# 第8章 进 程 控 制

## 8.1 引言

本章介绍UNIX的进程控制，包括创建新进程、执行程序和进程终止。还将说明进程的各种ID——实际、有效和保存的用户和组ID，以及它们如何受到进程控制原语的影响。本章也包括了解释器文件和system函数。本章以大多数UNIX系统所提供的进程会计机制结束。这使我们从一个不同角度了解进程控制功能。

## 8.2 进程标识

每个进程都有一个非负整型的唯一进程ID。因为进程ID标识符总是唯一的，常将其用做其他标识符的一部分以保证其唯一性。 5.13节中的tmpnam函数将进程ID作为名字的一部分创建一个唯一的路径名。

有某些专用的进程：进程ID 0是调度进程，常常被称为交换进程(swapper)。该进程并不执行任何磁盘上的程序——它是内核的一部分，因此也被称为系统进程。进程 ID 1通常是init进程，在自举过程结束时由内核调用。该进程的程序文件在 UNIX的早期版本中是/etc/init，在较新版本中是/sbin/init。此进程负责在内核自举后起动一个 UNIX系统。init通常读与系统有关的初始化文件(/etc/rc*文件)，并将系统引导到一个状态(例如多用户)。init进程决不会终止。它是一个普通的用户进程(与交换进程不同，它不是内核中的系统进程 )，但是它以超级用户特权运行。本章稍后部分会说明init如何成为所有孤儿进程的父进程。

在某些UNIX的虚存实现中，进程ID 2是页精灵进程(pagedaemon)。此进程负责支持虚存系统的请页操作。与交换进程一样，页精灵进程也是内核进程。

除了进程ID，每个进程还有一些其他标识符。下列函数返回这些标识符。

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);                          返回：调用进程的进程ID

pid_t getppid(void);                         返回：调用进程的父进程ID

uid_t getuid(void);                          返回：调用进程的实际用户ID

uid_t geteuid(void);                         返回：调用进程的有效用户ID

gid_t getgid(void);                          返回：调用进程的实际组ID

gid_t getegid(void);                         返回：调用进程的有效组ID
```

注意，这些函数都没有出错返回，在下一节中讨论 fork函数时，将进一步讨论父进程ID。4.4节中已讨论了实际和有效用户及组ID。

## 8.3 fork函数

一个现存进程调用fork函数是UNIX内核创建一个新进程的唯一方法(这并不适用于前节提及的交换进程、init进程和页精灵进程。这些进程是由内核作为自举过程的一部分以特殊方式创建的)。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

返回：子进程中为0，父进程中为子进程ID，出错为-1

由fork创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新子进程的进程ID。将子进程ID返回给父进程的理由是：因为一个进程的子进程可以多于一个，所以没有一个函数使一个进程可以获得其所有子进程的进程ID。fork使子进程得到返回值0的理由是：一个进程只会有一个父进程，所以子进程总是可以调用getppid以获得其父进程的进程ID(进程ID 0总是由交换进程使用，所以一个子进程的进程ID不可能为0)。

子进程和父进程继续执行fork之后的指令。子进程是父进程的复制品。例如，子进程获得父进程数据空间、堆和栈的复制品。注意，这是子进程所拥有的拷贝。父、子进程并不共享这些存储空间部分。如果正文段是只读的，则父、子进程共享正文段(见7.6节)。

现在很多的实现并不做一个父进程数据段和堆的完全拷贝，因为在 fork之后经常跟随着exec。作为替代，使用了在写时复制(Copy-On-Write, COW)的技术。这些区域由父、子进程共享，而且内核将它们的存取许可权改变为只读的。如果有进程试图修改这些区域，则内核为有关部分，典型的是虚存系统中的“页”，做一个拷贝。Bach〔1986〕的9.2节和Leffler等〔1989〕的5.7节对这种特征做了更详细的说明。

**实例**

程序8-1例示了fork函数。如果执行此程序则得到：

```
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89 子进程的变量值改变了
pid = 429, glob = 6, var = 88 父进程的变量值没有改变
$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

一般来说，在fork之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。如果要求父、子进程之间相互同步，则要求某种形式的进程间通信。在程序 8-1中，父进程使自己睡眠2秒钟，以此使子进程先执行。但并不保证 2秒钟已经足够，在8.8节说明竞争条件时，还将谈及这一问题及其他类型的同步方法。在 10.6节中，在fork之后将用信号使父、

子进程同步。

注意，程序8-1中fork与I/O函数之间的关系。回忆第3章中所述，write函数是不带缓存的。因为在fork之前调用write，所以其数据写到标准输出一次。但是，标准 I/O库是带缓存的。回忆一下5.12节，如果标准输出连到终端设备，则它是行缓存的，否则它是全缓存的。当以交互方式运行该程序时，只得到printf输出的行一次，其原因是标准输出缓存由新行符刷新。但是当将标准输出重新定向到一个文件时，却得到printf输出行两次。其原因是，在fork之前调用了printf一次，但当调用fork时，该行数据仍在缓存中，然后在父进程数据空间复制到子进程中时，该缓存数据也被复制到子进程中。于是那时父、子进程各自有了带该行内容的缓存。在 exit之前的第二个printf将其数据添加到现存的缓存中。当每个进程终止时，其缓存中的内容被写到相应文件中。

**程序8-1 fork函数实例**

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");     /* we don't flush stdout */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {         /* child */
        glob++;                  /* modify variables */
        var++;
    } else
        sleep(2);                /* parent */

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

### 文件共享

对程序8-1需注意的另一点是：在重新定向父进程的标准输出时，子进程的标准输出也被重新定向。实际上， fork的一个特性是所有由父进程打开的描述符都被复制到子进程中。父、子进程每个相同的打开描述符共享一个文件表项(见图3-3)。

考虑下述情况，一个进程打开了三个不同文件，它们是：标准输入、标准输出和标准出错。在从fork返回时，我们有了如图8-1中所示的安排。

这种共享文件的方式使父、子进程对同一文件使用了一个文件位移量。考虑下述情况：一个进程fork了一个子进程，然后等待子进程终止。假定，作为普通处理的一部分，父、子进程都向标准输出执行写操作。如果父进程使其标准输出重新定向 (很可能是由shell实现的)，那么子进程写到该标准输出时，它将更新与父进程共享的该文件的位移量。在我们所考虑的例子中，当父进程等待子进程时，子进程写到标准输出；而在子进程终止后，父进程也写到标准输出上，

并且知道其输出会添加在子进程所写数据之后。如果父、子进程不共享同一文件位移量，这种形式的交互就很难实现。

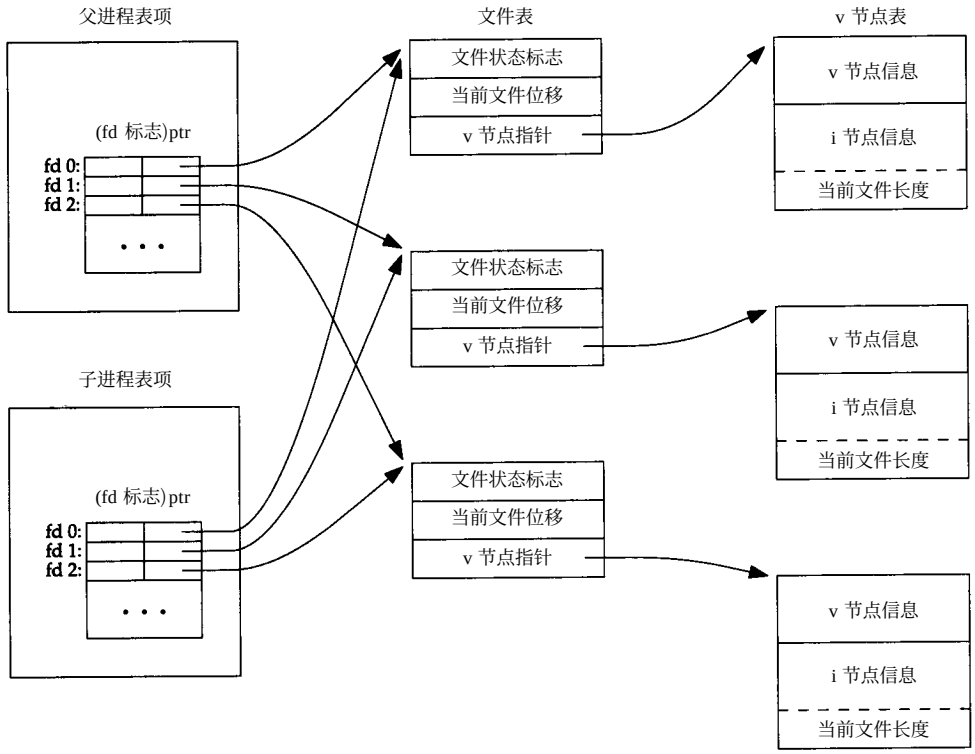


图8-1 fork之后父、子进程之间对打开文件的共享

如果父、子进程写到同一描述符文件，但又没有任何形式的同步（例如使父进程等待子进程），那么它们的输出就会相互混合（假定所用的描述符是在 fork之前打开的）。虽然这种情况是可能发生的（见程序8-1），但这并不是常用的操作方式。

在fork之后处理文件描述符有两种常见的情况：

(1) 父进程等待子进程完成。在这种情况下，父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的文件位移量已做了相应更新。

(2) 父、子进程各自执行不同的程序段。在这种情况下，在 fork之后，父、子进程各自关闭它们不需使用的文件描述符，并且不干扰对方使用的文件描述符。这种方法是网络服务进程中经常使用的。

除了打开文件之外，很多父进程的其他性质也由子进程继承：

- 实际用户ID、实际组ID、有效用户ID、有效组ID。
- 添加组ID。
- 进程组ID。
- 对话期ID。
- 控制终端。
- 设置-用户-ID标志和设置-组-ID标志。
- 当前工作目录。

- 根目录。
- 文件方式创建屏蔽字。
- 信号屏蔽和排列。
- 对任一打开文件描述符的在执行时关闭标志。
- 环境。
- 连接的共享存储段。
- 资源限制。

父、子进程之间的区别是：

- fork的返回值。
- 进程ID。
- 不同的父进程ID。
- 子进程的tms_utime,tms_stime,tms_cutime以及tms_ustime设置为0。
- 父进程设置的锁，子进程不继承。
- 子进程的未决告警被清除。
- 子进程的未决信号集设置为空集。

其中很多特性至今尚未讨论过，我们将在以后几章中对它们进行说明。

使fork失败的两个主要原因是：(a)系统中已经有了太多的进程(通常意味着某个方面出了问题)，或者(b)该实际用户ID的进程总数超过了系统限制。回忆表2-7，其中CHILD_MAX规定了每个实际用户ID在任一时刻可具有的最大进程数。

fork有两种用法：

(1) 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待委托者的服务请求。当这种请求到达时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求。

(2) 一个进程要执行一个不同的程序。这对 shell是常见的情况。在这种情况下，子进程在从fork返回后立即调用exec(我们将在8.9节说明exec)。

某些操作系统将(2)中的两个操作(fork之后执行exec)组合成一个，并称其为spawn。UNIX将这两个操作分开，因为在很多场合需要单独使用 fork，其后并不跟随exec。另外，将这两个操作分开，使得子进程在fork和exec之间可以更改自己的属性。例如I/O重新定向、用户ID、信号排列等。在第14章中有很多这方面的例子。

## 8.4 vfork函数

vfork函数的调用序列和返回值与fork相同，但两者的语义不同。

> vfork起源于较早的4BSD虚存版本。在Leffler 等〔1989〕的5.7节中指出：“虽然它是特别有效率的，但是vfork的语义很奇特，通常认为它具有结构上的缺陷。”
>
> 尽管如此SVR4和4.3+BSD仍支持vfork。
>
> 某些系统具有头文件<vfork.h>，当调用vfork时，应当包括该头文件。

vfork用于创建一个新进程，而该新进程的目的是 exec一个新程序(如上节(2) 中一样)。程序1-5中的shell基本部分就是这种类型程序的一个例子。 vfork与fork一样都创建一个子进程，但是它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec(或exit)，于

是也就不会存访该地址空间。不过在子进程调用 exec或exit之前，它在父进程的空间中运行。这种工作方式在某些UNIX的页式虚存实现中提高了效率（与上节中提及的，在 fork之后跟随exec，并采用在写时复制技术相类似）。

vfork和fork之间的另一个区别是：vfork保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。（如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。）

**实例**

在程序8-1中使用vfork代替fork，并做其他相应修改得到程序8-2。

**程序8-2 vfork 函数实例**

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

运行该程序得到：

```
$ a.out
before vfork
pid = 607, glob = 7, var = 89
```

子进程对变量glob和var做增1操作，结果改变了父进程中的变量值。因为子进程在父进程的地址空间中运行，所以这并不令人惊讶。但是其作用的确与fork不同。

注意，在程序8-2中，调用了_exit而不是exit。正如8.5节所述，_exit并不执行标准I/O缓存的刷新操作。如果用exit而不是_exit，则该程序的输出是：

```
$ a.out
before vfork
```

从中可见，父进程printf的输出消失了。其原因是子进程调用了exit，它刷新开关闭了所有标准I/O流，这包括标准输出。虽然这是由子进程执行的，但却是在父进程的地址空间中进行的，所以所有受到影响的标准I/O FILE对象都是在父进程中的。当父进程调用printf时，标准输出已

被关闭了，于是printf返回-1。

Leffler 等 [1989] 在5.7节中包含了fork和vfork实现方面的更多信息。习题8.1和8.2则继续了对vfork的讨论。

## 8.5 exit函数

如同7.3节所述，进程有三种正常终止法及两种异常终止法。

(1) 正常终止：

(a) 在main函数内执行return语句。如在7.3节中所述，这等效于调用exit。

(b) 调用exit函数。此函数由ANSI C定义，其操作包括调用各终止处理程序（终止处理程序在调用atexit函数时登录），然后关闭所有标准I/O流等。因为ANSI C并不处理文件描述符、多进程（父、子进程）以及作业控制，所以这一定义对UNIX系统而言是不完整的。

(c) 调用_exit系统调用函数。此函数由 exit调用，它处理UNIX特定的细节。 _exit是由POSIX.1说明的。

(2) 异常终止：

(a) 调用abort。它产生SIGABRT信号，所以是下一种异常终止的一种特例。

(b) 当进程接收到某个信号时。（第10章将较详细地说明信号。）进程本身（例如调用abort函数）、其他进程和内核都能产生传送到某一进程的信号。例如，进程越出其地址空间访问存储单元，或者除以0，内核就会为该进程产生相应的信号。

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

对上述任意一种终止情形，我们都希望终止进程能够通知其父进程它是如何终止的。对于exit和_exit，这是依靠传递给它们的退出状态（ exit status）参数来实现的。在异常终止情况，内核（不是进程本身）产生一个指示其异常终止原因的终止状态（ termination status）。在任意一种情况下，该终止进程的父进程都能用 wait或waitpid函数(在下一节说明)取得其终止状态。

注意，这里使用了“ 退出状态 ”（它是传向exit或_exit的参数，或main的返回值）和“ 终止状态 ”两个术语，以表示有所区别。在最后调用 _exit时内核将其退出状态转换成终止状态（回忆图7-1）。下一节中的表 8-1说明了父进程检查子进程的终止状态的不同方法。如果子进程正常终止，则父进程可以获得子进程的退出状态。

在说明fork函数时，一定是一个父进程生成一个子进程。上面又说明了子进程将其终止状态返回给父进程。但是如果父进程在子进程之前终止，则将如何呢 ?其回答是对于其父进程已经终止的所有进程，它们的父进程都改变为 init进程。我们称这些进程由 init进程领养。其操作过程大致是：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止的进程的子进程，如果是，则该进程的父进程 ID就更改为1(init进程的ID)。这种处理方法保证了每个进程有一个父进程。

另一个我们关心的情况是如果子进程在父进程之前终止，那么父进程又如何能在做相应检查时得到子进程的终止状态呢？对此问题的回答是内核为每个终止子进程保存了一定量的信息，所以当终止进程的父进程调用 wait或waitpid 时，可以得到有关信息。这种信息至少包括进程ID、该进程的终止状态、以及该进程使用的 CPU时间总量。内核可以释放终止进程所使用的所有存储器，关闭其所有打开文件。在 UNIX术语中，一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的有关信息、释放它仍占用的资源）的进程被称为僵死

进程（zombie）。ps(1)命令将僵死进程的状态打印为 Z。如果编写一个长期运行的程序，它fork了很多子进程，那么除非父进程等待取得子进程的终止状态，否则这些子进程就会变成僵死进程。

> 系统V提供了一种避免僵死进程的非标准化方法，这将在10.7中介绍。

最后一个要考虑的问题是：一个由init进程领养的进程终止时会发生什么?它会不会变成一个僵死进程？对此问题的回答是“否”，因为init被编写成只要有一个子进程终止，init就会调用一个wait函数取得其终止状态。这样也就防止了在系统中有很多僵死进程。当提及“一个init的子进程”时，这指的是init直接产生的进程(例如，将在9.2节说明的getty进程)，或者是其父进程已终止，由init 领养的进程。

## 8.6 wait和waitpid函数

当一个进程正常或异常终止时，内核就向其父进程发送 SIGCHLD信号。因为子进程终止是个异步事件(这可以在父进程运行的任何时候发生)，所以这种信号也是内核向父进程发的异步通知。父进程可以忽略该信号，或者提供一个该信号发生时即被调用执行的函数 (信号处理程序)。对于这种信号的系统默认动作是忽略它。第 10章将说明这些选择项。现在需要知道的是调用wait或waitpid的进程可能会：

- 阻塞(如果其所有子进程都还在运行)。
- 带子进程的终止状态立即返回(如果一个子进程已终止，正等待父进程存取其终止状态)。
- 出错立即返回(如果它没有任何子进程)。

如果进程由于接收到SIGCHLD信号而调用wait，则可期望wait会立即返回。但是如果在一个任一时刻调用wait，则进程可能会阻塞。

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

两个函数返回：若成功则为进程ID，若出错则为-1

这两个函数的区别是：

- 在一个子进程终止前，wait 使其调用者阻塞，而waitpid 有一选择项，可使调用者不阻塞。
- waitpid并不等待第一个终止的子进程——它有若干个选择项，可以控制它所等待的进程。

如果一个子进程已经终止，是一个僵死进程，则 wait立即返回并取得该子进程的状态，否则wait使其调用者阻塞直到一个子进程终止。如调用者阻塞而且它有多个子进程，则在其一个子进程终止时，wait就立即返回。因为wait返回终止子进程的进程ID，所以它总能了解是哪一个子进程终止了。

这两个函数的参数*statloc*是一个整型指针。如果*statloc*不是一个空指针，则终止进程的终止状态就存放在它所指向的单元内。如果不关心终止状态，则可将该参数指定为空指针。

依据传统，这两个函数返回的整型状态字是由实现定义的。其中某些位表示退出状态（正

常返回），其他位则指示信号编号（异常返回），有一位指示是否产生了一个 core文件等等。POSIX.1规定终止状态用定义在<sys/wait.h>中的各个宏来查看。有三个互斥的宏可用来取得进程终止的原因，它们的名字都以WIF开始。基于这三个宏中哪一个值是真，就可选用其他宏来取得终止状态、信号编号等。这些都在表 8-1中给出。在8.9节中讨论作业控制时，将说明如何停止一个进程。

**表8-1 检查wait和waitpid所返回的终止状态的宏**

| 宏 | 说 明 |
|---|---|
| WIFEXITED(*status*) | 若为正常终止子进程返回的状态，则为真。对于这种情况可执行<br>WEXITSTATUS(*status*)<br>取子进程传送给exit或_exit参数的低8位 |
| WIFSIGNALED(*status*) | 若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行<br>WTERMSIG(*status*)<br>取使子进程终止的信号编号。<br>另外，SVR4和4.3+BSD（但是，非POSIX.1）定义宏：<br>WCOREDUMP(*status*)<br>若已产生终止进程的core文件，则它返回真 |
| WIFSTOPPED(*status*) | 若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行<br>WSTOPSIG(*status*)<br>取使子进程暂停的信号编号 |

**实例**

程序8-3中的函数pr_exit使用表8-1中的宏以打印进程的终止状态。本章的很多程序都将调用此函数。注意，如果定义了WCOREDUMP，则此函数也处理该宏。

程序8-4调用pr_exit函数，例示终止状态的不同值。运行程序8-4可得：

```
$ a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

**程序8-3 打印exit状态的说明**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
```

```
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

**程序8-4　例示不同的exit值**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;
    int     status;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        abort();                /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        status /= 0;            /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

不幸的是，没有一种可移植的方法将 WTERMSIG得到的信号编号映射为说明性的名字。(10.21节中说明了一种方法。 )我们必须查看 <signal.h>头文件才能知道 SIGABRT的值是6，SIGFPE的值是8。

正如前面所述，如果一个进程有几个子进程，那么只要有一个子进程终止， wait就返回。如果要等待一个指定的进程终止 (如果知道要等待进程的ID)，那么该如何做呢?在早期的UNIX版本中，必须调用wait，然后将其返回的进程ID和所期望的进程ID相比较。如果终止进程不是所期望的，则将该进程ID和终止状态保存起来，然后再次调用 wait。反复这样做直到所期望的进程终止。下一次又想等待一个特定进程时，先查看已终止的进程表，若其中已有要等待的进程，则取有关信息，否则调用 wait。其实，我们需要的是等待一个特定进程的函数。 POSIX.1

定义了waitpid函数以提供这种功能(以及其他一些功能)。

> waitpid函数是新由POSIX.1定义的。SVR4和4.3+BSD都提供此函数，但早期的系统V和4.3BSD并不提供此函数。

对于waitpid的*pid*参数的解释与其值有关：

- *pid* == -1 等待任一子进程。于是在这一功能方面waitpid与wait等效。
- *pid* > 0 等待其进程ID与*pid*相等的子进程。
- *pid* == 0 等待其组ID等于调用进程的组ID的任一子进程。
- *pid* < -1 等待其组ID等于*pid*的绝对值的任一子进程。

(9.4节将说明进程组。)waitpid返回终止子进程的进程ID，而该子进程的终止状态则通过*statloc*返回。对于wait，其唯一的出错是调用进程没有子进程(函数调用被一个信号中断时，也可能返回另一种出错。第10章将对此进行讨论)。但是对于waitpid，如果指定的进程或进程组不存在，或者调用进程没有子进程都能出错。

*options*参数使我们能进一步控制waitpid的操作。此参数或者是0，或者是表8-2中常数的逐位或运算。

**表8-2 waitpid的选择项常数**

| 常 数 | 说 明 |
|---|---|
| WNOHANG | 若由*pid*指定的子进程并不立即可用，则waitpid不阻塞，此时其返回值为0 |
| WUNTRACED | 若某实现支持作业控制，则由*pid*指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态。WIFSTOPPED宏确定返回值是否对应于一个暂停子进程 |

> SVR4支持两个附加的非标准的*options*常数。WNOWAIT使系统将其终止状态已由waitpid返回的进程保持在等待状态，于是该进程就可被再次等待。对于WCONTINUED，返回由*pid*指定的某一子进程的状态，该子进程已被继续，其状态尚未报告过。

waitpid函数提供了wait函数没有提供的三个功能：

(1) waitpid等待一个特定的进程(而wait则返回任一终止子进程的状态)。在讨论popen函数时会再说明这一功能。

(2) waitpid提供了一个wait的非阻塞版本。有时希望取得一个子进程的状态，但不想阻塞。

(3) waitpid支持作业控制（以WUNTRACED选择项)。

**实例**

回忆一下8.5节中有关僵死进程的讨论。如果一个进程要fork一个子进程，但不要求它等待子进程终止，也不希望子进程处于僵死状态直到父进程终止，实现这一要求的诀窍是调用fork两次。程序8-5实现了这一点。

在第二个子进程中调用sleep以保证在打印父进程ID时第一个子进程已终止。在fork之后，父、子进程都可继续执行——我们无法预知哪一个会先执行。如果不使第二个子进程睡眠，则在fork之后，它可能比其父进程先执行，于是它打印的父进程ID将是创建它的父进程，而不是

init进程（进程ID 1）。

**程序8-5 fork两次以避免僵死进程**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon
           as our real parent calls exit() in the statement above.
           Here's where we'd continue executing, knowing that when
           we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
       knowing that we're not the parent of the second child. */

    exit(0);
}
```

执行程序8-5得到：

```
$ a.out
$ second child, parent pid = 1
```

注意，当原先的进程(也就是exec本程序的进程)终止时，shell打印其指示符，这在第二个子进程打印其父进程ID之前。

## 8.7 wait3和wait4函数

4.3+BSD提供了两个附加函数wait3和wait4。这两个函数提供的功能比POSIX.1函数wait和waitpid所提供的分别要多一个，这与附加参数*rusage*有关。该参数要求内核返回由终止进程及其所有子进程使用的资源摘要。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);
```

```
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

两个函数返回：若成功则为进程ID，若出错则为-1

SVR4在其BSD兼容库中也提供了wait3函数。

资源信息包括用户CPU时间总量、系统CPU时间总量、缺页次数、接收到信号的次数等。有关细节请参阅getrusage(2)手册页。这些资源信息只包括终止子进程，并不包括处于停止状态的子进程（这种资源信息与7.11节中所述的资源限制不同）。表8-3中列出了各个wait函数所支持的不同的参数。

**表8-3 不同系统上各个wait函数所支持的参数**

| 函 数 | *pid* | *options* | *rusage* | POSIX.1 | SVR4 | 4.3+BSD |
|---|---|---|---|---|---|---|
| wait | | | | • | • | • |
| waitpid | • | • | | • | • | • |
| wait3 | | • | • | | • | • |
| wait4 | • | • | • | | | • |

## 8.8 竞态条件

从本书的目的出发，当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞态条件（race condition）。如果在fork之后的某种逻辑显式或隐式地依赖于在fork之后是父进程先运行还是子进程先运行，那么fork函数就会是竞态条件活跃的孳生地。通常，我们不能预料哪一个进程先运行。即使知道哪一个进程先运行，那么在该进程开始运行后，所发生的事情也依赖于系统负载以及内核的调度算法。

在程序8-5中，当第二个子进程打印其父进程ID时，我们看到了一个潜在的竞态条件。如果第二个子进程在第一个子进程之前运行，则其父进程将会是第一个子进程。但是，如果第一个子进程先运行，并有足够的时间到达并执行exit，则第二个子进程的父进程就是init。即使在程序中调用sleep，这也不保证什么。如果系统负担很重，那么在第二个子进程从sleep返回时，可能第一个子进程还没有得到机会运行。这种形式的问题很难排除，因为在大部分时间，这种问题并不出现。

如果一个进程希望等待一个子进程终止，则它必须调用wait函数。如果一个进程要等待其父进程终止(如程序8-5中一样)，则可使用下列形式的循环：

```
while(getppid() !=1)
    sleep(1);
```

这种形式的循环（称为定期询问（polling））的问题是它浪费了CPU时间，因为调用者每隔1秒都被唤醒，然后进行条件测试。

为了避免竞态条件和定期询问，在多个进程之间需要有某种形式的信号机制。在UNIX中可以使用信号机制，在10.16节将说明它的一种用法。各种形式的进程间通信(IPC)也可使用，在第14、15章将对此进行讨论。

在父、子进程的关系中，常常出现下述情况。在fork之后，父、子进程都有一些事情要做。

例如，父进程可能以子进程ID更新日志文件中的一个记录，而子进程则可能要为父进程创建一个文件。在本例中，要求每个进程在执行完它的一套初始化操作后要通知对方，并且在继续运行之前，要等待另一方完成其初始化操作。这种情况可以描述如下：

```
#include  "ourhdr.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */

    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */
    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */

WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */
exit(0);
```

假定在头文件ourhdr.h中定义了各个需要使用的变量。五个例程TELL_WAIT、TELL_ PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD可以是宏，也可以是函数。

在后面的一些章中会说明实现这些 TELL和WAIT例程的不同方法：10.16节中说明用信号的一种实现，程序14-3中说明用流管道的一种实现。下面先看一个使用这五个例程的实例。

**实例**

程序8-6输出两个字符串：一个由子进程输出，一个由父进程输出。因为输出依赖于内核使进程运行的顺序及每个进程运行的时间长度，所以该程序包含了一个竞态条件。

**程序8-6　具有竞态条件的程序**

```
#include    <sys/types.h>
#include    "ourhdr.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
```

```
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

在程序中将标准输出设置为不带缓存的，于是每个字符输出都需调用一次 write。本例的目的是使内核能尽可能多次地在两个进程之间进行切换，以例示竞态条件。 (如果不这样做，可能也就决不会见到下面所示的输出。没有看到具有错误的输出并不意味着竞态条件不存在，这只是意味着在此特定的系统上未能见到它。 )下面的实际输出说明该程序的运行结果是会改变的。

```
$ a.out
output from child
output from parent
$ a.out
oouuttppuutt ffrroomm cphairledn
t
$ a.out
oouuttppuutt ffrroomm pcahrielndt

$ a.out
ooutput from parent
utput from child
```

修改程序8-6，使其使用TELL和WAIT函数，于是形成了程序8-7。行首标以+号的行是新增加的行。

**程序8-7 修改程序8-6以避免竞态条件**

```
  #include    <sys/types.h>
  #include    "ourhdr.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ( (pid = fork()) < 0)
          err_sys("fork error");
      else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
```

```
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

运行此程序则能得到所预期的输出——两个进程的输出不再交叉混合。

程序8-7是使父进程先运行。如果将fork之后的行改变成：

```
else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();       /* child goes first */
    charatatime("output from parent\n");
}
```

则子进程先运行。习题8.3继续这一实例。

## 8.9 exec函数

8.3节曾提及用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其 main函数开始执行。因为调用exec并不创建新进程，所以前后的进程 ID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

有六种不同的exec函数可供使用，它们常常被统称为exec函数。这些exec函数都是UNIX进程控制原语。用fork可以创建新进程，用exec可以执行新的程序。exit函数和两个wait函数处理终止和等待终止。这些是我们需要的基本的进程控制原语。在后面各节中将使用这些原语构造另外一些如popen和system之类的函数。

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);

int execv(const char *pathname, char *const argv[] );

int execle(const char *pathname, const char *arg0, ...
          /* (char *)0, char *const envp [] */);

int execve(const char *pathname, char *const argv[], char *const envp[] );

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);

int execvp(const char *filename, char *const argv[] );
```

六个函数返回：若出错则为-1，若成功则不返回

这些函数之间的第一个区别是前四个取路径名作为参数，后两个则取文件名作为参数。当指定*filename*作为参数时：

• 如果*filename*中包含/，则就将其视为路径名。

• 否则就按PATH环境变量，在有关目录中搜寻可执行文件。

PATH变量包含了一张目录表(称为路径前缀)，目录之间用冒号(:)分隔。例如下列*name=value*环境字符串：

```
PATH=/bin:/usr/bin:/usr/local/bin:.
```

指定在四个目录中进行搜索。(零长前缀也表示当前目录。在*value*的开始处可用：表示，在行中间则要用：:表示，在行尾以：表示。)

> 有很多出于安全性方面的考虑，要求在搜索路径中决不要包括当前目录。请参见Garfinkel 和Spafford [1991]。

如果execlp和execvp中的任意一个使用路径前缀中的一个找到了一个可执行文件，但是该文件不是由连接编辑程序产生的机器可执行代码文件，则就认为该文件是一个 shell脚本，于是试着调用/bin/sh，并以该*filename*作为shell的输入。

第二个区别与参数表的传递有关(l表示表(list)，v表示矢量(vector))。函数execl、execlp和execle要求将新程序的每个命令行参数都说明为一个单独的参数。这种参数表以空指针结尾。对于另外三个函数(execv,execvp和execve)，则应先构造一个指向各参数的指针数组，然后将该数组地址作为这三个函数的参数。

在使用ANSI C原型之前，对execl,execle和execlp三个函数表示命令行参数的一般方法是：

```
char *arg0, char *arg1, ..., char *argn, (char *) 0
```

应当特别指出的是：在最后一个命令行参数之后跟了一个空指针。如果用常数 0来表示一个空指针，则必须将它强制转换为一个字符指针，否则它将被解释为整型参数。如果一个整型数的长度与char *的长度不同，exec函数实际参数就将出错。

最后一个区别与向新程序传递环境表相关。以 e结尾的两个函数（ execle和execve）可以传递一个指向环境字符串指针数组的指针。其他四个函数则使用调用进程中的environ变量为新程序复制现存的环境。(回忆7.9节及表7-2中对环境字符串的讨论。其中曾提及如果系统支持 setenv和putenv这样的函数，则可更改当前环境和后面生成的子进程的环境，但不能影响父进程的环境。)通常，一个进程允许将其环境传播给其子进程，但有时也有这种情况，进程想要为子进程指定一个确定的环境。例如，在初始化一个新登录的shell时，login程序创建一个只定义少数几个变量的特殊环境，而在我们登录时，可以通过 shell起动文件，将其他变量加到环境中。在使用 ANSI C原型之前， execle 的参数是：

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

从中可见，最后一个参数是指向环境字符串的各字符指针构成的数组的指针。而在 ANSI C原型中，所有命令行参数，包括空指针， *envp*指针都用省略号(…)表示。

这六个exec函数的参数很难记忆。函数名中的字符会给我们一些帮助。字母 p表示该函数取*filename*作为参数，并且用PATH环境变量寻找可执行文件。字母l表示该函数取一个参数表，它与字母v互斥。v表示该函数取一个*argv*[ ]。最后，字母e表示该函数取*envp*[ ] 数组，而不使用当前环境。表8-4显示了这六个函数之间的区别。

表8-4 六个exec函数之间的区别

| 函 数 | *pathname* | *filename* | 参 数 表 | *argv*[] | environ | *envp*[] |
|---|---|---|---|---|---|---|
| execl | • | | • | | • | |
| execlp | | • | • | | • | |
| execle | • | | • | | | • |
| execv | • | | | • | • | |
| execvp | | • | | • | • | |
| execve | • | | | • | | • |
| (字母表示) | | p | l | v | | e |

每个系统对参数表和环境表的总长度都有一个限制。在表 2-7中，这种限制是ARG_MAX。在POSIX.1系统中，此值至少是4096字节。当使用shell的文件名扩充功能产生一个文件名表时，可能会受到此值的限制。例如，命令

```
grep _POSIX_SOURCE /usr/include/*/*.h
```

在某些系统上可能产生下列形式的shell错误：

```
arg list too long
```

> 由于历史原因，系统V中此限制是5120字节。4.3BSD和4.3+BSD在分发时此限制是20 480字节。作者所用的系统则允许多至1M字节（见程序2-1的输出）！

前面曾提及在执行exec后，进程ID没有改变。除此之外，执行新程序的进程还保持了原进程的下列特征：

- 进程ID和父进程ID。
- 实际用户ID和实际组ID。
- 添加组ID。
- 进程组ID。
- 对话期ID。
- 控制终端。
- 闹钟尚余留的时间。
- 当前工作目录。
- 根目录。
- 文件方式创建屏蔽字。
- 文件锁。
- 进程信号屏蔽。
- 未决信号。
- 资源限制。
- tms_utime, tms_stime, tms_cutime以及tms_ustime值。

对打开文件的处理与每个描述符的 exec关闭标志值有关。见图 3-1以及3.13节中对FD_CLOEXEC的说明，进程中每个打开描述符都有一个 exec关闭标志。若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用 fcntl设置了该标志，否则系统的默认操作是在exec后仍保持这种描述符打开。

POSIX.1明确要求在exec时关闭打开目录流（见4.21节中所述的opendir函数）。这通常是由opendir函数实现的，它调用fcntl函数为对应于打开目录流的描述符设置exec关闭标志。

注意，在exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序的文件的设置-用户-ID位和设置-组-ID位是否设置。如果新程序的设置-用户-ID位已设置，则有效用户ID变成程序文件所有者的ID，否则有效用户ID不变。对组ID的处理方式与此相同。

在很多UNIX实现中，这六个函数中只有一个execve是内核的系统调用。另外五个只是库函数，它们最终都要调用系统调用。这六个函数之间的关系示于图8-2中。在这种安排中，库函数execlp 和execvp 使用PATH环境变量查找第一个包含名为*filename*的可执行文件的路径名前缀。

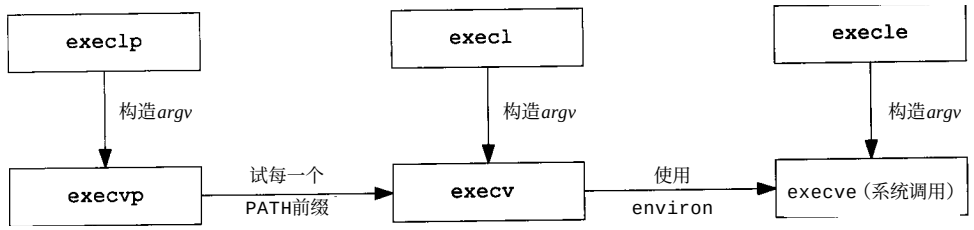


图8-2 六个exec函数之间的关系

**实例**

程序8-8例示了exec函数。

**程序8-8 exec函数实例**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify pathname, specify environment */
        if (execle("/home/stevens/bin/echoall",
                   "echoall", "myarg1", "MY ARG2", (char *) 0,
                   env_init) < 0)
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify filename, inherit environment */
        if (execlp("echoall",
                   "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execlp error");
    }
    exit(0);
}
```

在该程序中先调用execle，它要求一个路径名和一个特定的环境。下一个调用的是 execlp，它用一个文件名，并将调用者的环境传送给新程序。 execlp在这里能够工作的原因是因为目录/home/stevens/bin是当前路径前缀之一。注意，我们将第一个参数（新程序中的 argv [0]）设置为路径名的文件名分量。某些shell将此参数设置为完全的路径名。

在程序8-8中要执行两次的程序echoall示于程序8-9中。这是一个普通程序，它回送其所有命令行参数及其全部环境表。

**程序8-9　回送所有命令行参数和所有环境字符串**

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

执行程序8-8时得到：

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=stevens
HOME=/home/stevens
LOGNAME=stevens
                        其中31行没有显示
EDITOR=/usr/ucb/vi
```

注意，shell提示出现在第二个exec打印argv[0]和argv[1]之间。这是因为父进程并不等待该子进程结束。

## 8.10　更改用户ID和组ID

可以用setuid函数设置实际用户ID和有效用户ID。与此类似，可以用setgid函数设置实际组ID和有效组ID。

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
```

```
int setgid(gid_t gid);
```

两个函数返回：若成功则为0，若出错则为-1

关于谁能更改ID有若干规则。现在先考虑有关改变用户ID的规则（在这里关于用户ID所说明的一切都适用于组ID）。

(1) 若进程具有超级用户特权，则setuid函数将实际用户ID、有效用户ID，以及保存的设置-用户-ID设置为uid。

(2) 若进程没有超级用户特权，但是*uid*等于实际用户ID或保存的设置-用户-ID，则setuid只将有效用户ID设置为*uid*。不改变实际用户ID和保存的设置-用户-ID。

(3) 如果上面两个条件都不满足，则errno设置为EPERM，并返回出错。

在这里假定_POSIX_SAVED_IDS为真。如果没有提供这种功能，则上面所说的关于保存的设置-用户-ID部分都无效。

> FIPS 151-1要求此功能。
> SVR4支持_POSIX_SAVED_IDS功能。

关于内核所维护的三个用户ID，还要注意下列几点：

(1) 只有超级用户进程可以更改实际用户ID。通常，实际用户ID是在用户登录时，由login(1)程序设置的，而且决不会改变它。因为login是一个超级用户进程，当它调用setuid时，设置所有三个用户ID。

(2) 仅当对程序文件设置了设置-用户-ID位时，exec函数设置有效用户ID。如果设置-用户-ID位没有设置，则exec函数不会改变有效用户ID，而将其维持为原先值。任何时候都可以调用setuid，将有效用户ID设置为实际用户ID或保存的设置-用户-ID。自然，不能将有效用户ID设置为任一随机值。

(3) 保存的设置-用户-ID是由exec从有效用户ID复制的。在exec按文件用户ID设置了有效用户ID后，即进行这种复制，并将此副本保存起来。

表8-5列出了改变这三个用户ID的不同方法。

**表8-5 改变三个用户ID的不同方法**

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

注意，用8.2节中所述的getuid和geteuid函数只能获得实际用户ID和有效用户ID的当前值。我们不能获得所保存的设置-用户-ID的当前值。

**实例**

为了说明保存的设置-用户-ID特征的用法，先观察一个使用该特征的程序。我们所观察的是伯克利tip(1)程序(系统V的cu(1)程序与此类似)。这两个程序都连接到一个远程系统，或者是直接连接，或者是拨号一个调制解调器。当tip使用调制解调器时，它必须通过使用锁文件来独

占使用它。此锁文件与UUCP程序共享，因为这两个程序可能要同时使用同一调制解调器。对其工作步骤说明如下：

(1) tip程序文件是由用户uucp拥有的，并且其设置-用户-ID位已设置。当exec此程序时，则关于用户ID得到下列结果：

实际用户ID＝我们的用户ID

有效用户ID＝uucp

保存设置-用户-ID＝uucp

(2) tip存取所要求的锁文件。这些锁文件是由名为uucp的用户所拥有的，因为有效用户ID是uucp，所以tip可以存取这些锁文件。

(3) tip执行setuid(getuid())。因为tip不是超级用户进程，所以这仅仅改变有效用户ID。此时得到：

实际用户ID＝我们的用户ID(未改变)

有效用户-ID＝我们的用户ID(未改变)

保存设置-用户-ID＝uucp(未改变)

现在，tip进程是以我们的用户ID作为其有效用户ID而运行的。这就意味着能存取的只有我们通常可以存取的，没有额外的许可权。

(4) 当执行完所需的操作后，tip执行setuid（*uucpuid*），其中*uucpuid*是用户uucp的数值用户ID（tip很可能在起动时调用geteuid，得到uucp的用户ID，然后将其保存起来，我们并不认为tip会搜索口令文件以得到这一数值用户ID）。因为setuid的参数等于保存的设置-用户-ID，所以这种调用是许可的（这就是为什么需要保存的设置-用户-ID的原因）。现在得到：

实际用户ID=我们的用户ID（未改变）

有效用户ID=uucp

保存设置-用户-ID=uucp（未改变）

(5) tip现在可对其锁文件进行操作以释放它们，因为tip的有效用户ID是uucp。

以这种方法使用保存的设置-用户-ID，在进程的开始和结束部分就可以使用由于程序文件的设置用户ID而得到的额外优先权。但是，进程在其运行的大部分时间只具有普通的许可权。如果进程不能在其结束部分切换回保存的设置-用户-ID，那么就不得不在全部运行时间都保持额外的许可权(这可能会造成麻烦)。

下面来看一看如果在tip运行时为我们生成一个shell进程(先fork，然后exec)将发生什么。因为实际用户ID和有效用户ID都是我们的普通用户ID(上面的第(3)步)，所以该shell没有额外的许可权。它不能存取tip运行时设置成uucp的保存的设置-用户-ID，因为该shell所保存的设置-用户-ID是由exec复制有效用户ID而得到的。所以在执行exec的子进程中，所有三个用户ID都是我们的普通用户ID。

如果程序是设置-用户-ID为root,那么我们关于tip如何使用setuid所做的说明是不正确的。因为以超级用户特权调用setuid就会设置所有三个用户ID。使上述实例按我们所说明的进行工作，只需setuid设置有效用户ID。

### 8.10.1 setreuid和setregid函数

4.3+BSD支持setregid函数，其功能是交换实际用户ID和有效用户ID的值。

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

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

两个函数返回：若成功则为0，若出错则为-1

其作用很简单：一个非特权用户总能交换实际用户ID和有效用户ID。这就允许一个设置-用户-ID程序转换成只具有用户的普通许可权，以后又可再次转换回设置-用户-ID所得到的额外许可权。POSIX.1引进了保存的设置-用户-ID特征后，其作用也相应加强，它也允许一个非特权用户将其有效用户ID设置为保存的设置-用户-ID。

> SVR4在其BSD兼容库中也提供这两个函数。
>
> 4.3BSD并没有上面所说的保存的设置-用户-ID功能。它用setreuid和setregid来代替。这就允许一个非特权用户前、后交换这两个用户ID的值，而4.3BSD中的tip程序就是用这种功能编写的。但是要知道，当此版本生成shell进程时，它必须在exec之前，先将实际用户ID设置为普通用户ID。如果不这样做的话，那么实际用户ID就可能是uucp（由setreuid的交换操作造成。），然后shell进程可能会调用setreuid交换两个用户ID值并取得uucp许可权。作为一个保护性的程序设计措施，tip将子进程的实际用户ID和有效用户ID都设置成普通用户ID。

### 8.10.2 seteuid和setegid函数

在对POIX.1的建议更改中包含了两个函数seteuid和setegid。它们只更改有效用户ID和有效组ID。

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

两个函数返回：若成功则为0，若出错则为-1

一个非特权用户可将其有效用户ID设置为其实际用户ID或其保存的设置-用户-ID。对于一个特权用户则可将有效用户ID设置为*uid*。(这区别于setuid函数，它更改三个用户ID。)这一建议更改也要求支持保存的设置-用户-ID。

> SVR4和4.3+BSD都支持这两种函数。

图8-3给出了本节所述的修改三个不同用户ID的各个函数。

### 8.10.3 组ID

本章中所说明的一切都以类似方式适用于各个组ID。添加组ID不受setgid函数的影响。

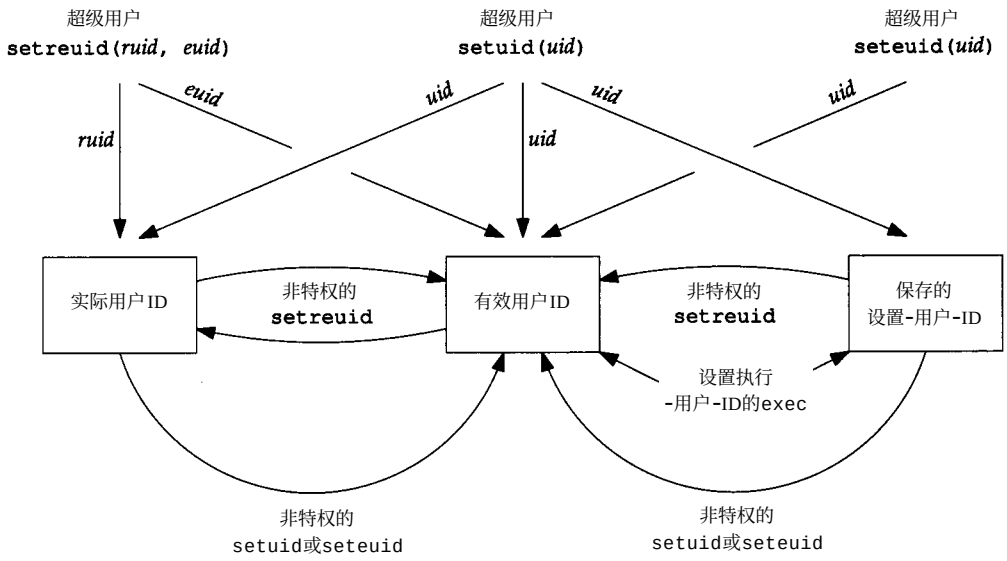


图8-3 设置不同的用户ID的各函数

## 8.11 解释器文件

SVR4和4.3+BSD都支持解释器文件。这种文件是文本文件，其起始行的形式是：

```
#! pathname [optional-argument]
```

在惊叹号和*pathname*之间的空格是可任选的。最常见的是以下列行开始：

```
#!/bin/sh
```

*pathname*通常是个绝对路径名，对它不进行什么特殊的处理(不使用PATH进行路径搜索)。对这种文件的识别是由内核作为exec系统调用处理的一部分来完成的。内核使调用 exec函数的进程实际执行的文件并不是该解释器文件，而是在该解释器文件的第一行中*pathname*所指定的文件。一定要将解释器文件（文本文件，它以 #！开头）和解释器（由该解释器文件第一行中的*pathname*指定）区分开来。

很多系统对解释器文件第一行有长度限制 (32个字符)。这包括#！、*pathname*、可选参数以及空格数。

### 实例

让我们观察一个实例，从中了解当被执行的文件是个解释器文件时，内核对 exec函数的参数及该解释器文件第一行的可选参数做何种处理。程序 8-10调用exec执行一个解释器文件。

**程序8-10 执行一个解释器文件的程序**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;
```

```
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {            /* child */
        if (execl("/home/stevens/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *) 0) < 0)
            err_sys("execl error");
    }

    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

下面先显示要被执行的该解释器文件（只有一行）的内容，接着是运行程序8-10的结果。

```
$ cat /home/stevens/bin/testinterp
#!/home/stevens/bin/echoarg foo
$ a.out
argv[0]: /home/stevens/bin/echoarg
argv[1]: foo
argv[2]: /home/stevens/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

程序echoarg（解释器）回送每一个命令行参数（它就是程序7-2）。注意，当内核exec该解释器(/home/stevens/bin/echoarg)时，argv[0]是该解释器的*pathname*，argv[1]是解释器文件中的可选参数，其余参数是*pathname*（/home/stevens/bin/testinterp），以及程序8-10中调用execl的第二和第三个参数(myarg1和MY ARG2)。调用execl时的argv[1]和argv[2]已右移了两个位置。注意，内核取execl中的*pathname*代替第一个参数(testinterp)，因为一般*pathname*包含了较第一个参数更多的信息。

**实例**

在解释器pathname后可跟随可选参数，它们常用于为支持-f选择项的程序指定该选择项。例如，可以以下列方式执行awk(1)程序：

```
awk -f myfile
```

它告诉awk从文件myfile中读awk程序。

> 在很多系统中，有awk的两个版本。awk常常被称为“老awk”，它是与V7一起分发的原始版本。nawk(新awk)包含了很多增强功能，对应于在Aho、Kernighan和Weinberger〔1988〕中说明的语言。此新版本提供了对命令行参数的存取，这是下面的例子所需的。SVR4提供了两者，老的awk既可用awk也可用oawk调用，但是SVR4已说明在将来的版本中awk将是nawk。POSIX.2将新awk语句就称为awk，这正是本书所使用的。

在解释器文件中使用-f选择项，可以写成：

```
#!/bin/awk -f
(在解释器文件中后随awk程序)
```

例如，程序8-11是一个在/usr/local/bin/awkexample解释器文件中的程序。

**程序8-11 解释器文件中的awk程序**

```
#!/bin/awk -f

BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

如果路径前缀之一是/usr/local/bin，则可以下列方式执行程序8-11(假定我们已打开了该文件的执行位)：

```
$ awkexample filel FILENAME2 f3
ARGV[0] = /bin/awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

执行/bin/awk时，其命令行参数是：

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

解释器文件的路径名（/usr/local/bin/awkexample）被传送给解释器。因为不能期望该解释器（在本例中是/bin/awk）会使用PATH变量定位该解释器文件，所以只传送其路径名中的文件名是不够的。当awk读解释器文件时，因为#是awk的注释字符，所以在awk读解释器文件时，它忽略第一行。

可以用下列命令验证上述命令行参数。

```
$ su                                  成为超级用户
Password:                             输入超级用户口令
# mv /bin/awk /bin/awk.save           保存原先的程序
# cp /home/stevens/bin/echoarg /bin/aw暂时替换它
# suspend                             用作业控制挂起超级用户 shell
[1] + Stopped          su
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                  用作业控制恢复超级用户 shell
su
# mv /bin/awk.save /bin/awk           恢复原先的程序
# exit                                终止超级用户 shell
```

在此例子中，解释器的-f选择项是必需的。正如前述，它告诉awk在什么地方得到awk程序。如果在解释器文件中删除-f选择项，则其结果是：

```
$ awkexample file1 FILENAME2 f3
/bin/awk:syntax error at source line 1
 context is
        >>> /usr/local <<< /bin/awkexample
```

```
/bin/awk: bailing out at source line 1
```

因为在这种情况下命令行参数是：

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

于是awk企图将字符串/usr/local/bin/awkexample解释为一个awk程序。如果不能向解释器传递至少一个可选参数（在本例中是-f），那么这些解释器文件只有对shell才是有用的。

是否一定需要解释器文件呢?那也不完全如此。但是它们确实使用户得到效率方面的好处，其代价是内核的额外开销（因为内核需要识别解释器文件）。由于下述理由，解释器文件是有用的：

(1) 某些程序是用某种语言写的脚本，这一事实可以隐藏起来。例如，为了执行程序 8-11，只需使用下列命令行：

```
awkexample optional-arguments
```

而并不需要知道该程序实际上是一个awk脚本，否则就要以下列方式执行该程序：

```
awk -f awkexample optional-arguments
```

(2) 解释器脚本在效率方面也提供了好处。再考虑一下前面的例子。仍旧隐藏该程序是一个awk脚本的事实，但是将其放在一个shell脚本中：

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

这种解决方法的问题是要求做更多的工作。首先， shell读此命令，然后试图execlp此文件名。因为shell脚本是一个可执行文件，但却不是机器可执行的，于是返回一个错误， execlp就认为该文件是一个shell脚本(它实际上就是这种文件)。然后执行/bin/sh，并以该shell脚本的路径名作为其参数。shell正确地执行我们的shell脚本，但是为了运行awk程序，它调用fork,exec和wait。用一个shell脚本代替解释器脚本需要更多的开销。

(3) 解释器脚本使我们可以使用除/bin/sh以外的其他shell来编写shell脚本。当execlp找到一个非机器可执行的可执行文件时，它总是调用/bin/sh来解释执行该文件。但是，用解释器脚本，则可编写成：

```
#!/bin/csh
(在解释器文件中后随C shell脚本)
```

再一次，我们也可将此放在一个/bin/sh脚本中(然后由其调用C shell)，但是要有更多的开销。

如果三个shell和awk没有用#作为注释符，则上面所说的都无效。

## 8.12 system函数

在程序中执行一个命令字符串很方便。例如，假定要将时间和日期放到一个文件中，则可使用6.9节中的函数实现这一点。调用time得到当前日历时间，接着调用localtime将日历时间变换为年、月、日、时、分、秒、周日形式，然后调用 strftime对上面的结果进行格式化处理，最后将结果写到文件中。但是用下面的system函数则更容易做到这一点。

```
system("date > file");
```

ANSI C定义了system函数，但是其操作对系统的依赖性很强。

> 因为system不属于操作系统界面而是shell界面，所以POSIX.1没有定义它，POSIX.2则正在对其进行标准化。下列说明与POSIX.2标准的草案11.2相一致。

```
#include <stdlib.h>

int system(const char *cmdstring);
```

返回：（见下）

如果*cmdstring*是一个空指针，则仅当命令处理程序可用时， system返回非0值，这一特征可以决定在一个给定的操作系统上是否支持system函数。在UNIX中，system总是可用的。

因为system在其实现中调用了fork、exec和waitpid，因此有三种返回值：

(1) 如果fork失败或者waitpid返回除EINTR之外的出错，则system返回－1，而且errno中设置了错误类型。

(2) 如果exec失败(表示不能执行shell)，则其返回值如同shell执行了exit(127)一样。

(3) 否则所有三个函数(fork,exec和waitpid)都成功，并且system的返回值是shell的终止状态，其格式已在waitpid中说明。

> 如果waitpid由一个捕捉到的信号中断，则system很多当前的实现都返回一个错误(EINTR)，在这种情况下system不返回一个错误的要求已被加到POSIX.2的最近草案中。(10.5节中将讨论被中断的系统调用。)

程序8-12是system函数的一种实现。它对信号没有进行处理。 10.18节中将修改此函数使其进行信号处理。

shell的-c选择项告诉shell程序取下一个命令行参数(在这里是*cmdstring*)作为命令输入(而不是从标准输入或从一个给定的文件中读命令)。shell对以null字符终止的命令字符串进行语法分析，将它们分成分隔开的命令行参数。传递给shell的实际命令串可以包含任一有效的shell命令。例如，可以用<和>对输入和输出重新定向。

如果不使用shell执行此命令，而是试图由我们自己去执行它，那么将相当困难。首先，我们必须用execlp而不是execl，像shell那样使用PATH变量。我们必须将null符结尾的命令字符串分成各个命令行参数，以便调用execlp。最后，我们也不能使用任何一个shell元字符。

注意，我们调用_exit而不是exit。这是为了防止任一标准I/O缓存(这些缓存会在fork中由父进程复制到子进程)在子进程中被刷新。

**程序8-12　system函数**(没有对信号进行处理)

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
```

```
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with Unix */

    if ( (pid = fork()) < 0) {
        status = -1;    /* probably out of processes */

    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127);     /* execl error */

    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    return(status);
}
```

用程序8-13对这种实现的system函数进行测试(pr_exit函数定义在程序8-3中)。运行程序8-13得到：

```
$ a.out
Thu Aug 29 14:24:19 MST 1991
normal termination, exit status = 0          对于date
sh: nosuchcommand: not found
normal termination, exit status = 1          对于无此种命令
stevens  console  Aug 25 11:49
stevens  ttyp0    Aug 29 05:56
stevens  ttyp1    Aug 29 05:56
stevens  ttyp2    Aug 29 05:56
normal termination, exit status = 44         对于exit
```

**程序8-13 调用system函数**

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    int     status;

    if ( (status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

使用system而不是直接使用fork和exec的优点是：system进行了所需的各种出错处理，以及各种信号处理(在10.18节中的下一个版本system函数中)。

在UNIX的早期版本中，包括SVR3.2和4.3BSD，都没有waitpid函数，于是父进程用下列形式的语句等待子进程：

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

如果调用system的进程在调用它之前已经生成一个子进程（并执行一个程序），那么将引起问题。因为上面的while语句一直循环执行，直到由system产生的子进程终止才停止，如果其任意一个不是用pid标识的子进程在此之前终止，则它们的进程ID和终止状态都被while语句丢弃。实际上，由于wait不能等待一个指定的进程，POSIX.1才为此及其他一些原因定义了waitpid函数。如果不提供waitpid，对于popen和pclose函数也会发生同样的问题。

### 设置-用户-ID程序

如果在一个设置-用户-ID程序中调用system，那么发生什么呢?这是一个安全性方面的漏洞，决不应当这样做。程序8-14是一个简单程序，它只是对其命令行参数调用system函数。

**程序8-14　用system执行命令行参数**

```
#include    "ourhdr.h"

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

    if (argc < 2)
        err_quit("command-line argument required");

    if ( (status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

将此程序编译成可执行目标文件tsys。

程序8-15是另一个简单程序，它打印其实际和有效用户ID。

**程序8-15　打印实际和有效用户ID**

```
#include    "ourhdr.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

将此程序编译成可执行目标文件printuids。运行这两个程序，得到下列结果：

```
$ tsys printuids                    正常执行，无特权
real uid = 224, effective uid = 224
```

```
normal termination, exit status = 0
$ su                                    成为超级用户
Password:                               输入超级用户口令
# chown root tsys                       更改所有者
# chmod u+s tsys                        增加设置-用户-ID
# ls -l tsys                            检验文件许可权和所有者
-rwsrwxr-x  1 root    105737 Aug 18 11:21 tsys
# exit                                  终止超级用户 shell
$ tsys printuids
real uid = 224, effective uid = 0       哎呀！这是一个安全性漏洞
normal termination, exit status = 0
```

我们给予tsys程序的超级用户许可权在system中执行了fork和exec之后仍被保持下来，也就是说执行system中shell命令的进程也具有了超级用户许可权。

如果一个进程正以特殊的许可权 (设置-用户-ID或设置-组-ID)运行，它又想生成另一个进程执行另一个程序，则它应当直接使用fork和exec，而且在fork之后、exec之前要改回到普通许可权。设置-用户-ID或设置-组-ID程序决不应调用system函数。

> 这种警告的一个理由是：system调用shell对命令字符串进行语法分析，而shell则使用IFS变量作为其输入字段分隔符。早期的shell版本在被调用时不将此变量恢复为普通字符集。这就允许一个不怀好意的用户在调用 system之前设置IFS，造成system执行一个不同的程序。

## 8.13 进程会计

很多UNIX系统提供了一个选择项以进行进程会计事务处理。当取了这种选择项后，每当进程结束时内核就写一个会计记录。典型的会计记录是 32字节长的二进制数据，包括命令名、所使用的CPU时间总量、用户ID和组ID、起动时间等。本节将比较详细地说明这种会计记录，这样也使我们得到了一个再次观察进程的机会，得到了使用 5.9节中所介绍的fread函数的机会。

> 任一标准都没有对进程会计进行过说明。本节的说明依据 SVR4和4.3+BSD实现。SVR4提供了很多程序处理这种原始的会计数据——例如 runacct和acctcom。4.3+BSD提供sa(8)命令处理并总结原始会计数据。

一个至今没有说明过的函数 (acct)起动和终止进程会计。唯一使用这一函数的是 SVR4和4.3+BSD的accton(8)命令。超级用户执行一个带路径名参数的 accton命令起动会计处理。该路径名通常是/var/adm/pacct（早期系统中为/usr/adm/acct)。执行不带任何参数的accton命令则停止会计处理。

会计记录结构定义在头文件<sys/acct.h>中，其样式如下：

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

struct  acct
{
  char    ac_flag;  /* flag (see Figure 8.9) */
  char    ac_stat;  /* termination status (signal & core flag only) */
                    /* (not provided by BSD systems) */
  uid_t   ac_uid;   /* real user ID */
```

```
  gid_t   ac_gid;   /* real group ID */
  dev_t   ac_tty;   /* controlling terminal */
  time_t  ac_btime; /* starting calendar time */
  comp_t  ac_utime; /* user CPU time (clock ticks) */
  comp_t  ac_stime; /* system CPU time (clock ticks) */
  comp_t  ac_etime; /* elapsed time (clock ticks) */
  comp_t  ac_mem;   /* average memory usage */
  comp_t  ac_io;    /* bytes transferred (by read and write) */
  comp_t  ac_rw;    /* blocks read or written */
  char    ac_comm[8];/* command name: [8] for SVR4, [10] for 4.3+BSD */
};
```

由于历史原因，伯克利系统，包括4.3+BSD都不提供ac_stat变量。

其中，ac_flag记录了进程执行期间的某些事件。这些事件见表8-6。

**表8-6 会计记录中的ac_flag值**

| ac_flag | 说 明 |
|---|---|
| AFORK | 进程是由fork产生的，但从未调用exec |
| ASU | 进程使用超级用户优先权 |
| ACOMPAT | 进程使用兼容方式（仅VAX） |
| ACORE | 进程转储core（不在SVR4） |
| AXSIG | 进程由信号消灭（不在SVR4） |

会计记录所需的各个数据（各CPU时间、传输的字符数等）都由内核保存在进程表中，并在一个新进程被创建时置初值(例如fork之后在子进程中)。进程终止时写一个会计记录。这就意味着在会计文件中记录的顺序对应于进程终止的顺序，而不是它们起动的顺序。为了确定起动顺序，需要读全部会计文件，并按起动日历时间进行排序。这不是一种很完善的方法，因为日历时间的单位是秒（见1.10节），在一个给定的秒中可能起动了多个进程。而墙上时钟时间的单位是时钟滴答（通常，每秒滴答数在50～100之间）。但是我们并不知道进程的终止时间，所知道的只是起动时间和终止顺序。这就意味着，即使墙上时间比起动时间要精确得多，但是仍不能按照会计文件中的数据重构各进程的精确起动顺序。

会计记录对应于进程而不是程序。在fork之后，内核为子进程初始化一个记录，而不是在一个新程序被执行时。虽然exec并不创建一个新的会计记录，但相应记录中的命令名改变了，AFORK标志则被清除。这意味着，如果一个进程顺序执行了三个程序(A exec B,B exec C,最后C exit)，但只写一个会计记录。在该记录中的命令名对应于程序C，但CPU时间是程序A、B、C之和。

**实例**

为了得到某些会计数据以便查看，运行程序8-16，它调用fork四次。每个子进程做不同的事情，然后终止。此程序所做的基本工作示于图8-4中。

程序8-17则从会计记录中选择一些字段并打印出来。

**程序8-16 产生会计数据的程序**

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

int
main(void)
```

```
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {          /* parent */
        sleep(2);
        exit(2);                  /* terminate with exit status 2 */
    }

                                  /* first child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                  /* terminate with core dump */
    }

                                  /* second child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL);
        exit(7);                  /* shouldn't get here */
    }

                                  /* third child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                  /* normal exit */
    }

                                  /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);      /* terminate with signal, no core dump */
    exit(6);                      /* shouldn't get here */
}
```

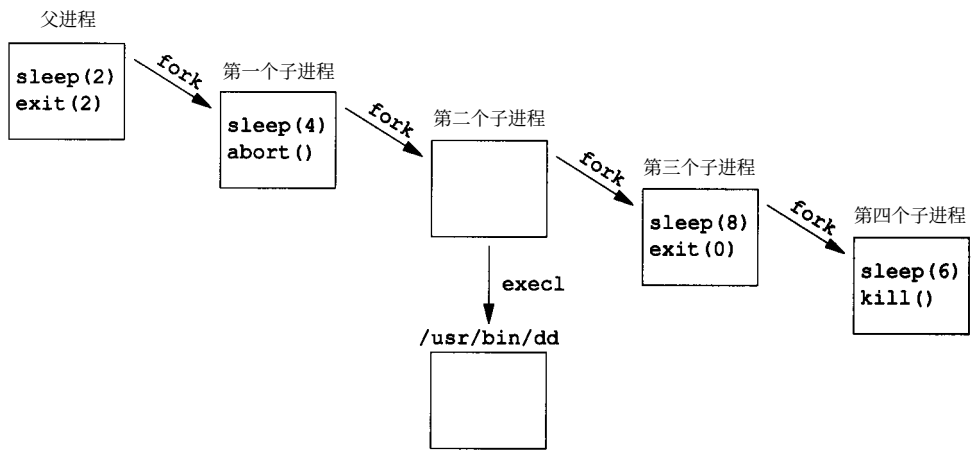


图8-4 会计处理实例的进程结构

然后，执行下列操作步骤：

(1) 成为超级用户，用accton命令起动会计事务处理。注意，当此命令结束时，会计事务处

理已经起动，因此在会计文件中的第一个记录应来自这一命令。

(2) 运行程序8-16。这会加五个记录到会计文件中(父进程一个，四个子进程各一个)。在第二个子进程中，execl并不创建一个新进程，所以对第二个进程只有一个会计记录。

(3) 成为超级用户，停止会计事务处理。因为在accton命令终止时已停止处理会计事务，所以不会在会计文件中增加一个记录。

(4) 运行程序8-17，从打印文件中选出字段并打印。

第(4)步的输出如下面所示。在每一行中都对进程加了说明，以便后面讨论。

```
accton    e =      7, chars =       64, stat =    0:    S
dd        e =     37, chars =   221888, stat =    0:    第二个子进程
a.out     e =    128, chars =        0, stat =    0:    父进程
a.out     e =    274, chars =        0, stat =  134: F  第一个子进程
a.out     e =    360, chars =        0, stat =    9: F  第四个子进程
a.out     e =    484, chars =        0, stat =    0: F  第三个子进程
```

**程序8-17　打印从系统会计文件中选出的字段**

```
#include    <sys/types.h>
#include    <sys/acct.h>
#include    "ourhdr.h"

#define ACCTFILE        "/var/adm/pacct"
static unsigned long    compt2ulong(comp_t);

int
main(void)
{
    struct acct     acdata;
    FILE            *fp;

    if ( (fp = fopen(ACCTFILE, "r")) == NULL)
        err_sys("can't open %s", ACCTFILE);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf("%-*.*s  e = %6ld, chars = %7ld, "
               "stat = %3u: %c %c %c %c\n", sizeof(acdata.ac_comm),
                sizeof(acdata.ac_comm), acdata.ac_comm,
                compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
                (unsigned char) acdata.ac_stat,
#ifdef  ACORE       /* SVR4 doesn't define ACORE */
                acdata.ac_flag & ACORE ? 'D' : ' ',
#else
                                                ' ',
#endif
#ifdef  AXSIG       /* SVR4 doesn't define AXSIG */
                acdata.ac_flag & AXSIG ? 'X' : ' ',
#else
                                                ' ',
#endif
                acdata.ac_flag & AFORK ? 'F' : ' ',
                acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}

static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;
```

```
        val = comptime & 017777;     /* 13-bit fraction */
        exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
        while (exp-- > 0)
            val *= 8;
        return(val);
    }
```

墙上日历时间值的单位是CLK_TCK。从表2-6中可见，本系统的值是60。例如，在父进程中的sleep(2)对应于墙上日历时间128个时钟滴答。对于第一个子进程， sleep(4)变成274时钟滴答。注意，一个进程睡眠的时间总量并不精确。（第10章将返回到sleep函数。）调用fork和exit也要一些时间。

注意，ac_stat并不是进程的真正终止状态。它只是8.6节中讨论的终止状态的一部分。如果进程异常终止，则此字节中的信息只是 core标志位(一般是最高位)以及信号编号数(一般是低7位)。如果进程正常终止，则从会计文件不能得到进程的退出 (exit)状态。对于第一个进程，此值是128+6。128是core标志位，6是此系统上信号 SIGABRT的值（它是由调用 abort产生的）。第四个子进程的值是 9，它对应于SIGKILL的值。从会计文件的数据中不能了解到，父进程在退出时所用的参数值是2，三个子进程退出时所用的参数值是0。

dd进程复制到第二个子进程中的文件 /boot的长度是110 888字节。而I/O字符数是此值的二倍，因为读了110 888字节，然后又写了110 888字节。即使输出到null设备，仍对I/O字符数进行计算。

ac_flag值与我们所预料的相同。除调用了execl的第二个子进程以外，其他子进程都设置了F标志。父进程没有设置F标志，其原因是交互式shell曾调用过fork生成父进程，然后执行a.out文件。调用了abort的第一个子进程的core转储标志(D)打开。因为abort产生信号SIGABRT以产生core转储。该进程的X标志也打开，因为它是由信号终止的。第四个子进程的 X标志也打开，但是SIGKILL信号并不产生core转储，它只是终止该进程。

最后要说明的是：第一个子进程的 I/O字符数为0，但是该进程产生了一个 core文件。其原因是写core文件所需的I/O并不由该进程负担。

## 8.14 用户标识

任一进程都可以得到其实际和有效用户 ID及组ID。但是有时希望找到运行该程序的用户的登录名。我们可以调用 getpwuid(getuid())，但是如果一个用户有多个登录名，这些登录名又对应着同一个用户 ID，那么又将如何呢？（一个人在口令文件中可以有多个登录项，它们的用户ID相同，但登录shell则不同。）系统通常保存用户的登录名（见 6.7节），用getlogin函数可以存取此登录名。

```
#include <unistd.h>

char *getlogin(void);
```

返回：若成功则为指向登录名字符串的指针，若出错则为 NULL

如果调用此函数的进程没有连接到用户登录时所用的终端，则本函数会失败。通常称这些进程为精灵进程（daemon），第13章将对这种进程专门进行讨论。

得到了登录名，就可用getpwnam在口令文件中查找相应记录以确定其登录shell等。

为了找到登录名，UNIX系统在历史上一直是调用ttyname函数（见11.9节），然后在utmp文件（见6.7节）中找匹配项。4.3+BSD将登录名存放在进程表项中，并提供系统调用存取该登录名。

系统V提供cuserid函数返回登录名。此函数先调用getlogin函数，如果失败则再调用getpwuid(getuid())。IEEE Std.1003.1-1988说明了cuserid，但是它以有效用户ID而不是实际用户ID来调用。POSIX.1的1990最后版本删除了cuserid函数。

FIPS 151-1要求登录shell定义一个环境变量LOGNAME,其值为用户的登录名。在4.3+BSD中，此变量由login设置，并由登录shell继承。但是，用户可以改变环境变量，所以不能使用LOGNAME来确认用户，而应当使用getlogin函数。

## 8.15 进程时间

在1.10节中说明了墙上时钟时间、用户CPU时间和系统CPU时间。任一进程都可调用times函数以获得它自己及终止子进程的上述值。

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

clock_t times(struct tms *buf);
```

返回：若成功则为经过的墙上时钟时间（单位：滴答），若出错则为-1

此函数填写由*buf*指向的tms结构，该结构定义如下：

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

注意，此结构没有包含墙上时钟时间。作为代替，times函数返回墙上时钟时间作为函数值。此值是相对于过去的某一时刻度量的，所以不能用其绝对值而必须使用其相对值。例如，调用times，保存其返回值。在以后某个时间再次调用times，从新返回的值中减去以前返回的值，此差值就是墙上时钟时间。（一个长期运行的进程可能其墙上时钟时间会溢出，当然这种可能性极小。）

结构中两个针对子进程的字段包含了此进程已等待到的各子进程的值。

所有由此函数返回的clock_t值都用_SC_CLK_TCK(由sysconf函数返回的每秒时钟滴答数，见2.5.4节)变换成秒数。

伯克利系统，包括4.3BSD继承了V7的times版本，它不返回墙上时钟时间。这一老版本如执行成功则返回0，如失败则返回－1。4.3+BSD支持POSIX.1版本。

4.3+BSD和SVR4(在BSD兼容库中)提供了getrusage(2)函数，此函数返回CPU时间，以及指示资源使用情况的另外14个值。

**实例**

程序8-18将每个命令行参数作为shell命令串执行，对每个命令计时，并打印从tms结构取

得的值。按下列方式运行此程序，得到：

```
$ a.out "sleep 5" "date"

command: sleep 5
  real:      5.25
  user:      0.00
  sys:       0.00
  child user:       0.02
  child sys:        0.13
normal termination, exit status = 0

command: date
Sun Aug 18 09:25:38 MST 1991
  real :     0.27
  user:      0.00
  sys:       0.00
  child user:      0.05
  child sys:       0.10
normal termination, exit status = 0
```

在这个实例中，在 child user和child sys行中显示的时间是执行 shell和命令的子进程所使用的CPU时间。

**程序8-18 时间以及执行命令行参数**

```
#include    <sys/times.h>
#include    "ourhdr.h"

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

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

    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}
static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    fprintf(stderr, "\ncommand: %s\n", cmd);

    if ( (start = times(&tmsstart)) == -1)  /* starting values */
        err_sys("times error");

    if ( (status = system(cmd)) < 0)        /* execute command */
        err_sys("system() error");

    if ( (end = times(&tmsend)) == -1)      /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}
static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
```

```
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");
    fprintf(stderr, "  real:  %7.2f\n", real / (double) clktck);
    fprintf(stderr, "  user:  %7.2f\n",
            (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    fprintf(stderr, "  sys:   %7.2f\n",
            (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    fprintf(stderr, "  child user:  %7.2f\n",
            (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    fprintf(stderr, "  child sys:   %7.2f\n",
            (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

让我们再运行1.10节中的例子：

```
$ a.out "cd /usr/include; grep _POSIX_SOURCE */*.h > /dev/null"

command: cd /usr/include; grep _POSIX_SOURCE */*.h > /dev/null
  real:    18.67
  user:     0.00
  sys:      0.02
  child user:     0.43
  child sys:      4.13
normal termination, exit status = 0
```

如同所期望的那样，所有三个值(实际时间和子进程CPU时间)都与1.10节中的值相近。

## 8.16 小结

对在UNIX环境中的高级程序设计而言，完整地了解UNIX的进程控制非常重要。其中必须熟练掌握的只有几个——fork、exec族、_exit、wait和waitpid。很多应用程序都使用这些原语。fork原语也给了我们一个了解竞态条件的机会。

本章说明了system函数和进程会计，以及这些进程控制函数的应用情况。本章还说明了exec函数的另一种变体：解释器文件及它们的工作方式。对各种不同的用户 ID和组ID(实际,有效和保存的)的理解和编写安全的设置-用户-ID程序是至关重要的。

在了解进程和子进程的基础上，下一章将进一步说明进程和其他进程的关系 —— 对话期和作业控制。第10章将说明信号机制并以此结束对进程的讨论。

## 习题

8.1 在程序8-2中用exit取代_exit将关闭标准输出，修改程序验证printf确实返回－1。

8.2 调用vfork后，子进程运行在父进程的地址空间中。如果不是在main函数中调用vfork,而是在vfork以后子进程从这个函数返回，那将会如何？请编写一段程序验证并且画出堆栈中的映像。

8.3 当用$a.out执行程序8-7一次，其输出是正确的。但是若将该程序按下列方式执行多次,则其输出不正确。

```
$ a.out ; a.out ; a.out
output from parent
ooutput from parent
ouotuptut from child
```

```
put from parent
output from child
utput from child
```

原因是什么？怎样才能更正此种错误？如果使子进程首先输出，还会发生此问题吗？

8.4 在程序8-10中，调用execl，指定解释文件为*pathname*。如果调用execlp，指定testinterp为*filename*，并且目录/home/stevens/bin是路径前缀，则运行该程序时，argv[2]的打印输出是什么？

8.5 一个进程怎样才能获得其保存的设置-用户-ID？

8.6 编写一段程序，用于创建一个僵死进程，然后调用system执行ps(1)命令以验证该进程是僵死进程。

8.7 8.9节中提及POSIX.1要求在exec时关闭打开目录流。按下列方法对此进行验证：对根目录调用opendir，查看DIR结构，然后打印exec关闭标志。接着打开同一目录读并打印exec关闭标志。