。通常 SIGCHLD 处理程序都简单地由以下代码组成，仅仅捕获已终止子进程而不关心其退出状态。

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    continue;
```

上述循环会一直持续下去，直至 waitpid()返回 0，表明再无僵尸子进程存在，或-1，表示有

更多资源请访问稀酷客(www.ckook.com)

错误发生（可能是 ECHILD，意即再无更多的子进程）。

SIGCHLD 处理程序的设计问题

假设创建 SIGCHLD 处理程序的时候，该进程已经有子进程终止。那么内核会立即为父进程产生 SIGCHLD 信号吗？SUSv3 对这一点并未规定。一些源自系统 V（System V）的实现在这种情况下会产生 SIGCHLD 信号；而另一些系统，包括 Linux，则不这么做。为保障可移植性，应用应在创建任何子进程之前就设置好 SIGCHLD 处理程序，将这一隐患消解于无形。（无疑，这也是顺其自然的处事之道。）

需要更深入考虑的问题是可重入性（reentrancy）。21.1.2 节特别指出，在信号处理程序中使用系统调用（如 waitpid()）可能会改变全局变量 errno 的值。当主程序企图显式设置 errno（参考 35.1 节对 getpriority()的讨论）或是在系统调用失败后检查 errno 值时，这一变化会与之发生冲突。出于这一原因，有时在编写 SIGCHLD 信号处理程序时，需要在一进入处理程序时就使用本地变量来保存 errno 值，而在返回前加以恢复。请参考程序清单 26-5。

范例程序

程序清单 26-5 提供了一个更为复杂的 SIGCHLD 信号处理程序示例。该处理程序为所捕获的每个子进程输出进程号及其等待状态①。为了模拟调用处理程序期间产生多个 SIGCHLD 信号而无法排队的效果，利用 sleep()调用②人为地拉长了处理程序的执行时间。主程序为每个（整型）命令行参数创建一个子进程④。每个子进程持续休眠其对应命令行参数所指定的秒数，随即退出⑤。从程序下面的执行例子可以看出，尽管有 3 个子进程退出，而父进程只捕获到两次 SIGCHLD 信号。

```
$ ./multi_SIGCHLD 1 2 4
16:45:18 Child 1 (PID=17767) exiting
16:45:18 handler: Caught SIGCHLD              First invocation of handler
16:45:18 handler: Reaped child 17767 - child exited, status=0
16:45:19 Child 2 (PID=17768) exiting          These children terminate during...
16:45:21 Child 3 (PID=17769) exiting          first invocation of handler
16:45:23 handler: returning                   End of first invocation of handler
16:45:23 handler: Caught SIGCHLD              Second invocation of handler
16:45:23 handler: Reaped child 17768 - child exited, status=0
16:45:23 handler: Reaped child 17769 - child exited, status=0
16:45:28 handler: returning
16:45:28 All 3 children have terminated; SIGCHLD was caught 2 times
```

请注意，在程序清单 26-5 中，在创建子进程之前使用了 sigprocmask()来阻塞 SIGCHLD 信号③。这一做法确保了父进程中 sigsuspend()循环的正确操作。如果以此方式未能阻塞 SIGCHLD 信号，而某一子进程又在对 numLiveChildren 的检查和执行 sigsuspend()调用（也可以是 pause()调用）之间终止，那么 sigsuspend()调用会永远阻塞，等待一个早已捕获过的信号⑥。22.9 节详细描述了处理此类竞争条件的要求。

程序清单 26-5：通过 SIGCHLD 信号处理程序捕获已终止的子进程

procexec/multi_SIGCHLD.c

```
#include <signal.h>
#include <sys/wait.h>
#include "print_wait_status.h"
#include "curr_time.h"
#include "tlpi_hdr.h"

static volatile int numLiveChildren = 0;
                /* Number of children started but not yet waited on */
```


更多资源请访问稀酷客(www.ckook.com)

```
static void
sigchldHandler(int sig)
{
    int status, savedErrno;
    pid_t childPid;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), printWaitStatus(), currTime(); see Section 21.1.2) */

    savedErrno = errno;     /* In case we modify 'errno' */

    printf("%s handler: Caught SIGCHLD\n", currTime("%T"));

    while ((childPid = waitpid(-1, &status, WNOHANG)) > 0) {
①       printf("%s handler: Reaped child %ld - ", currTime("%T"),
                (long) childPid);
        printWaitStatus(NULL, status);
        numLiveChildren--;
    }

    if (childPid == -1 && errno != ECHILD)
        errMsg("waitpid");
②   sleep(5);               /* Artificially lengthen execution of handler */
    printf("%s handler: returning\n", currTime("%T"));

    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    int j, sigCnt;
    sigset_t blockMask, emptyMask;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s child-sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);       /* Disable buffering of stdout */

    sigCnt = 0;
    numLiveChildren = argc - 1;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigchldHandler;
    if (sigaction(SIGCHLD, &sa, NULL) == -1)
        errExit("sigaction");

    /* Block SIGCHLD to prevent its delivery if a child terminates
       before the parent commences the sigsuspend() loop below */

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SIGCHLD);
③   if (sigprocmask(SIG_SETMASK, &blockMask, NULL) == -1)
        errExit("sigprocmask");

④   for (j = 1; j < argc; j++) {
```

```
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:         /* Child - sleeps and then exits */
⑤           sleep(getInt(argv[j], GN_NONNEG, "child-sleep-time"));
            printf("%s Child %d (PID=%ld) exiting\n", currTime("%T"),
                    j, (long) getpid());
            _exit(EXIT_SUCCESS);

        default:        /* Parent - loops to create next child */
            break;
        }
    }
    /* Parent comes here: wait for SIGCHLD until all children are dead */

    sigemptyset(&emptyMask);
    while (numLiveChildren > 0) {
⑥       if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        sigCnt++;
    }

    printf("%s All %d children have terminated; SIGCHLD was caught "
            "%d times\n", currTime("%T"), argc - 1, sigCnt);

    exit(EXIT_SUCCESS);
}
```

procexec/multi_SIGCHLD.c

26.3.2 向已停止的子进程发送 SIGCHLD 信号

正如可以使用 waitpid()来监测已停止的子进程一样，当信号导致子进程停止时，父进程也就有可能收到 SIGCHLD 信号。调用 sigaction()设置 SIGCHLD 信号处理程序时，如传入 SA_NOCLDSTOP 标志即可控制这一行为。若未使用该标志，系统会在子进程停止时向父进程发送 SIGCHLD 信号；反之，如果使用了这一标志，那么就不会因子进程的停止而发出 SIGCHLD 信号。（22.7 节中对 signal()的实现就未指定 SA_NOCLDSTOP。）

> 因为默认情况下会忽略信号 SIGCHLD，SA_NOCLDSTOP 标志仅在设置 SIGCHLD 信号处理程序时才有意义。而且，SA_NOCLDSTOP 只对 SIGCHLD 信号起作用。

SUSv3 也允许，当信号 SIGCONT 导致已停止的子进程恢复执行时，向其父进程发送 SIGCHLD 信号。（相当于 waitpid()的 WCONTINUED 标志。）始于版本 2.6.9，Linux 内核实现了这一特性。

26.3.3 忽略终止的子进程

更有可能像这样处理终止子进程：将对 SIGCHLD 的处置（disposition）显式置为 SIG_ IGN，系统从而会将其后终止的子进程立即删除，毋庸转为僵尸进程。这时，会将子进程的状态弃之不问，故而所有后续的 wait()（或类似）调用不会返回子进程的任何信息。

> 注意，虽然对信号 SIGCHLD 的默认处置就是将其忽略，但显式设置对 SIG_IGN 标志的处置还是会导致这里所描述的行为差异。在这方面，对信号 SIGCHLD 的处理非常独特，不同于其他信号。

更多资源请访问稀酷客(www.ckook.com)

如同许多 UNIX 实现一样，在 Linux 系统中将对 SIGCHLD 信号的处置置为 SIG_IGN 并不会影响任何既有僵尸进程的状态，对它们的等待仍然要照常进行。在其他一些 UNIX 实现中（例如 Solaris 8），将对 SIGCHLD 的处置设置为 SIG_IGN 确实会删除所有已有的僵尸进程。

信号 SIGCHLD 的 SIG_IGN 语义由来已久，源于系统 V（System V）。SUSv3 也规定了此处所描述的行为，不过原始的 POSIX.1 标准对此则未作表述。因此，在一些较老的 UNIX 实现中，忽略 SIGCHLD 并不影响僵尸进程的创建。要防止产生僵尸进程，唯一完全可移植的方法就是（可能是从 SIGCHLD 信号处理程序的内部）调用 wait()或者 waitpid()。

老版本 Linux 内核实现与 SUSv3 标准的差异

SUSv3 规定，如果将对 SIGCHLD 的处置设置为 SIG_IGN，那么将丢弃子进程的资源使用信息，且若指定 RUSAGE_CHILDREN 标志调用 getrusage()函数，其返回总量中也将不包含该项信息（36.1 节）。然而，在版本 2.6.9 之前的 Linux 内核中，还是会记录子进程的 CPU 使用时间以及资源的使用情况，并可通过 getrusage()调用获取。这一违规行为直至 Linux 2.6.9 才得以修正。

将对 SIGCHLD 的处置设置为 SIG_IGN 还会阻止 times()（10.7 节）返回的结构中包含子进程的 CPU 使用时间。不过，在 Linux 2.6.9 之前，times()所返回的信息同样存在违规行为。

SUSv3 规定，如果将对 SIGCHLD 的处置设置为 SIG_IGN，同时，父进程已终止的子进程中并无处于僵尸状态且未被等待的情况，那么 wait()（或 waitpid()）调用将一直阻塞，直至所有子进程都终止，届时该调用将返回错误 ECHILD。Linux 2.6 符合这一要求。不过在 Linux 2.4 以及更早期的版本中，wait()只会阻塞到下一个子进程终止的时刻，随即返回该子进程的进程 ID 及状态（亦即，此行为与未将对 SIGCHLD 信号的处置置为 SIG_IGN 时一样）。

sigaction()的 SA_NOCLDWAIT 标志

SUSv3 规定了 SA_NOCLDWAIT 标志，可在调用 sigaction()对 SIGCHLD 信号的处置进行设置时使用此标志。设置该标志的作用类似于将对 SIGCHLD 的处置置为 SIG_IGN 时的效果。Linux 2.4 及其早期版本并未实现该标志，直至 Linux 2.6 才实现对其支持。

将对 SIGCHLD 的处置置为 SIG_IGN 与采用 SA_NOCLDWAIT 之间最主要的区别在于，当以 SA_NOCLDWAIT 设置信号处理程序时，SUSv3 并未规定系统在子进程终止时是否向其父进程发送 SIGCHLD 信号。换言之，当指定 SA_NOCLDWAIT 时允许系统发送 SIGCHLD 信号，则应用程序即可捕捉这一信号（尽管由于内核已经丢弃了僵尸进程，造成 SIGCHLD 处理程序无法用 wait()来获得子进程状态）。在包括 Linux 在内的一些 UNIX 实现中，内核确实会为父进程产生 SIGCHLD 信号。而在另一些 UNIX 实现中，则不会。

当为 SIGCHLD 信号设置 SA_NOCLDWAIT 标志时，老版本 Linux 内核的行为细节同样与 SUSv3 不符，正如之前在将对 SIGCHLD 的处置置为 SIG_IGN 处所讨论的那样。

系统 V 的 SIGCLD 信号

Linux 系统中，信号 SIGCLD 与信号 SIGCHLD 意义相同。之所以两个名称并存，是由历史原因造成的。SIGCHLD 信号源自 BSD，POSIX 标准采用了这一名称，同时对 BSD 信号模型做了大量标准化工作。系统 V 则提供了相应的 SIGCLD 信号，在语义上稍许有些不同。

BSD SIGCHLD 信号与系统 V SIGCLD 间的主要差别在于，将信号处置为 SIG_IGN 时不

更多资源请访问稀酷客(www.ckook.com)

同的处理方式。

- 在历史（和一些现代）的 BSD 实现中，即使忽略了 SIGCHLD 信号，系统仍会继续将无人等待的子进程变为僵尸进程。
- 在系统 V 上，使用 signal()（而非 sigaction()）忽略 SIGCLD 信号将导致子进程在终止时不会转为僵尸进程。

如前所述，原始的 POSIX.1 标准对于忽略 SIGCHLD 的后果未作规定，从而也认可了系统 V 的行为。而今，系统 V 的行为已成为 SUSv3 标准的一部分（不过将仍然使用 SIGCHLD 的名称）。衍生自系统 V 的现代系统实现中对该信号使用了 SIGCHLD 这一标准名称，同时继续提供具有相同含义的 SIGCLD 信号。关于 SIGCLD 的更多信息可参考[Stevens & Rago, 2005]。

26.4 总结

使用 wait()和 waitpid()（以及其他相关函数），父进程可以得到其终止或停止子进程的状态。该状态表明子进程是正常终止（带有表示成功或失败的退出状态），还是异常中止，因收到某个信号而停止，还是因收到 SIGCONT 信号而恢复执行。

如果子进程的父进程终止，那么子进程将变为孤儿进程，并为进程 ID 为 1 的 init 进程接管。

子进程终止后会变为僵尸进程，仅当其父进程调用 wait()（或类似函数）获取子进程退出状态时，才能将其从系统中删除。在设计长时间运行的程序，诸如 shell 程序以及守护进程（daemon）时，应总是捕获其所创建子进程的状态，因为系统无法杀死僵尸进程，而未处理的僵尸进程最终将塞满内核进程表。

捕获终止子进程的一般方法是为信号 SIGCHLD 设置信号处理程序。当子进程终止时（也可选择子进程因信号而停止时），其父进程会收到 SIGCHLD 信号。还有另一种移植性稍差的处理方法，进程可选择将对 SIGCHLD 信号的处置置为忽略（SIG_IGN），这时将立即丢弃终止子进程的状态（因此其父进程从此也无法获取到这些信息），子进程也不会成为僵尸进程。

更多信息

请参考列于 24.6 节中的更多信息来源。

26.5 练习

26-1. 编写一程序以验证当一子进程的父进程终止时，调用 getppid()将返回 1（进程 init 的进程 ID）。

26-2. 假设存在 3 个相互关联的进程（祖父、父及子进程），祖父进程没有在父进程退出之后立即执行 wait()，所以父进程变成僵尸进程。那么请指出孙进程何时被 init 进程收养（即孙进程调用 getppid()将返回 1），是在父进程终止后，还是祖父进程调用 wait()后？请编写程序验证结果。

26-3. 使用 waitid()替换程序清单 26-3（child_status.c）中的 waitpid()。需要将对函数 printWaitStatus()的调用替换为打印 waitid()所返回 siginfo_t 结构中相关字段的代码。

26-4. 程序清单 26-4（make_zombie.c）调用了 sleep()，以便允许子进程在父进程执行函数 system()前得到机会去运行并终止。这一方法理论上存在产生竞争条件的可能。修改此程序，使用信号来同步父子进程以消除该竞争条件。

更多资源请访问稀酷客(www.ckook.com)

第27章

程序的执行

承接前几章对于进程创建和终止的探讨，本章首先将介绍系统调用 execve()，通过该调用，进程能以全新程序来替换当前运行的程序；接下来会讨论函数 system()的实现，该函数可允许调用者执行任意 shell 命令。

27.1 执行新程序：execve()

系统调用 execve()可以将新程序加载到某一进程的内存空间。在这一操作过程中，将丢弃旧有程序，而进程的栈、数据以及堆段会被新程序的相应部件所替换。在执行了各种 C 语言函数库的运行时启动代码以及程序的初始化代码后，例如，C++静态构造函数，或者以 gcc constructor 属性（见 42.4 节）声明的 C 语言函数，新程序会从 main()函数处开始执行。

由 fork()生成的子进程对 execve()的调用最为频繁，不以 fork()调用为先导而单独调用 execve()的做法在应用中实属罕见。

基于系统调用 execve()，还提供了一系列冠以 exec 来命名的上层库函数，虽然接口方式各异，但功能相同。通常将调用这些函数加载一个新程序的过程称作 exec 操作，或是简单地以 exec()来表示。下面将先描述 execve()，然后再对相关库函数进行说明。

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Never returns on success; returns −1 on error

参数 pathname 包含准备载入当前进程空间的新程序的路径名，既可以是绝对路径（冠之以/），也可以是相对于调用进程当前工作目录（current working directory）的相对路径。

参数 argv 则指定了传递给新进程的命令行参数。该数组对应于 C 语言 main()函数的第 2 个参数（argv），且格式也与之相同：是由字符串指针所组成的列表，以 NULL 结束。argv[0]的值则对应于命令名。通常情况下，该值与 pathname 中的 basename（路径名的最后部分）相同。

最后一个参数 envp 指定了新程序的环境列表。参数 envp 对应于新程序的 environ 数组：

更多资源请访问稀酷客(www.ckook.com)

也是由字符串指针组成的列表，以 NULL 结束，所指向的字符串格式为 name=value（6.7 节）。

Linux 所特有的/proc/PID/exe 文件是一个符号链接，包含 PID 对应进程中正在运行可执行文件的绝对路径名。

调用 execve()之后，因为同一进程依然存在，所以进程 ID 仍保持不变。如 28.4 节所述，还有少量其他的进程属性也未发生变化。

如果对 pathname 所指定的程序文件设置了 set-user-ID（set-group-ID）权限位，那么系统调用会在执行此文件时将进程的有效（effective）用户（组）ID 置为程序文件的属主（组）ID。利用这一机制，可令用户在运行特定程序时临时获取特权。（参考 9.3 节）。

无论是否更改了有效 ID，也不管这一变化是否生效，execve()都会以进程的有效用户 ID 去覆盖已保存的（saved）set-user-ID，以进程的有效组 ID 去覆盖已保存的（saved）set-group-ID。

由于是将调用程序取而代之，对 execve()的成功调用将永不返回，而且也无需检查 execve()的返回值，因为该值总是雷打不动地等于-1。实际上，一旦函数返回，就表明发生了错误。通常，可以通过 errno 来判断出错原因。可能自 errno 返回的错误如下：

EACCES

参数 pathname 没有指向一个常规（regular）文件，未对该文件赋予可执行权限，或者因为 pathname 中某一级目录不可搜索（not searchable）（即，关闭了该目录的可执行权限）。还有一种可能，是以 MS_NOEXEC 标志（14.8.1 节）来挂载（mount）文件所在的文件系统，从而导致这一错误。

ENOENT

pathname 所指代的文件并不存在。

ENOEXEC

尽管对 pathname 所指代文件赋予了可执行权限，但系统却无法识别其文件格式。一个脚本文件，如果没有包含用于指定脚本解释器（interpreter）（以字符#!开头）的起始行，就可能导致这一错误。

ETXTBSY

存在一个或多个进程已经以写入方式打开 pathname 所指代的文件（4.3.2 节）。

E2BIG

参数列表和环境列表所需空间总和超出了允许的最大值。

当上述任一条件作用于执行脚本的脚本解释器，或是执行程序的 ELF 解释器时，同样会产生相应错误。

ELF（Executable and Linking Format）是一种广为实现的标准，描述了可执行文件的布局。在执行期间，进程映像（image）通常是由可执行文件的各段（segment）构造而成（6.3 节）。不过，ELF 规格也允许定义一个解释器（ELF 程序头部的 PT_INTERP 元素）来运行程序。如果定义了解释器，内核则基于指定解释器可执行文件的各段来构建进程映像，转而由解释器负责加载和执行程序。第 41 章会对 ELF 解释器做进一步描述，并给出对深层信息的一些指引。

更多资源请访问稀酷客(www.ckook.com)

示例程序

程序清单 27-1 展示了 execve()的用法。该程序首先为新程序创建参数列表和环境列表，接着调用 execve()来执行由命令行参数（argv[1]）所指定的程序路径名。

程序清单 27-2 中所展示的程序，是设计专供程序清单 27-1 中程序来执行的。该程序只是简单显示一下自身的命令行参数以及环境列表（对后者的访问使用了全局变量 environ，如 6.7 节所述）。

如下 shell 会话（session）演示了对程序清单 27-1 和程序清单 27-2 的使用（本例在指定执行程序时使用的是相对路径名）：

```
$ ./t_execve ./envargs                    All of the output is printed by envargs
argv[0] = envargs
argv[1] = hello world
argv[2] = goodbye
environ: GREET=salut
environ: BYE=adieu
```

程序清单 27-1：调用函数 execve()来执行新程序

procexec/t_execve.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *argVec[10];           /* Larger than required */
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    argVec[0] = strrchr(argv[1], '/');      /* Get basename from argv[1] */
    if (argVec[0] != NULL)
        argVec[0]++;
    else
        argVec[0] = argv[1];
    argVec[1] = "hello world";
    argVec[2] = "goodbye";
    argVec[3] = NULL;           /* List must be NULL-terminated */

    execve(argv[1], argVec, envVec);
    errExit("execve");          /* If we get here, something went wrong */
}
```

procexec/t_execve.c

程序清单 27-2：显示参数列表和环境列表

procexec/envargs.c

```
#include "tlpi_hdr.h"

extern char **environ;

int
main(int argc, char *argv[])
{
    int j;
    char **ep;
```


更多资源请访问稀酷客(www.ckook.com)

```
    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);

    for (ep = environ; *ep != NULL; ep++)
        printf("environ: %s\n", *ep);

    exit(EXIT_SUCCESS);
}
```

procexec/envargs.c

27.2 exec()库函数

本节所讨论的库函数为执行 exec()提供了多种 API 选择。所有这些函数均构建于 execve()调用之上，只是在为新程序指定程序名、参数列表以及环境变量的方式上有所不同。

```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
                /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return −1 on error

各函数名称的最后一个字母为区分这些函数提供了线索。表 27-1 总结了这些差异，下面则是详细说明。

- 大部分 exec()函数要求提供欲加载新程序的路径名。而 execlp()和 execvp()则允许只提供程序的文件名。系统会在由环境变量 PATH 所指定的目录列表中寻找相应的执行文件（稍后将详细解释）。这与 shell 对键入命令的搜索方式一致。这些函数名都包含字母 p（表示 PATH），以示在操作上有所不同。如果文件名中包含“/”，则将其视为相对或绝对路径名，不再使用变量 PATH 来搜索文件。
- 函数 execle()、execlp()和 execl()要求开发者在调用中以字符串列表形式来指定参数，而不使用数组来描述 argv 列表。首个参数对应于新程序 main()函数的 argv[0]，因而通常与参数 filename 或 pathname 的 basename 部分相同。必须以 NULL 指针来终止参数列表，以便于各调用定位列表的尾部。（上述各原型注释中的(char*)NULL 部分透露了这一要求。至于为何需要对 NULL 进行强制类型转换，请参考附录 C。）这些函数的名称都包含字母 l（表示 list），以示与那些将以 NULL 结尾的数组作为参数列表的函数有所区别。后者（execve()、execvp()和 execv()）名称中则包含字母 v（表示 vector）。
- 函数 execve()和 execle()则允许开发者通过 envp 为新程序显式指定环境变量，其中 envp 是一个以 NULL 结束的字符串指针数组。这些函数命名均以字母 e（environment）结尾。其他 exec()函数将使用调用者的当前环境（即 environ 中内容）作为新程序的环境。

glibc 2.11 曾加入一个非标准函数 execve(file, argv, envp)。该函数与 execvp()类似，不过并非通过 environ 来取得新程序的环境，而是通过参数 envp（类似于函数 execve()和 execle()）来指定新环境。

更多资源请访问稀酷客(www.ckook.com)

后面几页会演示部分 exec()函数变体的使用。

表 27-1：exec()函数间的差异总结

函　　数	对程序文件的描述（-, p）	对参数的描述（v, l）	环境变量来源（e, -）
execve()	路径名	数组	*envp* 参数
execle()	路径名	列表	*envp* 参数
execlp()	文件名+PATH	列表	调用者的 *environ*
execvp()	文件名+PATH	数组	调用者的 *environ*
execv()	路径名	数组	调用者的 *environ*
excel()	路径名	列表	调用者的 *environ*

27.2.1　环境变量 PATH

函数 execvp()和 execlp()允许调用者只提供欲执行程序的文件名。二者均使用环境变量 PATH 来搜索文件。PATH 的值是一个以冒号（:）分隔，由多个目录名，也将其称为路径前缀（path prefixes）组成的字符串。下例中的 PATH 包含 5 个目录：

```
$ echo $PATH
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin:.
```

对于一个登录 shell 而言，其 PATH 值将由系统级和特定用户的 shell 启动脚本来设置。由于子进程继承其父进程的环境变量，shell 执行每个命令时所创建的进程也就继承了 shell 的 PATH。

PATH 中指定的路径名既可以是绝对路径名（以/开始），也可以是相对路径名。对相对路径名的诠释是基于调用进程的当前工作目录（current working directory）。正如前面例子中所示，可以以.（点）来表示当前工作目录。

> 在 PATH 中包含一个长度为 0 的前缀，也可以用来指定当前工作目录。表示方式有：连续的冒号、起始冒号或尾部冒号（例如，/usr/bin:/bin: ）。SUSv3 废止了这一技术；当前工作目录应该用.（点）来显式指定。

如果没有定义变量 PATH，那么 execvp()和 execlp()会采用默认的路径列表：.:/usr/bin:/bin。

出于安全方面的考虑，通常会将当前工作目录排除在超级用户（root）的 PATH 之外。这是为了防止 root 用户发生如下意外情况：执行当前工作目录下与标准命令同名的程序（事先由恶意用户故意放置），或者将常用命令拼错而执行了当前工作目录下的其他程序（例如，输入 sl 而非 ls）。一些 Linux 发行版还将当前工作目录排除在非特权用户的 PATH 缺省值之外。这里假定，在本书展示的所有 shell 会话日志中，对 PATH 的定义均不包含当前工作目录，而书中示例在执行当前工作目录下的程序时都冠以前缀./，原因也正在于此。（同时还有一重妙用：在本书的 shell 会话日志中，从表现形式上将示例程序与标准命令区分开来。）

函数 execvp()和 execlp()会在 PATH 包含的每个目录中搜索文件，从列表开头的目录开始，直至成功执行了既定文件。如果不清楚可执行程序的具体位置，或是不想因硬编码（hard- code）而对具体位置产生依赖，对 PATH 环境变量的这种使用方式是非常有效的。

应该避免在设置了 set-user-ID 或 set-group-ID 的程序中调用 execvp()和 execlp()，至少应当慎用。需要特别谨慎地控制 PATH 环境变量，以防运行恶意程序。在实际操作中，这意味着应用程序应该使用已知安全的目录列表来覆盖之前定义的任何 PATH 值。

更多资源请访问稀酷客(www.ckook.com)

程序清单 27-3 提供了一个使用 execlp()的例子。下面的 shell 会话日志则演示了如何通过该程序来调用 echo 命令（/bin/echo）：

```
$ which echo
/bin/echo
$ ls -l /bin/echo
-rwxr-xr-x    1 root      15428 Mar 19 21:28 /bin/echo
$ echo $PATH                          Show contents of PATH environment variable
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin            /bin is in PATH
$ ./t_execlp echo                     execlp() uses PATH to successfully find echo
hello world
```

在上例中，程序清单 27-3 程序将字符串 hello world 作为第 3 个参数传递给 execlp()调用。接下来，重新对 PATH 进行定义，从中移去包含程序 echo 的目录/bin：

```
$ PATH=/home/mtk/bin:/usr/local/bin:/usr/bin
$ ./t_execlp echo
ERROR [ENOENT No such file or directory] execlp
$ ./t_execlp /bin/echo
hello world
```

如你所见，当仅向 execlp()提供文件名（即，字符串中不包含斜杠“/”）时，调用会失败。这是因为在 PATH 包含的目录列表中无法找到名为 echo 的文件。另一方面，当提供了包含一个或多个斜杠的路径名时，execlp()则会忽略 PATH 的内容。

程序清单 27-3：使用 execlp()在 PATH 中搜索文件

procexec/t_execlp.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    execlp(argv[1], argv[1], "hello world", (char *) NULL);
    errExit("execlp");          /* If we get here, something went wrong */
}
```

procexec/t_execlp.c

27.2.2 将程序参数指定为列表

如果在编程时已知某个 exec()的参数个数，调用 execle()、execlp()或者 execl()时就可以将参数作为列表传入。较之于将参数装配于一个 argv 向量中，代码要少一些，便于使用。程序清单 27-4 中程序收效与程序清单 27-1 相同，只是调用了 execle()而非 execve()。

程序清单 27-4：使用 execle()，将程序参数指定为列表

procexec/t_execle.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };
    char *filename;
```

```
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    filename = strrchr(argv[1], '/');       /* Get basename from argv[1] */
    if (filename != NULL)
        filename++;
    else
        filename = argv[1];

    execle(argv[1], filename, "hello world", (char *) NULL, envVec);
    errExit("execle");          /* If we get here, something went wrong */
}
```

procexec/t_execle.c

27.2.3 将调用者的环境传递给新程序

函数 execlp()、execvp()、execl()和 execv()不允许开发者显式指定环境列表，新程序的环境继承自调用进程（6.7 节）。这一举措的后果可谓是喜忧参半。出于安全方面的考虑，有时希望确保程序在一个已知（安全）的环境列表下运行。38.8 节将对此做深入讨论。

程序清单 27-5 演示了如何运用函数 execl()使新程序继承调用者的环境。对于通过 fork()从 shell 处所继承的环境，程序首先用函数 putenv()进行了修改，接着执行 printenv 程序来显示环境变量 USER 和 SHELL 的值。运行程序的输出如下：

```
$ echo $USER $SHELL          Display some of the shell's environment variables
blv /bin/bash
$ ./t_execl
Initial value of USER: blv   Copy of environment was inherited from the shell
britta                       These two lines are displayed by execed printenv
/bin/bash
```

程序清单 27-5：调用函数 execl()，将调用者的环境传递给新程序

procexec/t_execl.c

```
#include <stdlib.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Initial value of USER: %s\n", getenv("USER"));
    if (putenv("USER=britta") != 0)
        errExit("putenv");

    execl("/usr/bin/printenv", "printenv", "USER", "SHELL", (char *) NULL);
    errExit("execl");           /* If we get here, something went wrong */
}
```

procexec/t_execl.c

27.2.4 执行由文件描述符指代的程序：fexecve()

glibc 自版本 2.3.2 开始提供函数 fexecve()，其行为与 execve()类似，只是指定将要执行的程序是以打开文件描述符 fd 的方式，而非通过路径名。有些应用程序需要打开某个程序文件，通过执行校验和（checksum）来验证文件内容，然后再运行该程序，这一场景就较为适宜使用函数 fexecve()。

```
#define _GNU_SOURCE
```

```
#include <unistd.h>

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Doesn't return on success; returns -1 on error

当然，即便没有 fexecve()函数，也可以调用 open()来打开文件，读取并验证其内容，并最终运行。然而，在打开与执行文件之间，存在将该文件替换的可能性（持有打开文件描述符并不能阻止创建同名新文件），最终造成验证者并非执行者的情况。

27.3 解释器脚本

所谓解释器（interpreter），就是能够读取并执行文本格式命令的程序。（相形之下，编译器则是将输入源代码译为可在真实或虚拟机器上执行的机器语言。）各种 UNIX shell，以及诸如 awk、sed、perl、python 和 ruby 之类的程序都属于解释器。除了能够交互式地读取和执行命令之外，解释器通常还具备这样一种能力：从被称为脚本（script）的文本文件中读取和执行命令。

UNIX 内核运行解释器脚本的方式与二进制（binary）程序无异，前提是脚本必须满足下面两点要求：首先，必须赋予脚本文件可执行权限；其次，文件的起始行（initial line）必须指定运行脚本解释器的路径名。格式如下：

#! *interpreter-path* [*optional-arg*]

字符#!必须置于该行起始处，这两个字符串与解释器路径名之间可以以空格分隔。在解释该路径名时不会使用环境变量 PATH，因而一般应采用绝对路径。使用相对路径固然可行，但很少见。对其解释则相对于启动解释器进程的当前工作目录。解释器路径名后还可跟随可选参数（稍后将解释其目的），二者之间以空格分隔。可选参数中不应包含空格。

作为例子，UNIX shell 脚本通常以下面这行开始，指定运行该脚本的 shell：

```
#!/bin/sh
```

解释器脚本文件首行中的可选参数不应包含空格，因为空格此处所起的作用完全取决于实现。Linux 系统不会对可选参数（optional-arg）中的空格做特殊解释，将从参数起始直至行尾的所有文本视为一个单词（正如后面所述，再将其作为一整个参数传递给解释器）。注意，对空格的这种处理方式与 shell 的做法形成鲜明对比，后者总是将其视为命令行中各单词的界定符。

其他 UNIX 实现在处理可选参数中的空格时，其做法与 Linux 有同有异。在 6.0 版本之前的 FreeBSD 上，可在解释器路径（interpreter-path）之后跟随多个以空格分隔的可选参数（并作为多个独立的单词传递给解释器）；而到了 6.0 版本，其行为又转而与 Linux 趋同。而 Solaris 8 则使用空格来表征可选参数的结束，同时忽略#!行中之后的任何剩余文本。

Linux 内核要求脚本的#!起始行不得超过 127 个字节，其中不包括行尾的换行符（newline）。超出部分会被悄无声息地略去。

SUSv3 并未对脚本解释器的#!行技术加以规范，不过大多数 UNIX 实现都支持这一特性。

不同的 UNIX 实现对于#!行的长度限制有所不同。例如，OpenBSD 3.1 的限制为 64 个字节，而 Tru64 5.1 则为 1024 字节。在一些早期的实现（例如 SunOS 4）中，这一限制甚至低至 32 字节。

更多资源请访问稀酷客(www.ckook.com)

解释器脚本的执行

因为脚本并不包含二进制机器码，所以当调用 execve()来运行脚本时，显然发生了一些不同寻常的事件。execve()如果检测到传入的文件以两字节序列“#!”开始，就会析取该行的剩余部分（路径名以及参数），然后按如下参数列表来执行解释器程序：

interpreter-path [*optional-arg*] *script-path arg*...

这里，interpreter-path（解释器路径）和 optional-arg（可选参数）都取自脚本的#!行，script-path（脚本路径）是传递给 execve()的路径名，arg ... 则是通过变量 argv 传递给 execve()的参数列表（不过将 argv[0]排除在外）。图 27-1 对每个脚本参数的起源做了总结。

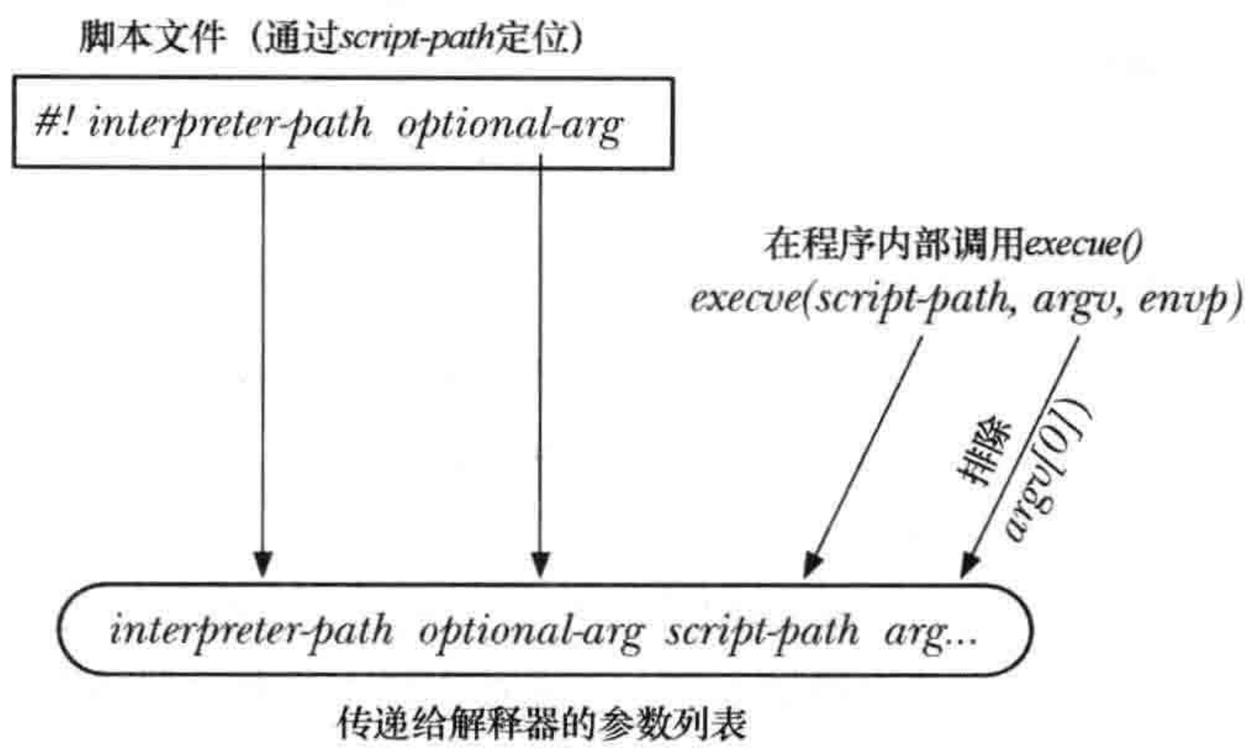

图 27-1：提供给可执行脚本的参数列表

编写一个脚本，用程序清单 6-2（necho.c）程序作为解释器，用于说明解释器参数的来源。该程序只是简单地输出所有的命令行参数。接着，再使用 27-1 中程序来执行该脚本：

```
$ cat > necho.script                  Create script
#!/home/mtk/bin/necho some argument
Some junk
Type Control-D
$ chmod +x necho.script               Make script executable
$ ./t_execve necho.script             And exec the script
argv[0] = /home/mtk/bin/necho         First 3 arguments are generated by kernel
argv[1] = some argument               Script argument is treated as a single word
argv[2] = necho.script                This is the script path
argv[3] = hello world                 This was argVec[1] given to execve()
argv[4] = goodbye                     And this was argVec[2]
```

在本例中，“解释器（necho）”并不关心脚本的内容（necho.script），脚本的第 2 行（Some junk）在执行时不起作用。

> 2.2 内核在执行脚本时将只传递 interpreter-path（解释器路径）的 basename 部分，以作为调用脚本的首个参数。所以，对于 Linux 2.2 来说，argv[0]的输出行会只显示值 necho。

大多数 UNIX shell 和解释器会视字符#为注释的开始。因此，这些解释器在解释脚本时会忽略带有#!的初始行。

使用脚本的 optional-arg（可选参数）

在脚本的#!起始行中，optional-arg 的用途之一是为解释器指定命令行参数。对于 awk 之

更多资源请访问稀酷客(www.ckook.com)

类的解释器而言，这是非常实用的特性。

> 自 20 世纪 70 年代末期开始，awk 解释器业已成为 UNIX 系统的一部分。在介绍 awk 语言的诸多书籍之中，就有一本[Aho 等，1988]是由该语言的 3 位发明者所著，而该语言的命名也源于 3 人名字的首字母。Awk 的长处在于，能快速为文本处理程序创建原型。作为一门弱类型语言，其设计中富含多种文本处理原素，语法结构则以 C 语言为基础。对于时下风光无限的诸多脚本语言（诸如 JavaScript 和 PHP）而言，awk 的始祖地位毋庸置疑。

向 awk 提供脚本有两种不同方式。默认方式是将脚本作为 awk 的首个命令行参数：

```
$ awk 'script' input-file...
```

也可以将 awk 脚本保存于文件之中，正如下面显示最长输入行长度的例子那样：

```
$ cat longest_line.awk
#!/usr/bin/awk
length > max { max = length; }
END          { print max; }
```

假设使用如下 C 代码来执行这一脚本：

```
execl("longest_line.awk", "longest_line.awk", "input.txt", (char *) NULL);
```

execl()转而调用 execve()，以如下参数列表来运行 awk：

```
/usr/bin/awk longest_line.awk input.txt
```

由于 awk 会把字符串 longest_line.awk 解释为一个包含无效 awk 命令的脚本，故而 execve()调用将以失败告终。这就需要有一种方法来通知 awk：该参数实际上是包含脚本的文件名称。在脚本的#!起始行中加入-f 可选参数，就可达到这一目的。这等于告诉 awk，后面的参数是一个脚本文件：

```
#!/usr/bin/awk -f
length > max { max = length; }
END          { print max; }
```

现在，新的 execl()调用会使用如下参数列表：

```
/usr/bin/awk -f longest_line.awk input.txt
```

这样，awk 就可以成功地执行 longest_line.awk 脚本来处理输入文件 input.txt。

使用 execlp()和 execvp()执行脚本

通常，脚本缺少#!起始行将导致 exec()函数执行失败。不过，execlp()和 execvp()的行事方式多少有些不同。前面提到，这些函数会通过环境变量 PATH 来获取目录列表，并在其中搜索将要执行的文件。两个函数无论谁找到该文件，如果既具有可执行权限，又并非二进制格式，且起始行也不以#!开始，那么就会使用 shell 来解释这一文件。Linux 中，会将这类文件视同于包含#!/bin/sh 起始行的文件来进行处理。

27.4 文件描述符与 exec()

默认情况下，由 exec()的调用程序所打开的所有文件描述符在 exec()的执行过程中会保持打开状态，且在新程序中依然有效。这通常很实用，因为调用程序可能会以特定的描述符来打开文件，而在新程序中这些文件将自动有效，无需再去了解文件名或是把它们重新打开。

shell 利用这一特性为其所执行的程序处理 I/O 重定向。例如，假设键入如下的 shell 命令：

```
$ ls /tmp > dir.txt
```


更多资源请访问稀酷客(www.ckook.com)

shell 运行该命令时，执行了以下步骤。

1. 调用 fork()创建子进程，子进程会也运行 shell 的一份拷贝（因此命令行也有一份拷贝）。
2. 子 shell 以描述符 1（标准输出）打开文件 dir.txt 用于输出。可能会采取以下任一方式。
   a）子 shell 关闭描述符 1（STDOUT_FILENO）后，随即打开文件 dir.txt。因为 open()在为描述符取值时总是取最小值，而标准输入（描述符 0）又仍处于打开状态，所以会以描述符 1 来打开文件。
   b）shell 打开文件 dir.txt，获取一个新的文件描述符。之后，如果该文件描述符不是标准输出，那么 shell 会使用 dup2()强制将标准输出复制为新描述符的副本，并将此时已然无用的新描述符关闭。（这种方法较之前者更为安全，因为它并不依赖于打开文件描述符的低值取数原则。）源代码顺序大体如下：

```
fd = open("dir.txt", O_WRONLY | O_CREAT,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
           /* rw-rw-rw- */
if (fd != STDOUT_FILENO) {
    dup2(fd, STDOUT_FILENO);
    close(fd);
}
```

3. 子 shell 执行程序 ls。ls 将其结果输出到标准输出，亦即文件 dir.txt 中。

此处对 shell 处理 I/O 重定向的解释有所简化。特别是，某些命令，即所谓 shell 内建命令，是由 shell 直接运行的，并未调用 fork()或者 exec()。在处理 I/O 重定向时，针对这样的命令必须进行特殊处理。

将某一 shell 命令实现为内建命令，不外乎如下两个目的：效率以及会对 shell 产生副作用（side effect）。一些频繁使用的命令（如 pwd、echo 和 test）逻辑都很简单，放在 shell 内部实现效率会更高。将其他命令内置于 shell 实现，则是希望命令对 shell 本身能产生副作用：更改 shell 所存储的信息，修改 shell 进程的属性，亦或是影响 shell 进程的运行。例如，cd 命令必须改变 shell 自身的工作目录，故而不应在一个独立进程中执行。产生副作用的内建命令还包括 exec、exit、read、set、source、ulimit、umask、wait 以及 shell 的作业控制（job-control）命令（jobs、fg 和 bg）。想了解 shell 支持的全套内建命令，可参考 shell 手册页（manual page）文档。

执行时关闭（close-on-exec）标志（FD_CLOEXEC）

在执行 exec()之前，程序有时需要确保关闭某些特定的文件描述符。尤其是在特权进程中调用 exec()来启动一个未知程序时（并非自己编写），亦或是启动程序并不需要使用这些已打开的文件描述符时，从安全编程的角度出发，应当在加载新程序之前确保关闭那些不必要的文件描述符。对所有此类描述符施以 close()调用就可达到这一目的，然而这一做法存在如下局限性。

- 某些描述符可能是由库函数打开的。但库函数无法使主程序在执行 exec()之前关闭相应的文件描述符。作为基本原则，库函数应总是为其打开的文件设置执行时关闭（close-on-exec）标志，稍后将介绍所使用的技术。
- 如果 exec()因某种原因而调用失败，可能还需要使描述符保持打开状态。如果这些描述符已然关闭，将它们重新打开并指向相同文件的难度很大，基本上不太可能。

为此，内核为每个文件描述符提供了执行时关闭标志。如果设置了这一标志，那么在成功执行 exec()时，会自动关闭该文件描述符，如果调用 exec()失败，文件描述符则会保持打开状态。可以通过系统调用 fcntl()（5.2 节）来访问执行时关闭标志。fcntl()的 F_GETFD 操作可

更多资源请访问稀酷客(www.ckook.com)

以获取文件描述符标志的一份拷贝：

```
int flags;

flags = fcntl(fd, F_GETFD);
if (flags == -1)
    errExit("fcntl");
```

获取这些标志后，可以对 FD_CLOEXEC 位进行修改，再调用 fcntl()的 F_SETFD 操作令其生效：

```
flags |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, flags) == -1)
    errExit("fcntl");
```

> 实际上 FD_CLOEXEC 是文件描述符标志中唯一可以操作的一位。该位对应值为 1。在较老一些的程序中，有时可以看到以 fcntl(fd, F_SETFD,1)的调用方式来设置执行时关闭标志，其事实依据是这种操作不可能影响到其他位。但理论上，情况不会总是一成不变（未来，一些 UNIX 系统可能会实现其他的标志位），所以还是使用正文所示的技术较为稳妥。
>
> 包括 Linux 在内的许多 UNIX 实现，还允许以另外两种非标准的 ioctl()调用来修改执行时关闭标志：以 ioctl(fd, FIOCLEX)为 fd 设置此标志，以 ioctl(fd, FIONCLEX)来清除此标志。

当使用 dup()、dup2()或 fcntl()为一文件描述符创建副本时，总是会清除副本描述符的执行时关闭标志。（这一现象既有其历史渊源，也顺应了 SUSv3 的要求。）

程序清单 27-6 展示了对执行时关闭标志的操作。如果运行时带有命令行参数（可为任意字符串），该程序首先为标准输出设置执行时关闭标志，随后执行 ls 命令。程序运行的结果如下：

```
$ ./closeonexec                    Exec ls without closing standard output
-rwxr-xr-x   1 mtk    users    28098 Jun 15 13:59 closeonexec
$ ./closeonexec n                  Sets close-on-exec flag for standard output
ls: write error: Bad file descriptor
```

上例中第 2 次运行该程序时，ls 检测出其标准输出已然关闭，故而向标准错误（stderr）输出了一条错误信息。

程序清单 27-6：为一文件描述符设置执行时关闭标志

procexec/closeonexec.c

```
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int flags;

    if (argc > 1) {
        flags = fcntl(STDOUT_FILENO, F_GETFD);              /* Fetch flags */
        if (flags == -1)
            errExit("fcntl - F_GETFD");

        flags |= FD_CLOEXEC;                        /* Turn on FD_CLOEXEC */

        if (fcntl(STDOUT_FILENO, F_SETFD, flags) == -1)     /* Update flags */
            errExit("fcntl - F_SETFD");
    }
```

```
    execlp("ls", "ls", "-l", argv[0], (char *) NULL);
    errExit("execlp");
}
```

procexec/closeonexec.c

27.5 信号与 exec()

exec()在执行时会将现有进程的文本段丢弃。该文本段可能包含了由调用进程创建的信号处理器程序。既然处理器已经不知所踪，内核就会将对所有已设信号的处置重置为 SIG_DFL。而对所有其他信号（即将处置置为 SIG_IGN 或 SIG_DFL 的信号）的处置则保持不变。这也符合 SUSv3 的要求。

不过，遭忽略的 SIGCHLD 信号属于 SUSv3 中的特例。（之前曾在 26.3.3 节提及，忽略 SIGCHLD 能够阻止僵尸进程的产生）。至于调用 exec()之后，是继续让遭忽略的 SIGCHLD 信号保持被忽略状态，还是将对其处置重置为 SIG_DFL，SUSv3 对此不置可否。Linux 的操作取其前者，而其他一些 UNIX 实现（如：Solaris）则采用后者。这就意味着，对于忽略 SIGCHLD 的程序而言，要最大限度的保证可移植性，就应该在调用 exec()之前执行 signal（SIGCHLD，SIG_DFL）。此外，程序也不应当假设对 SIGCHLD 处置的初始设置是 SIG_DFL 之外的其他值。

老程序的数据段、堆以及栈悉数被毁，这也意味着通过 sigaltstack()（21.3 节）所创建的任何备选信号栈都会丢失。由于 exec()在调用期间不会保护备选信号栈，故而也会将所有信号的 SA_ONSTACK 位清除掉。

在调用 exec()期间，进程信号掩码以及挂起（pending）信号的设置均得以保存。这一特性允许对新程序的信号进行阻塞和排队处理。不过，SUSv3 指出，许多现有应用程序的编写都基于如下的错误假设：程序启动时将对某些特定信号的处置置为 SIG_DFL，又或者并未阻塞这些信号。（特别是，C 语言标准对信号的规范很弱，对信号阻塞也未置一词，所以为非 UNIX 系统所编写的 C 程序也不可能去解除对信号的阻塞。）为此，SUSv3 建议，在调用 exec()执行任何程序的过程中，不应当阻塞或忽略信号。这里的“任何程序”是指并非由 exec()的调用者所编写的程序。至于说如果执行和被执行的程序均出自一人之手，又或者对运行程序处理信号的手法知根知底，那自然又另当别论。

27.6 执行 shell 命令：system()

程序可通过调用 system()函数来执行任意的 shell 命令。本节将讨论 system()的操作，下一节将介绍如何运用 fork()、exec()、wait()和 exit()来实现 system()。

44.5 节所介绍的 popen()和 pclose()函数同样可以用来执行 shell 命令，而且还允许调用程序向命令发送输入信息，或是读取命令的输出。

```
#include <stdlib.h>

int system(const char *command);
```

See main text for a description of return value

函数 system()创建一个子进程来运行 shell，并以之执行命令 command。其调用示例如下：

```
system("ls | wc");
```

system()的主要优点在于简便。

- 无需处理对 fork()、exec()、wait()和 exit()的调用细节。
- system()会代为处理错误和信号。
- 因为 system()使用 shell 来执行命令（command），所以会在执行 command 之前对其进行所有的常规 shell 处理、替换以及重定向操作。为应用增加“执行一条 shell 命令”的功能不过是举手之劳。（许多交互式应用程序以“！command”的形式提供了这一功能。）

但这些优点是以低效率为代价的。使用 system()运行命令需要创建至少两个进程。一个用于运行 shell，另外一个或多个则用于 shell 所执行的命令（执行每个命令都会调用一次 exec()）。如果对效率或者速度有所要求，最好还是直接调用 fork()和 exec()来执行既定程序。

system()的返回值如下。

- 当 command 为 NULL 指针时，如果 shell 可用则 system()返回非 0 值，若不可用则返回 0。这种返回值方式源于 C 语言标准，因为并未与任何操作系统绑定，所以如果 system()运行在非 UNIX 系统上，那么该系统可能是没有 shell 的。此外，即便所有 UNIX 实现都有 shell，如果程序在调用 system()之前又调用了 chroot()，那么 shell 依然可能无效。若 command 不为 NULL，则 system()的返回值由本列表中的余下规则决定。
- 如果无法创建子进程或是无法获取其终止状态，那么 system()返回-1。
- 若子进程不能执行 shell，则 system()的返回值会与子 shell 调用_exit(127)终止时一样。
- 如果所有的系统调用都成功，system()会返回执行 command 的子 shell 的终止状态。shell 的终止状态是其执行最后一条命令时的退出状态；如果命令为信号所杀，大多数 shell 会以值 128+n 退出，其中 n 为信号编号（如果是子 shell 为信号所杀，那么其终止状态如 26.1.3 节所述）。

> 至于调用失败是由于 system()无法执行 shell，还是 shell 以状态 127 退出（若 shell 未能发现并执行既定名称的程序，就会导致后一种情况的发生），（通过 system()的返回值）是无法区分的。

在最后两种情况中，system()的返回值与 waitpid()所返回的等待状态（wait status）形式相同。因此，可以使用 26.1.3 节所述函数来分析返回值，并以 printWaitStatus()函数（见程序清单 26-2）加以显示。

示例程序

程序清单 27-7 演示了 system()的用法。程序循环读取命令字符串，再使用 system()来执行命令，并对 system()的返回值进行分析和展示。下面是一个运行的例子：

```
$ ./t_system
Command: whoami
mtk
system() returned: status=0x0000 (0,0)
child exited, status=0
Command: ls | grep XYZ                    Shell terminates with the status of...
system() returned: status=0x0100 (1,0)    its last command (grep), which...
child exited, status=1                    found no match, and so did an exit(1)
Command: exit 127
system() returned: status=0x7f00 (127,0)
(Probably) could not invoke shell         Actually, not true in this case
Command: sleep 100
Type Control-Z to suspend foreground process group
```

```
[1]+  Stopped                 ./t_system
$ ps | grep sleep                          Find PID of sleep
29361 pts/6    00:00:00 sleep
$ kill 29361                               And send a signal to terminate it
$ fg                                       Bring t_system back into foreground
./t_system
system() returned: status=0x000f (0,15)
child killed by signal 15 (Terminated)
Command: ^D$                               Type Control-D to terminate program
```

程序清单 27-7：通过 system()执行 shell 命令

procexec/t_system.c

```
#include <sys/wait.h>
#include "print_wait_status.h"
#include "tlpi_hdr.h"

#define MAX_CMD_LEN 200

int
main(int argc, char *argv[])
{
    char str[MAX_CMD_LEN];      /* Command to be executed by system() */
    int status;                 /* Status return from system() */

    for (;;) {                  /* Read and execute a shell command */
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            break;              /* end-of-file */

        status = system(str);
        printf("system() returned: status=0x%04x (%d,%d)\n",
                (unsigned int) status, status >> 8, status & 0xff);

        if (status == -1) {
            errExit("system");
        } else {
            if (WIFEXITED(status) && WEXITSTATUS(status) == 127)
                printf("(Probably) could not invoke shell\n");
            else                /* Shell successfully executed command */
                printWaitStatus(NULL, status);
        }
    }

    exit(EXIT_SUCCESS);
}
```

procexec/t_system.c

在设置用户 ID（set-user-ID）和组 ID（set-group-ID）程序中避免使用 system()

当设置了用户 ID 和组 ID 的程序在特权模式下运行时，绝不能调用 system()。即便此类程序并未允许用户指定需要执行的命令文本，鉴于 shell 对操作的控制有赖于各种环境变量，故而使用 system()会不可避免地给系统带来安全隐患。

例如，在较老的 Bourne shell 中，环境变量 IFS（定义了用于将命令行拆分为独立单词的内部字段分隔符）就引发了若干起对系统入侵的成功案例。如果将 IFS 定义为 a，那么 shell 会将字

更多资源请访问稀酷客(www.ckook.com)

符串 shar 视为带有参数 r 的单词 sh，并启动另一 shell 进程来执行当前工作目录下名为 r 的脚本，这就一改命令的原意（执行名为 shar 的命令）。对这一安全漏洞的修复举措是，将 IFS 只应用于 shell 扩展所产生的单词。此外，现代 shell 会在启动时重置 IFS（为由空格、Tab 以及换行 3 个字符组成的字符串），以确保即使 IFS 的继承值很奇怪，脚本的行为也会保持一致。作为进一步的安全举措，当从设置用户（组）ID 程序中调用时，bash 会回转为实际用户（组）ID 的“真身”。

应用需要加载其他程序时，为确保安全过关，应当直接调用 fork()和 exec()系函数（execlp()和 execvp()除外）之一。

27.7 system()的实现

本节将说明如何实现 system()的功能。首先给出一个简化版实现，接着指出这一实现的缺失所在，最后再展示了一个完整的实现。

对 system()的简化实现

命令 sh 的参数-c 提供了一种简单的方法，可以执行包含任意命令的字符串：

```
$ sh -c "ls | wc"
     38      38     444
```

因此，为了实现 system()，需要使用 fork()来创建一个子进程，并以对应于上例 sh 命令的参数来调用 execl()：

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

为了收集 system()所创建的子进程状态，还以指定的子进程 ID 调用了 waitpid()。（使用 wait()并不合适，因为 wait()等待的是任一子进程，因而无意间所获取的子进程状态可能属于其他子进程。）程序清单 27-8 是对 system()的简化实现。

程序清单 27-8：一个缺乏信号处理的 system()实现

procexec/simple_system.c

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

int
system(char *command)
{
    int status;
    pid_t childPid;

    switch (childPid = fork()) {
    case -1: /* Error */
        return -1;
    case 0: /* Child */
        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
        _exit(127);                     /* Failed exec */

    default: /* Parent */
        if (waitpid(childPid, &status, 0) == -1)
            return -1;
        else
            return status;
    }
```

```
}
```

procexec/simple_system.c

在 system()内部正确处理信号

给 system()的实现带来复杂性的是对信号的正确处理。

首先需要考虑的信号是 SIGCHLD。假设调用 system()的程序还直接创建了其他子进程，对 SIGCHLD 的信号处理器自身也执行了 wait()。在这种情况下，当由 system()所创建的子进程退出并产生 SIGCHLD 信号时，在 system()有机会调用 waitpid()之前，主程序的信号处理器程序可能会率先得以执行（收集子进程的状态）。这是竞争条件（race condition）的又一例证。这会产生两种不良后果。

- 调用程序会误以为其所创建的某个子进程终止了。
- system()函数却无法获取其所创建子进程的终止状态。

所以，system()在运行期间必须阻塞 SIGCHLD 信号。

其他需要关注的信号则是分别由终端的中断（interrupt）（通常为 Ctrl-C）和退出（quit）（通常为 Ctrl-\）符所产生的 SIGINT 和 SIGQUIT 信号。考虑执行如下调用的后果：

```
system("sleep 20");
```

此时此刻，会有 3 个进程在运行：执行调用程序的进程、一个 shell 进程，以及 sleep 进程。如图 27-2 所示。

为提高效率，如果赋予-c 选项的是一条简单命令，较之于管道（pipeline）或序列（sequence），一些 shell（包括 bash）会直接执行该命令，而不会再去创建一个子 shell。对于采用此类优化的 shell 而言，因为只有两个进程（调用进程和 sleep 进程），所以图 27-2 有失准确。不过，本节关于 system()如何处理信号的论述仍然适用。

图 27-2 中所示的所有进程构成终端前台进程组的一部分。（34.2 节将详细讨论进程组。）所以，在输入中断或退出符时，会将相应信号发送给所有 3 个进程。shell 在等待子进程期间会忽略 SIGINT 和 SIGQUIT 信号。不过，默认情况下这些信号会杀死调用程序与 sleep 进程。

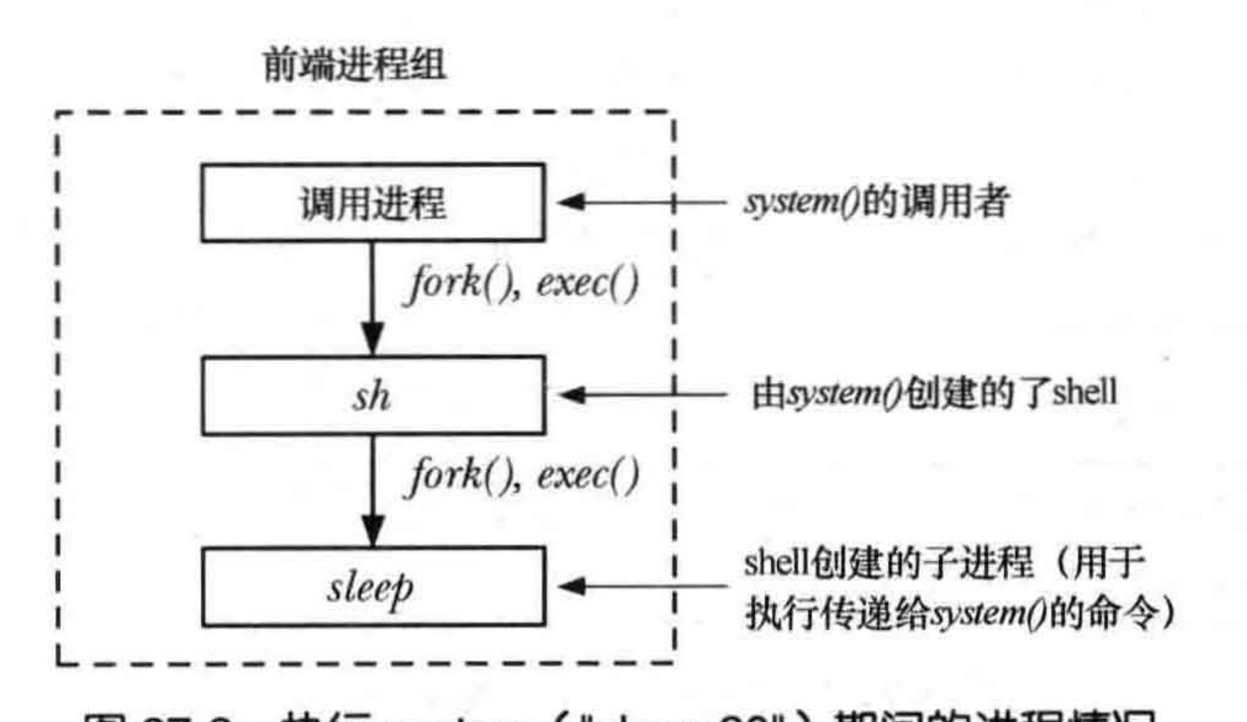

图 27-2：执行 system（"sleep 20"）期间的进程情况

调用进程和所执行的命令应当如何应对这些信号呢？SUSv3 的规定如下。

- 调用进程在执行命令期间应忽略 SIGINT 和 SIGQUIT 信号。
- 子进程对上述两信号的处理，如同调用进程调用了 fork()和 exec()一般，也就是说，将对已处理信号的处置重置为默认值，而对其他信号的处置则保持不变。

更多资源请访问稀酷客(www.ckook.com)

按照 SUSv3 所规范的方式来处理信号是最为合理的，其原因如下。

- 让所有的进程都对这些信号做出响应是没有意义的，用户会对应用的行为困惑不已。
- 与上述相类似，一面在执行命令的进程中忽略这些信号，而同时又在调用进程中按对它们的缺省处置来行事，这同样也说不通。用户借此可以将调用进程杀掉，同时放任其所执行的命令继续运行。这与实际情况也并不相符：当命令传递给 system()执行时，调用进程实际上已经放弃了控制权（即阻塞于 waitpid()调用中）。
- system()运行的可能是一个交互式应用，让此类应用响应终端产生的信号是合理的。

SUSv3 要求按上述方式来处理 SIGINT 和 SIGQUIT，但同时指出，对于暗中调用 system()来执行任务的程序，这一做法可能会产生不良后果。执行命令时如按下 Ctrl-C 或 Ctrl-\，将只会杀掉 system()的子进程，而应用程序会继续运行（用户并不希望如此）。以此方式调用 system()的程序应当检查 system()所返回的终止状态，一旦发现命令因信号而终止，应采取相应措施。

system()实现的改进版

程序清单 27-9 所示为遵循上述规则的 system()实现。关于该实现，需注意以下几点。

- 如前所述，当 command 为 NULL 指针时，若 shell 可用，则 system()应返回非 0 值；如不可用，则返回 0。要得出结论，唯一可靠的办法就是尝试运行 shell。程序这里的做法是：递归调用 system()去运行 shell 命令“:”并检查该递归调用的返回状态是否为 0 ①。“:”是一个 shell 内建命令，无所作为却总是返回成功。执行命令 exit 0 也可获得相同效果。（仅仅通过 access()来判断文件/bin/sh 存在与否，是否具有可执行权限的做法存在局限性。在 chroot()环境中，即使具有可执行权限的 shell 文件存在，如果与其进行动态链接的共享库无效，依然无法执行 shell。）
- 只有父进程（system()的调用者）才需要阻塞 SIGCHLD②，同时还需要忽略 SIGINT 和 SIGQUIT③。不过，必须在调用 fork()之前执行这些动作，因为如果在父进程的 fork()之后执行，将出现竞争条件。（假设：如果在父进程有机会阻塞 SIGCHLD 之前，子进程就退出了。）结果是，如同稍后所述，子进程必须取消对信号属性的这些变更。
- 父进程并未对 sigaction()以及 sigprocmask()调用进行错误检查②③⑨，这两者分别用于操作对信号的处置和信号掩码。这样做原因有二。其一，这些调用失手的可能性不大。实际上，只有指定参数有误时才会失败，而只要一开始调试就能搞定此类问题。其二，这里假定，较之于此类信号操控函数，调用者更关注 fork()或 waitpid()的成败与否。同理，在 system()尾部的信号处理操作前后，分别有代码来保存和恢复 errno，以便一旦 fork()或 waitpid()失败，调用者能查明原因。如果因信号操作失败而返回-1，那么调用者会误认为是 system()执行 command 失败所致。

> SUSv3 仅仅指出在创建子进程失败或无法获取子进程状态时，system()返回-1。并未提及 system()在处理信号失败时也会返回-1。

- 子进程中对于信号相关的系统调用也未执行错误检查④⑤。一方面，无法报告此类错误（_exit(127)是预留给执行 shell 时报告错误之用）；另一方面，这一失败也不会殃及 system()的调用者，二者分属于不同进程。
- 子进程刚从 fork()返回时，会将对 SIGINT 和 SIGQUIT 的处置置为 SIG_IGN（继承自父进程）。不过，如前所述，子进程处理这些信号时就如同 system()的调用者执行了 fork()和 exec()。fork()不会改变子进程对这些信号的处理方式。而 exec()则会将对

更多资源请访问稀酷客(www.ckook.com)

已处理信号的处置重置为默认值，但不改变对其他信号的处置（27.5 节）。因此，如果调用者对 SIGINT 和 SIGQUIT 的处置设置并非 SIG_IGN，那么子进程会将其置为 SIG_DFL④。

> 一些 system()实现反而会将子进程对 SIGINT 和 SIGQUIT 的处置重置为在调用者中生效的设置。这一做法的依据是，后续对 execl()的调用会自动将对这些已处理信号的处置重置为默认值。不过，如果调用者正在处理两个信号之一时，这可能会导致不希望的行为发生。在这种情况下，如果在调用 execl()之前的瞬间有信号送达子进程，那么在信号经由 sigprocmask()解除阻塞后，子进程还是会调用信号处理器程序。

- 子进程如果调用 execl()失败，就会以_exit()，而非 exit()来终止进程⑤。这是为了防止对子进程 stdio 缓冲区中的任何未写入数据进行刷新。
- 父进程必须使用 waitpid()来专候其所创建的特定子进程⑦。如果使用 wait()，不经意间可能会捕获到其他子进程的状态。
- 虽然 system()实现并未强制要求使用信号处理器程序，但调用程序还是可能会去创建它们，从而中断对 waitpid()的阻塞调用。SUSv3 明确要求在这种情况下必须重新等待。所以，如果发生 EINTR 错误⑦，则循环调用 waitpid()以期成功重启。如果是其他错误，则退出 waitpid()循环。

程序清单 27-9：system()的实现

procexec/system.c

```
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <errno.h>

int
system(const char *command)
{
    sigset_t blockMask, origMask;
    struct sigaction saIgnore, saOrigQuit, saOrigInt, saDefault;
    pid_t childPid;
    int status, savedErrno;

①   if (command == NULL)                /* Is a shell available? */
        return system(":") == 0;
    sigemptyset(&blockMask);            /* Block SIGCHLD */
    sigaddset(&blockMask, SIGCHLD);
②   sigprocmask(SIG_BLOCK, &blockMask, &origMask);

    saIgnore.sa_handler = SIG_IGN;      /* Ignore SIGINT and SIGQUIT */
    saIgnore.sa_flags = 0;
    sigemptyset(&saIgnore.sa_mask);
③   sigaction(SIGINT, &saIgnore, &saOrigInt);
    sigaction(SIGQUIT, &saIgnore, &saOrigQuit);

    switch (childPid = fork()) {
    case -1: /* fork() failed */
        status = -1;
        break;                  /* Carry on to reset signal attributes */
```

```
        case 0: /* Child: exec command */
            saDefault.sa_handler = SIG_DFL;
            saDefault.sa_flags = 0;
            sigemptyset(&saDefault.sa_mask);

④           if (saOrigInt.sa_handler != SIG_IGN)
                sigaction(SIGINT, &saDefault, NULL);
            if (saOrigQuit.sa_handler != SIG_IGN)
                sigaction(SIGQUIT, &saDefault, NULL);

⑤           sigprocmask(SIG_SETMASK, &origMask, NULL);

            execl("/bin/sh", "sh", "-c", command, (char *) NULL);
⑥           _exit(127);                     /* We could not exec the shell */

        default: /* Parent: wait for our child to terminate */
⑦           while (waitpid(childPid, &status, 0) == -1) {
                if (errno != EINTR) {       /* Error other than EINTR */
                    status = -1;
                    break;                  /* So exit loop */
                }
            }
            break;
        }

        /* Unblock SIGCHLD, restore dispositions of SIGINT and SIGQUIT */

⑧       savedErrno = errno;                 /* The following may change 'errno' */

⑨       sigprocmask(SIG_SETMASK, &origMask, NULL);
        sigaction(SIGINT, &saOrigInt, NULL);
        sigaction(SIGQUIT, &saOrigQuit, NULL);

⑩       errno = savedErrno;

        return status;
    }
```

procexec/system.c

关于 system()的更多细节

为保障应用程序的可移植性，应确保在将对 SIGCHLD 的处置置为 SIG_IGN 的情况下不去调用 system()，因为此时 waitpid()将无法获取子进程的状态。（忽略 SIGCHLD 会导致立即丢弃子进程状态，如 26.3.3 节所述。）

在一些 UNIX 实现中，如果在将对 SIGCHLD 的处置置为 SIG_IGN 的情况下调用 system()，system()的应对策略是：临时将其改为 SIG_DFL。只要在把对 SIGCHLD 的处置重置为 SIG_ IGN 时，UNIX 实现能够处理僵尸子进程（Linux 不在此列），这种方法就是可行的。（如果系统做不到这一点，按此方式实现 system()将产生如下不良后果：在调用者执行 system()期间，如果另一子进程终止了，那么该子进程将成为僵尸进程，且无法回收。）

对于一些 UNIX 实现（尤其是 Solaris），/bin/sh 并非标准的 POSIX shell。若希望确保执行标准 shell，则必须使用库函数 confstr()来获取配置变量_CS_PATH 的值。该值的风格与 PATH 相同，包含了标准系统工具的目录列表。可以将该列表赋给变量 PATH，随即调用 execlp()以执行标准 shell，具体如下：

更多资源请访问稀酷客(www.ckook.com)

```
char path[PATH_MAX];

if (confstr(_CS_PATH, path, PATH_MAX) == 0)
    _exit(127);
if (setenv("PATH", path, 1) == -1)
    _exit(127);
execlp("sh", "sh", "-c", command, (char *) NULL);
_exit(127);
```

27.8 总结

进程可使用 execve()以一新程序替换当前增长运行的程序。execve()的参数允许为新程序指定参数列表（argv）和环境列表。构建于 execve()之上，存在多种命名相似的函数，功能相同，但提供的接口不同。

所有的 exec()函数均可用于加载二进制的可执行文件或是执行解释器脚本。当进程执行脚本时，脚本解释器程序将替换进程当前执行的程序。脚本的起始行（以#!开头）指定了解释器的路径名，供识别解释器之用。如果没有这一起始行，那么只能通过 execlp()或 execvp()来执行脚本，并默认把 shell 作为脚本解释器。

本章还展示了如何组合使用 fork()、exec()、exit()和 wait()来实现 system()函数，该函数可用于执行任意 shell 命令。

更多的信息

请参考 24.6 节所列的更多信息来源。

27.9 练习

27-1. 如下 shell 会话的最后一条命令使用程序清单 27-3 程序来执行程序 xyz。结果如何？

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:./dir1:./dir2
$ ls -l dir1
total 8
-rw-r--r--   1 mtk      users        7860 Jun 13 11:55 xyz
$ ls -l dir2
total 28
-rwxr-xr-x   1 mtk      users       27452 Jun 13 11:55 xyz
$ ./t_execlp xyz
```

27-2. 试用 execve()实现 execlp()。需使用 stdarg(3) API 来处理 execlp()所提供的变长参数列表。还需要使用 malloc 函数库中函数为参数以及环境向量分配空间。最后，请注意，要检查特定目录下某个文件是否存在且可以执行，有一种简单方法：尝试执行该文件即可。

27-3. 如果赋予如下脚本可执行权限并以 exec()运行，输出结果如何？

```
#!/bin/cat -n
Hello world
```

27-4. 下列代码会有什么效果？在何种情况下会起作用？

```
childPid = fork();
if (childPid == -1)
    errExit("fork1");
```


更多资源请访问稀酷客(www.ckook.com)

```
if (childPid == 0) {    /* Child */
    switch (fork()) {
    case -1: errExit("fork2");

    case 0:             /* Grandchild */
        /* ----- Do real work here ----- */
        exit(EXIT_SUCCESS);             /* After doing real work */

    default:
        exit(EXIT_SUCCESS);             /* Make grandchild an orphan */
    }
}

/* Parent falls through to here */

if (waitpid(childPid, &status, 0) == -1)
    errExit("waitpid");

/* Parent carries on to do other things */
```

27-5. 运行如下程序时无输出。试问原因何在？

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world");
    execlp("sleep", "sleep", "0", (char *) NULL);
}
```

27-6. 假设父进程为信号 SIGCHLD 创建了一处理器程序，同时阻塞该信号。随后，其某一子进程退出，父进程接着执行 wait()以获取该子进程的状态。当父进程解除对 SIGCHLD 的阻塞时，会发生什么？编写一个程序来验证答案。这一结果与调用 system()函数的程序之间有什么关联？

更多资源请访问稀酷客(www.ckook.com)

第28章

详述进程创建和程序执行

本章对第 24 章到第 27 章的内容进行了拓展，涵盖进程创建和程序执行的多个主题。首先是进程记账（process accounting），这一内核特性会使系统在每个进程结束后记录一条账单信息。接着，会讨论 Linux 特有的系统调用 clone()，Linux 系统创建线程就有赖于这一底层 API。然后对 fork()、vfork()和 clone()的性能进行了比较。最后，本章就 fork()和 exec()对进程属性的影响做了总结。

28.1 进程记账

打开进程记账功能后，内核会在每个进程终止时将一条记账信息写入系统级的进程记账文件。这条账单记录包含了内核为该进程所维护的多种信息，包括终止状态以及进程消耗的 CPU 时间。借助于标准工具（sa(8) 对账单文件进行汇总，lastcomm(1)则就先前执行的命令列出相关信息）或是定制应用，可对记账文件进行分析。

> 内核 2.6.10 之前，内核会为基于 NPTL 线程实现所创建的每个线程单独记录一条进程记账信息。自内核 2.6.10 开始，只有当最后一个线程退出时才会为整个进程保存一条账单记录。至于更老的 LinuxThread 线程实现，则会为每个线程单独记录一条进程记账信息。

从历史上看，进程记账主要用于在多用户 UNIX 系统上针对用户所消耗的系统资源进行计费。不过，如果进程的信息并未由其父进程进行监控和报告，那么就可以使用进程记账来获取。

虽然大部分 UNIX 实现都支持进程记账功能，但 SUSv3 并未对其进行规范。账单记录的格式、记账文件的位置也随系统实现的不同而多少存在差别。本节所述是针对 Linux 系统的细节，但会在论述过程中点出其与其他 Unix 系统的差异。

> Linux 系统的进程记账功能属于可选内核组件，可以通过 CONFIG_BSD_PROCESS_ACCT 选项进行配置。

更多资源请访问稀酷客(www.ckook.com)

打开和关闭进程记账功能

特权进程可利用系统调用 acct()来打开和关闭进程记账功能。应用程序很少使用这一系统调用。一般会将相应命令置于系统启动脚本中，在系统每次重启时开启进程记账功能。

```
#define _BSD_SOURCE
#include <unistd.h>

int acct(const char *acctfile);
```

Returns 0 on success, or -1 on error

为了打开进程账单功能，需要在参数 acctfile 中指定一个现有常规文件的路径名。记账文件通常的路径名是/var/log/pacct 或/usr/account/pacct。若想关闭进程记账功能，则指定 acctfile 为 NULL 即可。

程序清单 28-1 中程序使用 acct()来开关进程的记账功能。该程序的作用类似于 shell 命令 accton(8)。

程序清单 28-1：打开和关闭进程记账功能

procexec/acct_on.c

```
#define _BSD_SOURCE
#include <unistd.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc > 2 || (argc > 1 && strcmp(argv[1], "--help") == 0))
        usageErr("%s [file]\n");
    if (acct(argv[1]) == -1)
        errExit("acct");

    printf("Process accounting %s\n",
            (argv[1] == NULL) ? "disabled" : "enabled");
    exit(EXIT_SUCCESS);
}
```

procexec/acct_on.c

进程账单记录

一旦打开进程记账功能，每当一进程终止时，就会有一条 acct 记录写入记账文件。acct 结构定义于头文件<sys/acct.h>中，具体如下：

```
typedef u_int16_t comp_t;  /* See text */

struct acct {
    char      ac_flag;     /* Accounting flags (see text) */
    u_int16_t ac_uid;      /* User ID of process */
    u_int16_t ac_gid;      /* Group ID of process */
    u_int16_t ac_tty;      /* Controlling terminal for process (may be
                              0 if none, e.g., for a daemon) */
    u_int32_t ac_btime;    /* Start time (time_t; seconds since the Epoch) */
    comp_t    ac_utime;    /* User CPU time (clock ticks) */
    comp_t    ac_stime;    /* System CPU time (clock ticks) */
```


更多资源请访问稀酷客(www.ckook.com)

```
    comp_t    ac_etime;    /* Elapsed (real) time (clock ticks) */
    comp_t    ac_mem;      /* Average memory usage (kilobytes) */
    comp_t    ac_io;       /* Bytes transferred by read(2) and write(2)
                              (unused) */
    comp_t    ac_rw;       /* Blocks read/written (unused) */
    comp_t    ac_minflt;   /* Minor page faults (Linux-specific) */
    comp_t    ac_majflt;   /* Major page faults (Linux-specific) */
    comp_t    ac_swaps;    /* Number of swaps (unused; Linux-specific) */
    u_int32_t ac_exitcode; /* Process termination status */
#define ACCT_COMM 16
    char      ac_comm[ACCT_COMM+1];
                           /* (Null-terminated) command name
                              (basename of last execed file) */
    char      ac_pad[10];  /* Padding (reserved for future use) */
};
```

关于 acct 结构需要注意以下几点。

- 数据类型 u_int16_t 和 u_int32_t 分别是 16 位和 32 位的无符号整数类型。
- ac_flag 字段（field）是为进程记录多种事件（event）的位掩码（bit mask）。表 28-1 展示了在该字段中可能出现的位。如表中所示，并非所有的 UNIX 实现都支持这些位。另有少数实现为该字段还提供了一些附加的位。
- ac_comm 字段记录了该进程最后执行的命令（程序文件）名称。内核会在每次调用 execve()时记录该值。一些 UNIX 实现将该字段的大小限制在 8 个字节以内。
- 类型 comp_t 是一种浮点型（floating-point）数字。有时也将该类型值称为压缩时钟周期（compressed clock tick）。该浮点值由 3 位（bit）以 8 为底的指数以及 13 位（bit）小数组成，指数用来表示值范围在 $8^0=1 \sim 8^7$（2097152）之间的因子。例如，尾数为 125，指数部分为 1 就表示值为 1000。程序清单 28-2 中定义的函数（comptToLL()）可以将该类型转换为 long long。因为在 x86-32 架构下的系统中，用于表示无符号长整型的 32 位数并不足以保存 comp_t 型的最大值：$(2^{13}-1) \times 8^7$。
- 3 个定义为 comp_t 型的时间字段其度量单位为系统时钟周期。要将它们转换成秒，必须除以 sysconf(_SC_CLK_TCK)的返回值。
- ac_exitcode 字段保存着进程的退出状态（如 26.1.3 节所述）。其他大多数 UNIX 实现则提供了一个名为 ac_stat 的单字节字段来代替 ac_exitcode，其中仅记录了杀死进程的信号值（如果进程为信号所杀）和一个标志位，用于标识是否因该信号而导致进程转储核心（dump core）。二者在源于 BSD 的实现中均未提供。

表 28-1：进程账单记录中 ac_flag 字段各位的值

位	说　明
AFORK	由 fork()创建的进程，终止前并未调用 exec()
ASU	拥有超级用户特权的进程
AXSIG	进程因信号而终止（有些实现未支持）
ACORE	进程产生了核心转储（有些实现未支持）

因为只在进程终止时才记录账单信息，所以对这些记录的排序也是按照进程的终止时间

更多资源请访问稀酷客(www.ckook.com)

（并未写入记录），而非启动时间（ac_btime）。

如果系统崩溃，也不会为当前运行的进程记录任何记账信息。

由于向记账文件中写入信息可能会加速对磁盘空间的消耗，为了对进程记账行为加以控制，Linux 系统提供了名为/proc/sys/kernel/acct 的虚拟文件。此文件包含 3 个值，按顺序分别定义了如下参数：高水位（high-water）、低水位（low-water）和频率（frequency）。3 个参数通常的默认值为 4、2 和 30。如果开启进程记账特性且磁盘空闲空间低于低水位（low-water）百分比，将暂停记账。如果磁盘空闲空间升至高水位百分比之上，则恢复记账。频率值则规定了两次检查空闲磁盘空间占比之间的间隔时间（以秒为单位）。

示例程序

程序清单 28-2 中程序显示了某进程记账文件记录中特定字段的信息。以下 shell 会话演示了对该程序的使用。首先新建一个空的进程记账文件，同时开启进程记账功能。

```
$ su                                  Need privilege to enable process accounting
Password:
# touch pacct
# ./acct_on pacct                     This process will be first entry in accounting file
Process accounting enabled
# exit                                Cease being superuser
```

从开启进程记账功能到现在，已经有 3 个进程退出，分别执行了 acct_on、su 和 bash 程序。进程 bash 由 su 启动，负责运行特权级 shell 会话。

接着运行一系列命令，借此向记账文件加入更多记录：

```
$ sleep 15 &
[1] 18063
$ ulimit -c unlimited                 Allow core dumps (shell built-in)
$ cat                                 Create a process
Type Control-\ (generates SIGQUIT, signal 3) to kill cat process
Quit (core dumped)
$
Press Enter to see shell notification of completion of sleep before next shell prompt
[1]+  Done            sleep 15
$ grep xxx badfile                    grep fails with status of 2
grep: badfile: No such file or directory
$ echo $?                             The shell obtained status of grep (shell built-in)
2
```

下面两个命令执行的是前面章节中展示过的两个程序（程序清单 27-1 和程序清单 24-1）。第一条命令运行的程序执行了/bin/echo，因此，写入账单记录中的命令名是 echo。第二条命令创建了一个子进程，该子进程并未调用 exec()。

```
$ ./t_execve /bin/echo
hello world goodbye
$ ./t_fork
PID=18350 (child) idata=333 istack=666
PID=18349 (parent) idata=111 istack=222
```

最后，运行程序清单 28-2 中程序来查看记账文件的内容。

```
$ ./acct_view pacct
command  flags   term.  user     start time            CPU   elapsed
                status                                 time    time
acct_on   -S--      0   root     2010-07-23 17:19:05   0.00    0.00
```


更多资源请访问稀酷客(www.ckook.com)

```
bash      ----      0   root     2010-07-23 17:18:55   0.02   21.10
su        -S--      0   root     2010-07-23 17:18:51   0.01   24.94
cat       --XC   0x83   mtk      2010-07-23 17:19:55   0.00    1.72
sleep     ----      0   mtk      2010-07-23 17:19:42   0.00   15.01
grep      ----  0x200   mtk      2010-07-23 17:20:12   0.00    0.00
echo      ----      0   mtk      2010-07-23 17:21:15   0.01    0.01
t_fork    F---      0   mtk      2010-07-23 17:21:36   0.00    0.00
t_fork    ----      0   mtk      2010-07-23 17:21:36   0.00    3.01
```

输出中的每行都对应于 shell 会话所创建的一个进程。ulimit 和 echo 都是 shell 的内建命令，所以并不会创建新进程。注意，记账文件中 sleep 之所以出现在 cat 之后，是因为 sleep 在 cat 之后才终止。

大部分输出的含义均不言而喻。flags 列中的各个字母表示每条记录对哪些 ac_flag 位进行了置位（参考表 28-1）。至于应如何解释 term.status 列中的终止状态，26.1.3 节有相关描述。

程序清单 28-2：显示进程记账文件中的数据

procexec/acct_view.c

```
#include <fcntl.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/acct.h>
#include <limits.h>
#include "ugid_functions.h"             /* Declaration of userNameFromId() */
#include "tlpi_hdr.h"

#define TIME_BUF_SIZE 100

static long long                /* Convert comp_t value into long long */
comptToLL(comp_t ct)
{
    const int EXP_SIZE = 3;             /* 3-bit, base-8 exponent */
    const int MANTISSA_SIZE = 13;       /* Followed by 13-bit mantissa */
    const int MANTISSA_MASK = (1 << MANTISSA_SIZE) - 1;
    long long mantissa, exp;

    mantissa = ct & MANTISSA_MASK;
    exp = (ct >> MANTISSA_SIZE) & ((1 << EXP_SIZE) - 1);
    return mantissa << (exp * 3);       /* Power of 8 = left shift 3 bits */
}

int
main(int argc, char *argv[])
{
    int acctFile;
    struct acct ac;
    ssize_t numRead;
    char *s;
    char timeBuf[TIME_BUF_SIZE];
    struct tm *loc;
    time_t t;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    acctFile = open(argv[1], O_RDONLY);
    if (acctFile == -1)
        errExit("open");
```

```
    printf("command  flags   term.  user     "
            "start time            CPU   elapsed\n");
    printf("                 status           "
            "                      time    time\n");

    while ((numRead = read(acctFile, &ac, sizeof(struct acct))) > 0) {
        if (numRead != sizeof(struct acct))
            fatal("partial read");

        printf("%-8.8s  ", ac.ac_comm);

        printf("%c", (ac.ac_flag & AFORK) ? 'F' : '-') ;
        printf("%c", (ac.ac_flag & ASU)   ? 'S' : '-') ;
        printf("%c", (ac.ac_flag & AXSIG) ? 'X' : '-') ;
        printf("%c", (ac.ac_flag & ACORE) ? 'C' : '-') ;

#ifdef __linux__
        printf(" %#6lx   ", (unsigned long) ac.ac_exitcode);
#else   /* Many other implementations provide ac_stat instead */
        printf(" %#6lx   ", (unsigned long) ac.ac_stat);
#endif

        s = userNameFromId(ac.ac_uid);
        printf("%-8.8s ", (s == NULL) ? "???" : s);

        t = ac.ac_btime;
        loc = localtime(&t);
        if (loc == NULL) {
            printf("???Unknown time???  ");
        } else {
            strftime(timeBuf, TIME_BUF_SIZE, "%Y-%m-%d %T ", loc);
            printf("%s ", timeBuf);
        }

        printf("%5.2f %7.2f ", (double) (comptToLL(ac.ac_utime) +
                    comptToLL(ac.ac_stime)) / sysconf(_SC_CLK_TCK),
                (double) comptToLL(ac.ac_etime) / sysconf(_SC_CLK_TCK));
        printf("\n");
    }

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);
}
```

procexec/acct_view.c

进程记账文件格式（版本 3）

从内核 2.6.8 开始，Linux 引入了另一版本的进程记账文件以备选用，意在突破传统记账文件的一些限制。若有意使用这种被称为版本 3 的备选格式，需要在编译内核前打开内核配置选项 CONFIG_BSD_PROCESS_ACCT_V3。

使用版本 3，操作进程记账时唯一的差别在于，写入记账文件的记录格式不同。新格式的定义如下：

```
struct acct_v3 {
    char      ac_flag;        /* Accounting flags */
```

```
char      ac_version;     /* Accounting version (3) */
u_int16_t ac_tty;         /* Controlling terminal for process */
u_int32_t ac_exitcode;    /* Process termination status */
u_int32_t ac_uid;         /* 32-bit user ID of process */
u_int32_t ac_gid;         /* 32-bit group ID of process */
u_int32_t ac_pid;         /* Process ID */
u_int32_t ac_ppid;        /* Parent process ID */
u_int32_t ac_btime;       /* Start time (time_t) */
float     ac_etime;       /* Elapsed (real) time (clock ticks) */
comp_t    ac_utime;       /* User CPU time (clock ticks) */
comp_t    ac_stime;       /* System CPU time (clock ticks) */
comp_t    ac_mem;         /* Average memory usage (kilobytes) */
comp_t    ac_io;          /* Bytes read/written (unused) */
comp_t    ac_rw;          /* Blocks read/written (unused) */
comp_t    ac_minflt;      /* Minor page faults */
comp_t    ac_majflt;      /* Major page faults */
comp_t    ac_swaps;       /* Number of swaps (unused; Linux-specific) */
#define ACCT_COMM 16
    char      ac_comm[ACCT_COMM];   /* Command name */
};
```

以下是 acct_v3 结构与传统 Linux acct 结构的主要差别。

- 增加 ac_version 字段。该字段包含本类型账单记录的版本号。对于 acct_v3 来说，总是等于 3。
- 增加 ac_pid 和 ac_ppid 字段，分别包含终止进程的进程 ID 及其父进程 ID。
- 字段 ac_uid 和 ac_gid 从 16 位扩展至 32 位，旨在容纳 Linux 2.4 所引入的 32 位用户 ID 和组 ID。（传统 acct 文件无法正确记录大数值的用户和组 ID。）
- 将 ac_etime 字段类型从 comp_t 改为 float，意在能够记录更长的逝去时间。

随本书发布的源代码文件 procexec/acct_v3_view.c 中提供了程序清单 28-2 中程序基于 v3 格式的新版本。

28.2 系统调用 clone()

类似于 fork()和 vfork()，Linux 特有的系统调用 clone()也能创建一个新进程。与前两者不同的是，后者在进程创建期间对步骤的控制更为精准。clone()主要用于线程库的实现。由于 clone()有损于程序的可移植性，故而应避免在应用程序中直接使用。之所以在这里讨论 clone()，意在对第 29 章至第 33 章所论述的 POSIX 线程有所铺垫，同时也利于进一步阐明 fork()和 vfork()的操作。

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Returns process ID of child on success, or -1 on error

如同 fork()，由 clone()创建的新进程几近于父进程的翻版。

但与 fork()不同的是，克隆生成的子进程继续运行时不以调用处为起点，转而去调用以参数 func 所指定的函数，func 又称为子函数（child function）。调用子函数时的参数由 func_arg

更多资源请访问稀酷客(www.ckook.com)

指定。经过适当转换，子函数可对该参数的含义自由解读，例如，可以作为整型值（int），也可视为指向结构的指针。（之所以可以作为指针处理，是因为克隆产生的子进程对调用进程的内存既可获取，也可共享。）

对于内核而言，fork()、vfork()以及 clone()最终均由同一函数实现（kernel/fork.c 中的 do_fork()）。在这一层次上，clone 与 fork 更为接近：sys_clone()并没有 func 和 func_arg 参数，且调用后 sys_clone()在子进程中返回的方式也与 fork()相同。正文所述的 clone()是由 glibc 为 sys_clone()提供的封装函数。（对该函数的定义位于 glibc 针对特定架构的汇编源码中，例如 sysdeps/unix/sysv/linux/i386/clone.S。）sys_clone()在子进程中返回之后，由 clone()发起对 func 函数的调用。

当函数 func 返回（此时其返回值即为进程的退出状态）或是调用 exit()（或_exit()）之后，克隆产生的子进程就会终止。照例，父进程可以通过 wait()一类函数来等待克隆子进程。

因为克隆产生的子进程可能（类似 vfork()）共享父进程的内存，所以它不能使用父进程的栈。相反，调用者必须分配一块大小适中的内存空间供子进程的栈使用，同时将这块内存的指针置于参数 child_stack 中。在大多数硬件架构中，栈空间的增长方向是向下的，所以参数 child_stack 应当指向所分配内存块的高端。

栈增长方向对架构的依赖是 clone()设计的一处缺陷。Interl IA-64 架构就提供了一款经过改善的克隆 API，称为 clone2()。该系统调用对子进程栈范围的定义方式不依赖于栈的增长方向，只需要提供栈的起始地址以及大小即可。详情请参阅手册页。

函数 clone()的参数 flags 服务于双重目的。首先，其低字节中存放着子进程的终止信号（terminateion signal），子进程退出时其父进程将收到这一信号。（如果克隆产生的子进程因信号而终止，父进程依然会收到 SIGCHLD 信号。）该字节也可能为 0，这时将不会产生任何信号。（借助于 Linux 特有的/proc/PID/stat 文件，可以判定任何进程的终止信号，详情请参阅 proc(5)手册页。）

对于 fork()和 vfork()而言，就无从选择终止信号，只能是 SIGCHLD。

参数 flags 的剩余字节则存放了位掩码，用于控制 clone()的操作。表 28-2 对这些位掩码值进行了总结，28.2.1 节会进一步加以说明。

表 28-2：clone()参数 flags 的位掩码值

标　志	设置后的效果
CLONE_CHILD_CLEARTID	当子进程调用 exec()或_exit()时，清除 ctid（从版本 2.6 开始）
CLONE_CHILD_SETTID	将子进程的线程 ID 写入 ctid（从 2.6 版本开始）
CLONE_FILES	父、子进程共享打开文件描述符表
CLONE_FS	父、子进程共享与文件系统相关的属性
CLONE_IO	子进程共享父进程的 I/O 上下文环境（从 2.6.25 版本开始）
CLONE_NEWIPC	子进程获得新的 System V IPC 命名空间（从 2.6.19 开始）
CLONE_NEWNET	子进程获得新的网络命名空间（从 2.4.24 版本开始）

更多资源请访问稀酷客(www.ckook.com)

续表

标　志	设置后的效果
CLONE_NEWNS	子进程获得父进程挂载（mount）命名空间的副本（从 2.4.19 版本开始）
CLONE_NEWPID	子进程获得新的进程 ID 命名空间（从 2.6.23 版本开始）
CLONE_NEWUSER	子进程获得新的用户 ID 命名空间（从 2.6.23 版本开始）
CLONE_NEWUTS	子进程获得新的 UTS（utsname()）命名空间（从 2.6.19 版本开始）
CLONE_PARENT	将子进程的父进程置为调用者的父进程（从 2.4 版本开始）
CLONE_PARENT_SETTID	将子进程的线程 ID 写入 ptid（从 2.6 版本开始）
CLONE_PID	标志已废止，仅用于系统启动进程（直至 2.4 版本为止）
CLONE_PTRACE	如果正在跟踪父进程，那么子进程也照此办理
CLONE_SETTLS	tls 描述子进程的线程本地存储（从 2.6 开始）
CLONE_SIGHAND	父、子进程共享对信号的处置设置
CLONE_SYSVSEM	父、子进程共享信号量还原（undo）值（从 2.6 版本开始）
CLONE_THREAD	将子进程置于父进程所属的线程组中（从 2.4 开始）
CLONE_UNTRACED	不强制对子进程设置 CLONE_PTRACE（从 2.6 版本开始）
CLONE_VFORK	挂起父进程直至子进程调用 exec()或_exit()
CLONE_VM	父、子进程共享虚拟内存

clone()的余下参数分别是：ptid、tls 和 ctid。这些参数与线程的实现相关，尤其是在针对线程 ID 以及线程本地存储的使用方面。28.2.1 节在说明 flags 位掩码值时，会论及这些参数的使用。（在 Linux 2.4 及其之前的版本中，clone()尚未提供上述 3 个参数。直到 Linux 2.6，为了支持 NPTL POSIX 的线程实现，才特意加入了这些参数。）

示例程序

程序清单 28-3 是使用 clone()创建子进程的一个简单例子。主程序所做工作如下。

- 打开一个文件描述符（打开设备/dev/null），在子进程中将其关闭②。
- 若提供有命令行参数，则将 clone()的 flags 参数置为 CLONE_FILES③，以便父、子进程共享同一文件描述符表。若没有提供命令行参数，则将 flags 置 0。
- 分配一个栈供子进程使用④。
- 若 CHILD_SIG 非 0 且不等于 SIGCHLD⑤，则忽略之，以防该信号将子进程终止。之所以未忽略 SIGCHLD，是因为那将导致无法收集子进程的退出状态。
- 调用 clone()创建子进程⑥。第三个参数（位掩码）包含了终止信号。第四个参数（func_arg）指定了之前打开的文件描述符（在②处）。
- 等待子进程终止⑦。
- 尝试调用 write()，以检查文件描述符（在②处打开）是否仍处于打开状态⑧。程序报告 write()操作是否成功。

克隆产生的子进程从 childFunc()处开始执行，该函数（利用参数 arg）接收由主程序打开的文件描述符（在②处）。子进程关闭文件描述符并调用 return 以终止①。

更多资源请访问稀酷客(www.ckook.com)

程序清单 28-3：使用 clone()创建子进程

procexec/t_clone.c

```
#define _GNU_SOURCE
#include <signal.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sched.h>
#include "tlpi_hdr.h"

#ifndef CHILD_SIG
#define CHILD_SIG SIGUSR1       /* Signal to be generated on termination
                                   of cloned child */
#endif

static int                      /* Startup function for cloned child */
childFunc(void *arg)
{
①   if (close(*((int *) arg)) == -1)
        errExit("close");

    return 0;                           /* Child terminates now */
}

int
main(int argc, char *argv[])
{
    const int STACK_SIZE = 65536;       /* Stack size for cloned child */
    char *stack;                        /* Start of stack buffer */
    char *stackTop;                     /* End of stack buffer */
    int s, fd, flags;

②   fd = open("/dev/null", O_RDWR);     /* Child will close this fd */
    if (fd == -1)
        errExit("open");
    /* If argc > 1, child shares file descriptor table with parent */

③   flags = (argc > 1) ? CLONE_FILES : 0;

    /* Allocate stack for child */

④   stack = malloc(STACK_SIZE);
    if (stack == NULL)
        errExit("malloc");
    stackTop = stack + STACK_SIZE;      /* Assume stack grows downward */

    /* Ignore CHILD_SIG, in case it is a signal whose default is to
       terminate the process; but don't ignore SIGCHLD (which is ignored
       by default), since that would prevent the creation of a zombie. */

⑤   if (CHILD_SIG != 0 && CHILD_SIG != SIGCHLD)
        if (signal(CHILD_SIG, SIG_IGN) == SIG_ERR)
            errExit("signal");

    /* Create child; child commences execution in childFunc() */

⑥   if (clone(childFunc, stackTop, flags | CHILD_SIG, (void *) &fd) == -1)
        errExit("clone");
```

```
    /* Parent falls through to here. Wait for child; __WCLONE is
       needed for child notifying with signal other than SIGCHLD. */

⑦   if (waitpid(-1, NULL, (CHILD_SIG != SIGCHLD) ? __WCLONE : 0) == -1)
        errExit("waitpid");
    printf("child has terminated\n");

    /* Did close() of file descriptor in child affect parent? */

⑧   s = write(fd, "x", 1);
    if (s == -1 && errno == EBADF)
        printf("file descriptor %d has been closed\n", fd);
    else if (s == -1)
        printf("write() on file descriptor %d failed "
                "unexpectedly (%s)\n", fd, strerror(errno));
    else
        printf("write() on file descriptor %d succeeded\n", fd);

    exit(EXIT_SUCCESS);
}
```

procexec/t_clone.c

运行程序清单 28-3 中程序，没有命令行参数时输出如下：

```
$ ./t_clone                                  Doesn't use CLONE_FILES
child has terminated
write() on file descriptor 3 succeeded       Child's close() did not affect parent
```

带有命令行参数运行程序时，两个进程将共享文件描述符表：

```
$ ./t_clone x                                Uses CLONE_FILES
child has terminated
file descriptor 3 has been closed            Child's close() affected parent
```

随本书发布的源代码文件 procexec/demo_clone.c 提供一个更为复杂的 clone()用例。

28.2.1 clone()的 flags 参数

clone()的 flags 参数是各种位掩码的组合（“或”操作），下面将对它们一一说明。讲述时并未按字母顺序展开，而是着眼于促进对概念理解，从实现 POSIX 线程所使用的标志开始。从线程实现的角度来看，下文多次出现的“进程”一词都可用“线程”替代。

这里需要指出，某种意义上，对术语“线程”和“进程”的区分不过是在玩弄文字游戏而已。引入术语“内核调度实体（KSE，kernel scheduling entity）”（某些教科书以之来指代内核调度器所处理的对象）的概念对解释这一点会有所助益。实际上，线程和进程都是 KSE，只是与其他 KSE 之间对属性（虚拟内存、打开文件描述符、对信号的处置、进程 ID 等）的共享程度不同。针对线程间属性共享的方案不少，POSIX 线程规范只是其中之一。

在下面的说明中，有时会提及 Linux 平台对 POSIX 线程的两种主要实现：年长的 LinuxThreads，以及较为年轻的 NPTL。关于这两种实现的更多细节可以在 33.5 节找到。

从内核 2.6.16 开始，Linux 提供了新的系统调用 unshare()，由 clone()（或 fork()、vfork()）创建的子进程利用该调用可以撤销对某些属性的共享（即反转一些 clone() flags 位的效果）。详细情况请参考 unshare(2)手册页。

更多资源请访问稀酷客(www.ckook.com)

共享文件描述符表：CLONE_FILES

如果指定了 CLONE_FILES 标志，父、子进程会共享同一个打开文件描述符表。也就是说，无论哪个进程对文件描述符的分配和释放（open()、close()、dup()、pipe()、socket()等），都会影响到另一进程。如果未设置 CLONE_FILES，那么也就不会共享文件描述符表，子进程获取的是父进程调用 clone()时文件描述符表的一份拷贝。这些描述符副本与其父进程中的相应描述符均指向相同的打开文件（和 fork()和 vfork()的情况一样）。

POSIX 线程规范要求进程中的所有线程共享相同的打开文件描述符。

共享与文件系统相关的信息：CLONE_FS

如果指定了 CLONE_FS 标志，那么父、子进程将共享与文件系统相关的信息（file system-related information）：权限掩码（umask）、根目录以及当前工作目录。也就是说，无论在哪个进程中调用 umask ()、chdir()或者 chroot()，都将影响到另一个进程。如果未设置 CLONE_ FS，那么父、子进程对此类信息则会各持一份（与 fork()和 vfork()的情况相同）。

POSIX 线程规范要求实现 CLONE_FS 标志所提供的属性共享。

共享对信号的处置设置：CLONE_SIGHAND

如果设置了 CLONE_SIGHAND，那么父、子进程将共享同一个信号处置表。无论在哪个进程中调用 sigaction()或 signal()来改变对信号处置的设置，都会影响其他进程对信号的处置。若未设置 CLONE_SIGHAND，则不共享对信号的处置设置，子进程只是获取父进程信号处置表的一份副本（如同 fork()和 vfork()）。CLONE_SIGHAND 不会影响到进程的信号掩码以及对挂起（pending）信号的设置，父子进程的此类设置是绝不相同的。从 Linux 2.6 开始，如果设置了 CLONE_SIGHAND，就必须同时设置 CLONE_VM。

POSIX 线程规范要求共享对信号的处置设置。

共享父进程的虚拟内存：CLONE_VM

如果设置了 CLONE_VM 标志，父、子进程会共享同一份虚拟内存页（如同 vfork()）。无论哪个进程更新了内存，或是调用了 mmap()、munmap()，另一进程同样会观察到这些变化。如果未设置 CLONE_VM，那么子进程得到的是对父进程虚拟内存的拷贝（如同 fork()）。

共享同一虚拟内存是线程的关键属性之一，POSIX 线程标准对此也有要求。

线程组：CLONE_THREAD

若设置了 CLONE_THREAD，则会将子进程置于父进程的线程组中。如果未设置该标志，那么会将子进程置于新的线程组中。

POSIX 标准规定，进程的所有线程共享同一进程 ID（即每个线程调用 getpid()都应返回相同值），Linux 从 2.4 版本开始引入了线程组（threads group），以满足这一需求。如图 28-1 所示，线程组就是共享同一线程组标识（TGID）（thread group identifier）的一组 KSE。在对 CLONE_THREAD 的后续讨论中，会将 KSE 视同线程看待。

始于 Linux2.4，getpid()所返回的就是调用者的 TGID。换言之，TGID 和进程 ID 是一回事。

在 2.2 以及更早的 Linux 系统中，对 clone()的实现并不支持 CLONE_THREAD。相反，

更多资源请访问稀酷客(www.ckook.com)

LinuxThreads 曾将 POSIX 线程实现为共享了多种属性（例如，虚拟内存）、进程 ID 又各不相同的进程。考虑到兼容性因素，即便是在当前的 Linux 内核中，LinuxThreads 实现也未提供 CLONE_THREAD，因为按此方式实现的线程就可以继续拥有不同的进程 ID。

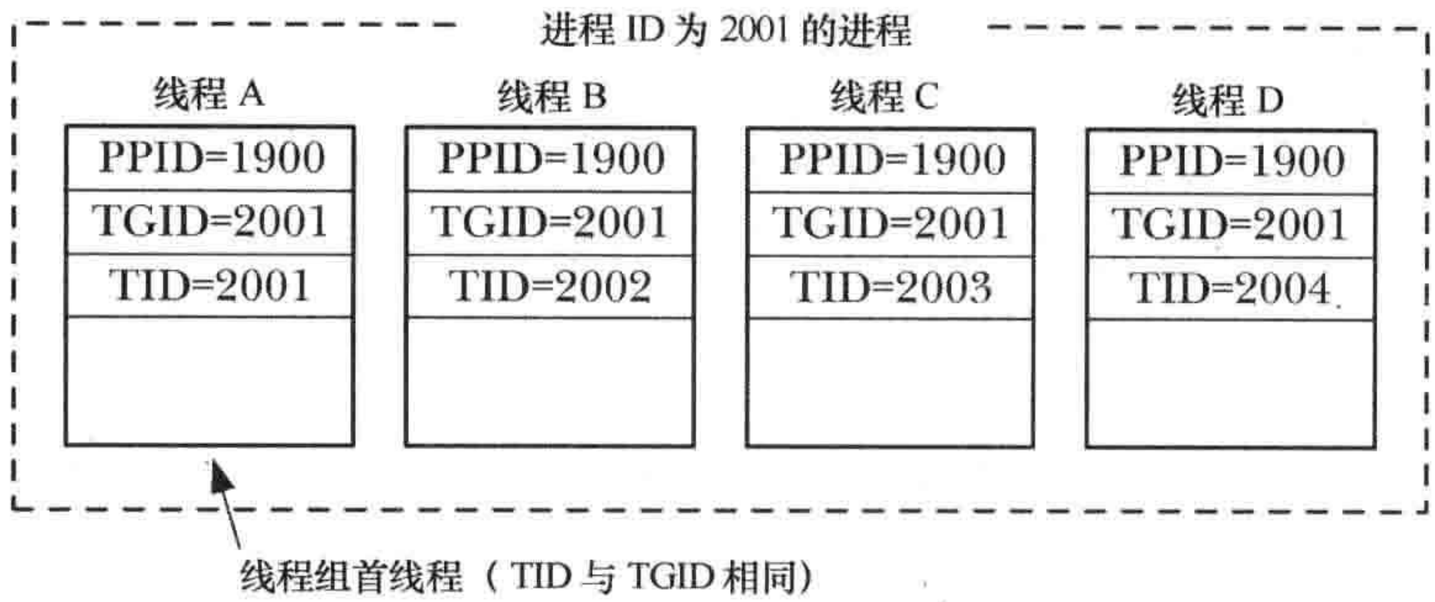

图 28-1：包含 4 个线程的线程组

一个线程组内的每个线程都拥有一个唯一的线程标识符（thread identifier，TID），用以标识自身。Linux 2.4 提供了一个新的系统调用 gettid()，线程可通过该调用来获取自己的线程 ID（与线程调用 clone()时的返回值相同）。线程 ID 与进程 ID 都使用相同的数据类型 pid_t 来表示。线程 ID 在整个系统中是唯一的，且除了线程担当进程中线程组首线程的情况之外，内核能够保证系统中不会出现线程 ID 与进程 ID 相同的情况。

线程组中首个线程的线程 ID 与其线程组 ID 相同，也将该线程称之为线程组首线程（thread group leader）。

此处讨论的线程 ID 与 POSIX 线程所使用的线程 ID（以数据类型 pthread_t 表示）不同。后者由 POSIX 线程实现（在用户空间）自行生成并维护。

线程组中的所有线程拥有同一父进程 ID，即与线程组首线程 ID 相同。仅当线程组中的所有线程都终止后，其父进程才会收到 SIGCHLD 信号（或其他终止信号）。这些行为符合 POSIX 线程规范的要求。

当一个设置了 CLONE_THREAD 的线程终止时，并没有信号会发送给该线程的创建者（即调用 clone()创建终止线程的线程）。相应的，也不可能调用 wait()（或类似函数）来等待一个以 CLONE_THREAD 标志创建的线程。这与 POSIX 的要求一致。POSIX 线程与进程不同，不能使用 wait()等待，相反，必须调用 pthread_join()来加入。为检测以 CLONE_THREAD 标志创建的线程是否终止，需要使用一种特殊的同步原语——futex（参考下文对 CLONE_PARENT_SETTID 标志的讨论）。

如果一个线程组中的任一线程调用了 exec()，那么除了首线程之外的其他线程都会终止（这一行为也符合 POSIX 线程规范的要求），新进程将在首线程中执行。换言之，新程序中的 gettid()调用将会返回首线程的线程 ID。调用 exec()期间，会将该进程发送给其父进程的终止信号重置为 SIGCHLD。

如果线程组中的某个线程调用 fork()或 vfork()创建了子进程，那么组中的任何线程都可使用 wait()或类似函数来监控该子进程。

从 Linux2.6 开始，如果设置了 CLONE_THREAD，同时也必须设置 CLONE_SIGHAND。这也与 POSIX 线程标准的深入要求相契合，详细内容可参考 33.2 节关于 POSIX 线程与信号交互的相关讨论。（内核针对 CLONE_THREAD 线程组的信号处理对应于 POSIX 标准对进程中线程如何处理信号的规范。）

更多资源请访问稀酷客(www.ckook.com)

线程库支持：CLONE_PARENT_SETTID、CLONE_CHILD_SETTID 和 CLONE_CHILD_CLEARTID

为实现 POSIX 线程，Linux 2.6 提供了对 CLONE_PARENT_SETTID、CLONE_CHILD_SETTID 和 CLONE_CHILD_CLEARTID 的支持。这些标志会影响 clone()对参数 ptid 和 ctid 的处理。NPTL 的线程实现使用了 CLONE_CHILD_SETTID 和 CLONE_CHILD_CLEARTID。

如果设置了 CLONE_PARENT_SETTID，内核会将子线程的线程 ID 写入 ptid 所指向的位置。在对父进程的内存进行复制之前，会将线程 ID 复制到 ptid 所指位置。这也意味着，即使没有设置 CLONE_VM，父、子进程均能在此位置获得子进程的线程 ID。（如上所述，创建 POSIX 线程时总是指定了 CLONE_VM 标志。）

CLONE_PARENT_SETTID 之所以存在，意在为线程实现获取新线程 ID 提供一种可靠的手段。注意，通过 clone()的返回值并不足以获取新线程的线程 ID。

```
tid = clone(...);
```

问题在于，因为赋值操作只能在 clone()返回后才会发生，所以以上代码会导致各种竞争条件。例如，假设新线程终止，而在完成对 tid 的赋值前就调用了终止信号的处理器程序。此时，处理器程序无法有效访问 tid。（在线程库内部，可能会将 tid 置于一个用以跟踪所有线程状态的全局结构中。）程序通常可以通过直接调用 clone()来规避这种竞争条件。不过，线程库无法控制其调用者程序的行为。使用 CLONE_PARENT_SETTID 可以保证在 clone()返回之前就将新线程的 ID 赋值给 ptid 指针，从而使线程库避免了这种竞争条件。

如果设置了 CLONE_CHILD_SETTID，那么 clone()会将子线程的线程 ID 写入指针 ctid 所指向的位置。对 ctid 的设置只会发生在子进程的内存中，不过如果设置了 CLONE_VM，还是会影响到父进程。虽然 NPTL 并不需要 CLONE_CHILD_SETTID，但这一标识还是能给其他的线程库实现带来灵活性。

如果设置了 CLONE_CHILD_CLEARTID 标志，那么 clone()会在子进程终止时将 ctid 所指向的内存内容清零。

借助于参数 ctid 所提供的机制（稍后描述），NPTL 线程实现可以获得线程终止的通知。函数 pthread_join()正需要这样的通知，POSIX 线程利用该函数来等待另一线程的终止。

使用 pthread_create()创建线程时，NPTL 会调用 clone()，其 ptid 和 ctid 均指向同一位置。（这正是 NPTL 不需要 CLONE_CHILD_SETTID 的原因所在。）设置了 CLONE_PARENT_ SETTID 标志，就会以新的线程 ID 对该位置进行初始化。当子进程终止，ctid 遭清除时，进程中的所有线程都会目睹这一变化（因为设置了 CLONE_VM）。

内核将 ctid 指向的位置视同 futex——一种有效的同步机制来处理。（关于 futex 的更多内容请参考 futex(2)手册页。）执行系统调用 futex()来监测 ctid 所指位置的内容变化，就可获得线程终止的通知。（这正是 pthread_join()所做的幕后工作。）内核在清除 ctid 的同时，也会唤醒那些调用了 futex()来监控该地址内容变化的任一内核调度实体（即线程）。（在 POSIX 线程的层面上，这会导致 pthread_join()调用去解除阻塞。）

线程本地存储：CLONE_SETTLS

如果设置了 CLONE_SETTLS，那么参数 tls 所指向的 user_desc 结构会对该线程所使用的线程本地存储缓冲区加以描述。为了支持 NPTL 对线程本地存储的实现，Linux 2.6 开始加入这一标志（31.4 节）。关于 user_desc 结构的详情，可参考 2.6 内核代码中对该结构的定义和使用，以及 set_thread_area(2)手册页。

更多资源请访问稀酷客(www.ckook.com)

共享 System V 信号量的撤销值：CLONE_SYSVSEM

如果设置了 CLONE_SYSVSEM，父、子进程将共享同一个 System V 信号量撤销值列表（47.8 节）。如果未设置该标志，父、子进程各自持有取消列表，且子进程的列表初始为空。

内核从 2.6 版本开始支持 CLONE_SYSVSEM，提供 POSIX 线程规范所要求的共享语义。

每进程挂载命名空间：CLONE_NEWNS

Linux 从内核 2.4.19 开始支持每进程挂载（mount）命名空间的概念。挂载命名空间是由对 mount()和 umount()的调用来维护的一组挂载点。挂载命名空间会影响将路径名解析为真实文件的过程，也会波及诸如 chdir()和 chroot()之类的系统调用。

默认情况下，父、子进程共享同一挂载命名空间，一个进程调用 mount()或 umount()对命名空间所做的改变，也会为其他进程所见（如同 fork()和 vfork()）。特权级（CAP_ SYS_ADMIN）进程可以指定 CONE_NEWNS 标志，以便子进程去获取对父进程挂载命名空间的一份拷贝。这样一来，进程对命名空间的修改就不会为其他进程所见。（早期的 2.4.x 内核以及更老的版本认为，系统的所有进程共享同一个系统级挂载命名空间。）

可以利用每进程挂载命名空间来创建类似于 chroot()监禁区（jail）的环境，而且更加安全、灵活，例如，可以向遭到监禁的进程提供一个挂载点，而该点对于其他进程是不可见的。设置虚拟服务器环境时也会用到挂载命名空间。

在同一 clone()调用中同时指定 CLONE_NEWNS 和 CLONE_FS 纯属无聊，也不允许这样做。

将子进程的父进程置为调用者的父进程：CLONE_PARENT

默认情况下，当调用 clone()创建新进程时，新进程的父进程（由 getppid()返回）就是调用 clone()的进程（同 fork()和 vfork()）。如果设置了 CLONE_PARENT，那么调用者的父进程就成为子进程的父进程。换言之，CLONE_PARENT 等同于这样的设置：子进程.PPID = 调用者.PPID。（未设置 CLONE_PARENT 的默认情况是：子进程.PPID = 调用者.PID。）子进程终止时会向父进程（子进程.PPID）发出信号。

Linux 从版本 2.4 之后开始支持 CLONE_PARENT。其设计初衷意图是对 POSIX 线程的实现提供支持，不过内核 2.6 找出一种无需此标志而支持线程（之前所述的 CLONE_ THREAD）的新方法。

将子进程的进程 ID 置为与父进程相同：CLONE_PID（已废止）

如果设置了 CLONE_PID，那么子进程就拥有与父进程相同的进程 ID。若未设置此标志，那么父、子进程的进程 ID 则不同（如同 fork()和 vfork()）。只有系统引导进程（进程 ID 为 0）可能会使用该标志，用于初始化多处理器系统。

CLONE_PID 的设计初衷并非供用户级应用使用。Linux 2.6 已将其移除，并以 CLONE_IDLETASK 取而代之，将新进程的 ID 置为 0。CLONE_IDLETASK 仅供内核内部使用（即使在 clone()的参数中指定，系统也会对其视而不见）。使用此标志可为每颗 CPU 创建隐身的空闲进程（idle process），在多处理器系统中可能存在有多个实例。

进程跟踪：CLONE_PTRACE 和 CLONE_UNTRACED

如果设置了 CLONE_PTRACE 且正在跟踪调用进程，那么也会对子进程进行跟踪。关于进程跟踪（由调试器和 strace 命令使用）的细节，请参考 ptrace(2)手册页。

更多资源请访问稀酷客(www.ckook.com)

从内核 2.6 开始，即可设置 CLONE_UNTRACED 标志，这也意味着跟踪进程不能强制将其子进程设置为 CLONE_PTRACE。CLONE_UNTRACED 标志供内核创建内核线程时内部使用。

挂起（suspending）父进程直至子进程退出或调用 exec()：CLONE_VFORK

如果设置了 CLONE_VFORK，父进程将一直挂起，直至子进程调用 exec()或_exit()来释放虚拟内存资源（如同 vfork()）为止。

支持容器（container）的 clone()新标志

Linux 从 2.6.19 版本开始给 clone()加入了一些新标志：CLONE_IO、CLONE_NEWIPC、CLONET_NEWNET、CLONE_NEWPID、CLONE_NEWUSER 和 CLONE_NEWUTS。（参考 clone(2)手册页可获得有关这些标志的详细说明。）

这些标志中的大部分都是为容器（container）的实现提供支持（[Bhattiprolu et al., 2008]）。容器是轻量级虚拟化的一种形式，将运行于同一内核的进程组从环境上彼此隔离，如同运行在不同机器上一样。容器可以嵌套，一个容器可以包含另一个容器。与完全虚拟化将每个虚拟环境运行于不同内核的手法相比，容器的运作方式可谓是大相径庭。

为实现容器，内核开发者不得不为内核中的各种全局系统资源提供一个间接层，以便每个容器能为这些资源提供各自的实例。这些资源包括：进程 ID、网络协议栈、uname()返回的 ID、System V IPC 对象、用户和组 ID 命名空间……

容器的用途很多，如下所示。

- 控制系统的资源分配，诸如网络带宽或 CPU 时间（例如，授予容器某甲 75%的 CPU 时间，某乙则获取 25%）。
- 在单台主机上提供多个轻量级虚拟服务器。
- 冻结某个容器，以此来挂起该容器中所有进程的执行，并于稍后重启，可能是在迁移到另一台机器之后。
- 允许转储应用程序的状态信息，记录于检查点（checkpointed），并于之后再行恢复（或许在应用程序崩溃之后，亦或是计划内、外的系统停机后），从检查点开始继续运行。

clone()标志的使用

大体上说来，fork()相当于仅设置 flags 为 SIGCHLD 的 clone()调用，而 vfork()则对应于设置如下 flags 的 clone()：

```
CLONE_VM | CLONE_VFORK | SIGCHLD
```

自 2.3.3 版本以来，作为 NPTL 线程实现的一部分，glibc 所提供的封装函数 fork()绕开了内核的 fork()系统调用，转而调用了 clone()。该封装函数会去调用任何由调用者通过 pthread_atfork()（参考 33.3 节）所设置的 fork 处理器程序。

LinuxThreads 线程实现使用 clone()（仅用到前 4 个参数）来创建线程，对 flags 的设置如下：

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

NPTL 线程实现则使用 clone()（使用了所有 7 个参数）来创建线程，对 flags 的设置如下：

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD |
CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```


更多资源请访问稀酷客(www.ckook.com)

28.2.2 因克隆生成的子进程而对 waitpid()进行的扩展

为等待由 clone()产生的子进程，waitpid()、wait3()和 wait4()的位掩码参数 options 可以包含如下附加（Linux 特有）值。

__WCLONE

一经设置，只会等待克隆子进程。如未设置，只会等待非克隆子进程。在这种情况下，克隆子进程终止时发送给其父进程的信号并非 SIGCHLD。如果同时还指定了__WALL，那么将忽略__WCLONE。

__WALL（自 Linux2.4 之后）

等待所有子进程，无论类型（克隆、非克隆通吃）。

__WNOTHREAD（自 Linux2.4 之后）

默认情况下，等待（wait）类调用所等待的子进程，其父进程的范围遍及与调用者隶属同一线程组的任何进程。指定__WNOTHREAD 标志则限制调用者只能等待自己的子进程。

waitid()不能使用上述标志。

28.3 进程的创建速度

表 28-3 对采用不同方法创建进程的速度进行了比较。测试程序在循环中反复创建子进程并等待子进程终止，从而获得了这一结果。比较过程中使用了 3 种不同大小的进程内存，如表中虚拟内存总量（total virtual memory）值所示。对不同大小内存的模拟，依赖于程序计时之前在堆中分配（malloc()）的额外内存。

表 28-3 中的进程大小（虚拟内存总量）取自命令 ps –o “pid vsz cmd” 输出的 VSZ 值。

表 28-3：使用 fork()、vfork()和 clone()创建 10 万个进程所需的时间

进程的创建方法	虚拟内存总量					
	1.70 MB		2.70 MB		11.70 MB	
	时间（秒）	速率	时间（秒）	速率	时间（秒）	速率
fork()	22.27 （7.99）	4544	26.38 （8.98）	4135	126.93 （52.55）	1276
vfork()	3.52 （2.49）	28955	3.55 （2.50）	28621	3.53 （2.51）	28810
clone()	2.97 （2.14）	34333	2.98 （2.13）	34217	2.93 （2.10）	34688
fork() + exec()	135.72 （12.39）	764	146.15 （16.69）	719	260.34 （61.86）	435
vfork() + exec()	107.36 （6.27）	969	107.81 （6.35）	964	107.97 （6.38）	960

更多资源请访问稀酷客(www.ckook.com)

表 28-3 对于每种进程大小都提供了两类统计数据。

- 第 1 项统计包含两种度量时间。以执行 10 万次进程创建期间所逝去的（实际）时间为主（较大值），以父进程所消耗的 CPU 时间（括号内的值）为辅。由于测试环境并无其他负载，两者之差应是测试期间创建子进程所消耗的时间总量。
- 第 2 项数据显示每（实际）秒创建的进程数，即创建速率，取各种情况下运行 20 次的平均值。实验基于 x86-32 系统，内核版本为 2.6.27。

前 3 行针对的是简单的进程创建（子进程不运行新程序）。子进程在创建后立即退出，父进程等待子进程终止后再去创建下一个子进程。

第 1 行取自系统调用 fork()。由数据可知，进程所占内存越大，fork()所需时间也就越长。额外时间花在了为子进程复制那些逐渐变大的页表，以及将数据段、堆段以及栈段的页记录标记为只读的工作上。因为子进程并未修改数据段或栈段，所以也没有对页（page）复制。

第 2 行取自 vfork()。可以看出，尽管进程大小在增加，但所用时间保持不变，因为调用 vfork()时并未复制页表或页，调用进程的虚拟内存大小并未造成影响。fork()和 vfork()在时间统计上的差值就是复制进程页表所需的时间总量。

表 28-3 中 vfork()和 clone()的各自数据在不同的进程内存大小下。之所以存在微小的差异，要归因于采样误差以及调度的变化。即使创建 300MB 大小的进程，两个系统调用的时间仍将保持不变。

第 3 行数据的统计信息来自对 clone()的调用，所使用的标志如下：

```
CLONE_VM | CLONE_VFORK | CLONE_FS | CLONE_SIGHAND | CLONE_FILES
```

前两个标志模拟 vfork()的行为。剩余的标志则要求父、子进程应当共享文件系统属性文件权限掩码（umask）、根目录和当前工作目录，信号处置表以及打开文件描述符表。clone()和 vfork()之间的数据差值则代表了 vfork()将这些信息拷贝到子进程的少量额外工作。拷贝文件系统属性和信号处置表的成本是固定的。不过，拷贝打开文件描述符表的开销则取决于描述符数量。例如：父进程打开 100 个文件，vfork()的实际时间（表中第 1 列）会从 3.52 秒增至 5.04 秒，但不会影响 clone()所需要的时间。

对 clone()的计时针对的是 glibc 库的封装函数 clone()，而非直接调用 sys_clone()。另有测试（在此恕不一一列出）对 sys_clone()和 clone()(以子函数调用并立即退出) 做了比较，实验结果表明，时间上的差异可以忽略不计。

fork()和 vfork()之间的差别非常明显，但仍需要注意以下几点。

- 最后一列数据表明，在大进程情况下，vfork()要比 fork()快逾 30 倍。而针对普通进程，则会近乎于表中前两列的数据。
- 因为进程的创建时间往往比 exec()的执行时间要少得多，所以如果随后接着执行 exec()，那么两者间的差异也就不再明显。表 28-3 的最后两行数据说明了这一点，其中的每个子进程都去调用 exec()，而非直接退出。程序执行的是 true 命令（/bin/true，选择该程序的原因是因为它不产生任何输出）。这时，fork()和 vfork()之间的相对差距就小了许多。

事实上，表 28-3 中所示数据并未揭示 exec()的全部开销，因为测试程序的每个循环中子进程均执行同一程序。根本就未计入把程序文件读入内存的磁盘 I/O 开销，因为第一次运行 exec()时就会将程序读入内核缓冲区，并一直保存在那里。如果测试每次循环执行的程序不同（例如，复制同一程序，并以不同文件名命名），那么应该可以观察到 exec()的开销要大出许多。

更多资源请访问稀酷客(www.ckook.com)

28.4 exec()和 fork()对进程属性的影响

进程有多种属性，其中一部分已经在前面几章有所说明，后续章节将讨论其他一些属性。关于这些属性，存在两个问题。

- 当进程执行 exec()时，这些属性将发生怎样的变化？
- 当执行 fork()时，子进程会继承哪些属性？

表 28-4 是对这些问题的回答。exec()列注明，调用 exec()期间哪些属性得以保存。fork()列则表明调用 fork()之后子进程继承，或（有时是）共享了哪些属性。除了标注为 Linux 特有的属性之外，列出的所有属性均获得了标准 UNIX 实现的支持，调用 exec()和 fork()期间对它们的处理也都符合 SUSv3 规范。

表 28-4：exec()和 fork()对进程属性的影响

进 程 属 性	exec()	fork()	影响属性的接口；额外说明
进程地址空间			
文本段	否	共享	子进程与父进程共享文本段
栈段	否	是	函数入口/出口；alloca()、longjmp()、siglongjmp()
数据段和堆段	否	是	brk()、sbrk()
环境变量	见注释	是	putenv()、setenv()；直接修改 environ。execle()和 execve()会对其改写，其他 exec()调用则会加以保护
内存映射	否	是；见注释	mmap()、munmap()。跨越 fork()进程，映射的 MAP_NORESERVE 标志得以继承。带有 madvise（MADV_ DONTFORK）标志的映射则不会跨 fork()继承
内存锁	否	否	mlock()、munlock()
进程标识符和凭证			
进程 ID	是	否	
父进程 ID	是	否	
进程组 ID	是	是	setpgid()
会话 ID	是	是	setsid()
实际 ID	是	是	setuid()、setgid()，以及相关调用
有效和保存设置（saved set）ID	见注释	是	setuid()、setgid()，以及相关调用。第 9 章解释了 exec()是如何影响这些 ID 的
补充组 ID	是	是	setgroups()、initgroups()
文件、文件 IO 和目录			
打开文件描述符	见注释	是	open()、close()、dup()、pipe()、socket()等。文件描述符在跨越 exec()调用的过程中得以保存，除非对其设置了执行时关闭（close-on-exec）标志。父、子进程中的描述符指向相同的打开文件描述，参考 5.4 节
执行时关闭（close-on-exec）标志	是（如果关闭）	是	fcntl（F_SETFD）
文件偏移	是	共用	lseek()、read()、write()、readv()、writev()。父、子进程共享文件偏移

更多资源请访问稀酷客(www.ckook.com)

续表

进 程 属 性	exec()	fork()	影响属性的接口；额外说明
文件、文件 IO 和目录			
打开文件状态标志	是	共用	open()、fcntl(F_SETFL)。父、子进程共享打开文件状态标志
异步 I/O 操作	见注释	否	aio_read()、aio_write()以及相关调用。调用 exec()期间，会取消尚未完成的操作
目录流	否	是：见注释	opendir()、readdir()。SUSv3 规定，子进程获得父进程目录流的一份副本，不过这些副本可以（也可以不）共享目录流的位置。Linux 系统不共享目录流的位置
文件系统			
当前工作目录	是	是	chdir()
根目录	是	是	chroot()
文件模式创建掩码	是	是	umask()
信号			
信号处置	见注释	是	signal()、sigaction()。将处置设置成默认或忽略的信号在执行 exec()期间保持不变；已捕获的信号会恢复为默认处置。参考 27.5 节
信号掩码	是	是	信号传递；sigprocmask()、sigaction()
挂起（pending）信号集合	是	否	信号传递；raise()、kill()、sigqueue()
备选信号栈	否	是	sigaltstack()
定时器			
间隔定时器	是	否	setitimer()
由 alarm()设置的定时器	是	否	alarm()
POSIX 定时器	否	否	timer_create()及其相关调用
POSIX 线程			
线程	否	见注释	fork()调用期间，子进程只会复制调用线程
线程可撤销状态与类型	否	是	exec()之后，将可撤销类型和状态分别重置为 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DEFERRED
互斥量与条件变量	否	是	关于调用 fork()期间对互斥量以及其他线程资源的处理细节可参考 33.3 节
优先级与调度			
nice 值	是	是	nice()、setpriority()
调度策略及优先级	是	是	sched_setscheduler()、sched_setparam()
资源与 CPU 时间			
资源限制	是	是	setrlimit()
进程和子进程的 CPU 时间	是	否	由 times()返回
资源使用量	是	否	由 getrusage()返回

更多资源请访问稀酷客(www.ckook.com)

续表

进 程 属 性	exec()	fork()	影响属性的接口；额外说明
进程间通信			
System V 共享内存段	否	是	shmat()、shmdt()
POSIX 共享内存	否	是	shm_open()及其相关调用
POSIX 消息队列	否	是	mq_open()及其相关调用。父、子进程的描述符都指向同一打开消息队列描述。子进程并不继承父进程的消息通知注册信息
POSIX 命名信号量	否	共用	sem_open()及其相关调用。子进程与父进程共享对相同信号量的引用
POSIX 未命名信号量	否	见注释	sem_init()及其相关调用。如果信号量位于共享内存区域，那么子进程与父进程共享信号量；否则，子进程拥有属于自己的信号量拷贝
System V 信号量调整	是	否	参考 47.8 节
文件锁	是	见注释	flock()。子进程自父进程处继承对同一锁的引用
记录锁	见注释	否	fcntl(F_SETLK)。除非将指代文件的文件描述符标记为执行时关闭，否则会跨越 exec()对锁加以保护
杂项			
地区设置	否	是	setlocale()。作为 C 运行时初始化的一部分，执行新程序后会调用 setlocale(LC_ALL，"C")的等效函数
浮点环境	否	是	运行新程序时，将浮点环境状态重置为默认值，参考 fenv(3)
控制终端	是	是	
退出处理器程序	否	是	atexit()、on_exit()
Linux 特有			
文件系统 ID	见注释	是	setfsuid()、setfsgid()。一旦相应的有效 ID 发生变化，那么这些 ID 也会随之改变
timeerfd 定时器	是	见注释	timerfd_create()，子进程继承的文件描述符与父进程指向相同的定时器
能力	见注释	是	capset()。执行 exec()期间对能力的处理一如 39.5 节所述
功能外延集合	是	是	
能力安全位（securebits）标志	见注释	是	执行 exec()期间，会保全所有的安全位标志， SECBIT_KEEP_CAPS 除外，总是会清除该标志
CPU 黏性（affinity）	是	是	sched_setaffinity()
SCHED_RESET_ON_FORK	是	否	参考 35.3.2 节
允许的 CPU	是	是	参考 cpuset(7)手册页
允许的内存节点	是	是	参考 cpuset(7)手册页
内存策略	是	是	参考 set_mempolicy(2)手册页
文件租约	是	见注释	fcntl（F_SETLEASE）。子进程从父进程处继承对相同租约的引用

更多资源请访问稀酷客(www.ckook.com)

续表

进 程 属 性	exec()	fork()	影响属性的接口；额外说明
Linux 特有			
目录变更通知	是	否	dnotify API，通过 fcntl（F_NOTIFY）来实现支持
prctl（PR_SET_DUMPABLE）	见注释	是	exec()执行期间会设置 PR_SET_DUMPABLE 标志，执行设置用户或组 ID 程序的情况除外，此时将清除该标志
prctl（PR_SET_PDEATHSIG）	是	否	
prctl（PR_SET_NAME）	否	是	
oom_adj	是	是	参考 49.9 节
coredump_filter	是	是	参考 22.1 节

28.5 总结

当打开进程记账功能时，内核会在系统中每一进程终止时将其账单记录写入一个文件。该记录包含进程使用资源的统计数据。

如同函数 fork()，Linux 特有的 clone()系统调用也会创建一个新进程，但其对父子间的共享属性有更为精确的控制。该系统调用主要用于线程库的实现。

本章对 fork()、vfork()和 clone()的进程创建速度做了比较。尽管 vfork()要快于 fork()，但较之于子进程随后调用 exec()所耗费的时间，二者间的时间差异也就微不足道了。

fork()创建的子进程会从其父进程处继承（有时是共享）某些进程属性的副本，而对其他进程属性则不做继承。例如，子进程继承了父进程文件描述符表和信号处置的副本，但并不继承父进程的间隔定时器、记录锁或是挂起信号集合。相应地，进程执行 exec()时，某些进程属性保持不变，而会将其他属性重置为缺省值。例如，进程 ID 保持不变，文件描述符保持打开（除非设置了执行时关闭标志），间隔定时器得以保存，挂起信号依然挂起，不过会将对已处理信号的处置重置为默认设置，同时与共享内存段“脱钩”。

更多信息

请参考 24.6 节列出的更多信息来源。[Frisch, 2002]第 17 章描述了对进程记账的管理，以及不同 UNIX 实现之间的差异。[Bovert & Vesati, 2005]介绍了系统调用 clone()的实现。

28.6 练习

28-1. 编写一程序，观察 fork()和 vfork()系统调用在读者系统中的速度。要求每个子进程必须立即退出，而父进程应在创建下一个子进程之前调用 wait()，等待当前子进程退出。将两个系统调用之间的差别与表 28-3 相比较。shell 内建命令 time 可用来测量程序的执行时间。

更多资源请访问稀酷客(www.ckook.com)

第29章

线程：介绍

本章及随后几章将讨论 POSIX 线程（thread），亦即 Pthreads。鉴于 Pthreads 范围极广，本书无意涵盖其所有 API。关于线程的深入信息，本章会在结尾处列出其来源。

后续各章将主要描述 Pthreads API 所规定的标准行为。33.5 节则会探讨 Linux 的两种主流线程实现——LinuxThreads 和 Native POSIX Threads Library（NPTL）——与线程标准间的出入。

本章会对线程操作加以概述，随之将述及线程的创建和销毁过程。此外，在应用程序设计中，是选择多线程还是多进程方式？本章将在最后讨论影响这一选择的因素。

29.1 概述

与进程（process）类似[1]，线程（thread）是允许应用程序并发执行多个任务的一种机制。如图 29-1 所示，一个进程可以包含多个线程。同一程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域，其中包括初始化数据段（initialized data）、未初始化数据段（uninitialized data），以及堆内存段（heap segment）。（传统意义上的 UNIX 进程只是多线程程序的一个特例，该进程只包含一个线程。）

图 29-1 其实做了一些简化。特别是，线程栈（thread stack）的位置可能会与共享库和共享内存区域混杂在一起，这取决于创建线程、加载共享库，以及映射共享内存的具体顺序。而且，对于不同的 Linux 发行版，线程栈地址也会有所不同。

同一进程中的多个线程可以并发执行。在多处理器环境下，多个线程可以同时并行。如果一线程因等待 I/O 操作而遭阻塞，那么其他线程依然可以继续运行。（虽然有时单独创建一个专门执行 I/O 操作的线程颇为有用，但采用另一种 I/O 模型则更为可取，第 63 章会对此加以描述。）

对于某些应用而言，线程要优于进程。传统 UNIX 通过创建多个进程来实现并行任务。以网络服务器的设计为例，服务器进程（父进程）在接受客户端的连接后，会调用 fork()来创建一个单独的子进程，以处理与客户端的通信（可参考 60.3 节）。采用这种设计，服务器就能

1 译者注：此处指多进程并发。

更多资源请访问稀酷客(www.ckook.com)

同时为多个客户端提供服务。虽然这种方法在很多情境下都屡试不爽，但对于某些应用来说也确实存在如下一些限制。

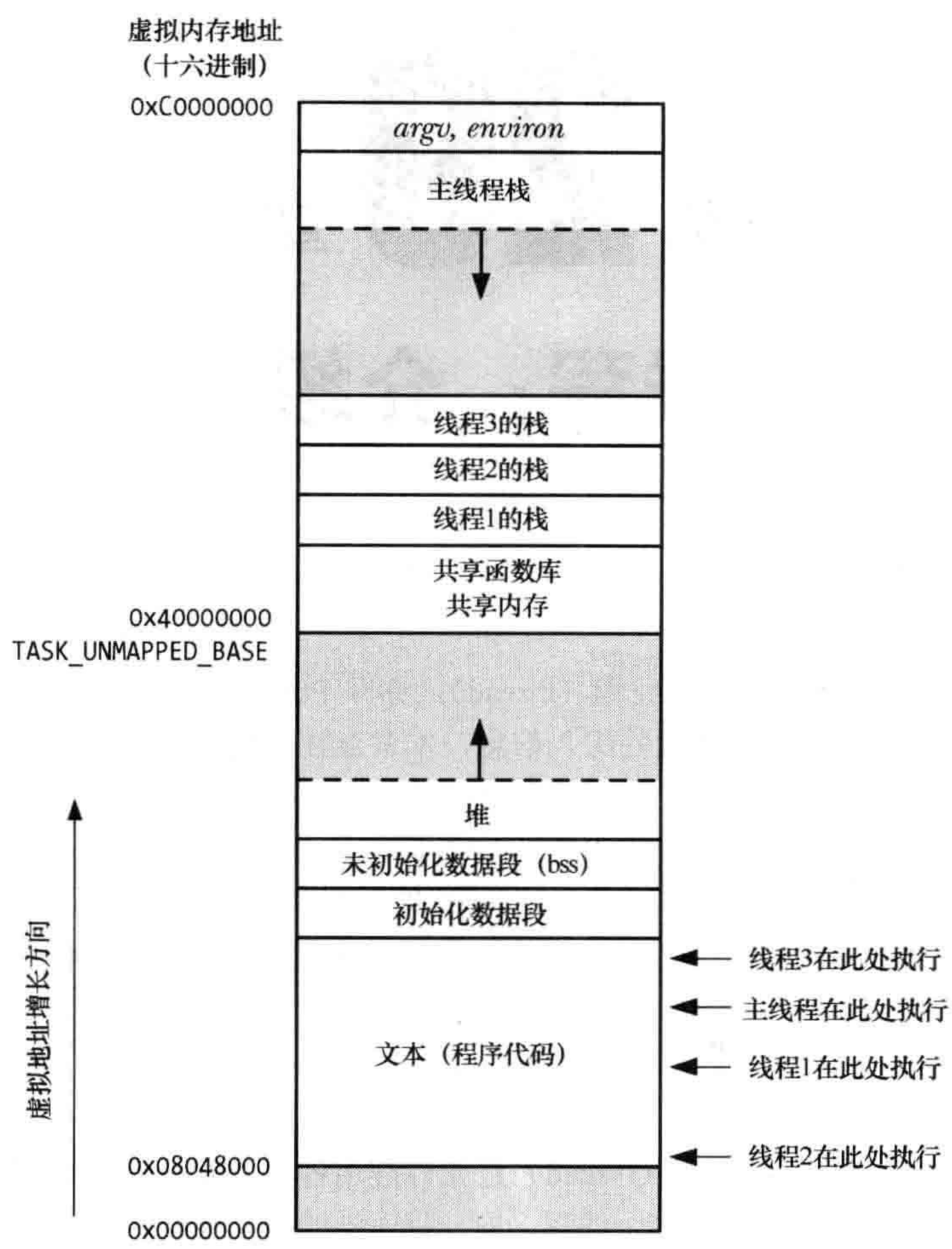

图 29-1：同时执行 4 个线程的进程（Linux/x86-32）

- 进程间的信息难以共享。由于除去只读代码段外，父子进程并未共享内存，因此必须采用一些进程间通信（inter-process communication，简称 IPC）方式，在进程间进行信息交换。
- 调用 fork()来创建进程的代价相对较高。即便利用 24.2.2 节所描述的写时复制（copy-on-write）技术，仍然需要复制诸如内存页表（page table）和文件描述符表（file descriptor table）之类的多种进程属性，这意味着 fork()调用在时间上的开销依然不菲。

线程解决了上述两个问题。

- 线程之间能够方便、快速地共享信息。只需将数据复制到共享（全局或堆）变量中即可。不过，要避免出现多个线程试图同时修改同一份信息的情况，这需要使用第 30 章描述的同步技术。
- 创建线程比创建进程通常要快 10 倍甚至更多。（在 Linux 中，是通过系统调用 clone()来实现线程的，表 28-3 展示了 fork()和 clone()在速度上的差异。）线程的创建之所以较快，是因为调用 fork()创建子进程时所需复制的诸多属性，在线程间本来就是共享的。特别是，既无需采用写时复制来复制内存页，也无需复制页表。

更多资源请访问稀酷客(www.ckook.com)

除了全局内存之外，线程还共享了一干其他属性（这些属性对于进程而言是全局性的，而并非针对某个特定线程），包括以下内容。

- 进程 ID（process ID）和父进程 ID。
- 进程组 ID 与会话 ID（session ID）。
- 控制终端。
- 进程凭证（process credential）（用户 ID 和组 ID ）。
- 打开的文件描述符。
- 由 fcntl()创建的记录锁（record lock）。
- 信号（signal）处置。
- 文件系统的相关信息：文件权限掩码（umask）、当前工作目录和根目录。
- 间隔定时器（setitimer()）和 POSIX 定时器（timer_create()）。
- 系统 V（system V）信号量撤销（undo，semadj）值（47.8 节）。
- 资源限制（resource limit）。
- CPU 时间消耗（由 times()返回）。
- 资源消耗（由 getrusage()返回）。
- nice 值（由 setpriority()和 nice()设置）。

各线程所独有的属性，如下列出了其中一部分。

- 线程 ID（thread ID，29.5 节）。
- 信号掩码（signal mask）。
- 线程特有数据（31.3 节）。
- 备选信号栈（sigaltstack()）。
- errno 变量。
- 浮点型（floating-point）环境（见 fenv(3)）。
- 实时调度策略（real-time scheduling policy）和优先级（35.2 节和 35.3 节）。
- CPU 亲和力（affinity，Linux 所特有，35.4 节将加以描述）。
- 能力（capability，Linux 所特有，第 39 章将加以描述）。
- 栈，本地变量和函数的调用链接（linkage）信息。

如图 29-1 所示，所有的线程栈均驻留于同一虚拟地址空间。这也意味着，利用一个合适的指针，各线程可以在对方栈中相互共享数据。这种方法偶尔也能派上用场，但由于局部变量的状态有效与否依赖于其所驻留栈帧的生命周期，故而需要在编程中谨慎处理这一问题。（当函数返回时，该函数栈帧所占用的内存区域有可能为后续的函数调用所重新使用。如果线程终止，那么新线程有可能会对已终止线程的栈所占用的内存空间重新加以利用）。若无法正确处理这一依赖关系，由此而产生的程序 bug 将难以捕获。

29.2 Pthreads API 的详细背景

20 世纪 80 年代末、90 年代初，存在着数种不同的线程接口。1995 年 POSIX.1c 对 POSIX 线程 API 进行了标准化，该标准后来为 SUSv3 所接纳。

有几个概念贯穿整个 Pthreads API，在深入探讨 API 之前，将简单予以介绍。

更多资源请访问稀酷客(www.ckook.com)

线程数据类型（Pthreads data type）

Pthreads API 定义了一干数据类型，表 29-1 列出了其中的一部分。后续章节会对这些数据类型中的绝大部分加以描述。

表 29-1：Pthreads 数据类型

数据类型	描述
pthread_t	线程 ID
pthread_mutex_t	互斥对象（Mutex）
pthread_mutexattr_t	互斥属性对象
pthread_cond_t	条件变量（condition variable）
pthread_condattr_t	条件变量的属性对象
pthread_key_t	线程特有数据的键（Key）
pthread_once_t	一次性初始化控制上下文（control context）
pthread_attr_t	线程的属性对象

SUSv3 并未规定如何实现这些数据类型，可移植的程序应将其视为“不透明”数据。亦即，程序应避免对此类数据类型变量的结构或内容产生任何依赖。尤其是，不能使用 C 语言的比较操作符（==）去比较这些类型的变量。

线程和 errno

在传统 UNIX API 中，errno 是一全局整型变量。然而，这无法满足多线程程序的需要。如果线程调用的函数通过全局 errno 返回错误时，会与其他发起函数调用并检查 errno 的线程混淆在一起。换言之，这将引发竞争条件（race condition）。因此，在多线程程序中，每个线程都有属于自己的 errno。在 Linux 中，线程特有 errno 的实现方式与大多数 UNIX 实现相类似：将 errno 定义为一个宏，可展开为函数调用，该函数返回一个可修改的左值（lvalue），且为每个线程所独有。（因为左值可以修改，多线程程序依然能以 errno=value 的方式对 errno 赋值。）

一言以蔽之，errno 机制在保留传统 UNIX API 报错方式的同时，也适应了多线程环境。

> 最初的 POSIX.1 标准沿袭 K&R 的 C 语言用法，允许程序将 errno 声明为 extern int errno。SUSv3 却不允许这一做法（这一变化实际发生于 1995 年的 POSIX.1c 标准之中）。如今，需要声明 errno 的程序必须包含<errno.h>，以启用对 errno 的线程级实现。

Pthreads 函数返回值

从系统调用和库函数中返回状态，传统的做法是：返回 0 表示成功，返回-1 表示失败，并设置 errno 以标识错误原因。Pthreads API 则反其道而行之。所有 Pthreads 函数均以返回 0 表示成功，返回一正值表示失败。这一失败时的返回值，与传统 UNIX 系统调用置于 errno 中的值含义相同。

由于多线程程序对 errno 的每次引用都会带来函数调用的开销，因此，本书示例并不会直接将 Pthreads 函数的返回值赋给 errno，而是使用一个中间变量，并利用自己实现的诊断函数 errExitEN()（3.5.2 节），如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
pthread_t *thread;
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

编译 Pthreads 程序

在 Linux 平台上，在编译调用了 Pthreads API 的程序时，需要设置 cc -pthread 的编译选项。使用该选项的效果如下。

- 定义_REENTRANT 预处理宏。这会公开对少数可重入（reentrant）函数的声明。
- 程序会与库 libpthread 进行链接（等价于-lpthread）。

编译多线程程序时的具体编译选项会因实现及编译器的不同而不同。其他一些实现（例如 Tru64）使用 cc –pthread，而 Solaris 和 HP-UX 则使用 cc –mt。

29.3 创建线程

启动程序时，产生的进程只有单条线程，称之为初始（initial）或主（main）线程。本节将讨论其他线程的创建过程。

函数 pthread_create()负责创建一条新线程。

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

Returns 0 on success, or a positive error number on error

新线程通过调用带有参数 arg 的函数 start（即 start(arg)）而开始执行。调用 pthread_create()的线程会继续执行该调用之后的语句。（如 28.2 节所述，这一行为与 glibc 库对系统调用 clone()的包装函数行为相同。）

将参数 arg 声明为 void*类型，意味着可以将指向任意对象的指针传递给 start()函数。一般情况下，arg 指向一个全局或堆变量，也可将其置为 NULL。如果需要向 start()传递多个参数，可以将 arg 指向一个结构，该结构的各个字段则对应于待传递的参数。通过审慎的类型强制转换，arg 甚至可以传递 int 类型的值。

严格说来，对于 int 与 void*之间相互强制转换的后果，C 语言标准并未加以定义。不过，大部分 C 语言编译器允许这样的操作，并且也能达成预期的目的，即 int j = (int) ((void*) j)。

start()的返回值类型为 void*，对其使用方式与参数 arg 相同。对后续 pthread_join()函数的描述中，将论及对该返回值的使用方式。

将经强制转换的整型数作为线程 start 函数的返回值时，必须小心谨慎。原因在于，取消线程（见第 32 章）时的返回值 PTHREAD_CANCELED，通常是由实现所定义的整型值，再经强制转换为 void*。若线程某甲的 start 函数将此整型值返回给正在执行 pthread_join()操作的线程某乙，某乙会误认为某甲遭到了取消。应用如果采用了线程取消技术并选择将 start 函数的返回值强制转换为整型，那么就必须确保线程正常结束时的返回值与当前 Pthreads 实现中的 PTHREAD_CANCELED 不同。如欲保证程序的可移植性，则在任何将要运行该应用的实现中，

更多资源请访问稀酷客(www.ckook.com)

正常退出线程的返回值应不同于相应的 PTHREAD_CANCELED 值。

参数 thread 指向 pthread_t 类型的缓冲区，在 pthread_create()返回前，会在此保存一个该线程的唯一标识。后续的 Pthreads 函数将使用该标识来引用此线程。

SUSv3 明确指出，在新线程开始执行之前，实现无需对 thread 参数所指向的缓冲区进行初始化，即新线程可能会在 pthread_create()返回给调用者之前已经开始运行。如新线程需要获取自己的线程 ID，则只能使用 pthread_self()（29.5 节描述）方法。

参数 attr 是指向 pthread_attr_t 对象的指针，该对象指定了新线程的各种属性。29.8 节将述及其中的部分属性。如果将 attr 设置为 NULL，那么创建新线程时将使用各种默认属性，本书的大部分示例程序都采用这一做法。

调用 pthread_create()后，应用程序无从确定系统接着会调度哪一个线程来使用 CPU 资源（在多处理器系统中，多个线程可能会在不同 CPU 上同时执行）。程序如隐含了对特定调度顺序的依赖，则无疑会对 24.4 节所述的竞争条件打开方便之门。如果对执行顺序确有强制要求，那么就必须采用第 30 章所描述的同步技术。

29.4 终止线程

可以如下方式终止线程的运行。

- 线程 start 函数执行 return 语句并返回指定值。
- 线程调用 pthread_exit()（详见后述）。
- 调用 pthread_cancel()取消线程（在 32.1 节讨论）。
- 任意线程调用了 exit()，或者主线程执行了 return 语句（在 main()函数中），都会导致进程中的所有线程立即终止。

pthread_exit()函数将终止调用线程，且其返回值可由另一线程通过调用 pthread_join()来获取。

```
include <pthread.h>

void pthread_exit(void *retval);
```

调用 pthread_exit()相当于在线程的 start 函数中执行 return，不同之处在于，可在线程 start 函数所调用的任意函数中调用 pthread_exit() 。

参数 retval 指定了线程的返回值。Retval 所指向的内容不应分配于线程栈中，因为线程终止后，将无法确定线程栈的内容是否有效。（例如，系统可能会立刻将该进程虚拟内存的这片区域重新分配，供一个新的线程栈使用。）出于同样的理由，也不应在线程栈中分配线程 start 函数的返回值。

如果主线程调用了 pthread_exit()，而非调用 exit()或是执行 return 语句，那么其他线程将继续运行。

29.5 线程 ID（Thread ID）

进程内部的每个线程都有一个唯一标识，称为线程 ID。线程 ID 会返回给 pthread_create()的调用者，一个线程可以通过 pthread_self()来获取自己的线程 ID。

```
include <pthread.h>

pthread_t pthread_self(void);
```


更多资源请访问稀酷客(www.ckook.com)

Returns the thread ID of the calling thread

线程 ID 在应用程序中非常有用，原因如下。

- 不同的 Pthreads 函数利用线程 ID 来标识要操作的目标线程。这些函数包括 pthread_join()、pthread_detach()、pthread_cancel()和 pthread_kill()等，后续章节将会加以讨论。
- 在一些应用程序中，以特定线程的线程 ID 作为动态数据结构的标签，这颇有用处，既可用来识别某个数据结构的创建者或属主线程，又可以确定随后对该数据结构执行操作的具体线程。

函数 pthread_equal()可检查两个线程的 ID 是否相同。

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if *t1* and *t2* are equal, otherwise 0

例如，为了检查调用线程的线程 ID 与保存于变量 t1 中的线程 ID 是否一致，可以编写如下代码：

```
if (pthread_equal(tid, pthread_self())
    printf("tid matches self\n");
```

因为必须将 pthread_t 作为一种不透明的数据类型加以对待，所以函数 pthread_equal()是必须的。Linux 将 pthread_t 定义为无符号长整型（unsigned long），但在其他实现中，则有可能是一个指针或结构。

在 NPTL 中，pthread_t 实际上是一个经强制转化而为无符号长整型的指针。

SUSv3 并未要求将 pthread_t 实现为一个标量（scalar）类型，该类型也可以是一个结构。因此，下列显示线程 ID 的代码实例并不具有可移植性（尽管该实例在包括 Linux 在内的许多实现上均可正常运行，而且有时在调试程序时还很实用）。

```
pthread_t thr;

printf("Thread ID = %ld\n", (long) thr);        /* WRONG! */
```

在 Linux 的线程实现中，线程 ID 在所有进程中都是唯一的。不过在其他实现中则未必如此，SUSv3 特别指出，应用程序若使用线程 ID 来标识其他进程的线程，其可移植性将无法得到保证。此外，在对已终止线程施以 pthread_join()，或者在已分离（detached）线程退出后，实现可以复用该线程的线程 ID。（下一节和 29.7 节将分别解释 pthread_join()和线程的分离。）

POSIX 线程 ID 与 Linux 专有的系统调用 gettid()所返回的线程 ID 并不相同。POSIX 线程 ID 由线程库实现来负责分配和维护。gettid()返回的线程 ID 是一个由内核（Kernel）分配的数字，类似于进程 ID（process ID）。虽然在 Linux NPTL 线程实现中，每个 POSIX 线程都对应一个唯一的内核线程 ID，但应用程序一般无需了解内核线程 ID（况且，如果程序依赖于这一信息，也将无法移植）。

29.6 连接（joining）已终止的线程

函数 pthread_join()等待由 thread 标识的线程终止。（如果线程已经终止，pthread_join()会

更多资源请访问稀酷客(www.ckook.com)

立即返回）。这种操作被称为连接(joining)。

```
include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number on error

若 retval 为一非空指针，将会保存线程终止时返回值的拷贝，该返回值亦即线程调用 return 或 pthred_exit()时所指定的值。

如向 pthread_join()传入一个之前已然连接过的线程 ID，将会导致无法预知的行为。例如，相同的线程 ID 在参与一次连接后恰好为另一新建线程所重用，再度连接的可能就是这个新线程。

若线程并未分离（detached，见 29.7 节），则必须使用 ptherad_join()来进行连接。如果未能连接，那么线程终止时将产生僵尸线程，与僵尸进程（zombie process）的概念相类似（参考 26.2 节）。除了浪费系统资源以外，僵尸线程若累积过多，应用将再也无法创建新的线程。

pthread_join()执行的功能类似于针对进程的 waitpid()调用，不过二者之间存在一些显著差别。

- 线程之间的关系是对等的（peers）。进程中的任意线程均可以调用 pthread_join()与该进程的任何其他线程连接起来。例如，如果线程 A 创建线程 B，线程 B 再创建线程 C，那么线程 A 可以连接线程 C，线程 C 也可以连接线程 A。这与进程间的层次关系不同，父进程如果使用 fork()创建了子进程，那么它也是唯一能够对子进程调用 wait()的进程。调用 pthread_create()创建的新线程与发起调用的线程之间，就没有这样的关系。
- 无法“连接任意线程”（对于进程，则可以通过调用 waitpid(-1, &status, options)做到这一点），也不能以非阻塞（nonblocking）方式进行连接（类似于设置 WHOHANG 标志的 waitpid()）。使用条件（condition）变量可以实现类似的功能，30.2.4 节会给出示例。

限制 pthread_join()只能连接特定线程 ID，这样做是“别有用心”的。其用意在于，程序应只能连接它所“知道的”线程。线程之间并无层次关系，如果听任“与任意线程连接”的操作发生，那么所谓“任意”线程就可以包括由库函数私自创建的线程，从而带来问题。（30.2.4 所展示的条件变量技术也只允许线程连接它“知道的”其他线程。）结果是，函数库在获取线程返回状态时将不再能与该线程连接[1]，只会一错再错，试图连接一个已然连接过的线程 ID。换言之，“连接任意线程”的操作与模块化的程序设计理念背道而驰。

示例程序

程序清单 29-1 中的程序创建了一个线程，并与之连接。

程序清单 29-1：一个使用 Pthreads 的简单程序

threads/simple_thread.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;
```

1 译者注：本句的两处线程均指前文由库函数私自创建的线程。

```
    printf("%s", s);

    return (void *) strlen(s);
}
int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

threads/simple_thread.c

当运行程序清单 29-1 的程序时，可以看到如下输出：

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

依赖于系统对两个线程的具体调度，第 1 行与第 2 行的输出顺序可能会颠倒过来。

29.7 线程的分离

默认情况下，线程是可连接的(joinable)，也就是说，当线程退出时，其他线程可以通过调用 pthread_join()获取其返回状态。有时，程序员并不关心线程的返回状态，只是希望系统在线程终止时能够自动清理并移除之。在这种情况下，可以调用 pthread_detach()并向 thread 参数传入指定线程的标识符，将该线程标记为处于分离（detached）状态。

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

如下例所示，使用 pthread_detach()，线程可以自行分离：

```
pthread_detach(pthread_self());
```

一旦线程处于分离状态，就不能再使用 pthread_join()来获取其状态，也无法使其重返“可连接”状态。

其他线程调用了 exit()，或是主线程执行 return 语句时，即便遭到分离的线程也还是会受到影响。此时，不管线程处于可连接状态还是已分离状态，进程的所有线程会立即终止。换

更多资源请访问稀酷客(www.ckook.com)

言之，pthread_detach()只是控制线程终止之后所发生的事情，而非何时或如何终止线程。

29.8 线程属性

前面已然提及 pthread_create()中类型为 pthread_attr_t 的 attr 参数，可利用其在创建线程时指定新线程的属性。本书无意深入这些属性的细节（关于此类细节，可参考本章结尾处的参考资料列表），也不会将操作 pthread_attr_t 对象的各种 Pthreads 函数原型一一列出，只会点出如下之类的一些属性：线程栈的位置和大小、线程调度策略和优先级（类似于 35.2 节和 35.3 节所描述的进程实时调度策略和优先级），以及线程是否处于可连接或分离状态。

作为线程属性的使用示例，程序清单 29-2 中的代码创建了一个新线程，该线程刚一创建即遭分离（而非之后再调用 pthread_detach()）。这段代码首先以缺省值对线程属性结构进行初始化，接着为创建分离线程而设置属性，最后再以此线程属性结构来创建新线程。线程一旦创建，就无需再保留该属性对象，故而程序将其销毁。

程序清单 29-2：使用分离属性创建线程

from **threads/detached_attrib.c**

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");

s = pthread_attr_destroy(&attr);             /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_attr_destroy");
```

from **threads/detached_attrib.c**

29.9 线程 VS 进程

将应用程序实现为一组线程还是进程？本节将简单考虑一下可能影响这一决定的部分因素。先从多线程方法的优点开始。

- 线程间的数据共享很简单。相形之下，进程间的数据共享需要更多的投入。（例如，创建共享内存段或者使用管道 pipe）。
- 创建线程要快于创建进程。线程间的上下文切换（context-switch），其消耗时间一般也比进程要短。

线程相对于进程的一些缺点如下所示。

- 多线程编程时，需要确保调用线程安全（thread-safe）的函数，或者以线程安全的方

更多资源请访问稀酷客(www.ckook.com)

式来调用函数。（31.1 节将讨论线程安全的概念。）多进程应用则无需关注这些。

- 某个线程中的 bug（例如，通过一个错误的指针来修改内存）可能会危及该进程的所有线程，因为它们共享着相同的地址空间和其他属性。相比之下，进程间的隔离更彻底。
- 每个线程都在争用宿主进程（host process）中有限的虚拟地址空间。特别是，一旦每个线程栈以及线程特有数据（或线程本地存储）消耗掉进程虚拟地址空间的一部分，则后续线程将无缘使用这些区域。虽然有效地址空间很大（例如，在 x86-32 平台上通常有 3GB），但当进程分配大量线程，亦或线程使用大量内存时，这一因素的限制作用也就突显出来。与之相反，每个进程都可以使用全部的有效虚拟内存，仅受制于实际内存和交换（swap）空间。

影响选择的还有如下几点。

- 在多线程应用中处理信号，需要小心设计。（作为通则，一般建议在多线程程序中避免使用信号。）关于线程与信号，33.2 节会做深入讨论。
- 在多线程应用中，所有线程必须运行同一个程序（尽管可能是位于不同函数中）。对于多进程应用，不同的进程可以运行不同的程序。
- 除了数据，线程还可以共享某些其他信息（例如，文件描述符、信号处置、当前工作目录，以及用户 ID 和组 ID）。优劣之判，视应用而定。

29.10 总结

在多线程程序中，多个线程并发执行同一程序。所有线程共享相同的全局和堆变量，但每个线程都配有用来存放局部变量的私有栈。同一进程中的线程还共享一干其他属性，包括进程 ID、打开的文件描述符、信号处置、当前工作目录以及资源限制。

线程与进程间的关键区别在于，线程比进程更易于共享信息，这也是许多应用程序舍进程而取线程的主要原因。对于某些操作来说（例如，创建线程比创建进程快），线程还可以提供更好的性能。但是，在程序设计的进程/线程之争中，这往往不会是决定性因素。

可使用 pthread_create()来创建线程。每个线程随后可调用 pthread_exit()独立退出。（如有任一线程调用了 exit()，那么所有线程将立即终止。）除非将线程标记为分离状态（例如通过调用 pthread_detached()），其他线程要连接该线程，则必须使用 pthread_join()，由其返回遭连接线程的退出状态。

进阶信息

[Butenhof, 1996]对 Pthreads 进行了透彻而清晰的阐述。[Robbins & Robbins, 2003]对 Pthreads 的各方面都有涉及。[Tanen-baum, 2007]对线程概念的介绍更具理论化，涵盖主题包括互斥量（mutex）、临界区（critical region）、条件变量以及死锁（deadlock）检测及规避。[Vahalia, 1996]提供了线程实现的背景知识。

29.11 练习

29-1. 若一线程执行了如下代码，可能会产生什么结果？

```
pthread_join(pthread_self(), NULL);
```

在 Linux 上编写一个程序，观察一下实际会发生什么情况。假设代码中有一变量 tid，其中包含了某个线程 ID，在自身发起 pthread_join(tid, NULL)调用时，要避免造成

更多资源请访问稀酷客(www.ckook.com)

与上述语句相同的后果，该线程应采取何种措施？

29-2. 除了缺少错误检查，以及对各种变量和结构的声明外，下列程序还有什么问题？

```
static void *
threadFunc(void *arg)
{
    struct someStruct *pbuf = (struct someStruct *) arg;

    /* Do some work with structure pointed to by 'pbuf' */
}

int
main(int argc, char *argv[])
{
    struct someStruct buf;

    pthread_create(&thr, NULL, threadFunc, (void *) &buf);
    pthread_exit(NULL);
}
```


更多资源请访问稀酷客(www.ckook.com)

第30章

线程：线程同步

本章介绍线程用来同步彼此行为的两个工具：互斥量（mutexe）和条件变量（condition variable）。互斥量可以帮助线程同步对共享资源的使用，以防如下情况发生：线程某甲试图访问一共享变量时，线程某乙正在对其进行修改。条件变量则是在此之外的拾遗补缺，允许线程相互通知共享变量（或其他共享资源）的状态发生了变化。

30.1 保护对共享变量的访问：互斥量

线程的主要优势在于，能够通过全局变量来共享信息。不过，这种便捷的共享是有代价的：必须确保多个线程不会同时修改同一变量，或者某一线程不会读取正由其他线程修改的变量。术语临界区（critical section）是指访问某一共享资源的代码片段，并且这段代码的执行应为原子（atomic）操作，亦即，同时访问同一共享资源的其他线程不应中断该片段的执行。

程序清单 30-1 中的简单示例，展示了以非原子方式访问共享资源时所发生的问题。该程序创建了两个线程，且均执行同一函数。该函数执行一个循环，重复以下步骤：将 glob 复制到本地变量 loc 中，然后递增 loc，再把 loc 复制回 glob，以此不断增加全局变量 glob 的值。因为 loc 是分配于线程栈中的自动变量（automatic variable），所以每个线程都有一份。循环重复的次数要么由命令行参数指定，要么取默认值。

程序清单 30-1：两线程以错误方式递增全局变量的值

threads/thread_incr.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;

static void *                    /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;
```


更多资源请访问稀酷客(www.ckook.com)

```
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

threads/thread_incr.c

运行程序清单 30-1 中的示例，并指定每个线程均对该变量递增 1000 次，看起来一切正常。

```
$ ./thread_incr 1000
glob = 2000
```

不过，很有可能会发生如下情况：在线程某乙尚未得以运行时，线程某甲已经执行完毕并且退出了。如果加大每个线程的工作量，结果将完全不同。

```
$ ./thread_incr 10000000
glob = 16517656
```

执行到最后，glob 的值本应为 2000 万。问题的原因是由于如下的执行序列（参见图 30-1）。

1. 线程 1 将 glob 值赋给局部变量 loc。假设 blog 的当前值为 2000。
2. 线程 1 的时间片期满，线程 2 开始执行。
3. 线程 2 执行多次循环：将全局变量 glob 的值置于局部变量 loc，递增 loc，再将结果写回变量 glob。第 1 次循环时，glob 的值为 2000。假设线程 2 的时间片到期时，glob 的值已经增至 3000。
4. 线程 1 获得另一时间片，并从上次停止处恢复执行。线程 1 在上次运行时，已将 glob 的值（2000）赋给 loc，现在递增 loc，再将 loc 的值 2001 写回 glob。此时，线程 2 此前递增操作的结果遭到覆盖。

如果使用同样的命令行参数将该程序运行多次，glob 的值会波动很大：

更多资源请访问稀酷客(www.ckook.com)

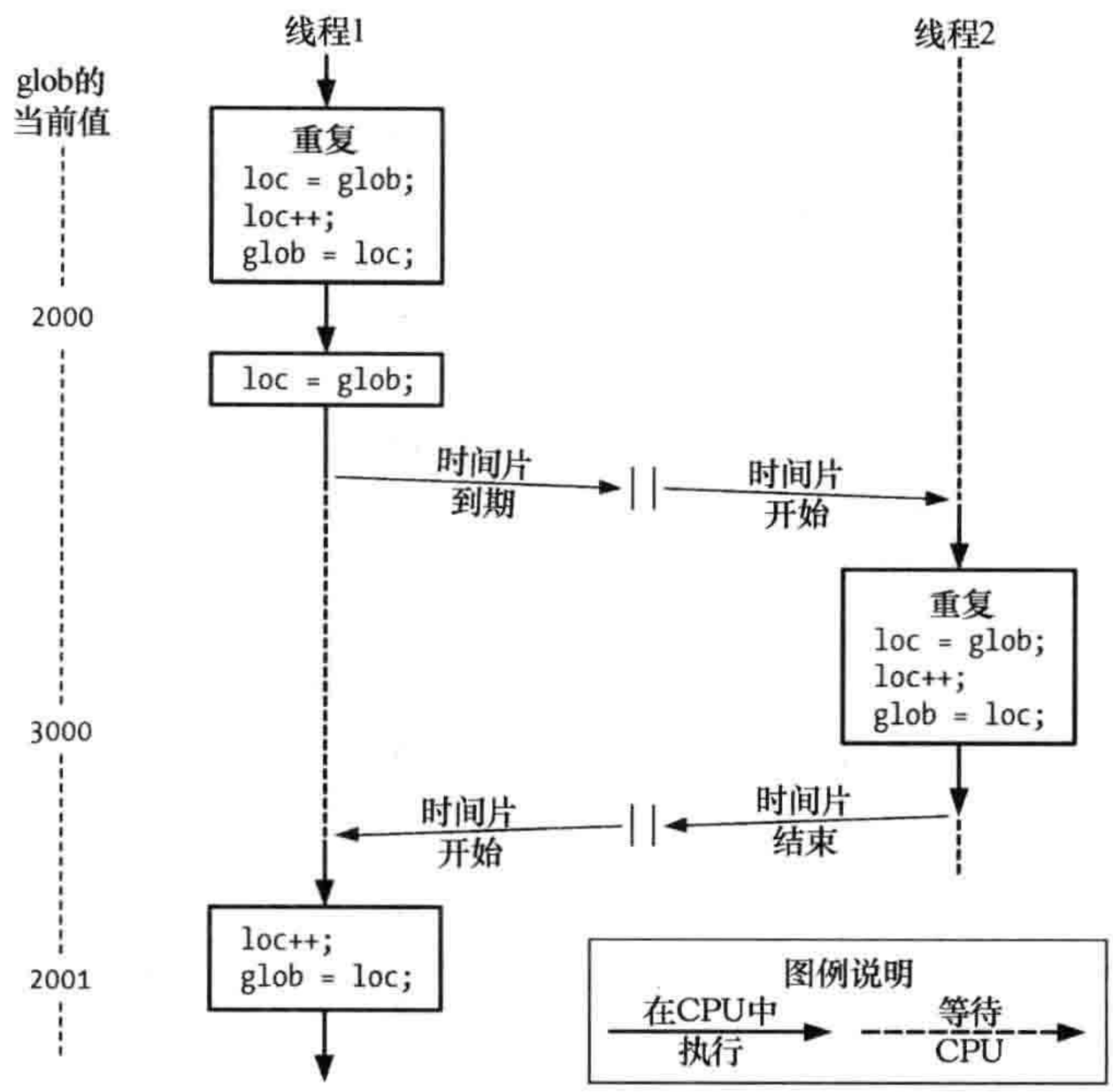

图 30-1：两个线程不使用同步技术递增全局变量的值

```
$ ./thread_incr 10000000
glob = 10880429
$ ./thread_incr 10000000
glob = 13493953
```

这一行为的不确定性，实应归咎于内核 CPU 调度决定的难以预见。若在复杂程序中发生这一不确定行为，则意味着此类错误将偶尔发作，难以重现，因此也很难发现。

使用如下语句，将程序清单 30-1 中函数 threadFunc()内 for 循环中的 3 条语句加以替换，似乎可以解决这一问题：

```
glob++;              /* or: ++glob; */
```

不过，在很多硬件架构上（例如，RISC 系统），编译器依然会将这条语句转换成机器码，其执行步骤仍旧等同于 threadFunc 循环内的 3 条语句。换言之，尽管 C 语言的递增符看似简单，其操作也未必就属于原子操作，依然可能发生上述行为。

为避免线程更新共享变量时所出现问题，必须使用互斥量（mutex 是 mutual exclusion 的缩写）来确保同时仅有一个线程可以访问某项共享资源。更为全面的说法是，可以使用互斥量来保证对任意共享资源的原子访问，而保护共享变量是其最常见的用法。

互斥量有两种状态：已锁定（locked）和未锁定（unlocked）。任何时候，至多只有一个线程可以锁定该互斥量。试图对已经锁定的某一互斥量再次加锁，将可能阻塞线程或者报错失败，具体取决于加锁时使用的方法。

一旦线程锁定互斥量，随即成为该互斥量的所有者。只有所有者才能给互斥量解锁。这一属性改善了使用互斥量的代码结构，也顾及到对互斥量实现的优化。因为所有权的关系，有时会使用术语获取（acquire）和释放（release）来替代加锁和解锁。

一般情况下，对每一共享资源（可能由多个相关变量组成）会使用不同的互斥量，每一线程在访问同一资源时将采用如下协议。

- 针对共享资源锁定互斥量。

更多资源请访问稀酷客(www.ckook.com)

- 访问共享资源。
- 对互斥量解锁。

如果多个线程试图执行这一代码块（一个临界区），事实上只有一个线程能够持有该互斥量（其他线程将遭到阻塞），即同时只有一个线程能够进入这段代码区域，如图 30-2 所示。

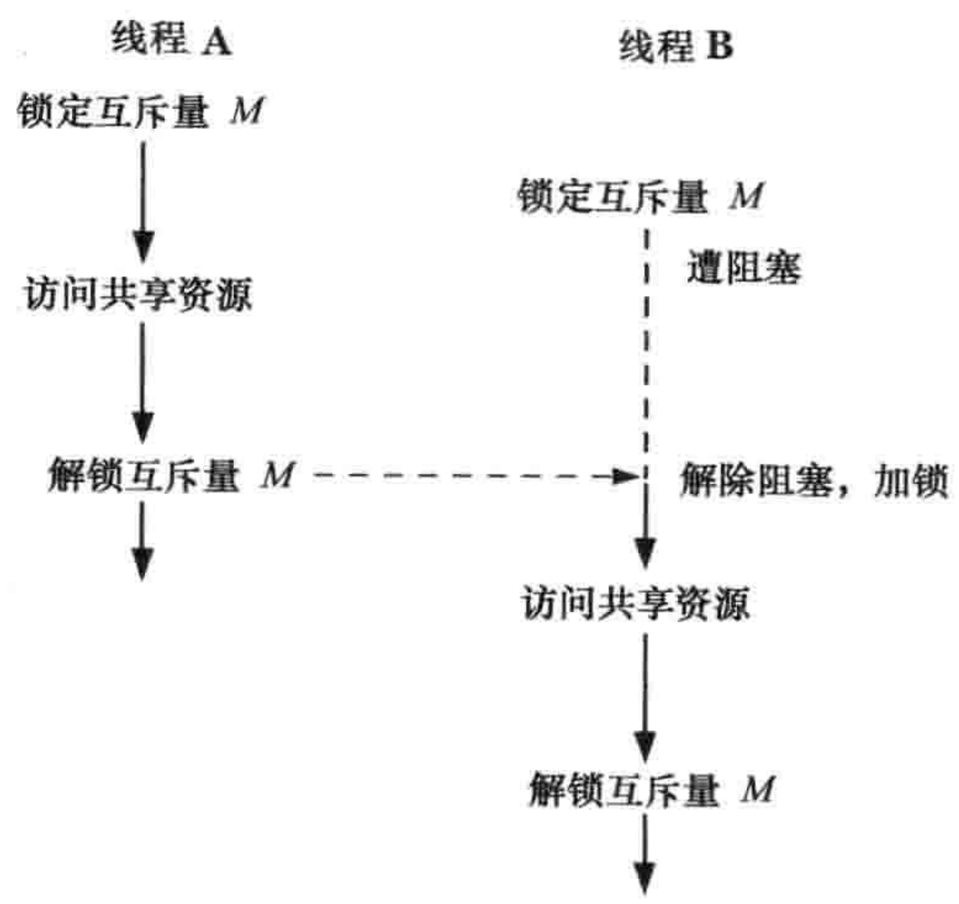

图 30-2：使用互斥量来保护临界区

最后请注意，使用互斥锁仅是一种建议，而非强制。亦即，线程可以考虑不使用互斥量而仅访问相应的共享变量。为了安全地处理共享变量，所有线程在使用互斥量时必须互相协调，遵守既定的锁定规则。

30.1.1 静态分配的互斥量

互斥量既可以像静态变量那样分配，也可以在运行时动态创建（例如，通过 malloc()在一块内存中分配）。动态互斥量的创建稍微有些复杂，将延后至 30.1.5 节再做讨论。

互斥量是属于 pthread_mutex_t 类型的变量。在使用之前必须对其初始化。对于静态分配的互斥量而言，可如下例所示，将 PTHREAD_MUTEX_INITIALIZER 赋给互斥量。

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

依照 SUSv3 的规定，对某一互斥量的副本（copy）执行本节（30.1 节）后续所述的操作将导致未定义的结果。此类操作只能施之于如下两类互斥量的“真身”，经由 PTHREAD_MUTEX_INITIALIZER 初始化的静态互斥量或者经由 pthrad_mutex_init()（在 30.1.5 节讨论）初始化的动态互斥量。

30.1.2 加锁和解锁互斥量

初始化之后，互斥量处于未锁定状态。函数 pthread_mutex_lock()可以锁定某一互斥量，而函数 pthread_mutex_unlock()则可以将一个互斥量解锁。

```
#include <pthread.h>

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

Both return 0 on success, or a positive error number on error

更多资源请访问稀酷客(www.ckook.com)

要锁定互斥量，在调用 pthread_mutex_lock()时需要指定互斥量。如果互斥量当前处于未锁定状态，该调用将锁定互斥量并立即返回。如果其他线程已经锁定了这一互斥量，那么 pthread_mutex_lock()调用会一直堵塞，直至该互斥量被解锁，到那时，调用将锁定互斥量并返回。

如果发起 pthread_mutex_lock()调用的线程自身之前已然将目标互斥量锁定，对于互斥量的默认类型而言，可能会产生两种后果——视具体实现而定：线程陷入死锁（deadlock），因试图锁定已为自己所持有的互斥量而遭到阻塞；或者调用失败，返回 EDEADLK 错误。在 Linux 上，默认情况下线程会发生死锁。（30.1.7 节在讨论互斥量类型时会述及一些其他的可能行为。）

函数 pthread_mutex_unlock()将解锁之前已遭调用线程锁定的互斥量。以下行为均属错误：对处于未锁定状态的互斥量进行解锁，或者解锁由其他线程锁定的互斥量。

如果有不止一个线程在等待获取由函数 pthread_mutex_unlock()解锁的互斥量，则无法判断究竟哪个线程将如愿以偿。

示例程序

程序清单 30-2 是对程序清单 30-1 的修改，使用了一个互斥量来保护对全局变量 glob 的访问。使用与之前类似的命令行来运行这个改版程序，可以看到对 glob 的累加总是能够保持正确。

```
$ ./thread_incr_mutex 10000000
glob = 20000000
```

程序清单 30-2：使用互斥量保护对全局变量的访问

threads/thread_incr_mutex.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                    /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
```

```
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

threads/thread_incr_mutex.c

pthread_mutex_trylock()和 pthread_mutex_timedlock()

Pthreads API 提供了 pthread_mutex_lock()函数的两个变体：pthread_mutex_trylock()和 pthread_mutex_timedlock()。可参考手册页（manual page）获取这些函数的原型。

如果信号量已然锁定，对其执行函数 pthread_mutex_trylock()会失败并返回 EBUSY 错误，除此之外，该函数与 pthread_mutex_lock()行为相同。

除了调用者可以指定一个附加参数 abstime（设置线程等待获取互斥量时休眠的时间限制）外，函数 pthread_mutex_timedlock()与 pthread_mutex_lock()没有差别。如果参数 abstime 指定的时间间隔期满，而调用线程又没有获得对互斥量的所有权，那么函数 pthread_mutex_timedlock()返回 ETIMEDOUT 错误。

函数 pthread_mutex_trylock() 和 pthread_mutex_timedlock()比 pthread_mutex_lock()的使用频率要低很多。在大多数经过良好设计的应用程序中，线程对互斥量的持有时间应尽可能短，以避免妨碍其他线程的并发执行。这也保证了遭堵塞的其他线程可以很快获取对互斥量的锁定。若某一线程使用 pthread_mutex_trylock()周期性地轮询是否可以对互斥量加锁，则有可能要承担这样的风险：当队列中的其他线程通过调用 pthread_mutex_lock()相继获得对互斥量的访问时，该线程将始终与此互斥量无缘。

30.1.3 互斥量的性能

使用互斥量的开销有多大？前面已经展示了递增共享变量程序的两个不同版本：没有使用互斥量的程序清单 30-1 和使用互斥量的程序清单 30-2。在 x86-32 架构的 Linux 2.6.31（含 NPTL）系统下运行这两个程序，如令单一线程循环 1000 万次，前者共花费了 0.35 秒（并产生错误结果），而后者则需要 3.1 秒。

乍看起来，代价极高。不过，考虑一下前者（程序清单 30-1）执行的主循环。在该版本中，函数 threadFunc()于 for 循环中，先递增循环控制变量，再将其与另一变量进行比较，随后执行两个复制操作和一个递增操作，最后返回循环起始处开始下一次循环。而后者——使用互斥量

更多资源请访问稀酷客(www.ckook.com)

的版本（程序清单 30-2）执行了相同步骤，不过在每次循环的前后多了加锁和解锁互斥量的工作。换言之，对互斥量加锁和解锁的开销略低于第 1 个程序的 10 次循环操作。成本相对比较低廉。此外，在通常情况下，线程会花费更多时间去做其他工作，对互斥量的加锁和解锁操作相对要少得多，因此使用互斥量对于大部分应用程序的性能并无显著影响。

进而言之，在相同系统上运行一些简单的测试程序，结果显示，如将使用函数 fcntl()（见 55.3 节）加锁、解锁一片文件区域的代码循环 2000 万次，需耗时 44 秒，而将对系统 V 信号量（semaphore）（见 47 章）的递增和递减代码循环 2000 万次，则需要 28 秒。文件锁和信号量的问题在于，其锁定和解锁总是需要发起系统调用（system call），而每个系统调用的开销虽小，但颇为可观（见 3.1 节）。与之相反，互斥量的实现采用了机器语言级的原子操作（在内存中执行，对所有线程可见），只有发生锁的争用时才会执行系统调用。

Linux 上，互斥量的实现采用了 futex（源自“快速用户空间互斥量”[fast user space mutex]的首字母缩写），而对锁争用的处理则使用了 futex()系统调用。本书无意描述 futex，其设计意图也并非供用户空间（user space）应用程序直接使用，不过[Drepper, 2004(a)]给出了详细描述，还讨论了如何使用 futexes 来实现互斥量。[Franke et al., 2002]是一篇由 futex 开发人员所撰写的论文（已经过时），介绍了 futex 的早期实现以及因其所带来的性能提升。

30.1.4 互斥量的死锁

有时，一个线程需要同时访问两个或更多不同的共享资源，而每个资源又都由不同的互斥量管理。当超过一个线程加锁同一组互斥量时，就有可能发生死锁。图 30-3 展示了一个死锁的例子，其中每个线程都成功地锁住一个互斥量，接着试图对已为另一线程锁定的互斥量加锁。两个线程将无限期地等待下去。

线程 A	线程 B
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);*	2. *pthread_mutex_lock(mutex1);*
阻塞	阻塞

图 30-3：两个线程分别锁定两个互斥量所导致的死锁

要避免此类死锁问题，最简单的方法是定义互斥量的层级关系。当多个线程对一组互斥量操作时，总是应该以相同顺序对该组互斥量进行锁定。例如，在图 30-3 所示场景中，如果两个线程总是先锁定 mutex1 再锁定 mutex2，死锁就不会出现。有时，互斥量间的层级关系逻辑清晰。不过，即便没有，依然可以设计出所有线程都必须遵循的强制层级顺序。

另一种方案的使用频率较低，就是“尝试一下，然后恢复”。在这种方案中，线程先使用函数 pthread_mutex_lock()锁定第 1 个互斥量，然后使用函数 pthread_mutex_trylock()来锁定其余互斥量。如果任一 pthread_mutex_trylock()调用失败（返回 EBUSY），那么该线程将释放所有互斥量，也许经过一段时间间隔，从头再试。较之于按锁的层级关系来规避死锁，这种方法效率要低一些，因为可能需要历经多次循环。另一方面，由于无需受制于严格的互斥量层级关系，该方法也更为灵活。[Butenhof, 1996]中载有这一方案的范例。

30.1.5 动态初始化互斥量

静态初始值 PTHREAD_MUTEX_INITIALIZER，只能用于对如下互斥量进行初始化：经由静态分配且携带默认属性。其他情况下，必须调用 pthread_mutex_init()对互斥量进行动态初始化。

更多资源请访问稀酷客(www.ckook.com)

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

参数 mutex 指定函数执行初始化操作的目标互斥量。参数 attr 是指向 pthread_mutexattr_t 类型对象的指针，该对象在函数调用之前已经过了初始化处理，用于定义互斥量的属性。（下节会介绍更多互斥量属性。）若将 attr 参数置为 NULL，则该互斥量的各种属性会取默认值。

SUSv3 规定，初始化一个业已初始化的互斥量将导致未定义的行为，应当避免这一行为。

在如下情况下，必须使用函数 pthread_mutex_init()，而非静态初始化互斥量。

- 动态分配于堆中的互斥量。例如，动态创建针对某一结构的链表，表中每个结构都包含一个 pthread_mutex_t 类型的字段来存放互斥量，借以保护对该结构的访问。
- 互斥量是在栈中分配的自动变量。
- 初始化经由静态分配，且不使用默认属性的互斥量。

当不再需要经由自动或动态分配的互斥量时，应使用 pthread_mutex_destroy()将其销毁。（对于使用 PTHREAD_MUTEX_INITIALIZER 静态初始化的互斥量，无需调用 pthread_mutex_destroy()。）

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Returns 0 on success, or a positive error number on error

只有当互斥量处于未锁定状态，且后续也无任何线程企图锁定它时，将其销毁才是安全的。若互斥量驻留于动态分配的一片内存区域中，应在释放（free）此内存区域前将其销毁。对于自动分配的互斥量，也应在宿主函数返回前将其销毁。

经由 pthread_mutex_destroy()销毁的互斥量，可调用 pthread_mutex_init()对其重新初始化。

30.1.6 互斥量的属性

如前所述，可以在 pthread_mutex_init()函数的 arg 参数中指定 pthread_mutexattr_t 类型对象，对互斥量的属性进行定义。通过 pthread_mutexattr_t 类型对象对互斥量属性进行初始化和读取操作的 Pthreads 函数有多个。本书不打算深入讨论互斥量属性的细节，也不会将初始化 pthread_mutexattr_t 对象内属性的各种函数原型一一列出。不过，下一节会讨论互斥量的属性之一：类型。

30.1.7 互斥量类型

前面几页对互斥量的行为做了若干论述。

- 同一线程不应对同一互斥量加锁两次。
- 线程不应对不为自己所拥有的互斥量解锁（亦即，尚未锁定互斥量）。
- 线程不应对一尚未锁定的互斥量做解锁动作。

准确地说，上述情况的结果将取决于互斥量类型（type）。SUSv3 定义了以下互斥量类型。

PTHREAD_MUTEX_NORMAL

该类型的互斥量不具有死锁检测（自检）功能。如线程试图对已由自己锁定的互斥量加

更多资源请访问稀酷客(www.ckook.com)

锁，则发生死锁。互斥量处于未锁定状态，或者已由其他线程锁定，对其解锁会导致不确定的结果。（在 Linux 上，对这类互斥量的上述两种操作都会成功。）

PTHREAD_MUTEX_ERRORCHECK

对此类互斥量的所有操作都会执行错误检查。所有上述 3 种情况都会导致相关 Pthreads 函数返回错误。这类互斥量运行起来比一般类型要慢，不过可将其作为调试工具，以发现程序在哪里违反了互斥量使用的基本原则。

PTHREAD_MUTEX_RECURSIVE

递归互斥量维护有一个锁计数器。当线程第 1 次取得互斥量时，会将锁计数器置 1。后续由同一线程执行的每次加锁操作会递增锁计数器的数值，而解锁操作则递减计数器计数。只有当锁计数器值降至 0 时，才会释放（release，亦即可为其他线程所用）该互斥量。解锁时如目标互斥量处于未锁定状态，或是已由其他线程锁定，操作都会失败。

Linux 的线程实现针对以上各种类型的互斥量提供了非标准的静态初始值（例如，PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP），以便对那些通过静态分配的互斥量进行初始化，而无需使用 pthread_mutex_init()函数。不过，为保证程序的可移植性，应该避免使用这些初始值。

除了上述类型，SUSv3 还定义了 PTHREAD_MUTEX_DEFAULT 类型。使用 PTHREAD_MUTEX_INITIALIZER 初始化的互斥量，或是经调用参数 attr 为 NULL 的 pthread_mutex_init()函数所创建的互斥量，都属于此类型。至于该类型互斥量在本节开始处 3 个场景中的行为，规范有意未作定义，意在为互斥量的高效实现保留最大的灵活性。Linux 上，PTHREAD_MUTEX_DEFAULT 类型互斥量的行为与 PTHREAD_MUTEX_NORMAL 类型相仿。

程序清单 30-3 演示了如何设置互斥量类型，本例创建了一个带有错误检查属性（error-checking）的互斥量。

程序清单 30-3：设置互斥量类型

```
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
int s, type;

s = pthread_mutexattr_init(&mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_init");
s = pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_ERRORCHECK);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_settype");

s = pthread_mutex_init(mtx, &mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutex_init");

s = pthread_mutexattr_destroy(&mtxAttr);        /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_mutexattr_destroy");
```

30.2 通知状态的改变：条件变量（Condition Variable）

互斥量防止多个线程同时访问同一共享变量。条件变量允许一个线程就某个共享变量（或

更多资源请访问稀酷客(www.ckook.com)

其他共享资源）的状态变化通知其他线程，并让其他线程等待（堵塞于）这一通知。

一个未使用条件变量的简单例子有助于展示条件变量的重要性。假设由若干线程生成一些“产品单元（result unit）”供主线程消费。还使用了一个由互斥量保护的变量 avail 来代表待消费产品的数量：

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static int avail = 0;
```

本节引用的代码片段摘自于随本书发布的源代码文件 threads/prod_no_condvar.c。

生产者线程的源代码如下：

```
/* Code to produce a unit omitted */

s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

avail++;    /* Let consumer know another unit is available */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");
```

主线程（消费者）的代码如下：

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail > 0) {         /* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}
```

上述代码虽然可行，但由于主线程不停地循环检查变量 avail 的状态，故而造成 CPU 资源的浪费。采用了条件变量（condition variable），这一问题就迎刃而解：允许一个线程休眠（等待）直至接获另一线程的通知（收到信号）去执行某些操作（例如，出现一些“情况”后，等待者必须立即做出响应）。

条件变量总是结合互斥量使用。条件变量就共享变量的状态改变发出通知，而互斥量则提供对该共享变量访问的互斥（mutual exclusion）。这里使用的术语“信号”（signal），与第 20 章至第 22 章所述信号（signal）无关，而是发出信号的意思。

30.2.1 由静态分配的条件变量

如同互斥量一样，条件变量的分配，有静态和动态之分。条件变量的动态创建延后到 30.2.5 节再行描述，这里先讨论一下静态分配。

条件变量的数据类型是 pthread_count_t。类似于互斥量，使用条件变量前必须对其初始化。对于经由静态分配的条件变量，将其赋值为 PTHREAD_COND_INITALIZER 即完成初始化操作。可参考下面的例子：

更多资源请访问稀酷客(www.ckook.com)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

依据 SUSv3 规定，将本节后续所描述的操作施之于一个条件变量的副本（copy）时，其结果未定义。所有操作仅能针对条件变量的原本执行，要么经由 PTHREAD_COND_INITIALIZE 进行了静态初始化，要么使用 pthread_cond_init()做了动态初始化（30.2.5 节描述）处理。

30.2.2 通知和等待条件变量

条件变量的主要操作是发送信号（signal）和等待（wait）。发送信号操作即通知一个或多个处于等待状态的线程，某个共享变量的状态已经改变。等待操作是指在收到一个通知前一直处于阻塞状态。

函数 pthread_cond_signal()和 pthread_cond_broadcast()均可针对由参数 cond 所指定的条件变量而发送信号。pthread_cond_wait()函数将阻塞一线程，直至收到条件变量 cond 的通知。

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

All return 0 on success, or a positive error number on error

函数 pthread_cond_signal()和 pthread_cond_broadcast()之间的差别在于，二者对阻塞于 pthread_cond_wait()的多个线程处理方式不同。pthread_cond_signal()函数只保证唤醒至少一条遭到阻塞的线程，而 pthread_cond_broadcast()则会唤醒所有遭阻塞的线程。

使用函数 pthread_cond_broadcast()总能产生正确结果（因为所有线程应都能处理多余和虚假的唤醒动作），但函数 pthread_cond_signal()会更为高效。不过，只有当仅需唤醒一条（且无论是其中哪条）等待线程来处理共享变量的状态变化时，才应使用 pthread_cond_signal()。应用这种方式的典型情况是，所有等待线程都在执行完全相同的任务。基于这些假设，函数 pthread_cond_signal()会比 pthread_cond_broadcast()更具效率，因为这可以避免发生如下情况。

1. 同时唤醒所有等待线程。
2. 某一线程首先获得调度。此线程检查了共享变量的状态（在相关互斥量的保护之下），发现还有任务需要完成。该线程执行了所需工作，并改变共享变量状态，以表明任务完成，最后释放对相关互斥量的锁定。
3. 剩余的每个线程轮流锁定互斥量并检测共享变量的状态。不过，由于第一个线程所做的工作，余下的线程发现无事可做，随即解锁互斥量转而休眠（即再次调用 pthread_cond_wait()）。

相形之下，函数 pthread_cond_broadcast()所处理的情况是：处于等待状态的所有线程执行的任务不同（即各线程关联于条件变量的判定条件不同）。

条件变量并不保存状态信息，只是传递应用程序状态信息的一种通讯机制。发送信号时若无任何线程在等待该条件变量，这个信号也就会不了了之。线程如在此后等待该条件变量，只有当再次收到此变量的下一信号时，方可解除阻塞状态。

函数 pthread_cond_timedwait()与函数 pthread_cond_wait()几近相同，唯一的区别在于，由参数 abstime 来指定一个线程等待条件变量通知时休眠时间的上限。

```
#include <pthread.h>
```


更多资源请访问稀酷客(www.ckook.com)

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

Returns 0 on success, or a positive error number on error

参数 abstime 是一个 timespec 类型的结构（见 23.4.2 节），用以指定自 Epoch（参考 10.1 节）以来以秒和纳秒（nanosecond）为单位表示的绝对（absolute）时间。如果 abstime 指定的时间间隔到期且无相关条件变量的通知，则返回 ETIMEOUT 错误。

在生产者-消费者（producer-consumer）示例中使用条件变量

下面对前面的示例作出修改，引入条件变量。对全局变量、相关互斥量以及条件变量的声明代码如下：

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int avail = 0;
```

本节中的代码片段摘自随本书发布的源代码文件 threads/prod_condvar.c。

除了增加对函数 pthread_cond_signal()的调用外，生产者线程的代码与之前并无变化：

```
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

avail++;                /* Let consumer know another unit is available */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

s = pthread_cond_signal(&cond);         /* Wake sleeping consumer */
if (s != 0)
    errExitEN(s, "pthread_cond_signal");
```

在分析消费者代码之前，需要对 pthread_cond_wait()函数做更为详细的解释。前文已经指出，条件变量总是要与一个互斥量相关。将这些对象通过函数参数传递给 pthread_cond_wait()，后者执行如下操作步骤。

- 解锁互斥量 mutex。
- 堵塞调用线程，直至另一线程就条件变量 cond 发出信号。
- 重新锁定 mutex。

设计 pthread_cond_wait()执行上述步骤，是因为通常情况下代码会以如下方式访问共享变量：

```
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

while (/* Check that shared variable is not in state we want */)
    pthread_cond_wait(&cond, &mtx);

/* Now shared variable is in desired state; do some work */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");
```


更多资源请访问棒酷客(www.ckook.com)

下一节将会介绍为何将 pthread_cond_wait()调用置于 while 循环中，而非 if 语句中。

在以上代码中，两处对共享变量的访问都必须置于互斥量的保护之下，其原因之前已做了解释。换言之，条件变量与互斥量之间存在着天然的关联关系。

1. 线程在准备检查共享变量状态时锁定互斥量。
2. 检查共享变量的状态。
3. 如果共享变量未处于预期状态，线程应在等待条件变量并进入休眠前解锁互斥量（以便其他线程能访问该共享变量）。
4. 当线程因为条件变量的通知而被再度唤醒时，必须对互斥量再次加锁，因为在典型情况下，线程会立即访问共享变量。

函数 pthread_cond_wait()会自动执行最后两步中对互斥量的解锁和加锁动作。第 3 步中互斥量的释放与陷入对条件变量的等待同属于一个原子操作。换句话说，在函数 pthread_cond_wait()的调用线程陷入对条件变量的等待之前，其他线程不可能获取到该互斥量，也不可能就该条件变量发出信号。

> 通过观察得出推论：条件变量与互斥量之间存在天然关系，同时等待相同条件变量的所有线程在调用 pthread_cond_wait()或 pthread_cond_timedwait()时必须指定同一互斥量。实际上，pthread_cond_wait()在调用期间能将条件变量与一个唯一的互斥量做动态绑定。SUSv3 规定，在针对同一条件变量并发调用 pthread_cond_wait()时，若使用多个互斥量会导致未定义的结果。

结合以上所有细节，使用 pthread_cond_wait()修改主（消费者）线程的代码如下：

```
for (;;) {
    s = pthread_mutex_lock(&mtx);

    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail == 0) {            /* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
    }

    while (avail > 0) {             /* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

    /* Perhaps do other work here that doesn't require mutex lock */
}
```

最后，再看一下 pthread_cond_signal()和 pthread_cond_broadcast()的使用。前面展示的生产者代码先调用了 pthread_mutex_unlock()，接着调用了 pthread_cond_signal()；换言之，先解锁与共享变量相关的互斥量，再就对应的条件变量发出信号。也可以将这两步颠倒执行，SUSv3 允许以任意顺序执行这两个调用。

更多资源请访问稀酷客(www.ckook.com)

[Butenhof, 1996]指出，在某些实现中，先解锁互斥量再通知条件变量可能比反序执行效率要高。如果仅在发出条件变量信号后才解锁互斥量，执行 pthread_cond_wait()调用的线程可能会在互斥量仍处于加锁状态时就醒来，当其发现互斥量仍未解锁，会立即再次休眠。这会导致两个多余的上下文切换（context switch）。有些实现运用等待变形（wait morphing）技术解决了这一问题：将等待接收信号的线程从条件变量的等待队列转移至互斥量等待队列。这样，即便互斥量处于加锁状态，也无需切换上下文。

30.2.3 测试条件变量的判断条件（predicate）

每个条件变量都有与之相关的判断条件，涉及一个或多个共享变量。例如，在上一节的代码中，与 cond 相关的判断是(avail == 0)。这段代码展示了一个通用的设计原则：必须由一个 while 循环，而不是 if 语句，来控制对 pthread_cond_wait()的调用。这是因为，当代码从 pthread_cond_wait()返回时，并不能确定判断条件的状态，所以应该立即重新检查判断条件，在条件不满足的情况下继续休眠等待。

从 pthread_cond_wait()返回时，之所以不能对判断条件的状态做任何假设，其理由如下。

- 其他线程可能会率先醒来。也许有多个线程在等待获取与条件变量相关的互斥量。即使就互斥量发出通知的线程将判断条件置为预期状态，其他线程依然有可能率先获取互斥量并改变相关共享变量的状态，进而改变判断条件的状态。
- 设计时设置“宽松的”判断条件或许更为简单。有时，用条件变量来表征可能性而非确定性，在设计应用程序时会更为简单。换言之，就条件变量发送信号意味着“可能有些事情”需要接收信号的线程去响应，而不是“一定有一些事情”要做。使用这种方法，可以基于判断条件的近似情况来发送条件变量通知，接收信号的线程可以通过再次检查判断条件来确定是否真的需要做些什么。
- 可能会发生虚假唤醒的情况。在一些实现中，即使没有任何其他线程真地就条件变量发出信号，等待此条件变量的线程仍有可能醒来。在一些多处理器系统上，为确保高效实现而采用的技术会导致此类（不常见的）虚假唤醒。SUSv3 对此予以明确认可。

30.2.4 示例程序：连接任意已终止线程

前面已然提及，使用 pthread_join()只能连接一个指定线程。且该函数也未提供任何机制去连接任意的已终止线程。本节展示如何使用条件变量绕过这一限制。

程序清单 30-4 为其每个命令行参数创建一个线程。每个线程休眠一段时间后随即退出，休眠时间由相应命令行参数所指定的秒数决定。这里用休眠间隔来模拟线程工作了一段时间。

该程序维护有一组全局变量，记录了所有已创建线程的信息。对于每个线程，全局数组 thread 中都含有一元素记录其线程 ID（字段 tid）以及当前状态（字段 state）。状态字段 state 可设置为以下值：TS_ALIVE，表示线程是活动的；TS_TERMINATED，代表线程已经终结但尚未被连接；TS_JOINED，表示线程终止且已被连接。

当线程终止时，将 TS_TERMINATED 赋给数组 thread 中对应元素的 state 字段，对表征已终止但尚未连接线程的全局计数器（numUnjoined）加一，并就条件变量 threadDied 发出信号。

主线程使用循环不断等待条件变量 threadDied。当收到 threadDied 信号，且存在已终止线程尚未被连接时，主线程将扫描 thread 数组，寻找 state 为 TS_TERMINATED 的数组元素。对

更多资源请访问稀酷客(www.ckook.com)

处于该状态的每个线程，以数组 thread 中的对应 tid 字段调用 pthread_join()函数，并将 state 置为 TS_JOINED。当由主线程创建的所有线程终止时，即全局变量 numLive 值为 0 时，主循环结束。

以下 shell 会话日志展示了对程序清单 30-4 中程序的调用：

```
$ ./thread_multijoin 1 1 2 3 3          Create 5 threads
Thread 0 terminating
Thread 1 terminating
Reaped thread 0 (numLive=4)
Reaped thread 1 (numLive=3)
Thread 2 terminating
Reaped thread 2 (numLive=2)
Thread 3 terminating
Thread 4 terminating
Reaped thread 3 (numLive=1)
Reaped thread 4 (numLive=0)
```

最后要指出，虽然示例中的线程都被创建为处于可连接状态，且终止后立即由 pthread_join()予以捕获，其实无需采用这一方法来发现线程的终止。可以将线程置为分离态（detached），无需使用 pthread_join()，简单地利用 thread 数组（及其他相关全局变量）作为记录每个线程终结的手段。

程序清单 30-4：可以连接任意已终止线程的主线程

threads/thread_multijoin.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_cond_t threadDied = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t threadMutex = PTHREAD_MUTEX_INITIALIZER;
                /* Protects all of the following global variables */

static int totThreads = 0;      /* Total number of threads created */
static int numLive = 0;         /* Total number of threads still alive or
                                   terminated but not yet joined */
static int numUnjoined = 0;     /* Number of terminated threads that
                                   have not yet been joined */
enum tstate {                   /* Thread states */
    TS_ALIVE,                   /* Thread is alive */
    TS_TERMINATED,              /* Thread terminated, not yet joined */
    TS_JOINED                   /* Thread terminated, and joined */
};

static struct {                 /* Info about each thread */
    pthread_t tid;              /* ID of this thread */
    enum tstate state;          /* Thread state (TS_* constants above) */
    int sleepTime;              /* Number seconds to live before terminating */
} *thread;

static void *                   /* Start function for thread */
threadFunc(void *arg)
{
    int idx = *((int *) arg);
    int s;
    sleep(thread[idx].sleepTime);       /* Simulate doing some work */
    printf("Thread %d terminating\n", idx);

    s = pthread_mutex_lock(&threadMutex);
    if (s != 0)
```

```
        errExitEN(s, "pthread_mutex_lock");

    numUnjoined++;
    thread[idx].state = TS_TERMINATED;

    s = pthread_mutex_unlock(&threadMutex);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
    s = pthread_cond_signal(&threadDied);
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");

    return NULL;
}

int
main(int argc, char *argv[])
{
    int s, idx;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s nsecs...\n", argv[0]);

    thread = calloc(argc - 1, sizeof(*thread));
    if (thread == NULL)
        errExit("calloc");

    /* Create all threads */

    for (idx = 0; idx < argc - 1; idx++) {
        thread[idx].sleepTime = getInt(argv[idx + 1], GN_NONNEG, NULL);
        thread[idx].state = TS_ALIVE;
        s = pthread_create(&thread[idx].tid, NULL, threadFunc, &idx);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    totThreads = argc - 1;
    numLive = totThreads;

    /* Join with terminated threads */

    while (numLive > 0) {
        s = pthread_mutex_lock(&threadMutex);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
        while (numUnjoined == 0) {
            s = pthread_cond_wait(&threadDied, &threadMutex);
            if (s != 0)
                errExitEN(s, "pthread_cond_wait");
        }

        for (idx = 0; idx < totThreads; idx++) {
            if (thread[idx].state == TS_TERMINATED){
                s = pthread_join(thread[idx].tid, NULL);
                if (s != 0)
                    errExitEN(s, "pthread_join");

                thread[idx].state = TS_JOINED;
```

```
                numLive--;
                numUnjoined--;

                printf("Reaped thread %d (numLive=%d)\n", idx, numLive);
            }
        }

        s = pthread_mutex_unlock(&threadMutex);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    exit(EXIT_SUCCESS);
}
```

threads/thread_multijoin.c

30.2.5 经由动态分配的条件变量

使用函数 pthread_cond_init()对条件变量进行动态初始化。需要使用 pthread_cond_init()的情形类似于使用 pthread_mutex_init()来动态初始化互斥量的情况。亦即，对自动或动态分配的条件变量进行初始化时，或是对未采用默认属性经由静态分配的条件变量进行初始化时，必须使用 pthread_cond_init()。

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

参数 cond 表示将要初始化的目标条件变量。类似于互斥量，可以指定之前经由初始化处理的 attr 参数来判定条件变量的属性。对于 attr 所指向的 pthread_condattr_t 类型对象，可使用多个 Pthreads 函数对其中属性进行初始化。若将 attr 置为 NULL，则使用一组缺省属性来设置条件变量。

SUSv3 规定，对业已初始化的条件变量进行再次初始化，将导致未定义的行为。应当避免这一做法。

当不再需要一个经由自动或动态分配的条件变量时，应调用 pthread_cond_destroy()函数予以销毁。对于使用 PTHREAD_COND_INITIALIZER 进行静态初始化的条件变量，无需调用 pthread_cond_destroy()。

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

Returns 0 on success, or a positive error number on error

对某个条件变量而言，仅当没有任何线程在等待它时，将其销毁才是安全的。如果条件变量驻留于某片动态创建的内存区域，那么应在释放该内存区域前就将其销毁。经由自动分配的条件变量应在宿主函数返回前予以销毁。

经 pthread_cond_destroy()销毁的条件变量，之后可以调用 pthread_cond_init()对其进行重新初始化。

更多资源请访问稀酷客(www.ckook.com)

30.3 总结

线程提供的强大共享是有代价的。多线程应用程序必须使用互斥量和条件变量等同步原语来协调对共享变量的访问。互斥量提供了对共享变量的独占式访问。条件变量允许一个或多个线程等候通知：其他线程改变了共享变量的状态。

更多信息

请参考 29.10 节所列的更多信息来源。

30.4 练习

30-1. 修改程序清单 30-1（thread_incr.c）中的程序，以便线程起始函数在每次循环中都能输出 glob 的当前值以及能对线程做唯一标识的标识符。可将线程的这一唯一标识指定为创建线程的函数 pthread_create()的调用参数。对于这一程序，需要将线程起始函数的参数改为指针，指向包含线程唯一标识和循环次数限制的数据结构。运行该程序，将输出重定向至一文件，查看内核在调度两线程交替执行时 glob 的变化情况。

30-2. 实现一组线程安全的函数，以更新和搜索一个不平衡二叉树。此函数库应该包含如下形式的函数（目的明显）：

```
initialize(tree);
add(tree, char *key, void *value);
delete(tree, char *key)
Boolean lookup(char *key, void **value)
```

上述函数原型中，tree 是一个指向根节点的结构（为此需要定义一个合适的结构）。树的每个节点保存有一个键-值对。还需为树中每个节点定义一数据结构，其中应包含互斥量，以确保同时仅有一个线程可以访问该节点。initialize()、add()和 lookup()函数的实现相对简单。delete()的实现需要较为深入的考虑。

> 无需维护平衡二叉树，这极大简化了实现对锁的需求，但同时也带来了风险，特定模式的输入会导致树的执行效率低下。要维护平衡二叉树，则在执行 add()和 delete()操作时必然要在子树间移动节点，这就需要更为复杂的锁定策略。

更多资源请访问稀酷客(www.ckook.com)

第 31 章

线程：线程安全和每线程存储

本章将拓展对 POSIX 线程 API 的探讨，描述线程安全（thread-safe）函数以及一次性初始化。同时讨论在不改变函数接口定义的前提下，如何通过线程特有数据（thread-specific data）或线程局部存储（thread-local storage）实现已有函数的线程安全。

31.1 线程安全（再论可重入性）

若函数可同时供多个线程安全调用，则称之为线程安全函数；反之，如果函数不是线程安全的，则不能并发调用。例如，如下函数（30.1 节也有类似代码）就不是线程安全的：

```
static int glob = 0;

static void
incr(int loops)
{
    int loc, j;

    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
}
```

如果多个线程并发调用该函数，glob 的最终值将不得而知。本例展示了导致线程不安全的典型原因：使用了在所有线程之间共享的全局或静态变量。

实现线程安全有多种方式。其一是将函数与互斥量关联使用（如果函数库中的所有函数都共享同样的全局变量，那么或许应将所有函数都与该互斥量相关联），在调用函数时将其锁定，在函数返回时解锁。这一方法的优点在于简单。另一方面，这也意味着同时只能有一个线程执行该函数，亦即，对该函数的访问是串行的（serialized）。如果各线程在执行此函数时都耗费了相当多的时间，那么串行化会导致并发能力的丧失，所有线程将不再并发执行。

另一种更为复杂的解决方案是：将共享变量与互斥量关联起来。这需要程序员们确认函数

更多资源请访问稀酷客(www.ckook.com)

的哪些部分是使用了共享变量的临界区，且仅在执行到临界区时去获取和释放互斥量。这将允许多线程同时执行一个函数并实现并行，除非出现多个线程需要同时执行同一临界区的情况。

非线程安全的函数

为便于开发多线程应用程序，除了表 31-1 所列函数以外（其中大部分并未在本书中提及），SUSv3 中的所有函数都需实现线程安全。

除了表 31-1 中所列函数，SUSv3 还做了如下规定。

- 如传参为 NULL 时，函数 ctermid()和 tmpnam()无需是线程安全的。
- 如果函数 wcrtomb()和 wcsrtombs()的最后一个参数（ps）为 NULL，那么这两个函数也无需是线程安全的。

SUSv4 对表 31-1 中的函数做了以下修改。

- 移除函数 ecvt()、fcvt()、gcvt()、gethostbyname()以及 gethostbyaddr()，因为已从标准中删除了这些函数。
- 增加函数 strsignal()和 system()。由于 system()函数就信号处置所做的操作将影响整个进程，故而是不可重入的。

标准并未禁止将表 31-1 中的函数实现为线程安全。不过，即使在某些实现中有些函数是线程安全的，为确保应用程序的可移植性，也不应该假设这些函数在所有实现中都是如此。

表 31-1：SUSv3 不要求这些函数是线程安全的

asctime()	fcvt()	getpwnam()	nl_langinfo()
basename()	ftw()	getpwuid()	ptsname()
catgets()	gcvt()	getservbyname()	putc_unlocked()
crypt()	getc_unlocked()	getservbyport()	putchar_unlocked()
ctime()	getchar_unlocked()	getservent()	putenv()
dbm_clearerr()	getdate()	getutxent()	pututxline()
dbm_close()	getenv()	getutxid()	rand()
dbm_delete()	getgrent()	getutxline()	readdir()
dbm_error()	getgrgid()	gmtime()	setenv()
dbm_fetch()	getgrnam()	hcreate()	setgrent()
dbm_firstkey()	gethostbyaddr()	hdestroy()	setkey()
dbm_nextkey()	gethostbyname()	hsearch()	setpwent()
dbm_open()	gethostent()	inet_ntoa()	setutxent()
dbm_store()	getlogin()	l64a()	strerror()
dirname()	getnetbyaddr()	lgamma()	strtok()
dlerror()	getnetbyname()	lgammaf()	ttyname()
drand48()	getnetent()	lgammal()	unsetenv()
ecvt()	getopt()	localeconv()	wcstombs()
encrypt()	getprotobyname()	localtime()	wctomb()
endgrent()	getprotobynumber()	lrand48()	
endpwent()	getprotoent()	mrand48()	
endutxent()	getpwent()	nftw()	

可重入和不可重入函数

较之于对整个函数使用互斥量，使用临界区实现线程安全虽然有明显改进，但由于存在对互斥量的加锁和解锁开销，所以多少还是有些低效。可重入函数则无需使用互斥量即可实

更多资源请访问稀酷客(www.ckook.com)

现线程安全。其要诀在于避免对全局和静态变量的使用。需要返回给调用者的任何信息，亦或是需要在对函数的历次调用间加以维护的信息，都存储于由调用者分配的缓冲区内。（初次碰到可重入问题，是在 21.1.2 节讨论信号处理器中的全局变量时。）不过，并非所有函数都可以实现为可重入，通常的原因如下。

- 根据其性质，有些函数必须访问全局数据结构。malloc 函数库中的函数就是这方面的典范。这些函数为堆中的空闲块维护有一个全局链表。malloc 库函数的线程安全是通过使用互斥量来实现的。
- 一些函数（在发明线程之前就已问世）的接口本身就定义为不可重入，要么返回指针，指向由函数自身静态分配的存储空间，要么利用静态存储对该函数（或相关函数）历次调用间的信息加以维护。表 31-1 所列函数大多属于此类。例如，函数 asctime()（10.2.3 节）就返回一个指针，指向经由静态分配的缓冲区，其内容为日期和时间字符串。

对于一些接口不可重入的函数，SUSv3 为其定义了以后缀_r 结尾的可重入“替身”。这些“替身”函数要求由调用者来分配缓冲区，并将缓存区地址传给函数用以返回结果。这使得调用线程可以使用局部（栈）变量来存放函数结果。出于这一目的，SUSv3 定义了如下函数：asctime_r()、ctime_r()、getgrgid_r()、getgrnam_r()、getlogin_r()、getpwnam_r()、getpwuid_r()、gmtime_r()、localtime_r()、rand_r()、readdir_r()、strerror_r()、strtok_r()和 ttyname_r()。

有些系统实现为一些传统的不可重入函数也提供了附加的可重入“替身”。例如，glibc 就提供了函数 crypt_r()、gethostbyname_r()、getservbyname_r()、getutent_r()、getutid_r()、getutline_r()和 ptsname_r()。不过，为确保应用程序的可移植性，不应假设这些函数在其他实现中也存在。某些情况下，SUSv3 并未规定这些等价的可重入函数，因为功能更强、又可重入的替代函数已然存在。例如，函数 getaddrinfo()就更新且可重入，可用来替代函数 gethostbyname()和 getservbyname()。

31.2 一次性初始化

多线程程序有时有这样的需求：不管创建了多少线程，有些初始化动作只能发生一次。例如，可能需要执行 pthread_mutex_init()对带有特殊属性的互斥量进行初始化，而且必须只能初始化一次。如果由主线程来创建新线程，那么这一点易如反掌：可以在创建依赖于该初始化的线程之前进行初始化。不过，对于库函数而言，这样处理就不可行，因为调用者在初次调用库函数之前可能已经创建了这些线程。故而需要这样的库函数：无论首次为任何线程所调用，都会执行初始化动作。

库函数可以通过函数 pthread_once()实现一次性初始化。

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control, void (*init)(void));
```

Returns 0 on success, or a positive error number on error

利用参数 once_control 的状态，函数 pthread_once()可以确保无论有多少线程对 pthread_once()调用了多少次，也只会执行一次由 init 指向的调用者定义函数。

init 函数没有任何参数，形式如下：

更多资源请访问稀酷客(www.ckook.com)

```
void
init(void)
{
    /* Function body */
}
```

另外，参数 once_control 必须是一指针，指向初始化为 PTHREAD_ONCE_INIT 的静态变量。

```
pthread_once_t once_var = PTHREAD_ONCE_INIT;
```

调用函数 pthread_once()时要指定一个指针，指向类型为 pthread_once_t 的特定变量，对该函数的首次调用将修改 once_control 所指向的内容，以便对其后续调用不会再次执行 init。

常常将 Pthread_once()和线程特有数据结合使用，相关内容会在下一节描述。

> Pthreads 的早期版本不能对互斥量进行静态初始化，只能使用 pthread_mutex_init()（[Butenbof，1996]），这也是函数 pthread_once()存在的主要原因。随着静态分配互斥量功能的问世，库函数可以使用一个经静态分配的互斥量和一个静态布尔型（Boolean）变量来实现一次性初始化。虽然如此，出于方便的考虑，函数 pthread_once()得以保留。

31.3 线程特有数据

实现函数线程安全最为有效的方式就是使其可重入，应以这种方式来实现所有新的函数库。不过，对于已有的不可重入函数库（可能问世于线程流行之前）来说，采用这种方法通常需要修改函数接口，这也意味着，需要修改所有使用此类函数的应用程序。

使用线程特有数据技术，可以无需修改函数接口而实现已有函数的线程安全。较之于可重入函数，采用线程特有数据的函数效率可能要略低一些，不过对于使用了这些调用的程序而言，则省去了修改程序之劳。

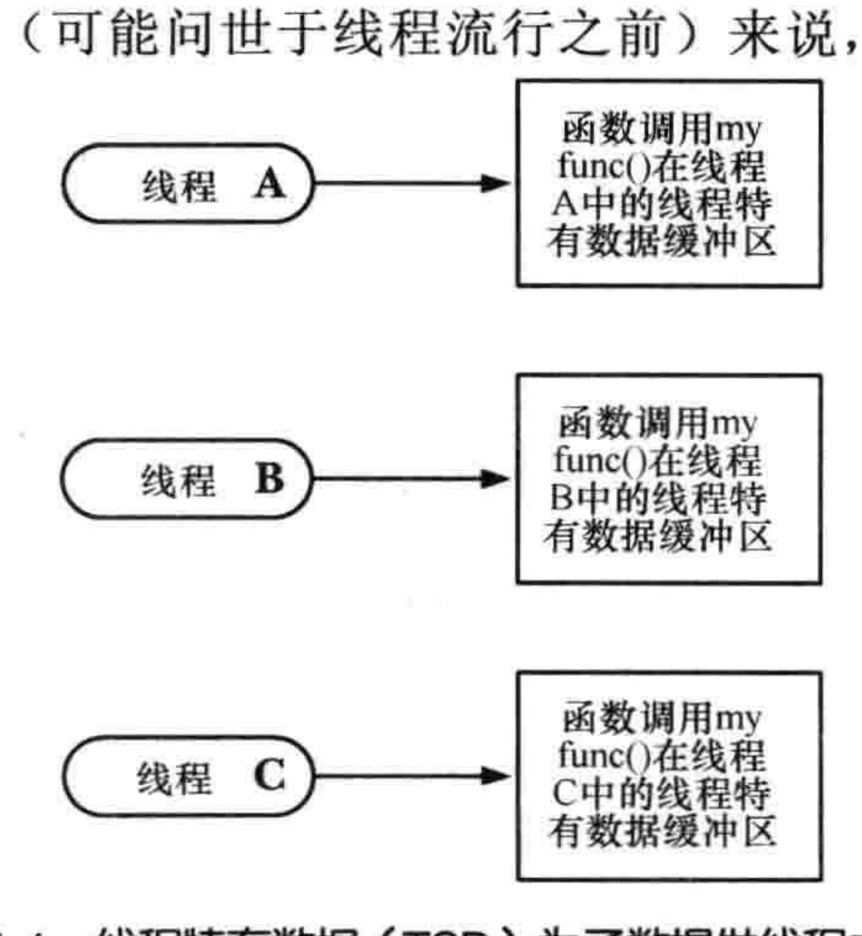

图 31-1：线程特有数据（TSD）为函数提供线程内存储

如图 31-1 所示，线程特有数据使函数得以为每个调用线程分别维护一份变量的副本（copy）。线程特有数据是长期存在的。在同一线程对相同函数的历次调用间，每个线程的变量会持续存在，函数可以向每个调用线程返回各自的结果缓冲区（如果需要的话）。

31.3.1 库函数视角下的线程特有数据

要了解线程特有数据相关 API 的使用，需要从使用这一技术的库函数角度来考虑如下问题。

- 该函数必须为每个调用者线程分配单独的存储，且只需在线程初次调用此函数时分配一次即可。
- 在同一线程对此函数的后续所有调用中，该函数都需要获取初次调用时线程分配的存

更多资源请访问稀酷客(www.ckook.com)

储块地址。由于函数调用结束时会释放自动变量，故而函数不应利用自动变量存放存储块指针，也不能将指针存放于静态变量中，因为静态变量在进程中只有一个实例。Pthreads API 提供了函数来处理这一情况。

- 不同（无相互依赖关系）函数各自可能都需要使用线程特有数据。每个函数都需要方法来标识其自身的线程特有数据（键），以便与其他函数所使用的线程特有数据有所区分。
- 当线程退出时，函数无法控制将要发生的情况。这时，线程可能会执行该函数之外的代码。不过，一定存在某些机制（解构器），在线程退出时会自动释放为该线程所分配的存储。若非如此，随着持续不断地创建线程，调用函数和终止线程，将会引发内存泄露。

31.3.2 线程特有数据 API 概述

要使用线程特有数据，库函数执行的一般步骤如下。

1. 函数创建一个键（key），用以将不同函数使用的线程特有数据项区分开来。调用函数 pthread_key_create()可创建此“键”，且只需在首个调用该函数的线程中创建一次，函数 pthread_once()的使用正是出于这一目的。键在创建时并未分配任何线程特有数据块。
2. 调用 pthread_key_create()还有另一个目的，即允许调用者指定一个自定义解构函数，用于释放为该键所分配的各个存储块（参见下一步）。当使用线程特有数据的线程终止时，Pthreads API 会自动调用此解构函数，同时将该线程的数据块指针作为参数传入。
3. 函数会为每个调用者线程创建线程特有数据块。这一分配通过调用 malloc()（或类似函数）完成，每个线程只分配一次，且只会在线程初次调用此函数时分配。
4. 为了保存上一步所分配存储块的地址，函数会使用两个 Pthreads 函数：pthread_setspecific()和 pthread_getspecific()。调用函数 pthread_setspecific()实际上是对 Pthreads 实现发起这样的请求：保存该指针，并记录其与特定键（该函数的键）以及特定线程（调用者线程）的关联性。调用 pthread_getspecific()所执行的是互补操作：返回之前所保存的、与给定键以及调用线程相关联的指针。如果还没有指针与特定的键及线程相关联，那么 pthread_getspecific()返回 NULL。函数可以利用这一点来判断自身是否是初次为某个线程所调用，若为初次，则必须为该线程分配空间。

31.3.3 线程特有数据 API 详述

本节将详述上节所提及的各个函数，并通过对线程特有数据的典型实现来说明其操作方法。下一节会演示如何使用线程特有数据来实现线程安全的标准 C 语言库函数 stderror()。

调用 pthread_key_create()函数为线程特有数据创建一个新键，并通过 key 所指向的缓冲区返回给调用者。

因为进程中的所有线程都可使用返回的键，所以参数 key 应指向一个全局变量。

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

Returns 0 on success, or a positive error number on error

参数 destructor 指向一个自定义函数，其格式如下：

更多资源请访问稀酷客(www.ckook.com)

```
void
dest(void *value)
{
    /* Release storage pointed to by 'value' */
}
```

只要线程终止时与 key 的关联值不为 NULL，Pthreads API 会自动执行解构函数，并将与 key 的关联值作为参数传入解构函数。传入的值通常是与该键关联，且指向线程特有数据块的指针。如果无需解构，那么可将 destructor 设置为 NULL。

> 如果一个线程有多个线程特有数据块，那么对各个解构函数的调用顺序是不确定的。对每个解构函数的设计应相互独立。

观察线程特有数据的实现有助于理解它们的使用方法。典型的实现（NPTL 即在此列）会包含以下数组。

- 一个全局（进程范围）数组，存放线程特有数据的键信息。
- 每个线程包含一个数组，存有为每个线程分配的线程特有数据块的指针（通过调用 pthread_setspecific()来存储指针）。

图 31-2：线程特有数据键的实现

在这一实现中，pthread_key_create()返回的 pthread_key_t 类型值只是对全局数组的索引（index），标记为 pthread_keys，其格式如图 31-2 所示。数组的每个元素都是一个包含两个字段（field）的结构。第一个字段标记该数组元素是否在用（即已由之前对 pthread_key_create()的调用分配）。第二个字段用于存放针对此键、线程特有数据块的解构函数指针（是函数 pthread_key_crate()中参数 destructor 的一份拷贝）。

函数 pthread_setspecific()要求 Pthreads API 将 value 的副本存储于一数据结构中，并将 value 与调用线程以及 key 相关联（key 由之前对 pthread_key_create()的调用返回）。Pthread_getspecific()函数执行的操作与之相反，返回之前与本线程及给定 key 相关的值（value）。

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
                    Returns 0 on success, or a positive error number on error
void *pthread_getspecific(pthread_key_t key);
            Returns pointer, or NULL if no thread-specific data isassociated with key
```

函数 pthread_setspecific()的参数 value 通常是一指针，指向由调用者分配的一块内存。当线程终止时，会将该指针作为参数传递给与 key 对应的解构函数。

> 参数 value 也可以不是一个指向内存区域的指针，而是任何可以赋值（通过强制转换）给 void*的标量值。在这种情况下，先前对 pthread_key_create()函数的调用应将 destructor 指定为 NULL。

图 31-3 展示了用于存储 value 的数据结构的常见实现。图中假设将 pthread_keys[1]分配给函数 myfunc()。Pthreads API 为每个函数维护指向线程特有数据块的一个指针数组。其中每个数

更多资源请访问稀酷客(www.ckook.com)

组元素都与图 31-2 中全局 pthread_keys 数组的元素一一对应。函数 pthread_ setspecific()在指针数组中为每个调用线程设置与 key 对应的元素。

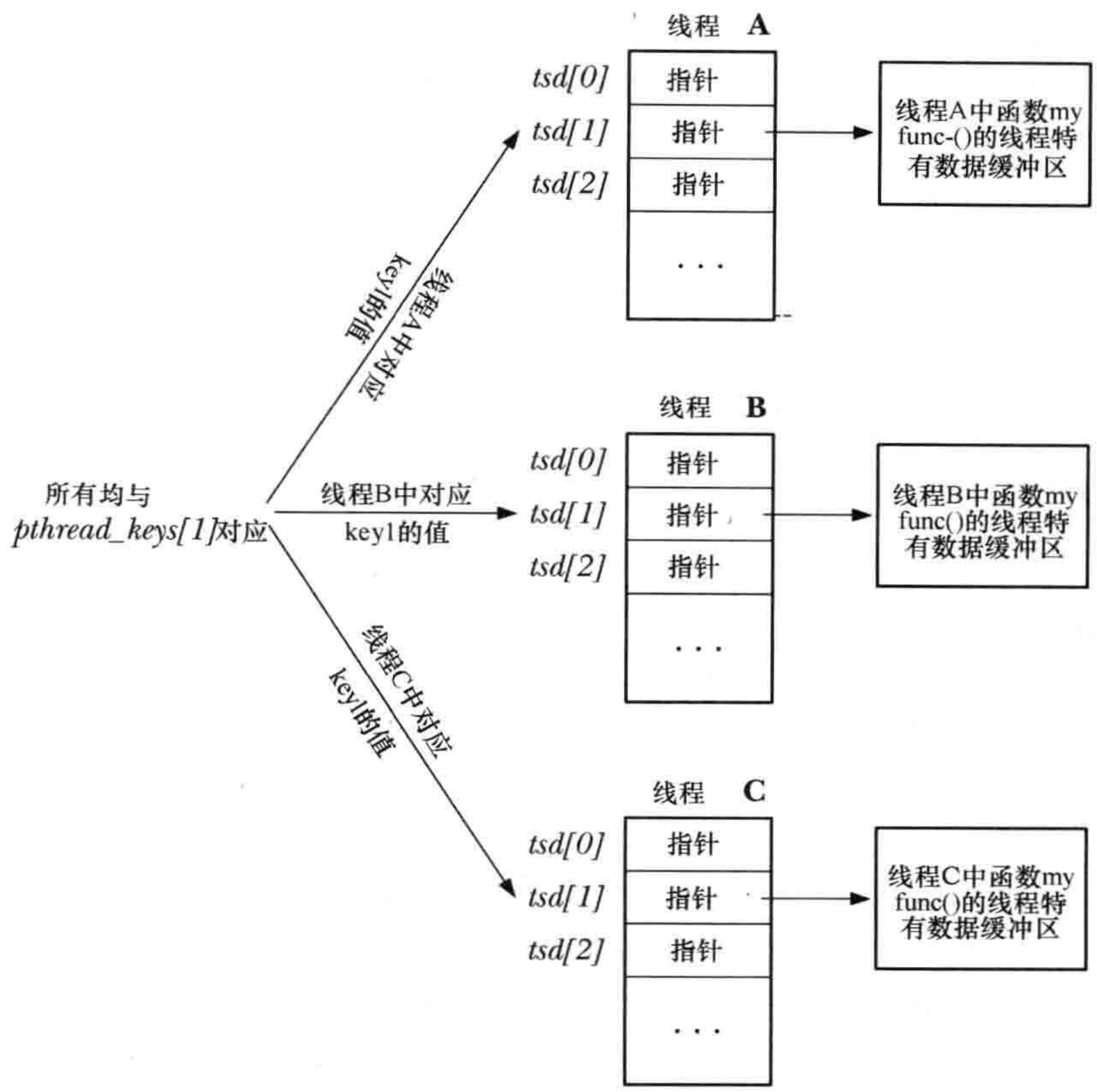

图 31-3：用于实现线程特有数据（TSD）指针的数据结构

当线程刚刚创建时，会将所有线程特有数据的指针都初始化为 NULL。这意味着当线程初次调用库函数时，必须使用 pthread_getspecific()函数来检查该线程是否已有与 key 对应的关联值。如果没有，那么此函数会分配一块内存并通过 pthread_setspecific()保存指向该内存块的指针。在下一节实现线程安全版的 stderror()函数时，将给出示例。

31.3.4 使用线程特有数据 API

3.4 节在首度论及标准 stderror()函数时曾指出，可能会返回一个指向静态分配字符串的指针作为函数结果。这意味着 stderror()可能不是线程安全的。后面将以数页篇幅讨论一下非线程安全的 stderror()实现，接着说明如何使用线程特有数据来实现该函数的线程安全。

在包括 Linux 在内的许多 UNIX 实现中，由标准 C 语言函数库提供的 stderror()函数都是线程安全的。不过，由于 SUSv3 并未规定该函数必须是线程安全的，而且这一 stderror()的实现又为使用线程特有数据提供了一个简单范例，故而在此将其作为示例。

程序清单 31-1 演示了非线程安全版 strerror()函数的一个简单实现。该函数利用了由 glibc 定义的一对全局变量：_sys_errlist 是一个指针数组，其每个元素指向一个与 errno 错误号相匹配的字符串（因此，例如，_sys_errlist[EINVAL]即指向字符串 Invalid operation）；_sys_nerr 表

示_sys_errlist 中的元素个数。

程序清单 31-1：非线程安全版 strerror()函数的一种实现

threads/strerror.c

```
#define _GNU_SOURCE                 /* Get '_sys_nerr' and '_sys_errlist'
                                       declarations from <stdio.h> */
#include <stdio.h>
#include <string.h>                 /* Get declaration of strerror() */

#define MAX_ERROR_LEN 256           /* Maximum length of string
                                       returned by strerror() */

static char buf[MAX_ERROR_LEN];     /* Statically allocated return buffer */

char *
strerror(int err)
{
    if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
    } else {
        strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';          /* Ensure null termination */
    }

    return buf;
}
```

threads/strerror.c

可以利用程序清单 31-2 中程序来展示程序清单 31-1 中非线程安全版的 streerror()实现所造成的后果。该程序分别从两个不同线程中调用 strerror()，并且均在两个线程调用 stderror()之后才显示返回结果。虽然两个线程为 strerror()指定的参数值不同（EINVAL 和 EPERM），在与程序清单 31-1 版的 strerror()链接、编译后，运行该程序将产生如下结果：

```
$ ./strerror_test
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x804a7c0) = Operation not permitted
Main thread:  str (0x804a7c0) = Operation not permitted
```

两个线程都显示与 EPERM 对应的 errno 字符串，因为第二个线程对 strerror()的调用（在函数 threadFunc()中）覆盖了主线程调用 strerror()时写入缓冲区的内容。检查输出结果可以发现，两个线程的局部变量 str 均指向同一内存地址。

程序清单 31-2：从两个不同线程调用 strerror()

threads/strerror_test.c

```
#include <stdio.h>
#include <string.h>                 /* Get declaration of strerror() */
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *str;

    printf("Other thread about to call strerror()\n");
    str = strerror(EPERM);
```

```
    printf("Other thread: str (%p) = %s\n", str, str);

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t;
    int s;
    char *str;

    str = strerror(EINVAL);
    printf("Main thread has called strerror()\n");

    s = pthread_create(&t, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Main thread:  str (%p) = %s\n", str, str);

    exit(EXIT_SUCCESS);
}
```

threads/strerror_test.c

程序清单 31-3 是对函数 strerror()的全新实现，使用了线程特有数据来确保线程安全。

程序清单 31-3：使用线程特有数据以实现线程安全的 strerror()函数

threads/strerror_tsd.c

```
#define _GNU_SOURCE             /* Get '_sys_nerr' and '_sys_errlist'
                                   declarations from <stdio.h> */
#include <stdio.h>
#include <string.h>             /* Get declaration of strerror() */
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t strerrorKey;

#define MAX_ERROR_LEN 256       /* Maximum length of string in per-thread
                                   buffer returned by strerror() */

static void                     /* Free thread-specific data buffer */
① destructor(void *buf)
  {
      free(buf);
  }

  static void                   /* One-time key creation function */
② createKey(void)
  {
      int s;

      /* Allocate a unique thread-specific data key and save the address
```

```
        of the destructor for thread-specific data buffers */

③       s = pthread_key_create(&strerrorKey, destructor);
        if (s != 0)
            errExitEN(s, "pthread_key_create");
    }
    char *
    strerror(int err)
    {
        int s;
        char *buf;

        /* Make first caller allocate key for thread-specific data */

④       s = pthread_once(&once, createKey);
        if (s != 0)
            errExitEN(s, "pthread_once");

⑤       buf = pthread_getspecific(strerrorKey);
        if (buf == NULL) {          /* If first call from this thread, allocate
                                       buffer for thread, and save its location */
⑥           buf = malloc(MAX_ERROR_LEN);
            if (buf == NULL)
                errExit("malloc");

⑦           s = pthread_setspecific(strerrorKey, buf);
            if (s != 0)
                errExitEN(s, "pthread_setspecific");
        }

        if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
            snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
        } else {
            strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
            buf[MAX_ERROR_LEN - 1] = '\0';          /* Ensure null termination */
        }

        return buf;
    }
```

threads/strerror_tsd.c

改进版 strerror()所做的第一步是调用 pthread_once()④，以确保（从任何线程）对该函数的首次调用将执行 createKey()②。函数 createKey()会调用 pthread_key_create()来分配一个线程特有数据的键（key），并将其存储于全局变量 strerrorKey③中。对 pthread_key_create()的调用同时也会记录解构函数①的地址，将使用该解构函数来释放与键对应的线程特有数据缓冲区。

接着，函数 strerror()调用 pthread_getspecific()⑤以获取该线程中对应于 strerrorKey 的唯一缓冲区地址。如果 pthread_getspecific()返回 NULL，这表明该线程是首次调用 strerror()函数，因此函数会调用 malloc()⑥分配一个新缓冲区，并使用 pthread_setspecific()⑦来保存该缓冲区的地址。如果 pthread_getspecific()的返回值非 NULL，那么该值指向业已存在的缓冲区，此缓冲区由之前对 strerror()的调用所分配。

这一 strerror()函数实现的剩余部分与非线程安全版的前述实现相类似，唯一的区别在于，buf 是线程特有数据的缓冲区地址，而非静态变量。

如果使用新版 strerror()（程序清单 31-3）编译链接测试程序（程序清单 31-2）strerror_test_tsd，程序运行会有如下结果：

```
$ ./strerror_test_tsd
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x804b158) = Operation not permitted
Main thread:  str (0x804b008) = Invalid argument
```

根据这一输出，可以看出新版 strerro()是线程安全的：两个线程中局部变量 str 所指向的地址是不同的。

31.3.5 线程特有数据的实现限制

正如对线程特有数据典型实现过程的描述所揭示的，实现可能要对其所支持的线程特有数据键的数量加以限制。SUSv3 要求至少支持 128（_POSIX_THREAD_KEYS_MAX）个键。应用程序要么通过对 PTHREAD_KEY_MAX（定义于<limits.h>）的定义，要么通过调用 sysconf(_SC_THREAD_KEYS_MAX)，来确定实际支持的键数量。Linux 支持多达 1024 个键。

即使 128 个键对于大多数应用来说也已经绰绰有余。这是因为，每个库函数应该只会使用到少量的键，通常会只用一个。如果一个函数需要多个线程特有数据的值，通常可将这些值置于一个结构中，并将该结构仅与一个线程特有数据的键关联。

31.4 线程局部存储

类似于线程特有数据，线程局部存储提供了持久的每线程存储。作为非标准特性，诸多其他的 UNIX 实现（例如 Solaris 和 FreeBSD）为其提供了相同，或类似的接口形式。

线程局部存储的主要优点在于，比线程特有数据的使用要简单。要创建线程局部变量，只需简单地在全局或静态变量的声明中包含__thread 说明符即可。

```
static __thread buf[MAX_ERROR_LEN];
```

但凡带有这种说明符的变量，每个线程都拥有一份对变量的拷贝。线程局部存储中的变量将一直存在，直至线程终止，届时会自动释放这一存储。

关于线程局部变量的声明和使用，需要注意如下几点。

- 如果变量声明中使用了关键字 static 或 extern，那么关键字__thread 必须紧随其后。
- 与一般的全局或静态变量声明一样，线程局部变量在声明时可设置一个初始值。
- 可以使用 C 语言取址操作符（&）来获取线程局部变量的地址。

线程局部存储需要内核（由 Linux 2.6 提供）、Pthreads 实现（由 NPTL 提供）以及 C 编译器（在 x86-32 平台上由 gcc 3.3 或后续版本提供）的支持。

程序清单 31-4 提供了使用线程局部存储实现线程安全版 strerror()函数的例子。如果用该版 strerror()与测试程序（程序清单 31-2）编译、链接、生成 strerror_test_tls，那么运行时将产生如下结果：

```
$ ./strerror_test_tls
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x40376ab0) = Operation not permitted
Main thread:  str (0x40175080) = Invalid argument
```


更多资源请访问稀酷客(www.ckook.com)

程序清单 31-4：使用线程局部存储实现线程安全版的 strerror()函数

threads/strerror_tls.c

```
#define _GNU_SOURCE                     /* Get '_sys_nerr' and '_sys_errlist'
                                           declarations from <stdio.h> */

#include <stdio.h>
#include <string.h>                     /* Get declaration of strerror() */
#include <pthread.h>

#define MAX_ERROR_LEN 256               /* Maximum length of string in per-thread
                                           buffer returned by strerror() */

static __thread char buf[MAX_ERROR_LEN];
                                        /* Thread-local return buffer */

char *
strerror(int err)
{
    if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
    } else {
        strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';          /* Ensure null termination */
    }

    return buf;
}
```

threads/strerror_tls.c

31.5 总结

若一函数可由多个线程同时安全调用，则称之为线程安全的函数。使用全局或静态变量是导致函数非线程安全的通常原因。在多线程应用中，保障非线程安全函数安全的手段之一是运用互斥锁来防护对该函数的所有调用。这种方法带来了并发性能的下降，因为同一时点只能有一个线程运行该函数。提升并发性能的另一方法是：仅在函数中操作共享变量（临界区）的代码前后加入互斥锁。

使用互斥量可以实现大部分函数的线程安全，不过由于互斥量的加、解锁开销，故而也带来了性能的下降。如能避免使用全局或静态变量，可重入函数则无需使用互斥量即可实现线程安全。

SUSv3 所规范的大部分函数都需实现线程安全。SUSv3 同时也列出了小部分无需实现线程安全的函数。一般情况下，这些函数将静态存储返回给调用者，或者在对函数的连续调用间进行信息维护。根据定义，这些函数是不可重入的，也不能使用互斥量来确保其线程安全。本章讨论了两种大致相当的编程技术——线程特有数据和线程局部存储——可在无需改变函数接口定义的情况下保障不安全函数的线程安全。这两种技术均允许函数分配持久的、基于线程的存储。

更多信息

请参考 29.10 节所列的更多信息来源。

更多资源请访问稀酷客(www.ckook.com)

31.6 练习

31-1. 试实现函数 one_time_init(control，init)，要求与函数 pthread_once()执行等同操作。参数 control 应为一指针，指向经静态分配的结构，其中包含一个布尔型变量和一个互斥量。布尔型变量用以标识函数 init 是否曾被调用过，而由互斥量来控制对变量的访问。为简化函数实现，可以忽略诸如 init()调用失败或者在由线程初次调用时被取消的情况（亦即，无需为此做特别设计，如果真发生了此类事件，那么下一个调用 one_time_init()的线程会重新调用 init()）。

31-2. 使用线程特有数据重新实现线程安全版的函数 dirname()和 basename()（18.14 节）。

更多资源请访问稀酷客(www.ckook.com)

第 32 章

线程：线程取消

在通常情况下，程序中的多个线程会并发执行，每个线程各司其职，直至其决意退出，随即会调用函数 pthread_exit()或者从线程启动函数中返回。

有时候，需要将一个线程取消（cancel）。亦即，向线程发送一个请求，要求其立即退出。比如，一组线程正在执行一个运算，一旦某个线程检测到错误发生，需要其他线程退出，取消线程的功能这时就派上用场。还有一种情况，一个由图形用户界面（GUI）驱动的应用程序可能会提供一个“取消”按钮，以便用户可以终止后台某一线程正在执行的任务。这种情况下，主线程（控制图形用户界面）需要请求后台线程退出。

本章就来讨论 POSIX 线程的取消机制。

32.1 取消一个线程

函数 pthread_cancel()向由 thread 指定的线程发送一个取消请求。

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

发出取消请求后，函数 pthread_cancel()当即返回，不会等待目标线程的退出。

准确地说，目标线程会发生什么？何时发生？这些都取决于下节将要述及的线程取消状态（state）和类型（type）。

32.2 取消状态及类型

函数 pthread_setcancelstate()和 pthread_setcanceltype()会设定标志，允许线程对取消请求的响应过程加以控制。

更多资源请访问稀酷客(www.ckook.com)

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Both return 0 on success, or a positive error number on error

函数 pthread_setcancelstate()会将调用线程的取消性状态置为参数 state 所给定的值。该参数的值如下。

PTHREAD_CANCEL_DISABLE

线程不可取消。如果此类线程收到取消请求，则会将请求挂起，直至将线程的取消状态置为启用。

PTHREAD_CANCEL_ENABLE

线程可以取消。这是新建线程取消性状态的默认值。

线程的前一取消性状态将返回至参数 oldstate 所指向的位置。

如果对前一状态没有兴趣，Linux 允许将 oldstate 置为 NULL。在很多其他的系统实现中，情况也是如此。不过，SUSv3 并没有规范这一特性，所以要保证应用的可移植性，就不能依赖这一特性。应该总是为 oldstate 设置一个非 NULL 的值。

如果线程执行的代码片段需要不间断地一气呵成，那么临时屏闭线程的取消性状态（PTHREAD_CANCEL_DISABLE）就变得很有必要。

如果线程的取消性状态为“启用”（PTHREAD_CANCEL_ENABLE），那么对取消请求的处理则取决于线程的取消性类型，该类型可以通过调用函数 pthread_setcanceltype()时的参数 type 给定。参数 type 有如下值：

PTHREAD_CANCEL_ASYNCHRONOUS

可能会在任何时点（也许是立即取消，但不一定）取消线程。异步取消的应用场景很少，将延后至 32.6 节再做讨论。

PTHREAD_CANCEL_DEFERED

取消请求保持挂起状态，直至到达取消点（cancellation point，见下节）。这也是新建线程的缺省类型。后续各节将介绍延迟取消（deferred cancelability）的更多细节。

线程原有的取消类型将返回至参数 oldtype 所指向的位置。

与函数 pthread_setcancelstate()的参数 oldstate 类似，如果不关心原有取消类型，许多系统实现（包括 Linux）允许将 oldtype 置为 NULL。同样，SUSv3 也没有规范这一行为，所以需要保障可移植性的应用不应使用这一特性，应该总是为 oldtype 设置一个非 NULL 值。

当某线程调用 fork()时，子进程会继承调用线程的取消性类型及状态。而当某线程调用 exec()时，会将新程序主线程的取消性类型及状态分别重置为 PTHREAD_ CANCEL_ NABLE 和 PTHREAD_CANCEL_DEFERRED。

32.3 取消点

若将线程的取消性状态和类型分别置为启用和延迟，仅当线程抵达某个取消点（cancellation

更多资源请访问稀酷客(www.ckook.com)

point）时，取消请求才会起作用。取消点即是对由实现定义的一组函数之一加以调用。

SUSv3 规定，实现若提供了表 32-1 中所列的函数，则这些函数必须是取消点。其中的大部分函数都有能力将线程无限期地堵塞起来。

表 32-1：SUSv3 规定必须是取消点的函数

accept()	nanosleep()	sem_timedwait()
aio_suspend()	open()	sem_wait()
clock_nanosleep()	pause()	send()
close()	poll()	sendmsg()
connect()	pread()	sendto()
creat()	pselect()	sigpause()
fcntl(F_SETLKW)	pthread_cond_timedwait()	sigsuspend()
fsync()	pthread_cond_wait()	sigtimedwait()
fdatasync()	pthread_join()	sigwait()
getmsg()	pthread_testcancel()	sigwaitinfo()
getpmsg()	putmsg()	sleep()
lockf(F_LOCK)	putpmsg()	system()
mq_receive()	pwrite()	tcdrain()
mq_send()	read()	usleep()
mq_timedreceive()	readv()	wait()
mq_timedsend()	recv()	waitid()
msgrcv()	recvfrom()	waitpid()
msgsnd()	recvmsg()	write()
msync()	select()	writev()

除表 32-1 所列函数之外，SUSv3 还指定了大量函数，系统实现可以将其定义为取消点。其中包括 stdio 函数、dlopen API、syslog API、nftw()、popen()、semop()、unlink()，以及从诸如 utmp 之类的系统文件中获取信息的各种函数。可移植应用程序必须正确处理这一情况：线程在调用这些函数时有可能遭到取消。

SUSv3 规定，除了上述两组必须或可能是可取消点的函数之外，不得将标准中的任何其他函数视为取消点（亦即，调用这些函数不会招致线程取消，可移植程序无需加以处理）。

SUSv4 在必须的可取消点函数列表中增加了 openat()，并移除了函数 sigpause()（将其移至“可能的”取消点函数列表中）和函数 usleep()（已从标准中删除）。

系统实现可随意将标准并未规范的其他函数标记为取消点。任何可能造成堵塞的函数（有可能是因为需要访问文件）都是取消点的理想候选对象。出于这一理由，glibc 将其中的许多非标准函数标记为取消点。

线程一旦收到取消请求，且启用了取消性状态并将类型置为延迟，则其会在下次抵达取消点时终止。如果该线程尚未分离（not detached），那么为防止其变为僵尸线程，必须由其他线程对其进行连接（join）。连接之后，返回至函数 pthread_join()中第二个参数的将是一个特殊值：PTHREAD_CANCELED。

示例程序

程序清单 32-1 是一个使用 pthread_cancel()的简单例子。主程序创建一个线程来执行无限循

更多资源请访问稀酷客(www.ckook.com)

环，每次都在休眠一秒后打印循环计数器的值。（仅当向其发送取消请求或者进程退出时，该线程才会终止。）同时，主程序将休眠 3 秒，随即向新创建的线程发送取消请求。程序运行结果如下：

```
$ ./t_pthread_cancel
New thread started
Loop 1
Loop 2
Loop 3
Thread was canceled
```

程序清单 32-1：调用 pthread_cancel()取消线程

threads/thread_cancel.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    int j;
    printf("New thread started\n");     /* May be a cancellation point */
    for (j = 1; ; j++) {
        printf("Loop %d\n", j);         /* May be a cancellation point */
        sleep(1);                       /* A cancellation point */
    }

    /* NOTREACHED */
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;

    s = pthread_create(&thr, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    sleep(3);                           /* Allow new thread to run a while */

    s = pthread_cancel(thr);
    if (s != 0)
        errExitEN(s, "pthread_cancel");

    s = pthread_join(thr, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    if (res == PTHREAD_CANCELED)
        printf("Thread was canceled\n");
    else
        printf("Thread was not canceled (should not happen!)\n");

    exit(EXIT_SUCCESS);
}
```

threads/thread_cancel.c

32.4 线程可取消性的检测

在程序清单 32-1 中，由 main()创建的线程会执行到属于取消点的函数（sleep()属于取消点，printf()可能也是），因而会接受取消请求。不过，假设线程执行的是一个不含取消点的循环（计算密集型［compute-bound］循环），这时，线程永远也不会响应取消请求。

函数 pthread_testcancel()的目的很简单，就是产生一个取消点。线程如果已有处于挂起状态的取消请求，那么只要调用该函数，线程就会随之终止。

```
#include <pthread.h>

void pthread_testcancel(void);
```

当线程执行的代码未包含取消点时，可以周期性地调用 pthread_testcancel()，以确保对其他线程向其发送的取消请求做出及时响应。

32.5 清理函数（cleanup handler）

一旦有处于挂起状态的取消请求，线程在执行到取消点时如果只是草草收场，这会将共享变量以及 Pthreads 对象（例如互斥量）置于一种不一致状态，可能导致进程中其他线程产生错误结果、死锁，甚至造成程序崩溃。为规避这一问题，线程可以设置一个或多个清理函数，当线程遭取消时会自动运行这些函数，在线程终止之前可执行诸如修改全局变量，解锁互斥量等动作。

每个线程都可以拥有一个清理函数栈。当线程遭取消时，会沿该栈自顶向下依次执行清理函数，首先会执行最近设置的函数，接着是次新的函数，以此类推。当执行完所有清理函数后，线程终止。

函数 pthread_cleanup_push()和 pthread_cleanup_pop()分别负责向调用线程的清理函数栈添加和移除清理函数。

```
#include <pthread.h>

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

pthread_cleanup_push()会将参数 routine 所含的函数地址添加到调用线程的清理函数栈顶。参数 routine 是一个函数指针，格式如下：

```
void
routine(void *arg)
{
    /* Code to perform cleanup */
}
```

执行 pthread_cleanup_push()时给定的 arg 值，会作为调用清理函数时的参数。其参数类型为 void*，如果强制装换使用得当，那么通过该参数可以传入各种类型的数据。

通常，线程如在执行一段特殊代码时遭到取消，才需要执行清理动作。如果线程顺利执行完这段代码而未遭取消，那么就不再需要清理。所以，每个对 pthread_cleanup_push()的调用都会伴随着对 pthread_cleanup_pop()的调用。此函数从清理函数栈中移除最顶层的函数。如

更多资源请访问稀酷客(www.ckook.com)

果参数 execute 非零，那么无论如何都会执行清理函数。在函数未遭取消而又希望执行清理动作的情况下，这会非常方便。

尽管这里把 pthread_cleanup_push()和 pthread_cleanup_pop()描述为函数，SUSv3 却允许将它们实现为宏（macro），可展开为分别由{和}所包裹的语句序列。并非所有的 UNIX 都这样做，不过包括 Linux 在内的很多系统都是使用宏来实现的。这意味着，pthread_ cleanup_push()和与其配对的 pthread_cleanup_pop()属于同一个语法块，必须一一对应。（一旦以此方式来实现 pthread_cleanup_push()和 pthread_cleanup_pop()，在对两者的调用间所声明的变量，其作用域将受限于这一语法块。）例如，以下代码就不正确：

```
pthread_cleanup_push(func, arg);
...
if (cond) {
    pthread_cleanup_pop(0);
}
```

为便于编码，若线程因调用 pthread_exit()而终止，则也会自动执行尚未从清理函数栈中弹出（pop）的清理函数。线程正常返回（return）时不会执行清理函数。

示例程序

程序清单 32-2 提供了一个使用清理函数的简单例子。主程序创建线程⑧，线程首先分配一块内存③，并将其地址存储于 buf 中，接着锁定互斥量 mtx④。因为线程可能会遭到取消，所以调用 pthread_cleanup_push()⑤设置清理函数，并将存储于 buf 中的地址作为参数传入。如果执行到清理函数，那么清理函数会释放内存①并解锁互斥量②。

线程接着进入循环，等待对条件变量 cond 的通知⑥。取决于可执行程序是否带有命令行参数，此循环会以以下两种方式结束。

- 若无命令行参数，则由 main()⑨函数取消线程。此时，取消操作发生在对 pthread_cond_wait()⑥的调用中，此函数可见于程序清单 32-1 中，属于取消点。作为取消动作的一部分，会自动调用由 pthread_cleanup_push()设置的清理函数。
- 如果指定了命令行参数，那么在将全局变量 glob 设置为非零后，通知条件变量⑩。此时，线程会一直执行到 pthread_cleanup_pop()⑦，因为向此函数传入了非零参数，所以依然会调用清理函数。

程序清单 32-2：使用清理函数

threads/thread_cleanup.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int glob = 0;                    /* Predicate variable */

static void     /* Free memory pointed to by 'arg' and unlock mutex */
cleanupHandler(void *arg)
{
    int s;

    printf("cleanup: freeing block at %p\n", arg);
①   free(arg);

    printf("cleanup: unlocking mutex\n");
```

```
②       s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    static void *
    threadFunc(void *arg)
    {
        int s;
        void *buf = NULL;                   /* Buffer allocated by thread */

③       buf = malloc(0x10000);              /* Not a cancellation point */
        printf("thread:  allocated memory at %p\n", buf);

④       s = pthread_mutex_lock(&mtx);       /* Not a cancellation point */
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

⑤       pthread_cleanup_push(cleanupHandler, buf);

        while (glob == 0) {
⑥           s = pthread_cond_wait(&cond, &mtx);    /* A cancellation point */
            if (s != 0)
                errExitEN(s, "pthread_cond_wait");
        }

        printf("thread:  condition wait loop completed\n");
⑦       pthread_cleanup_pop(1);             /* Executes cleanup handler */
        return NULL;
    }

    int
    main(int argc, char *argv[])
    {
        pthread_t thr;
        void *res;
        int s;
⑧       s = pthread_create(&thr, NULL, threadFunc, NULL);
        if (s != 0)
            errExitEN(s, "pthread_create");

        sleep(2);                   /* Give thread a chance to get started */

        if (argc == 1) {            /* Cancel thread */
            printf("main:    about to cancel thread\n");
⑨           s = pthread_cancel(thr);
            if (s != 0)
                errExitEN(s, "pthread_cancel");

        } else {                    /* Signal condition variable */
            printf("main:    about to signal condition variable\n");
            glob = 1;
⑩           s = pthread_cond_signal(&cond);
            if (s != 0)
                errExitEN(s, "pthread_cond_signal");
        }

⑪       s = pthread_join(thr, &res);
        if (s != 0)
            errExitEN(s, "pthread_join");
```

```
    if (res == PTHREAD_CANCELED)
        printf("main:    thread was canceled\n");
    else
        printf("main:    thread terminated normally\n");

    exit(EXIT_SUCCESS);
}
```

threads/thread_cleanup.c

主程序与遭终止线程建立连接⑪，并报告线程是遭到取消还是正常终止。

如果执行程序清单 32-2 中程序且不带任何命令行参数，那么 main()函数会调用 pthread_cancel()，清理函数也会得以自动执行。输出如下：

```
$ ./thread_cleanup
thread:  allocated memory at 0x804b050
main:    about to cancel thread
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread was canceled
```

如果运行该程序且带有命令行参数，那么 main()将 glob 设置为 1 并通知条件变量，清理函数则通过 pthread_cleanup_pop()的调用执行，可以看到如下结果：

```
$ ./thread_cleanup s
thread:  allocated memory at 0x804b050
main:    about to signal condition variable
thread:  condition wait loop completed
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread terminated normally
```

32.6 异步取消

如果设定线程为可异步取消时（取消性类型为 PTHREAD_CANCEL_ASYNCHRONOUS），可以在任何时点将其取消（亦即，执行任何机器指令时），取消动作不会拖延到下一个取消点才执行。

异步取消的问题在于，尽管清理函数依然会得以执行，但处理函数却无从得知线程的具体状态。程序清单 32-2 采用了延时取消类型，只有在执行到 pthread_cond_wait()这一唯一的取消点时，线程才会遭到取消。此时可知，已将 buf 初始化为指向新分配的内存块，并且锁定了互斥量 mtx。不过，要是采用异步取消，就可以在任意点取消线程（例如，调用 malloc()之前，调用 malloc()与锁定互斥量之间，或者锁定互斥量之后）。清理函数无法知道将在哪里发生取消动作，或者准确地来说，清理函数不清楚需要执行哪些清理步骤。此外，线程也很可能在对 malloc()的调用期间被取消，这极有可能造成后续的混乱（见 7.1.3 节）。

作为一般性原则，可异步取消的线程不应该分配任何资源，也不能获取互斥量或锁。这导致大量库函数无法使用，其中就包括 Pthreads 函数的大部分。（SUSv3 中有 3 处例外 pthread_cancel()、pthread_setcancelstate()以及 pthread_setcanceltype()，规范明确要求将它们实现为“异步取消安全（async-cancel-safe）”，亦即，实现必须确保在可异步取消的线程中可以安全调用它们。）换言之，异步取消功能鲜有应用场景，其中之一就是：取消在执行计算密集型循环的线程。

更多资源请访问稀酷客(www.ckook.com)

32.7 总结

函数 pthread_cancel()允许某线程向另一个线程发送取消请求，要求目标线程终止。

目标线程如何响应，取决于其取消性状态和类型。如果禁用线程的取消性状态，那么请求会保持挂起（pending）状态，直至将线程的取消性状态置为启用。如果启用取消性状态，那么线程何时响应请求则依赖于取消性类型。若类型为延迟取消，则在线程下一次调用某个取消点（由 SUSv3 标准所规定的一系列函数之一）时，取消发生。如果为异步取消类型，取消动作随时可能发生（鲜有使用）。

线程可以设置一个清理函数栈，其中的清理函数属于由开发人员定义的函数，当线程遭到取消时，会自动调用这些函数以执行清理工作（例如，恢复共享变量状态，或解锁互斥量）。

更多信息

请参考列于 29.10 节的深入信息来源。

更多资源请访问稀酷客(www.ckook.com)

第 33 章

线程：更多细节

本章将就 POSIX 线程库各方面的细节做深入探讨，涉及线程与传统 UNIX API——尤其是信号以及进程控制原语（fork()、exec()和_exit()）——之间的交互，同时对 Linux 上的两个 POSIX 线程实现（LinuxThreads 和 NPTL）加以概括，并特别指出了这些实现与 SUSv3 Pthreads 标准间的偏差所在。

33.1 线程栈

创建线程时，每个线程都有一个属于自己的线程栈，且大小固定。在 Linux/x86-32 架构上，除主线程外的所有线程，其栈的缺省大小均为 2MB。（在一些 64 位架构下，默认尺寸要大一些，例如，IA-64 有 32MB。）为了应对栈的增长（参考图 29-1），主线程栈的空间要大出许多。

偶尔，也需要改变线程栈的大小。在通过线程属性对象创建线程时，调用函数 pthread_attr_setstacksize()所设置的线程属性（29.8 节）决定了线程栈的大小。而使用与之相关的另一函数 pthread_attr_setstack()，可以同时控制线程栈的大小和位置，不过设置栈的地址将降低程序的可移植性。手册页（manual page）提供了对这些函数的具体说明。

更大的线程栈可以容纳大型的自动变量或者深度的嵌套函数调用（也许是递归调用），这是改变每个线程栈大小的原因之一。而另一方面，应用程序可能希望减小每个线程栈，以便进程可以创建更多的线程。例如，在 x86-32 系统中，用户（模式）可访问的虚拟地址空间是 3GB，而 2MB 的缺省栈大小则意味着最多只能创建 1500 个线程。（更为准确的最大值还视乎文本段、数据段、共享函数库等对虚拟内存的消耗量。）特定架构的系统上，可采用的线程栈大小最小值可以通过调用 sysconf(_SC_THREAD_STACK_MION)来确定。在 Linux/x86-32 上的 NPTL 实现中，该调用返回 16384。

在 NPTL 线程实现中，如果对线程栈尺寸资源限制（RLIMIT_STACK）的设置不同于 unlimited，那么创建线程时会以其作为默认值。对该限制的设置必须在运行程序之前，通常通过执行 shell 内建命令 ulimit - s 完成（在 C shell 下命令为 limit stacksize）。在主程序中调用 setrlimit()来设置限制的办法可能行不通，因为 NPTL 在调用 main()之前的运行时初始化期间就已经确定了默认的栈大小。

更多资源请访问稀酷客(www.ckook.com)

33.2 线程和信号

UNIX 信号模型是基于 UNIX 进程模型而设计的，问世比 Pthreads 要早几十年。自然而然，信号与线程模型之间存在一些明显的冲突。主要是因为，一方面，针对单线程进程要保持传统的信号语义（Pthreads 不应改变传统进程的信号语义），与此同时，又需要开发出适用于多线程进程环境的新信号模型。

信号与线程模型之间的差异意味着，将二者结合使用，将会非常复杂，应尽可能加以避免。尽管如此，有的时候还是必须在多线程程序中处理信号问题。本节将讨论信号与线程间的交互，并描述在多线程程序中处理信号的各种有效函数。

33.2.1 UNIX 信号模型如何映射到线程中

要了解 UNIX 信号如何映射到 Pthreads 模型，就需要了解，信号模型的哪些方面属于进程层面（由进程中的所有线程所共享），哪些方面是属于进程中的单个线程层面。如下是对其关键点的汇总。

- 信号动作属于进程层面。如果某进程的任一线程收到任何未经（特殊）处理的信号，且其缺省动作为 stop 或 terminate，那么将停止或者终止该进程的所有线程。
- 对信号的处置属于进程层面，进程中的所有线程共享对每个信号的处置设置。如果某一线程使用函数 sigaction()为某类信号（比如，SIGINT）创建了处理函数，那么当收到 SIGINT 时，任何线程都会去调用该处理函数。与之类似，如果将对信号的处置设置为忽略（ignore），那么所有线程都会忽略该信号。
- 信号的发送既可针对整个进程，也可针对某个特定线程。满足如下三者之一的信号当属面向线程的。
  - 信号的产生源于线程上下文中对特定硬件指令的执行（即 22.4 节所描述的硬件异常：SIGBUS、SIGFPE、SIGILL 和 SIGSEGV）。
  - 当线程试图对已断开的（broken pipe）管道进行写操作时所产生的 SIGPIPE 信号。
  - 由函数 pthread_kill()或 pthread_sigqueue()所发出的信号，这些函数（由 33.2.3 节描述）允许线程向同一进程下的其他线程发送信号。

    由其他机制产生的所有信号都是面向进程的。例如，其他进程通过调用 kill()或者 sigqueue()所发送的信号；用户键入特殊的终端字符所产生的信号，诸如 SIGINT 和 SIGTSTP；还有一些信号由软件事件产生，例如终端窗口大小的调整（SIGWINCH）或者定时器到期（例如，SIGALRM）。
- 当多线程程序收到一个信号，且该进程已然为此信号创建了信号处理程序时，内核会任选一条线程来接收这一信号，并在该线程中调用信号处理程序对其进行处理。这种行为与信号的原始语义保持了一致。让进程针对单个信号重复处理多次是没有意义的。
- 信号掩码（mask）是针对每个线程而言。（对于多线程程序来说，并不存在一个作用于整个进程范围的信号掩码，可以管理所有线程。）使用 Pthreads API 所定义的函数 pthread_sigmask()，各线程可独立阻止或放行各种信号。通过操作每个线程的信号掩码，应用程序可以控制哪些线程可以处理进程收到的信号。
- 针对为整个进程所挂起（pending）的信号，以及为每条线程所挂起的信号，内核都分别维护有记录。调用函数 sigpending()会返回为整个进程和当前线程所挂起信号的并集。在新创

更多资源请访问稀酷客(www.ckook.com)

建的线程中，每线程的挂起信号集合初始时为空。可将一个针对线程的信号仅向目标线程投送。如果该信号遭线程阻塞，那么它会一直保持挂起，直至线程将其放行（或者线程终止）。

- 如果信号处理程序中断了对 pthread_mutex_lock()的调用，那么该调用总是会自动重新开始。如果一个信号处理函数中断了对 pthread_cond_wait()的调用，则该调用要么自动重新开始（Linux 就是如此），要么返回 0，表示遭遇了假唤醒（如 30.2.3 节所述，此时，设计良好的应用程序会重新检查相应的判断条件并重新发起调用）。SUSv3 对这两个函数的行为要求与此处的描述一致。
- 备选信号栈是每线程特有的（参考 21.3 节对函数 sigaltstack()的描述）。新创建的线程并不从创建者处继承备选信号栈。

更确切地说，SUSv3 规定每个内核调度实体（KSE）都有一个单独的备选信号栈。在按 1:1 比例实现线程的系统中，例如 Linux，每一个线程对应一个 KSE（见 33.4 节）。

33.2.2 操作线程信号掩码

刚创建的新线程会从其创建者处继承信号掩码的一份拷贝。线程可以使用 pthread_sigmask()来改变或/并获取当前的信号掩码。

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
                    Returns 0 on success, or a positive error number on error
```

除了所操作的是线程信号掩码之外，pthread_sigmask()与 sigprocmask()的用法完全相同（见 20.10 节）。

SUSv3 特别指出，注明在多线程程序中使用函数 sigprocmask()，其结果是未定义的，也无法保证程序的可移植性。事实上，函数 sigprocmask()和 pthread_sigmask()在包括 Linux 在内的很多系统实现中是相同的。

33.2.3 向线程发送信号

函数 pthread_kill()向同一进程下的另一线程发送信号 sig。目标线程由参数 thread 标识。

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
                    Returns 0 on success, or a positive error number on error
```

因为仅在同一进程中可保证线程 ID 的唯一性（参见 29.5 节），所以无法调用 pthread_ kill()向其他进程中的线程发送信号。

在实现函数 pthread_kill()时，使用了 Linux 特有的 tgkill（tgid，tid，sig）系统调用，将信号 sig 发送给由 tid（由 gettid()所返回的内核线程 ID）标识的线程，该线程从属于由 tgid 标识的线程组中。更多细节，请参考 tgkill(2)手册页。

Linux 特有的函数 pthread_sigqueue()将 pthread_kill()和 sigqueue()的功能合二为一（见

更多资源请访问稀酷客(www.ckook.com)

22.8.1 节）：向同一进程中的另一线程发送携带数据的信号。

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

int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

Returns 0 on success, or a positive error number on error

与函数 pthread_kill()一样，sig 表示将要发送的信号，thread 标识目标线程。参数 value 则指定了伴随信号的数据，其使用方式与函数 sigqueue()中的对应参数相同。

函数 pthread_sigqueue()从 2.11 版开始加入 glibc 函数库中，同时需要内核的支持。始于 Linux 2.6.31，内核通过系统调用 rt_tgsigqueueinfo()来提供这一支持。

33.2.4 妥善处理异步信号

第 20 章至第 22 章所探讨的各种因素（诸如，可重入问题、重启遭中断的系统调用，以及避免竞争条件），当使用信号处理函数对异步产生的信号加以处理时，这些都将导致情况变得复杂。另外，没有任何 Pthreads API 属于异步信号安全（async-signal-safe）函数，均无法在信号处理函数（21.1.2 节）中安全加以调用。因为这些原因，所以当多线程应用程序必须处理异步产生的信号时，通常不应该将信号处理函数作为接收信号到达的通知机制。相反，推荐的方法如下。

- 所有线程都阻塞进程可能接收的所有异步信号。最简单的方法是，在创建任何其他线程之前，由主线程阻塞这些信号。后续创建的每个线程都会继承主线程信号掩码的一份拷贝。
- 再创建一个专用线程，调用函数 sigwaitinfo()、sigtimedwait()或 sigwait()来接收收到的信号。22.10 节对 sigwaitinfo()和 sigtimedwait()做了说明。下面则对 sigwait()有所描述。

这一方法的优势在于，同步接收异步产生的信号。当接收到信号时，专有线程可以安全地修改共享变量（在互斥量的保护之下），并可调用并非异步信号安全（non-async-signal-safe）的函数。也可以就条件变量发出信号，并采用其他线程或进程的通讯及同步机制。

函数 sigwait()会等待 set 所指信号集合中任一信号的到达，接收该信号，且在参数 sig 中将其返回。

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

Returns 0 on success, or a positive error number on error

除了以下不同以外，sigwait()的操作与 sigwaitinfo()相同。

- 函数 sigwait()只返回信号编号，而非返回一个描述信号信息的 siginfo_t 类型结构。
- 并且返回值与其他线程相关函数保持一致（而非传统 UNIX 系统调用返回的 0 或-1）。

如有多个线程在调用 sigwait()等待同一信号，那么当信号到达时只有一个线程会实际接收到，也无法确定收到信号的会是哪条线程。

33.3 线程和进程控制

与信号机制类似，exec()、fork()和 exit()的问世均早于 Pthreads API。接下来的段落将指出

更多资源请访问稀酷客(www.ckook.com)

在多线程程序中使用此类系统调用所应关注的细节。

线程和 exec()

只要有任一线程调用了 exec()系列函数之一时，调用程序将被完全替换。除了调用 exec()的线程之外，其他所有线程都将立即消失。没有任何线程会针对线程特有数据执行解构函数（destructor），也不会调用清理函数（cleanup handler）。该进程的所有互斥量（为进程私有）和属于进程的条件变量都会消失。调用 exec()之后，调用线程的线程 ID 是不确定的。

线程和 fork()

当多线程进程调用 fork()时，仅会将发起调用的线程复制到子进程中。（子进程中该线程的线程 ID 与父进程中发起 fork()调用线程的线程 ID 相一致。）其他线程均在子进程中消失，也不会为这些线程调用清理函数以及针对线程特有数据的解构函数。这将导致如下一些问题。

- 虽然只将发起调用的线程复制到子进程中，但全局变量的状态以及所有的 Pthreads 对象（如互斥量、条件变量等）都会在子进程中得以保留。（因为在父进程中为这些 Pthreads 对象分配了内存，而子进程则获得了该内存的一份拷贝。）这会导致很棘手的问题。例如，假设在调用 fork()时，另一线程已然锁定了某一互斥量，且对某一全局数据结构的更新也做到了一半。此时，子进程中的该线程无法解锁这一互斥量（因为其并非该互斥量的属主），如果试图获取这一互斥量，线程会遭阻塞。此外，子进程中的全局数据结构拷贝可能也处于不一致状态，因为对其进行更新的线程在执行到一半时消失了。
- 因为并未执行清理函数和针对线程特有数据的解构函数，多线程程序的 fork()调用会导致子进程的内存泄漏。另外，子进程中的线程很可能无法访问（父进程中）由其他线程所创建的线程特有数据项，因为（子进程）没有相应的引用指针。

由于这些问题，推荐在多线程程序中调用 fork()的唯一情况是：其后紧跟对 exec()的调用。因为新程序会覆盖原有内存，exec()将导致子进程的所有 Pthreads 对象消失。

对于那些必须执行 fork()，而其后又无 exec()跟随的程序来说，Pthreads API 提供了一种机制：fork 处理函数（handler）。可以利用函数 pthread_atfork()来创建 fork 处理函数，格式如下：

```
pthread_atfork(prepare_func, parent_func, child_func);
```

每一次 pthread_atfork()调用都会将 prepare_func 添加到一个函数列表中，在调用 fork()创建新的子进程之前，会（按与注册次序相反的顺序）自动执行该函数列表中的函数。与之类似，会将 parent_func 和 child_func 添加到一函数列表中，在 fork()返回前，将分别在父、子进程中（按注册顺序）自动运行。

在使用线程的函数库中，有时候 fork 处理函数很实用。如果没有这一机制，对于那些随意调用了此函数库和 fork()，又对函数库创建的其他线程一无所知的应用程序，函数库还真是无计可施。

调用 fork()所产生的子进程从调用 fork()的线程处继承 fork 处理函数。执行 exec()期间，fork 处理函数将不再保留（因为处理函数的代码会在执行 exec()的过程中遭到覆盖）。

关于 fork 处理函数及其使用的更多细节可以参考[Butenhof，1996]。

> 在 Linux 上，如果使用 NPTL 线程库的程序执行了 vfork()，那么将不再调用 fork 处理函数。不过，在使用 LinuxThreads 程序的同一情况下却有效。

线程与 exit()

如果任何线程调用了 exit()，或者主线程执行了 return，那么所有线程都将消失，也不会

更多资源请访问稀酷客(www.ckook.com)

执行线程特有数据的解构函数以及清理函数。

33.4 线程实现模型

本节将涉及一些理论知识，简要阐述实现线程 API 的 3 种不同模型，从而为 33.5 节中关于 Linux 线程实现的讨论提供必要的背景知识。这 3 种实现模型的差异主要集中在线程如何与内核调度实体（KSE，Kernel Scheduling Entity）相映射。KSE 是内核分配 CPU 以及其他系统资源的（对象）单位。（在早于线程而出现的传统 UNIX 中，KSE 等同于进程。）

多对一（M:1）实现（用户级线程）

在 M:1 线程实现中，关乎线程创建、调度以及同步（互斥量的锁定，条件变量的等待等）的所有细节全部由进程内用户空间（user-space）的线程库来处理。对于进程中存在的多个线程，内核一无所知。

M:1 实现的优势不多，其中最大的优点在于，许多线程操作（例如线程的创建和终止、线程上下文间的切换、互斥量以及条件变量操作）速度都很快，因为无需切换到内核模式。此外，由于线程库无需内核支持，所以 M:1 实现在系统间的移植相对要容易一些。

不过，M:1 实现也存在一些严重缺陷。

- 当一线程发起系统调用（如 read()）时，控制由用户空间的线程库转交给内核。这就意味着，如果 read()调用遭到阻塞，那么所有的线程都会被阻塞。
- 内核无法调度进程中的这些线程。因为内核并不知晓进程中存在这些线程，也就无法在多处理器平台上将各线程调度给不同的处理器。另外，也不可能将一进程中某线程的优先级调整为高于其他进程中的线程，这是没有意义的，因为对线程的调度完全在进程中处理。

一对一（1:1）实现（内核级线程）

在 1:1 线程实现中，每一线程映射一个单独的 KSE。内核分别对每个线程做调度处理。线程同步操作通过内核系统调用实现。

1:1 实现消除了 M:1 实现的种种弊端。遭阻塞的系统调用不会导致进程的所有线程被阻塞，在多处理器硬件平台上，内核还可以将进程中的多个线程调度到不同的 CPU 上。

不过，因为需要切换到内核模式，所以诸如线程创建、上下文切换以及同步操作就要慢一些。另外，为每个线程分别维护一个 KSE 也需要开销，如果应用程序包含大量线程，则可能对内核调度器造成严重的负担，降低系统的整体性能。

尽管有这些缺点，1:1 实现通常更胜于 M:1 实现。LinuxThreads 和 NPTL 都采用 1:1 模型。

在 NPTL 的开发期间，为了使得包含数以千计线程的进程得以高效运行，投入了巨大的努力，对内核调度器进行了重写并设计了新的线程实现。后续的测试也显示了预期目标的达成。

多对多（M:N）实现（两级模型）

M:N 实现旨在结合 1:1 和 M:1 模型的优点，避免二者的缺点。

在 M:N 模型中，每个进程都可拥有多个与之相关的 KSE，并且也可以把多个线程映射到一个 KSE。这种设计允许内核将同一应用的线程调度到不同的 CPU 上运行，同时也解决了随线程数量而放大的性能问题。

M:N 模型的最大问题是过于复杂。线程调度任务由内核及用户空间的线程库共同承担，二者之

更多资源请访问稀酷客(www.ckook.com)

间势必要进行分工协作和信息交换。在 M:N 模型下，按照 SUSv3 标准要求来管理信号也极为复杂。

最初曾考虑采用 M:N 模型来实现 NPTL 线程库，但若要保证 Linux 调度器即使在处理大量 KSE 的情况下也能应对自如，则需要对内核所作的改动范围过大，可能也没有必要，故而否决了这一方案。

33.5 Linux POSIX 线程的实现

针对 Pthreads API：Linux 下有两种实现。

- LinuxThreads：这是最初的 Linux 线程实现，由 Xavier Leroy 开发。
- NPTL（Native POSIX Threads Library）：这是 Linux 线程实现的现代版，由 Ulrich Drepper 和 Ingo Molnar 开发，以取代 LinuxThreads。NPTL 的性能优于 LinuxThreads，也更符合 SUSv3 的 Pthreads 标准。对 NPTL 的支持需要修改内核，这始于 Linux 2.6。

一度，人们曾将由 IBM 开发的另一线程实现——NGPT（Next Generation POSIX Threads）视为 LinuxThreads 的继任者。NGPT 采用 M:N 模型设计，性能明显优于 LinuxThreads。不过，NPTL 的开发者决意推出新的实现。NPTL 的设计方法有所调整，采用 1:1 模型，性能也优于 NGPT。随着 NPTL 的发布，对 NGPT 的开发也随之终止。

后续各节将讨论这两种实现的更多细节，并将二者对 SUSv3 Pthreads 标准的背离之处一一指处。

此处值得强调的是：LinuxThreads 实现已经过时，并且 glibc 从 2.4 版本开始也已不再支持它，所有新的线程库开发都基于 NPTL。

33.5.1 LinuxThreads

多年以来，LinuxThreads 曾一直是 Linux 上的主流线程实现，也能够满足各种线程化应用程序实现的需要。LinuxThreads 实现的要点如下。

- 线程的创建使用了 clone()，并指定有如下标志：

  ```
  CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
  ```

  这意味着，LinuxThreads 线程共享虚拟内存、文件描述符、文件系统相关信息（umask，根目录和当前工作目录）以及信号处置。不过，线程间并不共享进程 ID 和父进程 ID。
- 除了由应用程序创建的线程以外，LinuxThreads 还会创建一个附加的管理线程，负责处理其他线程的创建和终止。
- LinuxThreads 利用信号来处理内部的操作。对于支持实时信号的内核来说（Linux 2.2 及以后版本），会使用头 3 个实时信号。对于老版本内核，则使用 SIGUSR1 和 SIGUSR2。应用程序不能使用这些信号。（对于各种线程同步操作而言，使用信号会导致较高延迟。）

LinuxThreds 对标准行为的背离之处

LinuxThreads 在很多方面与 SUSv3 Pthreads 标准并不一致。（LinuxThreads 实现受限于其开发时可用的内核特性，而在此范围内，则会尽量保持一致。）以下列表对背离之处作了概述。

- 在同一进程的不同线程中调用 getpid()会返回不同的值。调用 getppid()则反应了如下事实：除了主线程以外的所有线程都由进程的管理线程创建（getppid()会返回管理线程

更多资源请访问稀酷客(www.ckook.com)

的进程 ID）。在主线程和其他线程中，对 getppid()调用的返回值相同。

- 如果某线程调用 fork()创建了一子进程，那么任何其他线程都可使用 wait()或类似技术来获取子进程的终止状态。不过，事实并非如此，只有创建子进程的线程才能使用 wait()。
- 如果某线程执行了 exec()，那么按照 SUSv3 要求，将终止所有其他线程。不过，只要调用 exec()的是主线程之外的线程，那么产生进程则与调用线程拥有相同的进程 ID，而与主线程的进程 ID 不同。而依照 SUSv3 标准，该进程 ID 应与主线程的进程 ID 保持一致。
- 线程之间不会共享凭证（用户 ID 与用户组 ID）。当一多线程进程执行一个 set- user- ID 程序时，将导致线程之间无法通过 pthread_kill()来发送信号，因为这两个线程的凭证已经发生了改变，发送线程不再有权发信号给目标线程（请参考图 20-2）。另外，由于 LinuxThreads 实现在内部使用了信号，一旦线程改变了自身的凭证，那么各种 Pthreads 操作有可能失败或者挂起（hang）。
- 还未能顾及 SUSv3 关于线程与信号间交互规范的各个方面。
  - 采用 kill()或者 sigqueue()向某进程发送的信号，应该由目标进程中不阻塞该信号的任意线程来接收和处理。不过，因为 LinuxThreads 线程的进程 ID 不同，所以只能将信号送给特定线程。要是该线程阻塞这一信号，即使其他线程并不阻塞此信号，它也会一直保持挂起（pending）。
  - LinuxThreads 并不支持信号为整个进程挂起的概念，只支持每个线程分别挂起信号。
  - 如果信号针对包含某个多线程应用的进程组，那么应用中的所有线程（即每个创建了信号处理函数的线程）都会处理该信号，而不是由（任意）某个线程去处理。例如，键入某个终端字符就会产生针对前台进程组的任务控制（job-control）信号。
  - 背选信号栈设置（由 sigaltstack()建立）是针对每个线程的。不过，如果新线程不慎从 pthread_create()调用者那里继承了备选信号栈设置，那么这两个线程将共享同一备选信号栈。SUSv3 规定新线程启动时不应定义备选信号栈。LinuxThreads 这一“违规”操作的后果是，如果两个线程恰巧在它们共享的备选信号栈中同时处理不同的信号，很可能会导致混乱（例如，程序崩溃）。这一问题可能非常难以重现，也很难调试，因为问题的出现依赖于在同一时点处理两个信号，这种情况极其罕见。

在使用 LinuxThreads 的程序中，新线程可以通过调用 sigaltstack()来确保使用与其创建者线程不同的备选信号栈（或许根本就没有栈）。不过，可移植程序（以及创建线程的库函数）并不知道这些，因为在其他实现上这并非必须。另外，即使采用这种技术，依然可能产生竞争条件：新线程在有机会调用 sigaltstack()之前依然有可能接收并处理备选栈上的信号。

- 线程不共享一般的任务号以及进程组号。不能利用 setsid()和 setpgid()系统调用去改变多线程程序中的会话号以及进程组号。
- 使用 fcntl()建立的记录锁也不能共享。重复地对同一类型的锁的请求是无法合并在一起执行的。
- 线程不共享资源的限制。SUSv3 定义资源限制是一种进程范围的属性。
- 函数 times()返回的 CPU 时间以及由 getrusage()返回的资源使用信息也都是针对每一个线程的。这些系统调用应该返回这个进程的总量。
- 一些版本的 ps(1)会显示进程中的所有线程（包括管理线程），作为单独的项，且它们的进程号也不相同。
- 线程之间并不共享由 setpiority()设置的 nice 值。

更多资源请访问稀酷客(www.ckook.com)

- 使用 setitimer()创建的间隔定时器也无法在线程之间共享。
- 线程之间不能共享 System V 信号量还原（semadj）值。

LinuxThreads 的其他问题

除了以上与 SUSv3 标准的偏差外，LinuxThreads 实现还有如下问题。

- 如果管理线程被杀掉，那么余下的线程只能手工清理。
- 多线程程序的核心转储（core dump）可能并不包含所有的线程（甚至可能也不包含触发转储的线程）。
- 只有从主线程调用非标准的 ioctl()TIOCNOTTY 操作才能移除进程与控制终端的关联。

33.5.2 NPTL

设计 NPTL 是为了弥补 LinuxThreads 的大部分的缺陷。特别是如下部分。

- NPTL 更接近 SUSv3 Pthreads 标准。
- 使用 NPTL 的有大量线程的应用程序的性能要远优于 LinuxThreads。

NPTL 允许应用程序创建大量的线程。NPTL 实现的测试程序可以创建 10 万个线程。对于 LinuxThreads，实际线程数量的限制大约是一两千个。（应当承认，很少有程序需要创建这个多的线程。）

NPTL 实现的开发从 2002 年开始，大约在第 2 年完成。同时，Linux 内核为适应 NPTL 也做了各种调整。这些变动出现在 Linux 2.6 内核，并对 NPTL 的如下方面提供支持。

- 改进线程组的实现（28.2.1 节）。
- 增加 futex 作为一种同步机制（futex 作为一种通用机制，并不只是为 NPTL 而设计）。
- 增加新的系统调用（get_thread_area()和 set_thread_area()）以便支持线程本地存储。
- 支持线程化的核心转储和对多线程程序的调试功能。
- 修改并支持与 Pthreads 模型一样的信号处理。
- 增加新的系统调用 exit_group()，可以终止进程中的所有线程（从 glibc2.3 开始，库函数 exit()是 exit_group()的包装函数，而函数 pthread_exit()调用真正的内核系统调用 _exit()，仅终止调用的线程）。
- 重写内核调度程序以便能够有效地调度和处理大量（上千个）KSE 的情况。
- 提升内核进程终止的执行效率。
- 扩展系统调用 clone()（28.2 节）。

NPTL 实现最基本的部分如下：

- 线程使用函数 clone()创建并指定如下标志。

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |
CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

NPTL 比 LinuxThreads 可以共享更多的信息。标志 CLONE_THREAD 意思是新线程与创建者线程属于同一个线程组，并且共享同样的进程号以及父进程号。CLONE_SYSVSEM 表示新线程与创建者共享 System V 信号量还原值。

使用 ps(1)列出一个运行在 NPTL 下的多线程程序时，只会输出一条记录。为了看到进程中的线程信息，可以使用 ps-L 选项。

更多资源请访问稀酷客(www.ckook.com)

- 实现的内部使用前两个实时信号。应用程序不能使用这些信号。

其中一个信号用来实现线程的取消功能。另一个信号用于确保进程中的所有线程拥有同样的用户号和用户组号。而在内核模式，线程有不同的用户和组凭证。所以 NPTL 实现对每一个改变用户号和用户组号的系统调用（setuid()、setresuid()等以及类似的组操作函数）的包装函数都做了修改，以确保进程中的所有线程都做了相应的改变。

- 与 LinuxThreads 不同，NPTL 并不需要管理线程。

NPTL 标准一致性

这些改变意味着 NPTL 比 LinuxThreads 更接近 SUSv3 标准。在作者撰写本书的时候，遗留以下不一致的地方。

- 线程之间不共享 nice 值。

在早期的 2.6.x 内核中，还有一些额外的不一致的地方。

- 内核版本 2.6.16 之前，备选信号栈是针对每个线程的，但是新的线程从调用 pthread_create()函数的线程那里错误地继承了备选信号栈设置（通过 sigaltstack()产生），导致出现两个线程共享同一备选信号栈的问题。
- 内核 2.6.16 之前，只有一个线程组的组长（即主线程）可以通过调用函数 setsid()启动一个新的会话。
- 内核 2.6.16 之前，只有一个线程组的组长可以使用函数 setpgid()让宿主进程成为进程组主进程。
- 早于 2.6.12 的内核版本，在同一进程的线程之间无法共享使用 setitimer()创建的间隔定时器。
- 早于 2.6.10 的内核版本，同一进程中的所有线程并不共享资源限制的设置。
- 早于 2.6.9 的内核版本，函数 times()返回的 CPU 时间以及函数 getrusage()返回的资源使用信息都是针对每个线程的。

NPTL 设计与 LinuxThreads ABI 兼容。那些与提供 LinuxThreads 的 GNU C 库链接的程序换用 NPTL 时无需再重新编译。不过当程序运行在 NPTL 环境时某些行为可能会有些不同，主要是因为 NPTL 更接近于 SUSv3 Pthreads 标准。

33.5.3 哪一种线程实现

一些 Linux 发布版本附带包换 LinuxThreads 和 NPTL 的 GNU C 库，依据系统运行在何种内核上动态地确定链接哪一种 GNUC 库。（这些发布版本有其历史原因，自版本 2.4 后 glibc 不再提供 LinuxThreads。）所以有时候可能需要回答以下的问题。

- 特定的 Linux 发布版本中，哪一种线程实现是有效的？
- 在既提供 LinuxThreads 也提供 NPTL 的 Linux 发布版本中，缺省使用哪一种？如何明确地选择一个应用程序所使用的线程库？

找出线程实现

可以通过一些技术去找出某个特定系统使用的线程实现，也可以发现在提供两种线程实现的系统上运行的程序默认使用的实现版本。

在提供 glibc 2.3.2 或后续版本的系统上，可以使用如下命令找出系统提供的线程实现，如

更多资源请访问稀酷客(www.ckook.com)

果提供两种实现的，则显示默认的那个：

```
$ getconf GNU_LIBPTHREAD_VERSION
```

在只有 NPTL 或者将其作为默认实现的系统上，将会显示类似下面的信息：

```
$ ldd /bin/ls | grep libc.so
        libc.so.6 => /lib/tls/libc.so.6 (0x40050000)
```

NPTL 2.3.4

自 glibc 2.3.2 以来，程序可以通过 confstr(3)获得类似的信息，并取得 glibc 特定的配置变量_CS_GNU_LIBPTHREAD_VERSION 的值。

使用老版本 GNUC 库的系统上，需要做一些额外的动作。首现，下面的命令可以被用来显示程序运行时使用的 GNUC 库的路径（这里使用标准程序 ls 作为例子，其位置为/bin/ls）：

```
$ /lib/tls/libc.so.6 | egrep -i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

GNU C 库的路径显示在=>之后。如果作为命令执行这个路径的程序，那么 glibc 将会显示关于自身的一系列信息。可以通过 grep 选取与线程实现相关的信息。

在 egrep 正则表达式（regular expression）中包含 nptl，是因为某些包含 NPTL 的 glibc 发布显示如下的字符串信息：

```
NPTL 0.61 by Ulrich Drepper
```

因为 glibc 路径会随着不同的 Linux 发布而改变，可以使用 shell 的替换功能来产生一个显示 Linux 系统上使用的线程实现信息的命令行：

```
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

选择程序使用的线程实现

在即提供 NPTL 也提供 LinuxThreads 的 Linux 系统上，能够明确地控制具体使用的线程实现有时候非常地有用。最常见的例子是，当遇到一个旧有的依赖于某些 LinuxThreads 行为（可能非标准）的程序时，要能够强制程序使用指定的线程实现，而不是默认的 NPTL。

出于这个目的，可以使用一个动态链接器（dynamic linker）能够理解的特定的环境变量 LD_ASSUME_KERNEL。顾名思义，这个环境变量告诉动态链接器就好像运行在特定的 Linux 内核版本上一样。通过指定并不提供 NPTL 支持的内核版本（例如 2.2.5）可以确保 LinuxThreads 被使用到。所以可以使用如下命令运行一个基于 LinuxThreads 的多线程应用程序：

```
$ LD_ASSUME_KERNEL=2.2.5 ./prog
```

当环境变量设置与之前提及的显示使用的线程实现信息的命令行一起使用时，可以看到如下的一些信息：

```
$ export LD_ASSUME_KERNEL=2.2.5
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
        linuxthreads-0.10 by Xavier Leroy
```

> 可以通过 LD_ASSUME_KERNEL 设置的内核版本号的范围受一些限制因素的制约。在一些提供 NPTL 和 LinuxThreads 的一般发布版本中，将版本号指定为 2.2.5 已经足够保证会使用 LinuxThreads。此环境变量更完整的描述请参考 http://people. redhat. com/ drepper/assumkernel.html。

33.6 Pthread API 的高级特性

Pthreads API 还包括一些如下的高级特性。

- 实时调度（Realtime scheduling）：可以对线程设置实时调度策略以及优先级。类似于 35.3 节中描述的进程的实时调度的系统调用。
- 进程共享互斥量和条件变量：SUSv3 规定进程之间共享互斥量和条件变量是可选的（不只是针对进程中的线程而言）。这种情况，条件变量或者互斥量必须在进程间的共享内存中分配。NPTL 支持这种特性。
- 高级线程同步原语：这些功能包括障碍（barrier）、读写锁（read-write lock）以及自旋锁（spin lock）。

关于这些特性的更多的细节请参考[Butenhof，1996]。

33.7 总结

不要将线程与信号混合使用，只要可能多线程应用程序的设计应该避免使用信号。如果多线程应用必须处理异步信号的话，通常最简洁的方法是所有的线程都阻塞信号，创建一个专门的线程调用 sigwait()函数（或者类似的函数）来接收收到的信号。这个线程就可以安全地执行像修改共享内存（处于互斥量的保护之下）和调用非异步信号安全的函数。

一般有两种有效的 Linux 线程实现：LinuxThreads 和 NPTL。LinuxThreads 多年以来一直为 Linux 使用，但是很多方面并不遵循 SUSv3 的标准，而且已经过时。全新的 NPTL 实现更接近 SUSv3 标准并且提供更优的性能，现今的 Linux 发布也都提供这种实现。

更多的信息

请参考列于 29.10 节的更多的信息来源。

LinuxThreads 的作者编写了实现文档，可以在如下地址找到：http:// pauillac.inria.fr/～xleroy/linuxthreads/。NPTL 实现在如下的论文（有些过时）中有所描述：http:// people. redhat. com/drepper/nptl-design.pdf。

33.8 练习

33-1. 编写程序以便证明：作为函数 sigpending()的返回值，同一个进程中的的不同线程可以拥有不同的 pending 信号。可以使用函数 pthread_kill()分别发送不同的信号给阻塞这些信号的两个不同的线程，接着调用 sigpending()方法并显示这些 pending 信号的信息。（可能会发现程序清单 20-4 中函数的作用。）

33-2. 假设一个线程使用 fork()创建了一个子进程。当子进程终止时，可以保证由此产生的 SIGCHLD 信号一定会发送给调用 fork()的线程吗（可以用进程中的其他线程做对比）？

更多资源请访问稀酷客(www.ckook.com)