# 第7章 进程环境

## 7.1 引言

下一章将介绍进程控制原语，在此之前需先了解进程的环境。本章中将学习：当执行程序时，其main函数是如何被调用的，命令行参数是如何传送给执行程序的，典型的存储器布局是什么样式，如何分配另外的存储空间，进程如何使用环境变量；各种不同的进程终止方式等。本章结束之前，还将研究进程的资源限制。

## 7.2 main函数

C程序总是从main函数开始执行。main函数的原型是

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

其中，argc是命令行参数的数目，是指向参数的各个指针所构成的数组。后面将对命令行参数进行说明。

当内核执行C程序时（使用一个exec函数，下一章将说明exec函数），在调用main前先调用一个特殊的启动例程。可执行程序文件将此启动例程指定为程序的起始地址——这是由连接编辑器设置的，而连接编辑器则由C编译器（通常是gcc）调用。启动例程从内核取得命令行参数和环境变量值，然后为按上述方式调用main函数做好安排。

## 7.3进程终止

有8种方式使进程终止(termination),其中5种为正常终止，它们是

* 从main返回。
* 调用exit。
* 调用\_exit或\_Exit。
* 最后一个线程从其启动例程返回。
* 最后一个线程调用pthread\_exit。

异常终止有3种方式，它们是

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

本章我们暂不考虑专门针对线程的三种终止方式。

上一节提及的启动例程是这样编写的，使得从main返回后立即调用exit函数。如果将启动例程以C代码形式表示（实际上该例程常常用汇编语言编写），则它调用main函数的形式可能是：

exit(main(argc, argv));

### 1. exit函数

有三个函数用干正常终止一个程序：\_exit和\_Exit立即进入内核，exit则先执行一些清理处理（包括调用执行各终止处理程序，关闭所有标准I/O流等），然后进入内核。

|  |
| --- |
| #include <stdlib.h>  void exit (int status);  void \_Exit (int status);  #include <unistd.h>  void \_exit (int status); |

我们将在后续章节中讨论这三个函数对其他进程（例如正在终止进程的父、子进程）的影响。使用不同头文件的原因是：exit和\_Exit是由ISO C说明的，而\_exit则是由POSIX.1标准说明的。

由干历史原因，exit函数总是执行一个标准I/O库的清理关闭操作：为所有打开流调用fclose函数。前面章节讨论过，这会造成所有缓冲的输出数据都被写到文件上。

三个exit函数都带一个整型参数，称之为终止状态（或退出状态，exit status）。大多数LINUX/UNIX shell都提供检査进程终止状态的方法。如果

(a)若调用这些函数时不带终止状态，或

(b)main执行了一个无返回值的return语句，或

(c)main没有声明返回类型为整型，则

则该进程的终止状态是未定义的。但是，若main的返回类型是整型，并且main执行到最后一条语句时返回（隐式返回），那么该进程的终止状态是0。

main函数返回一整型值与用该值调用exit是等价的。于是在main函数中：

exit(0)；

等价于

return(0)；

实例

下面程序是经典的“hello, world”实例。

程序清单7-1经典的C程序

|  |
| --- |
| #include <stdio.h>  main()  {  printf{"hello, world\n");  } |

该程序进行编译，然后运行，则可见到其终止码是随机的。如果在不同的系统上编译该程序，我们很可能得到不同的终止码，这取决于main函数返回时栈和寄存器的内容：

|  |
| --- |
| $ cc hello.c  $ ./a.out  hello, world  $ echo $?  13 |

打印终止状态 现在，如果我们启用1999 ISO C编译器扩展，则可见到终止码改变了：

|  |
| --- |
| $ cc -std=c99 hello.c  hello.c：4： warning： return type defaults to "int"  $ ./a.out hello, world  $ echo $?  0 |

注意，当我们启用1999 ISO C扩展时，编译器发出I告消息。打印该縈告消息的原因是：main函数的类型没有显式地声明为整型。如果我们增加了这一声明，那么此警告消息就不会出现。

在下一章，我们将了解到任一进程如何引起一个程序被执行，如何等待进程完成，然后又如何取其终止状态。

### 2. atexit函数

按照ISO C的规定，一个进程可以登记多达32个函数，这些函数将由exit自动调用。我们称这些函数为终止处理程序（exit handler），并调用atexit函数来登记这些函数。

|  |
| --- |
| #include <stdlib.h>  int atexit (void (\*func) (void));  返回值：若成功则返回0，若出错则返回非0值 |

其中，atexit的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。exit调用这些函数的顺序与它们登记时候的顺序相反。同一函数如若登记多次，则也会被调用多次。

根据ISOC和POSIX.1，exit首先调用各终止处理程序，然后按需多次调用fclose，关闭所有打开流。POSIX.1扩展了ISO C标准，它指定如若程序调用exec函数族中的任一函数，则将清除所有已安装的终止处理程序。下图显示了一个C程序是如何启动的，以及它可以终止的各种方式。

|  |
| --- |
|  |
| 一个程序是如何启动和终止的 |

注意，内核使程序执行的唯一方法是调用一个exec函数。进程自愿终止的唯一方法是显式或隐式地（通过调用exit）调用\_exit或者\_Exit。进程也可非自愿地由一个信号使其终止，上图中没有表示。

下面程序演示了如何使用atexit函数。

|  |
| --- |
|  |

程序运行结果如下：

|  |
| --- |
|  |

终止处理程序每登记一次，就会被调用一次。在上述程序中，第一个终止处理程序被登记两次，所以也会被调用两次。注意，在main中没有调用exit,而是用了return语句。

## 7.4命令行参数

当执行一个程序时，调用exec的进程可将命令行参数传递给该新程序。这是LINUX/UNIX shell的一部分常规操作。

下面的程序将其所有命令行参数都回送到标准输出上。注意，通常的echo(1)程序不会回送第0个参数。

|  |
| --- |
|  |

程序的运行结果如下：

|  |
| --- |
|  |

注意数组的最后一个元素是一个空指针，所以在参数循环的时候应该写成：

for(i = 0; argv[i] != NULL, ++i)

## 7.5环境表

每个程序都会接收到一张环境表。与参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以null结束的C字符串的地址。全局变量environ则包含了该指针数组的地址：

extern char \*\*environ；

例如，如果该环境包含5个字符串，那么它看起来可能如下图。其中，每个字符串的结尾处都显式地有一个null字符。我们称environ为环境指针（environment pointer），指针数组为环境表，其中各指针指向的字符串为环境字符串。

|  |
| --- |
|  |
| 由5个C字符串组成的环境变量表 |

按照惯例，环境由

name=value

这样的字符串组成，这与上图中所示相同。大多数预定义的名字完全由大写字母组成，但这只是一个惯例。

在历史上，大多数LINUX/UNIX系统支持main函数带有三个参数，其中第三个参数就是环境表的地址：

int main (int argc, char \*argv[] , char \*envp [])；

因为ISO C规定main函数只有两个参数，而且第三个参数与全局变量environ相比也没有带来更多益处，所以POSIX.1也规定应使用environ而不使用第三个参数。通常用getenv和putenv函数来访问特定的环境变量，而不是用environ变量。但是，如果要查看整个环境，则必须使用environ指针。

## 7.6C程序的存储空间布局

从历史上讲，C程序一直由下面几部分组成：

* 正文段。这是由CPU执行的机器指令部分。通常，正文段是可共享的，所以即使是频繁执行的程序（如文本编辑器、C编译器和shell等）在存储器中也只需有一个副本，另外，正文段常常是只读的，以防止程序由于意外而修改其自身的指令。
* 初始化数据段。通常将此段称为数据段，它包含了程序中需明确地陚初值的变量。例如，C程序中出现在任何函数之外的声明：

int maxcount = 99；

使此变量带有其初值存放在初始化数据段中。

* 非初始化数据段。通常将此段称为bss段，这一名称来源于一个早期的汇编运算符，意思是“block started by symbol”（由符号开始的块），在程序开始执行之前，内核将此段中的数据初始化为0或空指针。出现在任何函数外的C声明

long sum[1000]；

使此变量存放在非初始化数据段中。

* 栈。自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次调用函数时，其返回地址以及调用者的环境信息（例如某些机器寄存器的值）都存放在栈中。然后，最近被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈，可以递归调用C函数。递归函数每次调用自身时，就使用一个新的栈帧，因此一个函数调用实例中的变量集不会影响另一个函数调用实例中的变量。
* 堆。通常在堆中进行动态存储分配。由于历史上形成的惯例，堆位于非初始化数据段和栈之间。

下图显示了这些段的一种典型安排方式。这是程序的逻辑布局，虽然并不要求一个具体实现一定以这种方式安排其存储空间，但这是一种我们便于说明的典型安排。对于x86处理器上的Linux，正文段从0x08048000单元开始，栈底则在0xC0000000之下开始（在这种特定结构中，栈从高地址向低地址方向增长）。堆顶和栈底之间未用的虚地址空间很大。

|  |
| --- |
|  |
| 典型的存储器安排 |

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

在上图中还可注意到，未初始化数据段的内容并不存放在磁盘上的程序文件中。其原因是，内核在程序开始运行前将它们都设置为0。需要存放在程序文件中的段只有正文段和初始化数据段。

size(l)命令报告正文段、数据段和bss段的长度（单位：字节）。例如：

$ size /uer/bin/cc /bin/sh

text data bss dec hex filename

79606 1536 916 82058 1408a /usr/bin/cc

619234 21120 18260 658614 a0cb6 /bin/sh

第4列和第5列是分别以十进制和十六进制表示的三个段的总长度。

## 7.7共享库

现在，大多数LINUX/UNIX系统支持共享库。共享库使得可执行文件中不再需要包含公用的库例程，而只需在所有进程都可引用的存储区中维护这种库例程的一个副本。程序第一次执行或者第一次调用某个库函数时，用动态链接方法将程序与共享库函数相链接。这减少了每个可执行文件的长度，但增加了一些运行时间开销。这种时间开销发生在该程序第一次被执行时，或者每个共享库函数第一次被调用时。共享库的另一个优点是可以用库函数的新版本代替老版本，而无需对使用该库的程序重新连接编辑。（假定参数的数目和类型都没有发生改变。）

在不同的系统中，程序可能使用不同的方法说明是否要使用共享库。比较典型的有cc(l)和ld(l)命令的选项。作为长度方面发生变化的例子，先用无共享库方式创建下列可执行文件（经典的hello.c程序）：

|  |
| --- |
| $ cc -static hellol.c //阻止gcc使用共享库  $ ls -1 a.out  -rwxrwxr- -x 1 sar 475570 Feb 18 23:17 a.out  $ size a. .out  text data bss dec hex filename  375657 37.80 3220 382657 5d6cl a.out |

如果编译此程序以使用共享库，则可执行文件的正文和数据段的长度都会显著减小

|  |
| --- |
| $ cc hellol.c //gcc默认使用共享库  $ ls -l a.out  -rwxrwxr-x 1 sar 11410 Feb 18 23:19 a.out  $ size a. out  text data bss dec hex filename  872 256 4 1132 46c a.out |

## 7.8存储器分配

ISO C说明了三个用于存储空间动态分配的函数。

(1)malloc。分配指定字节数的存储区。此存储区中的初始值不确定。

(2)calloc。为指定数量具指定长度的对象分配存储空间。该空间中的每一位都初始化为0。

(3)realloc。更改以前分配区的长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，以便在尾端提供增的存储区，而新增区域内的初始值则不确定。

|  |
| --- |
| #include <stdlib.h>  void \*malloc(size\_t size);  void \*calloc(size\_t nobj, size\_t size);  void \*realloc(void \*ptr, size\_t newsize);  三个函数返回值：若成功则返回非空指针，若出错则返回NULL  void free (void \*ptr) |

这三个分配函数所返回的指针一定是适当对齐的，使其可用于任何数据对象。例如，在一个特定的系统上，如果最苛刻的对齐要求是，double必须在8的倍数地址单元处开始，那么这三个函数返回的指针都应这样对齐。

因为这三个alloc函数都返回通用指针void\*，所以如果在程序中包括了#include <stdlib.h>（以获得函数原型），那么当我们将这些函数返回的指针赋予一个不同类型的指针时，就不需要显式地执行类型强制转换。

函数free释放ptr指向的存储空间。被释放的空间通常被送入可用存储区池，以后，可在调用上述三个分配函数时再分配。

realloc函数使我们可以增、减以前分配区的长度（最常见的用法是增加该区）。例如，如果先为一个数组分配存储空间，该数组长度为512，然后在运行时填充它，但运行一段时间后发现该数组原先的长度不够用，此时就可调用realloc扩充相应存储空间。如果在该存储区后有足够的空间可供扩充，则可在原存储区位置上向髙地址方向扩充，无需移动任何原先的内容，并返回传送给它的同样的指针值。如果在原存储区后没有足够的空间，则realloc分配另一个足够大的存储区，将现有的512个元素数组的内容复制到新分配的存储区。然后，释放原存储区，返回新分配区的指针。因为这种存储区可能会移动位置，所以不应当使任何指针指到该区中。

注意，realloc的最后一个参数是存储区的（新长度），它不是新、旧存储区长度之差。作为一个特例，若ptr是一个空指针，贝realloc的功能与malloc相同，用于分配一个指定长度为newsize的存储区。

这些分配例程通常用sbrk(2)系统调用实现。该系统调用扩充（或缩小）进程的堆。

虽然sbrk可以扩充或缩小进程的存储空间，但是大多数malloc和free的实现都不减小进程的存储空间。释放的空间可供以后再分配，但通常将它们保持在malloc池中而不返回给内核。

应当注意的是，大多数实现所分配的存储空间比所要求的要稍大一些，额外的空间用来记录管理信息——分配块的长度、指向下一个分配块的指针等等。这就意味着如果超过一个已分配区的尾端进行写操作，则会重写后一个块的管理记录。这种类型的错误是灾难性的，但是因为这种错误不会很快暴露出来，所以也就很难发现。同样，在已分配区起始位置之前进行写操作会重写本块的管理记录。

在动态分配的缓冲区前或后进行写操作，破坏的可能不仅仅是该区的管理记录信息。在动态分配的缓冲区前后的存储区很可能用于其他动态分配的对象。这些对象与破坏它们的代码可能无关，这造成寻求信息破坏的源头更加困难。

其他可能产生的致命性的错误是：释放一个已经释放了的块，调用free时所用的指针不是三个alloc函数的返回值等。如若一个进程调用malloc函数，但却忘记调用free函数，那么该进程占用的存储器就会连续增加，这被称为泄漏（leakage）。不调用free函数以释放不再使用的空间，那么进程地址空间长度就会漫慢增加，直至不再有空闲空间。此时，由于过度的分页开销，因而使性能下降。

因为存储器分配出错很难跟踪，所以某些系统提供了这些函数的另一种实现版本。每次调用这三个分配函数中的任意一个或free时，它们都进行附加的检错。在调用连接编辑器时指定一个专用库，则在程序中就可使用这种版本的函数。此外还有公共可用的资源，在对其进行编译时使用一个特殊标志就会使附加的运行时检查生效。

## 7.9环境变量

如同前述，环境字符串的形式通常如下：

name=value

LINUX/UNIX内核并不查看这些字符串，它们的解释完全取决于各个应用程序。例如，shell使用了大量的环境变量。其中某一些在登录时自动设置（如HOME、USER等），有些则由用户设置。我们通常在一个shell启动文件中设置环境变量以控制shell的动作。例如，若设置了环境变量MAILPATH，则它告诉shell到哪里去查看邮件。

ISO C定义了一个函数getenv，可以用其取环境变量值，但是该标准又称环境的内容是由实现定义的。

|  |
| --- |
| #include <stdlib.h>  char \*getenv(const char \*name);  返回值：指向与name关联的value的指针，若未找到则返回NULL |

注意，此函数返回一个指针，它指向name = value字符串中的value。我们应当使用getenv从环境中取一个指定环境变量的值，而不是直接访问environ。

LINUX/UNIX定义了某些环境变量，常见的如下表。

|  |  |
| --- | --- |
| 变量 | 说明 |
| COLUMNS | 终端宽度 |
| DATEMSK | getdate(3)模板文件路径名 |
| HOME | 起始目录 |
| LANG | 本地名 |
| LC\_ALL | 本地名 |
| LC\_COLLATE | 本地排序名 |
| LC\_CTYPE | 本地字符分类名 |
| LC\_MESSAGES | 本地消息名 |
| LC\_MONETARY | 本地货币编辑名 |
| LC\_NUMERIC | 本地数字编辑名 |
| LC\_TIME | 本地日期/时间格式名 |
| LINES | 终端高度 |
| LOGNAME | 登录名 |
| NLSPATH | 消息类模板序列 |
| PATH | 搜索可执行文件的路径前缀列表 |
| PWD | 当前工作目录的绝对路径名 |
| SHELL | 用户首选的shell名 |
| TERM | 终端类型 |
| TMPDIR | 在其中创建临时文件的目录路径名 |
| TZ | 时区信息 |

注意，ISO C没有定义任何环境变量。

除了取环境变量的值，有时也需要设置环境变量。我们可能希望改变现有变量的值，或者增加新的环境变量。在下一章将会了解到，我们能影响的只是当前进程及调用的任何子进程的环境，但不能影响父进程的环境，这通常是一个shell进程。尽管如此，修改环境表的能力仍然是很有用的。

常用的设置环境变量的三个函数的原型是：

|  |
| --- |
| #include <stdlib.h>  int putenv (char \*str);  int setenv (const char \*name, const char \* value, int rewrite);  int unsetenv(const char \*name);  三个函数返回值：若成功则返回0，若出错则返回非0值 |

这三个函数的操作是：

* putenv取形式为name=value的字符串，将其放到环境表中。如果name已经存在，则先删除其原来的定义。
* setenv将name设置为value。如果在环境中name已经存在，那么(a)若rewrite非0，则首先删除其现有的定义，(b)若rewrite为0，则不删除其现有定义（name不设置为新的value，而且也不出错）。
* unsetenv删除name的定义。即使不存在这种定义也不算出错。

注意putenv和setenv之间的区别。setenv必须分配存储区，以便依据其参数创建name=value字符串。同时，putenv则无需将传送给它的参数字符串直接放到环境中。确实，在LINUX中，putenv实现将传送给它的字符串地址作为参数直接放入环境表中。在这种情况下，将存放在栈中的字符串作为参数传送给该函数就会发生错误，其原因是，从当前函数返回时，其栈桢占用的存储区可能将被重用。

这些函数在修改环境表时是如何进行操作的呢？对这一问题进行研究考察是非常有益的。回忆进程结构分布图，其中，环境表（指向实际name=value字符串的指针数组）和环境字符串通常存放在进程存储空间的顶部（栈之上）。删除一个字符串很简单——只要先在环境表中找到该指针，然后将所有后续指针都向环境表首部顺次移动一个位置。但是增加一个字符串或修改一个现有的字符串就困难得多。环境表和环境字符串通常占用的是进程地址空间的顶部，所以它不能再向髙地址方向（向上）扩展：同时也不能移动在它之下的各栈帧，所以它也不能向低地址方向（向下）扩展。两者组合使得该空间的长度不能再增加。

(1) 如果修改一个现有的name:

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

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

(2) 如果要增加一个新的name，则操作就更加复杂。首先，调用malloc为name=value字符串分配空间，然后将该字符串复制到此空间中，则：

(a) 如果这是第一次增加一个新name，则必须调用malloc为新的指针表分配空间。接着，将原来的环境表复制到新分配区，并将指向新name=value字符串的指针存放在该指针表的表尾，然后又将一个空指针存放在其后。最后使environ指向新指针表。如果原来的环境表位于找顶之上（这是一种常见情况），那么必须将此表移至堆中。但是，此表中的大多数指针仍指向栈顶之上的各name=value字符串。

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

## 7.10 setjmp和longjmp函数

在C中，goto语句是不能跨越函数的，而执行这类跳转功能的是函数setjmp和longjmp。这两个函数对于处理发生在深层嵌套函数调用中的出错情况是非常有用的。

考虑下面程序。

|  |
| --- |
|  |

其主循环是从标准输入读1行，然后调用do\_line处理该输入行。do\_line函数调用get\_token从该输入行中取下一个标记。一行中的第一个标记假定是一条某种形式的命令，于是switch语句就实现命令选择。我们的程序只处理一条命令（add命令），对此命令调用cmd\_add函数。

上述程序框架典型地用于读命令，确定命令的类型，然后调用相应函数处理每一条命令这一类程序。下图显示了调用了cmd\_add之后栈的大致使用情况。

|  |
| --- |
|  |
| 调用cmd\_add后的各个栈帧 |

自动变量的存储单元在每个函数的栈帧中。数组line在main的栈帧中，整型cmd在do\_line的栈帧中，整型token在cmd\_add的找帧中。

如上所述，这种形式的栈安排是非常典型的，但并不要求非如此不可。栈并不一定要向低地址方向扩充。某些系统对栈并没有提供特殊的硬件支持，此时一个C实现可能要用链接表实现栈帧。

在上面程序时经常会遇到的一个问题是，如何处理非致命性的错误。例如，若cmd\_add函数发现一个错误（譬如说一个无效的数），那么它可能先打印一个出错消息，然后希望忽略输入行的余下部分，返回main函数并读下一输入行。但是如果这种情况出现在main函数中的深层嵌套中时，用C语言就比较难以做到这一点。（在本例中，cmd\_add函数只比main深两个层次，在有些程序中往往深五个层次或更多。）如果我们不得不以检查返回值的方法逐层返回，那就会变得很麻烦。

解决这种问题的方法就是使用非局部goto——setjmp和longjmp函数。非局部指的是，这不是由普通C语言goto语句在一个函数内实施的跳转，而是在栈上跳过若干调用帧，返回到当前函数调用路径上的某一个函数中。

|  |
| --- |
| #include <setjmp.h>  int setjmp(jmp\_buf env);  返回值：若直接调用则返回0，若从long jmp调用返回则返回非0值  void 1ongjmp(jmp\_buf env, int val); |

在希望返回到的位置调用setjmp，在本例中，此位置在main函数中。因为我们直接调用该函数，所以其返回值为0。setjmp参数的类型是一个特殊类型jmp\_buf。这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息。因为需在另一个函数中引用env变量，所以规范的处理方式是将变量定义为全局变量。

当检查到一个错误时，例如在cmd\_add函数中，则以两个参数调用longjmp函数。第一个就是在调用setjmp时所用的env，第二个参数是具有非0值的val，它将成为从setjmp处返回的值。使用第二个参数的原因是对于一个setjmp可以有多个longjmp。例如，可以在cmd\_add中以val为1调用longjmp,也可在get\_token中以val为2调用longjmp。在main函数中，setjmp的返回值就会是1或2，通过测试返回值就可判断造成返回的longjmp是在cmd\_add还是在get\_token中。

再回到程序实例中，下面程序给出了经修改过后的main和cmd\_add函数（另外两个函数do\_line和get\_token未经更改）。

|  |
| --- |
|  |

执行main时，调用setjmp，它将所需的信息记入变量jmpbuffer中并返回0。然后调用do\_line，它又调用cm\_add,假定在其中检测到一个错误。在cmd\_add中调用longjmp之前，栈的形式如下图所示。

|  |
| --- |
|  |
| 调用longjmp后的栈帧 |

但是longjmp使栈反绕到执行main函数时的情况，也就是抛弃了cmd\_add和do\_line的栈帧（如上图）。调用longjmp造成main中setjmp的返回，但是，这一次的返回值是1（longjmp的第二个参数）。

### 1. 自动、寄存器和易失变置

我们已经了解在调用longjmp后栈帧的基本结构，下一个问题是：在main函数中，自动变量和寄存器变量的状态如何？当longjmp返回到main函数时，这些变量的值是否能恢复到以前调用setjmp时的值（即回滚到原先值），或者这些变量的值保持为调用do\_line时的值（do\_line调用cmd\_add，cmd\_add又调用longjmp）？不幸的是，对此问题的回答是“看情况”。大多数实现并不回滚这些自动变量和寄存器变量的值，而所有标准则说它们的值是不确定的。如果你有一个自动变量，而又不想使其值回滚，则可定义其为具有volatile属性。声明为全局或静态变量的值在执行longjmp时保持不变。

下面程序演示了在调用longjmp后，自动变量、全局变量、寄存器变量、静态变量和易失变量的不同情况。

|  |
| --- |
|  |

如果以不带优化和带优化选项对此程序分别进行编译，然后运行它们，则得到的结果是不同的：

|  |
| --- |
|  |

注意，全局、静态和易失变量不受优化的影响，在调用longjmp之后，它们的值是最近所呈现的值。某个系统的setjmp(3)手册页上说明，存放在存储器中的变量将具有longjmp时的值，而在CPU和浮点寄存器中的变量则恢复为调用setjmp时的值。这确实就是运行上述程序时所观察到的值。不进行优化时，所有这5个变量都存放在存储器中（亦即忽略了对regival变量的register存储类说明）。而进行了优化后，autoval和regival都存放在寄存器中（即使autoval并未声明为register），volatile变量则仍存放在存储器中。通过这一实例我们可以理解到，如果要编写一个使用非局部跳转的可移植程序，则必须使用volatile属性。但是从一个系统移植到另一个系统，任何事情都可能改变。

在上述程序中，某些printf的格式字符串可能不适宜安排在程序文本的一行中。我们没有将其分成多个printf调用，而是使用了ISO C的字符串连接功能，于是两个字符串序列：

"stringl" "string2"

等价于

"stringlstring2"

后面讨论信号处理程序及其信号版本sigsetjmp和siglongjmp时，将再次涉及setjmp和longjmp函数。

### 2. 自动变置的潜在问题

前面已经说明了处理栈帧的一般方式，现在值得分析一下自动变量的一个潜在出错情况。基本规则是声明自动变量的函数已经返回后，不能再引用这些自动变量。在整个LINUX/UNIX系统手册中，关于这点有很多警告。

下列程序中示出了一个名为open\_data的函数，它打开了一个标准I/O流，然后为该流设置缓存。

|  |
| --- |
|  |

问题是：当open\_data返回时，它在栈上所使用的空间将由下一个被调用函数的栈帧使用。但是，标准I/O库函数仍将使用其流缓冲区的存储空间。这就产生了冲突和混乱。为了校正这一问题，应在全局存储空间静态地（如static或extern）或者动态地（使用一种alloc函数）为数组databuf分配空间。

## 7.11 getrlimit 和 setrlimit 函数

每个进程都有一组资源限制，其中一些可以用getrlimit和setrlimit函数查询和更改。

|  |
| --- |
| #include <sys/resource .h>  int getrlimit (int resource, struct rlimit \*rlptr);  int set rlimit (int resource, const struct rlimit \*rlptr);  两个函数返回值：若成功则返回0，若出错则返回非0值 |

进程的资源限制通常是在系统初始化时由进程0建立的，然后由每个后续进程继承。每种实现都可以用自己的方法对各种限制做出调整。

对这两个函数的每一次调用都会指定一个资源以及一个指向下列结构的指针。

struct rlimit {

rlim\_t rlim\_cur； /\* soft limit: current limit \*/

rlim\_t rlim\_max; /\* hard limit： maximum value for rlim\_cur \*/

}

在更改资源限制时，须遵循下列三条规则：

(1) 任何一个进程都可将一个软限制值更改为小于或等于其硬限制值。

(2) 任何一个进程都可降低其硬限制值，但它必须大于或等于其软限制值。这种降低对普通用户而言是不可逆的。

(3) 只有超级用户进程可以提髙硬限制值。

常量RLIM\_INFINI1T指定了一个无限量的限制。

这两个函数的resource参数取下列值之一。

|  |  |
| --- | --- |
| 限制 | 含义 |
| RLIMIT\_AS | 进程可用存储区的最大总长度（字节）。这会影响sbrk和mmap函数。 |
| RLIMIT\_CORE | core文件的最大字节数，若其值为0则阻止创建core文件。 |
| RLIMIT\_CPU | CPU时间的最大量值（秒），当超过此软限制时，向该进程发送SIGXCPU信号。 |
| RLIMIT\_DATA | 数据段的最大字节长度。这是初始化数据、非初始以及堆的总和。 |
| RLIMIT\_FSIZE | 可以创建的文件的最大字节长度。当超过此软限制时，则向该进程发送SIGXFSZ信号。 |
| RLIMIT\_LOCKS | 一个进程可持有的文件锁的最大数（此数也包括Linux特有的文件租借数）。 |
| RLIMIT\_MEMLOCK | 一个进程使用mlock(2)能够锁定在存储器中的最大字节长度。 |
| RLIMIT\_NOFILE | 每个进程能打开的最大文件数。 |
| RLIMIT\_NPROC | 每个实际用户ID可拥有的最大子进程数。 |
| RLIMIT\_RSS | 最大驻内存集的字节长度（resident set size in bytes, RSS）。如果物理存储器供不应求，则内核将从进程处取回超过RSS的部分。 |
| RLIMIT\_SBSIZE | 用户在任一给定时刻可以占用的套接字缓冲区的最大长度（字节）。 |
| RLIMIT\_STACK | 栈的最大字节长度。 |
| RLIMIT\_VMEM | 这是RLIMIT\_AS的同义词。 |

资源限制影响到调用进程并由其子进程继承。这就意味着为了影响一个用户的所有后续进程，需将资源限制的设置构造在shell之中。确实，Bourne shell（sh）, GNU Boume-again shell（bash）具有内置的ulimit命令，C shell具有内置的limit命令。（umask和chdir函数也必须是shell内置的。）

下面的程序打印由系统支持的所有资源当前的软限制和硬限制。为了在各种实现上编译该程序，我们已经有条件地包括了各种不同的资源名。另请注意，有些平台定义rlim\_t为unsigned long long而非unsigned long，对于此种平台，必须使用不同的printf格式。

|  |
| --- |
|  |

注意，在doit宏中使用了ISO C的字符串创建运算符（#），以便为每个资源名产生字符串值。例如：

doit（RLIMIT \_CORE）

这将由C预处理程序扩展为：

pr\_limits ("RLIMIT\_CORE", RLIMIT\_CORE);

下面是程序的运行结果：

|  |
| --- |
|  |

后面在介绍信号之后还将继续介绍资源限制。

## 7.12小结

理解LINUX/UNIX环境中C程序的环境是理解LINUX/UNIX进程控制特征的先决条件。本章说明了一个进程是如何启动和终止的，如何向其传递参数表和环境。虽然这两者都不是由内核进行解释的，但内核起到了从exec的调用者将这两者传递给新进程的作用。

本章也说明了C程序的典型存储空间布局，以及一个进程如何动态地分配和释放存储器。详细地了解用于维护环境的一些函数是值得的，因为它们涉及存储器分配。本章也介绍了setjmp和longjmp函数，它们提供了一种在进程内执行非局部转移的方法。最后介绍了各种实现提供的资源限制功能。

# 第8章 进程控制

## 8.1引言

本章介绍LINUX/UNIX的进程控制，包括创建新进程、执行程序和进程终止。还将说明进程属性的各种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。本章还包括了解释器文件和system函数。本章最后讲述大多数LINUX/UNIX系统所提供的进程会计机制。这种机制使我们能够从另一个角度了解进程的控制功能。

## 8.2进程标识符

每个进程都有一个非负整型表示的唯一进程ID。因为进程ID标识符总是唯一的，常将其用作其他标识符的一部分以保证其唯一性。例如，应用程序有时就把进程ID作为名字的一部分来创建一个唯一的文件名。

虽然是唯一的，但是进程ID可以重用。当一个进程终止后，其进程ID就可以再次使用了。大多数LINUX/UNIX系统实现延迟重用算法，使得赋予新建进程的ID不同于最近终止进程所使用的ID。这防止了将新进程误认为是使用同一ID的某个已终止的先前进程。

系统中有一些专用的进程，但具体细节因实现而异。ID为0的进程通常是调度进程，常常被称为交换进程（swapper）。该进程是内核的一部分，它并不执行任何磁盘上的程序，因此也被称为系统进程。进程ID1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LINUX/UNIX是/etc/init。init通常读与系统有关的初始化文件（/etc/rc\*文件或/etc/inittab文件，以及/etc/init.d中的文件），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。

每个LINUX/UNIX系统实现都有它自己的一套提供操作系统服务的内核进程，例如，在某些LINUX/UNIX的虚拟存储器实现中，进程ID 2是页守护进程（pagedaemon）。此进程负责支持虚拟存储系统的分页操作。

除了进程ID，每个进程还有一些其他的标识符。下列函数返回这些标识符。

|  |
| --- |
| #include <unistd.h>  pid\_t getpid(void)；  返回值：调用进程的进程ID  pid\_t getppid(void)；  返回值：调用进程的父进程ID  uid\_t getuid(void)；  返回值：调用进程的实际用户ID  uid\_t geteuid(void)；  返回值：调用进程的有效用户ID |  gid\_t getgid(void)；  返回值：调用进程的实际组ID  gid\_t getegid(void)；  返回值：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。

## 8.3 fork 函数

一个现有进程可以调用fork函数创建一个新进程。

|  |
| --- |
| #include <unistd.h>  pid\_t fork(void)；  返回值：子进程中返回0，父进程中返回子进程ID。出错返回-1 |

由fork创建的新进程被称为子进程（child process）。fork函数被调用一次，但返回两次。两次返回的唯一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有多个，并且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总是由内核交换进程使用，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调用之后的指令。子进程是父进程的副本。例如，子进程获得父进程数据空间、堆和栈的副本。注意，这是子进程所拥有的副本。父、子进程并不共享这些存储空间部分。但父、子进程共享正文段，这在上一张讨论进程空间的时候讨论过。

由于在fork之后经常跟随着exec，所以现在的很多实现并不执行一个父进程数据段、栈和堆的完全复制。作为替代，使用了写时复制（Copy-On-Write，COW）技术。这些区域由父、子进程共享，而且内核将它们的访问权限改变为只读的。如果父、子进程中的任一个试图修改这些区域，则内核只为修改区域的那块内存制作一个副本，通常是虚拟存储器系统中的一“页”。

下面的程序演示了fork函数，从中可以看到子进程对变量所作的改变并不影响父进程中该变量的值。

|  |
| --- |
|  |

程序的运行结果如下：

|  |
| --- |
|  |

子进程的变量值改变了 父进程的变量值没有改变

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。在上述程序中，父进程使自己休眠2秒钟，以使子进程先执行。但并不保证2秒钟已经足够，后面章节说明竞争条件时，还将谈及这一问题及其他类型的同步方法。后面章节，我们将说明在fork 之后如何使用信号使父、子进程同步。

当写到标准输出时，我们将buf长度减去1作为输出字节数，这是为了避免将终止null字节写出。strlen计算不包含终止null字节的字符串长度，而sizeof则计算包括终止null字节的缓冲区长度。两者之间的另一个差别是，使用strlen需进行一次函数调用，而对于sizeof而言，因为缓冲区已用已知字符串进行了初始化，其长度是固定的，所以sizeof在编译时计算缓冲区长度。

注意上述程序中fork与I/O函数之间的交互关系。如前面文件章节所述，write函数是不带缓冲的。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库是带缓冲的。如果标准输出连到终端设备，则它是行缓冲的，否则它是全缓冲的。当以交互方式运行该程序时，只得到该printf输出的行一次，其原因是标准输出缓冲区由换行符冲洗。但是当将标准输出重定向到一个文件时，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，该行数据仍在缓冲区中，然而在将父进程数据空间复制到子进程中时，该缓冲区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标准I/O缓冲区。在exit之前的第二个printf将其数据添加到现有的缓冲区中。当每个进程终止时，最终会冲洗其缓冲区中的副本。

文件共享

对上述程序需注意的另一点是：在重定向父进程的标准输出时，子进程的标准输出也被 重定向。实际上，fork的一个特性是父进程的所有打开文件描述符都被复制到子进程中。父、 子进程的每个相同的打开描述符共享一个文件表项，如同我们在文件中讨论的那样。

考虑下述情况，一个进程具有三个不同的打开文件，它们是标准输入、标准输出和标准出错。在从fork返回时，我们有了下图所示的结构。

|  |
| --- |
|  |
| 调用fork之后，父子进程对打开文件的共享 |

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偏移量。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进行写操作。如果父进程的标准输出已重定向（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偏移量。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偏移量，这种形式的交互就很难实现。

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（如上图），但这并不是常用的操作模式。

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偏移量已执行了相应更新。

(2) 父、子进程各自执行不同的程序段。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，这样就不会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，父进程的很多其他属性也由子进程继承，包括：

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

父、子进程之间的区别是：

* fork的返回值。
* 进程ID不同。
* 两个进程具有不同的父进程ID：子进程的父进程ID是创建它的进程的ID，而父进程的父进程ID则不变。
* 子进程的tms\_utime、tms\_stime、tms\_cutime以及tms\_ustime均被设置为0。
* 父进程设置的文件锁不会被子进程继承。
* 子进程的未处理的闹钟（alarm）被清除。
* 子进程的未处理信号集设置为空集。

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：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），或者该实际用户ID的进程总数超过了系统限制。

fork有下面两种用法：

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

(2) —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，子进程从fork返回后立即调用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。 LINUX/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重定向、用户ID、信号安排等。

## 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用于创建一个新进程，而该新进程的目的是exec一个新程序（与上一节末尾的(2) 中一样）。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。相反，在子进程调用exec或exit之前，它在父进程的空间中运行。这种优化工作方式在某些LINUX/UNIX的页式虚拟存储器实现中提高了效率。 这与上一节中提及的在fork之后跟随exec，并采用在写时复制技术相似，而且不复制比部分复制要更快一些。

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异步动作， 则会导致死锁。

下面这个程序是上一节中程序的修改版，其中用vfork代替了fork，删除了对于标准输出的write调用。另外，我们也不再需要让父进程调用sleep，vfork已保证在子进程调用exec或exit之前，内核会使父进程处于休眠状态。

|  |
| --- |
|  |

运行该程序得到：

|  |
| --- |
|  |

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，在上述程序中，调用了\_exit而不是exit。如上章所述，\_exit并不执行标准I/O缓冲的冲洗操作。如果调用的是exit而不是\_exit，则该程序的输出是不确定的。它依赖于标准I/O库的实现，我们可能会见到输出没有发生变化，或者发现没有出现父进程的printf输出。

如果子进程调用exit，而该实现冲洗所有标准I/O流。如果这是函数库采取的唯一动作，那么我们会见到输出与子进程调用\_exit所产生的输出完全相同，没有任何区别。如果该实现也关闭标准I/O流，那么表示标准输出FILE对象的相关存储区将被清0。因为子进程借用了父进程的地址空间，所以当父进程恢复运行并调用printf时，也就不会产生任何输出，printf返 回-1。注意，父进程的STDOUT\_FILENO仍旧有效，子进程得到的是父进程的文件描述符数组的副本。

大多数exit的现代实现不再在流的关闭方面自找麻烦。因为进程即将终止。那时内核将关闭在进程中已打开的所有文件描述符。在库中关闭它们，只是增加了开销而不会带来任何益处。

## 8.5 exit函数

上章所述，进程有下面5种正常终止方式：

(1) 在main函数内执行return语句，这等效于调用exit。

(2) 调用exit函数。此函数由ISO 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函数时登记），然后关闭所有标准I/O流等。因为ISO 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LINUX/UNIX系统而言是不完整的。

(3) 调用\_exit或\_Exit函数。ISO C定义\_Exit，其目的是为进程提供一种无需运行终止处理程序或信号处理程序而终止的方法。对标准I/O流是否进行冲洗，这取决于实现。在LINUX/UNIX系统中，\_Exit和\_exit是同义的，并不清洗标准I/O流。\_exit函数由exit调用，它处理LINUX/UNIX特定的细节。\_exit是由POSIX.1说明的。

在大多数LINUX/UNIX系统实现中，exit(3)是标准C库中的一个函数，而\_exit(2)则是一个系统调用。

(4) 进程的最后一个线程在其启动例程中执行返回语句。但是，该线程的返回值不会用作进程的返回值。当最后一个线程从其启动例程返回时，该进程以终止状态0返回。

(5) 进程的最后一个线程调用pthread\_exit函数。如同前面一样，在这种情况中，进程终止状态总是0，这与传送给pthread\_exit的参数无关。

三种异常终止方式如下：

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

(2) 当进程接收到某些信号时。后面将较详细地说明信号。信号可由进程自身(例如调用abort函数)、其他进程或内核产生。例如，若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。

(3) 最后一个线程对“取消”（cancellation）请求做出响应。按系统默认，“取消”以延迟方 式发生：一个线程要求取消另一个线程，一段时间之后，目标线程终止。

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

对上述任意一种终止情形。我们都希望终止进程能够通知其父进程它是如何终止的。对于三个终止函数（exit、\_exit和\_Exit），实现这一点的方法是，将其退出状态（exit status）作为参数传送给函数。在异常终止情况下，内核（不是进程本身）产生一个指示其异常终止原因的终止状态（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，内核将退出状态转换成终止状态。下一节将说明父进程检查子进程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，显而易见，子进程是在父进程调用fork后生成的。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。其操作过程大致如下：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止进程的子进程，如果是，则将该进程的父进程ID更改为1（init进程的ID）。 这种处理方法保证了每个进程都有一个父进程。

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，可以得到这些信息。这些信息至少包括进程ID、该进程的终止状态、以及该进程使用的CPU时间总量。内核可以释放终止进程所使用的所有存储区，关闭其所有打开文件。在LINUX/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(l)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，它调用fork产生了很多子进程，那么除非父进程等待取得子进程的终止状态，否则这些子进程终止后就会变成僵死进程。

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被编写成无论何时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，这指的可能是init直接产生的进程，也可能是其父进程已终止，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。父进程可以选择忽略该信号，或者提供一个该信号发生时即被调用执行的函数（信号处理程序）。对于这种信号的系统默认动作是忽略它。后面将说明这些选项。现在需要知道的是调用wait或waitpid的进程可能会发生什么情况：

* 如果其所有子进程都还在运行，则阻塞。
* 如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回。
* 如果它没有任何子进程，则立即出错返回。

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。但是如果在任意时刻调用wait，则进程可能会阻塞。

|  |
| --- |
| #include <sys/wait.h>  pid\_t wait (int \*statloc);  pid\_t waitpid (pid\_t pid, int \*statloc, int options);  两个函数返回值：若成功则返回进程ID.0（见后面的说明），若出错则返回-1 |

这两个函数的区別如下：

* 在一个子进程终止前，wait使其调用者阻塞，而waitpid有一个选项，可使调用者不阻塞。
* waitpid并不等待在其调用之后的第一个终止子进程，它有若干个选项，可以控制它所等待的进程。

如果一个子进程已经终止，并且是一个僵死进程，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。POSIX.1规定终止状态用定义在<sys/wait.h>中的各个宏来査看。有四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。基于这四个宏中哪一个值为真，就可选用其他宏来取得终止状态、信号编号等。这四个互斥的宏在下表中列出。

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

下面的函数pr\_exit使用上表中的宏以打印进程终止状态的说明。本书中的很多程序都将调用此函数。注意，如果定义了WCOREDUMP宏，则此函数也处理该宏。

|  |
| --- |
|  |

下面程序调用pr\_exit函数，演示终止状态的各种值。运行该程序可得：

|  |
| --- |
|  |

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。我们必须査看<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LINUX/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，先査看已终止的进程列表，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。新版的LINUX/UNIX定义了waitpid函数以提供这种功能（以及其他一些功能）。

对于waitpid函数中pid参数的作用解释如下：

pid == -1 等待任一子进程。就这一方面而言，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等于调用进程组ID的任一子进程。

pid < -1 等待其组ID等于pid绝对值的任一子进程。

waitpid函数返回终止子进程的进程ID，并将该子进程的终止状态存放在由statloc指向的存储单元中。对干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。后面将对此进行讨论）。但是对于waitpid，如果指定的进程或进程组不存在，或者参数指定的进程不是调用进程的子进程则都将出错。

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

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

waitpid函数提供了wait函数没有提供的三个功能：

(1) waitpid可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。有时用户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（利用WUNTRACED和WCONTINUED选项）。

回忆前面章节有关僵死进程的讨论。如果一个进程fork一个子进程，但不要它等待子进程终止，也不希望子进程处干僵死状态直到父进程终止，实现这一要求的技巧是调用fork两次。下面的程序实现了这一点。

|  |
| --- |
|  |

第二个子进程调用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，并且我们无法预知哪一个会先执行。在fork之后，如果不使第二个子进程休眠，那么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ID1）。

执行程序得到：

|  |
| --- |
|  |

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

## 8.7 waitid函数

LINUX/UNIX包括了另一个取进程终止状态的函数——waitid，此函数类似于waitpid，但提供了更多的灵活性。

|  |
| --- |
| #include <sys/wait.h>  int waitid (idtype\_t idtype, id\_t id, siginfo\_t \* infop, int options);  返回值：若成功则返回0，若出错则返回-1 |

与waitpid相似，waitid允许一个进程指定要等待的子进程。但它使用单独的参数表示要等待的子进程的类型，而不是将此与进程ID或进程组ID组合成一个参数。id参数的作用与idtype的值相关。该函数支持的idtype类型列出在下表中：

|  |  |
| --- | --- |
| waited的idtype常量 | |
| 常 量 | 说 明 |
| P\_PID | 等待一个特定的进程：id包含要等待子进程的进程ID |
| P\_PGID | 等待一个特定进程组中的任一子进程：id包含要等待子进程的进程组ID |
| P\_ALL | 等待任一子进程：忽略id |

options参数是下表中各标志的按位“或”这些标志指示调用者关注哪些状态变化。

|  |  |
| --- | --- |
| waitid的options常置 | |
| 常 量 | 说 明 |
| WCONTINUED | 等待一个进程，它以前曾被暂停，此后又已继续，但其状态尚未报告 |
| WEXITED | 等待已退出的进程 |
| WNOHANG | 如无可用的子进程退出状态，立即返回而非阻塞 |
| WNOWAIT | 不破坏子进程退出状态。该子进程退出状态可由后续的wait、waitid或waitpid调用取得 |
| WSTOPPED | 等待一个进程。它已经暂停，但其状态尚未报告 |

infop参数是指向siginfo结构的指针。该结构包含了有关引起子进程状态改变的生成信 号的详细信息。后面进一步讨论siginfo结构。

## 8.8 wait3和wait4函数

大多数LINUX/UNIX系统实现提供了另外两个函数wait3和wait4。它们提供的功能比函数wait、waitpid和waitid所提供的功能要多一个，这与附加参数有关。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汇总。

|  |
| --- |
| #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);  两个的数返回值：若成功则返回进程ID，若出错则返回1 |

资源统计信息包括用户CPU时间总量、系统CPU时间总量、页面出错次数、接收到信号的次数等。有关细节可以参阅getrUsage(2)手册页。

## 8.9竞争条件

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争条件（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函数就会是竞争条件活跃的滋生地。通常，我们不能预料哪个进程先运行。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。

在前面程序中，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争条件。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，这也不会保证什么。如果系统负载很重，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，这种问题并不会出现。

如果一个进程希望等待一个子进程终止，则它必须调用一种wait函数。如果一个进程要等待其父进程终止（如上一个程序一样)，则可使用下列形式的循环：

while (getppid() != 1 )

sleep(1)；

这种形式的循环（称为轮询（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。

为了避免竞争条件和轮询，在多个进程之间需要有某种形式的信号发送和接收的方法。在LINUX/UNIX中可以使用信号机制，在下一章将说明它在解决此方面问题的一种用法。也可使用各种形式的进程间通信（IPC），后面章节将对此进行讨论。

在父、子进程的关系中，常常出现下述情况。在调用fork之后，父、子进程都有一些事情要做。例如，父进程可能要用子进程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。这种方案可以用代码描述如下：

|  |
| --- |
|  |

假定在各个需要使用的变量已经定义。5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。

在后面几章中会说明实现这些TELL和WAIT例程的不同方法。

## 8.10 exec函数

前面曾提及用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，该进程执行的程序完全替换为新程序，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只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。

有6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函数使得LINUX/UNIX进程控制原语更加完善。用fork可以创建新进程，用exec可以执行新程序。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[]);  6个函数返回值：若出错则返回\_1，若成功则不返回值 |

这些函数之间的第一个区别是前4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，则将其视为路径名。
* 否则就按PATH环境变量，在它所指定的各目录中捜寻可执行文件。

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，name=value环境字符串

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

指定在4个目录中进行捜索。最后的路径前缀表示当前目录。（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，在行尾则以：表示。）

如果execlp或execvp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器产生的机器可执行文件，则认为该文件是一个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（1表示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ISO 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 \*argl, …, 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，否则将它解释为整型参数。如果一个整型数的长度与char \*的长度不同，那么exec函数的实际参数就将出错。

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有的环境变量。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，即进程想要为子进程指定某一个确定的环境。例如，在初始化一个新登录的shell时，login程序通常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启动文件，将其他变量加到环境中。

在使用ISO 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地址。而在ISO C原型中，所有命令行参数、空指针和指针都用省略号(...)表示。

这6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1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[]矢量。最后，字母e表示该函数取envp[]数组，而不使用当前环境。下表显示了这6个函数之间的区别。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 函数 | pathname | filename | 参数表 | argv[] | environ | Envp[] |
| execl |  |  |  |  |  |  |
| execlp |  |  |  |  |  |  |
| execle |  |  |  |  |  |  |
| execv |  |  |  |  |  |  |
| execvp |  |  |  |  |  |  |
| execve |  |  |  |  |  |  |
| 字母 |  | 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。在符合POSIX.1标准系统中，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 getrlimit /usr/share/man/\*/\*

在某些系统上可能产生下列形式的shell错误：

Argument list too long

为了摆脱对参数表长度的限制，我们可以使用xargs(l)命令，将长参数表分解成几部分。为了寻找在我们所用系统手册中的getrlimit,我们可以用

find /usr/share/man -type f -print | xargs grep getrlimit

如果所用的系统手册是压缩过的，则可使用

find /usr/share/man -type f -print | xargs bzgrep getrlimit

对于find命令，我们使用选项-type f限制输出列表只包含普通文件。这样做的原因是，grep命令不能在目录中捜索模式，我们也想避免不必要的出错消息。

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。
* 附加组ID。
* 进程组ID。
* 会话ID。
* 控制终端。
* 闹钟尚余留的时间。
* 当前工作目录。
* 根目录。
* 文件模式创建屏蔽字。
* 文件锁。
* 进程信号屏蔽。
* 未处理信号。
* 资源限制。
* tms\_utime、tms\_stime、tms\_cutime以及tms\_cstime 值。

对打开文件的处理与每个描述符的执行时关闭（close-on-exec）标志值有关。前面讨论过，进程中每个打开描述符都有一个执行时关闭标志。若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用fcntl设置了该标志，否则系统的默认操作是在执行exec后仍保持这种描述符打开。

POSIX.1明确要求在执行exec时关闭打开的目录流（前面讨论过的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执行时关闭标志。

注意，在执行exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序文件的设置用户ID位和设置组ID位是否设置。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LINUX/UNIX实现中，这6个函数中只有execve是内核的系统调用。另外5个只是库函数，它们最终都要调用该系统调用。这6个函数之间的关系如下图。

|  |
| --- |
|  |
| 6个exec函数之间的关系 |

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。

下面程序演示了exec函数：

|  |
| --- |
|  |

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/<username>/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将此参数设置为完整的路径名。这只是一个惯例。我们可将argv[0]设置为任何字符串。当login命令执行shell时就是这样做的。在执行shell之前，login在argv[0]之前加一个/作为前缀，这向shell指明它是作为登录shell被调用的。 登录shell将执行启动配置文件（start-up profile）命令，而非登录shell则不会执行这些命令。

下面程序要执行两次的程序echoall如下，这是一个很普通的程序, 它回送其所有命令行参数及全部环境表。

|  |
| --- |
|  |

执行exec例子程序得到如下输出：

|  |
| --- |
|  |

注意，shell提示符出现在第二个exec打印argv[0]之前。这是因为父进程并不等待该子 进程结束。

## 8.11更改用户ID和组ID

在LINUX/UNIX系统中，特权（例如能改变当前日期的表示法以及访问控制（例如，能否读、写一特定文件））是基于用户和组ID的。当程序需要增加特权，或需要访问当前并不允许访问的资源时，我们需要更换自己的用户ID或组ID，使得新ID具有合适的特权或访问权限。与此类似，当程序需要降低其特权或阻止对某些资源的访问时，也需要更换用户ID或组ID，从而使新ID不具有相应特权或访问这些资源的能力。

一般而言，在设计应用程序时，我们总是试图使用最小特权(least privilege)模型。依照此模型，我们的程序应当只具有为完成给定任务所需的最小特权。这减少了安全性受到损害的可能性，这种安全性损害是由恶意用户试图哄骗我们的程序以未预料的方式使用特权所造成的。

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 <unistd.h>  int setuid (uid\_t uid);  int setgid(gid\_t gid);  两个函数返回值：若成功则返回0，若出错则返回-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（我们关于用户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，则将errno设置为EPERM，并返回-1。

在这里假定系统支持保存设置用户ID。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。

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(l)程序设置的，而且永远不会改变它。因为login是一个超级用户进程，当它调用setuid时，会设置所有三个用户ID。

(2) 仅当对程序文件设置了设置用户位时，exec函数才会设置有效用户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设置为任意随机值。

(3) 保存的设置用户ID是由exec复制有效用户ID而得来的。如果设置了文件的设置用户ID位，则在exec根据文件的用户ID设置了进程的有效用户ID以后，就将这个副本保存起来。

下表列出了改变这三个用户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，前面讨论的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。

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man(l)程序，它用于显示联机手册页。man程序可被安装为设置用户ID或设置组ID程序，man程序文件的所有者及他所属的组通常是为man自身保留的用户或组。man程序可以被构造为读以及可能重写文件，通过配置文件（通常是/etc/man.config或/etc/manpath.config）或者使用命令行的选项选择读、写文件的位置。

man程序可能需要执行许多其他命令，以处理包含需显示手册页的文件。为了防止被欺骗运行错误的命令或重写错误的文件，man命令不得不在两种权限之间切换：运行man命令用户的权限，以及拥有man可执行文件用户的权限。下面列出了其工作步骤：

(1) man程序文件是由名为man的用户拥有的，并且其设置用户ID位已设置。当我们exec 此程序时，则关于用户ID得到

* 实际用户ID =我们的用户ID
* 有效用户ID =man
* 保存的设置用户ID =man

(2) man程序访问需要的配置文件和手册页。这些文件是由名为man的用户所拥有的，因为有效用户ID是man，所以可以访问这些文件。

(3) 在man代表我们运行任一命令之前，它调用setuid (getuid ())。因为我们不是超级用户进程，所以这仅仅会改变有效用户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
* 保存的设置用户ID =man (未改变)

现在，man进程是以我们的用户ID作为其有效用户ID而运行。这就意味着能访问的只有我们通常可以访问的，而没有额外的权限。它可以代表我们安全地执行任一过滤器程序(filter)。

(4) 当执行完过滤器操作后，man调用setuid (euid)，其中euid是用户man的数值用户ID （man调用geteuid，得到用户man的用户id，然后将其保存起来）。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 =man
* 保存的设置用户ID =man (未改变)

(5) 因为man程序的有效用户ID是man，所以现在它可对其文件进行操作。

以这种方式使用保存的设置用户ID，于是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而得到的额外特权。但是，进程在其运行的大部分时间内只具有普通的权限。如果进程不能在其结束时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权限（这可能会造成麻烦）。

下面来看一看如果在man运行时为我们生成一个shell进程（先fork，然后exec），这将发生什么？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没有额外权限。它不能存取man运行时设置成man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。

如果程序文件是设置用户ID为root，那么我们关于man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。要使上述实例按我们所说明的进行工作，只需setuid仅设置有效用户ID。

### 1. setreuid和setregid函数

LINUX/UNIX支持setreuid函数，其功能是交换实际用户ID和有效用户ID的值。

|  |
| --- |
| #include <unistd.h>  int setreuid (uid\_t ruid, uid\_t euid);  int setregid (gid\_t rgid, gid\_t egid);  两个函数返回值：若成功则返回0，若出错则返回-1 |

如若其中任一参数的值为-1，则表示相应的1D应当保持不变。

相关规则很简单：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权限，以后又可再次转换回设置用户ID所得到的额外权限。 POSIX.1引入了保存的设置用户ID特征后，其规则也相应加强，它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。

### 2. seteuid和setegid函数

LINUX/UNIX包含了两个函数seteuid和setegid。它们类似干setuid和setgid，但只更改有效用户ID和有效组ID。

|  |
| --- |
| #include <unistd.h>  int seteuid (uid\_t uid);  int setegid (gid\_t gid);  两个函数返回值：若成功则返回0，若出错则返回-1 |

一个非特权用户可将其有效用户ID设置为其实标用户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。这有别于setuid函数，它会更改所有三个用户。

下图给出了本节所述的修改三个不同用户ID的各个函数。

|  |
| --- |
|  |
| 设置不同用户ID的各个函数 |

### 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。附加组ID不受setgid、setregid或setegid函数的影响。

## 8.12解释器文件

所有现今的UNIX系统都支持解释器文件(interpreter file)。这种文件是文本文件，其起始行的形式是：

#! pathname [ optional-argument ]

感叹号和pathname之间的空格是可选的。最常见的解释器文件以下列行开始：

#!/bin/sh

pathname通常是绝对路径名，对它不进行什么特殊的处理（即不使用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函数的进程实标执行的并不是该解释器文件，而是该解释器文件第一行中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。

要知道很多系统对解释器文件的第一行有长度限制。这些限制包括#!、pathname、可选 参数、终止换行符以及空格数。

下面例子可帮助我们了解当被执行的文件是解释器文件时，内核如何处理exec函数的参数及该解释器文件第一行的可选参数。下面程序调用exec执行一个解释器文件。

|  |
| --- |
|  |

下面先显示要被执行的该解释器文件的内容（只有一行），接着是程序的运行结果。

|  |
| --- |
|  |

程序echoarg （解释器）回送每一个命令行参数，前面已经给出代码。注意，当内核exec该解释器(/home/<username>/bin/echoarg)时，argv[0]是该解释器的pathname，argv[l]是解释器文件中的可选参数，其余参数是pathname （/home/<username>/bin/testinterp），以及调用execl的第二个和第三个参数（myargl和myarg2）。调用execl时的argv[1]和argv[2]已右移了两个位置。注意，内核取execl调用中的pathname而非第一个参数（testinterp），因为一般而言，pathname含了比第一个参数更多的信息。

在解释器pathname后可跟随可选参数。如果一个解释器程序支持-f选项，那么在后经常使用的就是-f。例如，可以以下列方式执行awk(l)程序：

awk -f myfile

它告诉awk从文件myf ile中读awk程序。

在解释器文件中使用-f选项，可以写成

#!/bin/awk -f

（在此解释器文件中后随awk程序）

例如，下面程序给出了在/usr/local/bin/awkexample中的一个解释器文件。

|  |
| --- |
| #!/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，则可以用下列方式执行程序：

$ awkexample filel FILENAME2 f3

ARGV[0] = awk

ARGV[1] = filel

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 filel 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，所以要将解释器文件完整的路径名传送给解释器。当awk读解释器文件时，因为#是awk的注释字符，所以它会忽略第一行。

可以用下列命令验证上述命令行参数：

$ /bin/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/ear/bin/echoarg /bin/awk #暂时替换它

# suspend #用作业控制挂起超级用户shell

[1] + Stopped /bin/su

$ awkexample filel FILENAME2 f3

argv[0]: /bin/awk

argv[1]: -f

argv[2]: /usr/local/bin/awkexample

argv[3]: filel

argv[4]: FILENAME2

argv[5]: f3

$ fg #用作业控制恢复超级用户shell

/bin/su

# mv /bin/awk.save /bin/awk 恢复原先的程序

# exit #终止超级用户shell

在此示例中，解释器的-f选项是必需的。正如前述，它告诉awk在什么地方找到awk程序。如果从解释器文件中删除-f选项，则在试图运行该解释器文件时，通常输出一条出错消息。该出错消息的精确文本可能有所不同，这取决于解释器文件存放在何处，以及其余参数是否表示现有文件等。因为在这种情况下命令行参数是：

/bin/awk /usr/local/bin/awkexample filel 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（因为识别解释器文件的是内核）。由于下述理由，解释器文件是有用的：

(1) 有些程序是用某种语言编写的脚本，解释器文件可将这一事实隐藏起来。例如，为了执行上述程序，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，但是将其包装在一个shell脚本中：

awk ‘BEGIN {

for (i = 0； i < ARGC; i++)

printf "ARGV[%d] = %s\n", i, ARGV[i]

exit }’ $\*

这种解决方案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，它会调用fork、exec和wait。于是，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，使用解释器脚本，则可编写成：

#!/bin/csh

(在解释器文件中后接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，则上述方式无效。

## 8.13 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某一个文件中，则可使用time函数实现这一点。调用time得到当前日历时间，接着调用localtime 将日历时间转换为年、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");

一般来说LINUX/UNIX都支持system函数。

|  |
| --- |
| #include <stdlib.h>  int system (const char \* cmdstring); |

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，这一特征可以确定在一个给定的操作系统上是否支持system函数。在LINUX/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中设置了错误类型值。

(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)—样。

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

下面程序是system函数的一种实现。它对信号没有进行处理。后面章节中将修改此函数使其进行信号处理。

|  |
| --- |
|  |

shell的-c选项告诉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，将它们分成命令行参数。传递给shell的实际命令字符串可以包含任一有效的shell命令。例如，可以用<和>重定向输入和输出。

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缓冲区（这些缓冲区会在 fork中由父进程复制到子进程）在子进程中被冲洗。

下面程序对system的这种版本进行了测试，其中pr\_exi t函数定义在前面的程序中。

|  |
| --- |
|  |

程序的运行结果如下：

|  |
| --- |
|  |

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。

如果调用system的进程在调用它之前已经生成它自己的子进程，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，如果不是用pid标识的任一子进程在pid子进程之前终止，则它们的进程ID和终止状态都会被while语句丢弃。实际上，由于wait不能等待一个指定的进程以及其他一些原因，所以才有了 waitpid函数。如果不提供waitpid函数，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。下面是一个简单程序，它只是对其命令行参数调用system函数。

|  |
| --- |
|  |

将此程序编译成可执行文件tsys。

接下来是另一个简单程序，它打印其实际和有效用户ID。

|  |
| --- |
|  |

将此程序编译成可执行文件printuids。运行这两个程序，得到下列结果：

|  |
| --- |
|  |

我们给予tsys程序的超级用户权限在system中执行了fork和exec之后仍会保持下来。

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

## 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。）系统通常记录用户登录时使用的名字，用getlogin函数可以获取此登录名。

|  |
| --- |
| #include <unistd.h>  char \*getlogin(void);  返回值：若成功则返回指向登录名字符串的指针，若出错则返回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。通常称这些进程为守护进程(daemon)，后面章节将讨论它们。

给出了登录名，就可用getpwnam在口令文件中査找用户的相应记录，从而确定其登录shell等。

## 8.15进程时间

LINUX/UNIX中我们可以测量的三种时间：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已终止子进程的上述值。

|  |
| --- |
| #include <sys/times.h>  clock\_t times (struct tms \*buf);  返回值：若成功则返回流逝的墙上时钟时间（单位：时钟滴答数），若出错则返回-1 |

此函数填写由buf指向的tms结构，其定义如下：

struct tms {

clock\_t tms\_ut ime; // 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的任何测量值。作为替代，times函数返回墙上时钟时间作为其函数值。此值是相对于过去的某一时刻测量的，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，从新的返回值中减去以前的返回值，此差值就是墙上时钟时间。（一个长期运行的进程可能会使墙上时钟时间溢出，当然这种可能性极小。

该结构中两个针对子进程的字段包含了此进程用wait、waitid或waitpid已等待到的各个子进程的值。

下面程序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。

|  |
| --- |
|  |

在这两个实例中，子进程中显示的所有CPU时间都是执行shell和命令的子进程所使用的CPU时间。

## 8.16 小结

对于LINUX和境中的高级编程而言，完整地了解UNIX的进程控制是非常重要的。其中必须熟练掌握的只有几个函数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争条件的机会。

本章说明了system函数，这也使我们能进一步了解所有这些进程控制函数。

本章还说明了exec函数的另一种变体：解释器文件及其工作方式。理解各种不同的用户ID和组ID（实际、有效和保存的），对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一个进程和其他进程的关系——会话和作业控制。

# 第9章 进程关系

## 9.1引言

在上一章我们已了解到进程之间具有关系。首先，每个进程都有一个父进程（初始的内核进程并无父进程，也可以说其父进程就是它自己）。当子进程终止时，父进程得到通知并能取得子进程的退出状态。在前一章讨论waitpid函数时，我们也提到了进程组，以及如何等待进程组中的任意一个进程终止。

本章将更详细地说明进程组以及会话的概念。还将介绍登录shell （登录时所调用的）和所有从登录shell启动的进程之间的关系。

## 9.2终端登录

先说明当我们登录到LINUX/UNIX系统时所执行的各个程序。在早期的LINUX/UNIX系统中， 用户用哑终端（用硬连接连到主机）进行登录。终端要么是本地的（直接连接）要么是远程的（通过调制解调器连接）。在这两种情况下，登录都经由内核中的终端设备驱动程序。因为连到主机上的终端设备数已经确定，所以同时的登录数也就有了已知的上限。

随着位映射图形终端变成可用，开发出了窗口系统，它向用户提供了与主机系统进行交互的新方式。创建“终端窗口”的应用程序也被开发出来，它仿真了基于字符的终端，使得用户可以用熟悉的方式（即通过shell命令行）与主机交互。

现今，某些平台允许用户在登录后启动一个窗口系统，而另一些平台则自动为用户启动窗口系统。在后一种情况中，用户可能仍然需要登录，这取决于窗口系统是如何配置的（某些窗口系统可配置成自动登录用户）。

我们现在说明的过程用于经由终端登录至LINUX/UNIX系统。该过程是类似的，而与所使用的终端无关，终端可以是基于字符的终端、仿真简单的基于字符终端的图形终端，或者是运行窗口系统的图形终端。

### 1. BSD终端登录

在过去30年中，登录过程并没有多少改变。系统管理员创建通常名为/etc/ttys的文件，其中，每个终端设备都有一行，每一行说明设备名和传递给getty程序的参数，例如，参数之一说明了终端的波特率等。当系统自举时，内核创建进程ID为1的进程，也就是init进程。init进程使系统进入多用户状态。init进程读文件/etc/ttys，对每一个允许登录的终端设备，init调用一次fork，它所生成的子进程则执行（exec）getty程序。下图显示了这些进程。

|  |
| --- |
|  |
| init为允许终端登录而调用的进程 |

上图中各个进程的实际用户ID和有效用户ID都是0 （即它们都具有超级用户特权）。init 以空环境执行getty程序。

getty为终端设备调用open函数，以读、写方式将终端打开。如果设备是调制解调器， 则open可能会在设备驱动程序中滞留，直到用户拨号调制解调器，并且呼叫被应答。一旦设备被打开，则文件描述符0、1、2就被设置到该设备。然后getty输出“login:”之类的信息，并等待用户键入用户名。如果终端支持多种速度，则getty可以测试特殊字符以便适当地更改终端速度（波特率）。

当用户键入了用户名后，getty的工作就完成了。然后它以类似于下面的方式调用login程序：

execle ("/bin/login", "login", "-p", username, (char \*) 0, envp);

（在gettytab文件中可能会有一些选项使其调用其他程序，但系统默认的是login程序）。 init以一个空环境调用getty。getty以终端名（例如TERM=foo，其中终端foo的类型取自gettytab文件）和在gettytab中说明的环境字符串为login创建一个环境（envp参数）。 -p标志通知login保留传给它的环境，也可将其他环境字符串加到该环境中，但是不要替换它。下图显示了login刚被调用后这些进程的状态。

|  |
| --- |
|  |
| 调用login后的进程状态 |

因为最初的init进程具有超级用户特权，所以上图中的所有进程都有超级用户特权。上图中底部三个进程的进程ID相同，因为进程ID不会因执行exec而改变。并且，除了最初的init进程，所有进程的父进程ID均为1。

login能执行多项工作。因为它得到了用户名，所以能调用getpwnam取得相应用户的口令文件登录项。然后调用getpass(3)以显示提示“Password:”，接着读用户键入的口令（自然，禁止回送用户键入的口令）。它调用crypt(3)将用户键入的口令加密，并与该用户在阴影口令文件中登录项的pw\_passwd字段相比较。如果用户几次键入的口令都无效，则login 以参数1调用exit表示登录过程失败。父进程（init)）了解到子进程的终止情况后，将再次调用fork，其后接着执行getty，对此终端重复上述过程。

这是UNIX系统传统的用户身份验证过程。现代UNIX系统已发展到支持多个身份验证过程。例如，FreeBSD、Linux都支持被称为PAM（Pluggable Authentication Module，可插入式身份验证模块）的更加灵活的方案。PAM允许管理员配置使用何种身份验证方法来访问那些使用PAM库编写成的服务。

如果应用程序需验证一用户是否具有适当的权限去执行某个服务，那么我们可以将身份验证机制编写到应用中，或者使用PAM库来得到等价的功能。使用PAM的优点是，管理员可以基于本地策略、针对不同任务配置不同的验证用户身份的方法。

如果用户正确登录，login就将执行如下工作：

* 将当前工作目录更改为该用户的起始目录(chdir)。
* 调用chown改变该终端的所有权，使登录用户成为它的所有者。
* 将对该终端设备的访问权限改变成用户读和写。
* 调用setgid及initgroups设置进程的组ID。
* 用login所得到的所有信息初始化环境：起始目录（HOME）、shell （SHELL）、用户名（USER和LOGNAME），以及一个系统默认路径（PATH)。
* login进程改变为登录用户的用户ID （setuid）并调用该用户的登录shell，如下：

exec1("/bin/sh", "-sh", (char \*)0);

argv[0]的第一个字符是一个标志，表示该shell被调用为登录shell。shell可以查看此字符，并相应地修改其启动过程。

login程序实际所做的比上面说的要多。它可选择打印日期消息（message-of-the-day）文件，检査新邮件以及执行其他一些任务。但是我们主要关心上面所说的功能。回忆上一章中对setuid函数的讨论，因为setuid是由超级用户调用的，它更改所有三个用户ID：实际用户ID、有效用户ID和保存的用户ID。login在较早时间调用的setgid对所有三个组ID也有同样效果。

到此为止，登录用户的登录shell开始运行。其父进程ID是init进程ID（进程ID 1），所以当此登录shell终止时，init会得到通知（接到SIGCHLD信号），它会对该终端重复全部上述过程。

将登录shell的文件描述符0、1和2设置为终端设备。下图显示了这种安排。

|  |
| --- |
|  |
| 为终端登录完成各种设置后的进程安排 |

### 2. Linux终端登录

Linux的终端登录过程非常类似于BSD。确实，Linux login命令是从4.3BSD login命令派生而来的。BSD登录过程与LINUX登录过程的主要区别在于说明终端配置的方式。

在Linux中，/etc/inittab包含配置信息，它说明了init应当为之启动getty进程的各终端 设备。依赖于所使用的getty的版本，终端的各种特性要么在命令行上说明（如同使用agetty—样），要么在文件/etc/gettyefs中说明（如同使用mgetty—样）。

## 9.3网络登录

通过串行终端登录至系统和经由网络登录至系统两者之间的主要（物理上的）区别是：通过网络登录时，终端和计算机之间的连接不是点对点连接。在这种情况下，login只是一种可用的服务，这与其他网络服务（例如FTP或SMTP）的性质相同。

在上一节所述的终端登录中，init知道哪些终端设备可用来进行登录，并为每个设备生成一个getty进程。但是，在网络登录情况下，所有登录都经由内核的网络接口驱动程序（如 以太网驱动程序），事先并不知道将会有多少这样的登录。我们不是使一个进程等待每个可能的登录，而是必须等待一个网络连接请求的到达。

为使同一个软件既能处理终端login，又能处理网络login，系统使用了一种称为伪终端 （pseudo terminal）的软件驱动程序，它仿真串行终端的运行行为，并将终端操作映射为网络操作，反之亦然。

### 1. BSD网络登录

在BSD中，有一个称为inetd的进程（有时称之为因特网超级服务器），它等待大多数网络连接。

作为系统启动的一部分，init调用一个shell，使其执行shell脚本/etc/rc。由此shell脚本启动一个守护进程inetd。一旦此shell脚本终止，inetd的父进程就变成init。inetd等待TCP/IP连接请求到达主机，而当一个连接请求到达时，它执行一次fork，然后生成的子进程执行适当的程序。

我们假定到达了一个针对TELNET服务进程的TCP连接请求。TELNET是使用TCP协议的远程登录应用程序。在另一台主机（它通过某种形式的网络与服务进程的主机相连接）上的用户，或在同一台主机上的用户启动TELNET客户进程，由此启动登录过程：

telnet hostname

该客户进程打开一个到hostname主机的TCP连接，在hostname主机上启动的程序被称为 TELNET服务进程。然后，客户进程和服务进程之间使用TELNET应用协议通过TCP连接交换数据。所发生的是启动客户进程的用户现在登录到了服务进程所在的主机。（自然，用户需要在服务进程主机上有一个有效的账号）。下图显示了在执行TELNET服务进程（称为telnetd）时所涉及的进程序列。

|  |
| --- |
|  |
| 执行TELNET服务进程时涉及的进程序列 |

然后，telnetd进程打开一个伪终端设备，并用fork分成两个进程。父进程处理通过网络连接的通信，子进程则执行login程序。父、子进程通过伪终端相连接。在调用exec之前，子进程使其文件描述符0、1、2与伪终端相连。如果登录正确，login就执行上节中所述的同样步骤：更改当前工作目录为起始目录，设置登录用户的组ID和用户ID，以及登录用户的初始环境。然后login调用exec将其自身替换为登录用户的登录shell。下图显示了此时的进程安排。

|  |
| --- |
|  |
| 为网络登录完成各种设置后的进程安排 |

需要理解的重点是：当通过终端或网络登录时，我们得到一个登录shell，其标准输入、输出和标准出错连接到一个终端设备或者伪终端设备上。在下一节中我们会了解到这一登录shell是一个会话的开始，而此终端或伪终端则是会话的控制终端。

### 2. Linux网络登录

除了使用扩展的因特网服务守护进程xinetd代替inetd进程外，Linux网络登录的其他方面与BSD相同。xinetd进程对它所启动的各种服务的控制比inetd提供的更加精细。

## 9.4进程组

每个进程除了有一个进程ID之外，还属于一个进程组，下一张章讨论信号时还会涉及进程组。进程组是一个或多个进程的集合。通常，它们与同一作业相关联（后面会细讨论作业控制），可以接收来自同一终端的各种信号。每个进程组有一个唯一的进程组ID。进程组ID类似于进程ID——它是一个正整数，并可存放在pid\_t数据类型中。函数getpgrp返回调用进程的进程组ID。

|  |
| --- |
| #include <unistd.h>  pid\_t getpgrp (void);  返回值：调用进程的进程组0 |

另外一个函数也可以获得进程组的ID。

|  |
| --- |
| #include <unistd.h>  pid\_t getpgid (pid\_t pid);  返回值：若成功则返回进程组ID，若出错则返回-1 |

若为0，则返回调用进程的进程组ID，于是：

getpgid(0);

等价于

getpgrp()；

每个进程组都可以有一个组长进程。组长进程的标识是，其进程组ID等于其进程ID。组长进程可以创建一个进程组，创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生存期。进程组中的最后一个进程可以终止，或者转移到另一个进程组。

进程可以通过调用setpgid来加入一个现有的组或者创建一个新进程组（下一节中将说明用setsid也可以创建一个新的进程组）。

|  |
| --- |
| #include <unistd.h>  int setpgid (pid\_t pid, pid\_t pgid);  返回值：若成功则返回0，若出错则返回-1 |

setpgid函数将pid进程的进程组ID设置为pid。如果这两个参数相等，则由pid指定的进程变成进程组组长。如果是0，则使用调用者的进程ID。另外，如果pgid是0，则由pid指定的进程ID将用作进程组ID。

一个进程只能为它自己或它的子进程设置进程组ID。在它的子进程调用了exec函数之一后，它就不再能改变该子进程的进程组ID。

在大多数作业控制shell中，在fork之后调用此函数，使父进程设置其子进程的进程组ID，并且使子进程设置其自己的进程组ID。这两个调用中有一个是冗余的，但让父子进程都这么做可以保证，在父、子进程认为子进程已进入了该进程组时，这确实已经发生了。如果不这样做的话，那么fork之后，由于父、子进程运行先后次序的不确定，会造成在一段时间内（父、子 进程中只运行了其中一个）子进程组员身份的不确定（取决于哪个进程首先执行），这就产生了竞争条件。

## 9.5会话

会话(session)是一个或多个进程组的集合。例如，可以具有下图所示的安排。其中，在一个会话中有三个进程组。

|  |
| --- |
|  |
| 进程组和会话中的进程安排 |

通常是由shell的管道几个进程编成一组的。例如，上图中的安排可能是由下列形式的shell命令形成的：

procl | proc2 &

proc3 | proc4 丨 proc5

进程调用setsid函数建立一个新会话。

|  |
| --- |
| #include <unistd.h>  pid\_t setsid(void)；  返回值：若成功则返回进程组ID，若出错则返回-1 |

如果调用此函数的进程不是一个进程组的组长，则此函数就会创建一个新会话，结果将发生下面3件事：

(1) 该进程变成新会话首进程（session leader）。会话首进程是创建该会话的进程。此时，该进程是新会话中唯一的进程。

(2) 该进程成为一个新进程组的组长进程。新进程组ID是该调用进程的进程ID。

(3) 该进程没有控制终端（下一节将讨论控制终端）。如果在调用setsid之前该进程有一个控制终端，那么这种联系也会被中断。

如果该调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不会发生这种情况，通常先调用fork，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组ID，而其进程ID则是新分配的，两者不可能相等，所以这就保证子进程不会是一个进程组的组长。

显然，会话首进程是具有唯一进程ID的单个进程，所以可以将会话首进程的进程ID视为会话ID。getsid函数返回会话首进程的进程组ID。

|  |
| --- |
| #include <unistd.h> pid\_t getsid (pid\_t pid);  返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1 |

如若pid是0，getsid返回调用进程的会话首进程的进程组ID。出于安全方面的考虑，某些实现会有如下限制：如若pid并不属于调用者所在的会话，那么调用者就不能得到该会话首进程的进程组ID。

## 9.6控制终端

会话和进程组有一些其他特性：

* 一个会话可以有一个控制终端（controlling terminal）。这通常是登录到其上的终端设备（在终端登录情况下）或伪终端设备（在网络登录情况下）。
* 建立与控制终端连接的会话首进程被称为控制进程（controlling process）。
* 一个会话中的几个进程组可被分成一个前台进程组（foreground process group）以及一个或几个后台进程组（background process group）。
* 如果一个会话有一个控制终端，则它有一个前台进程组，会话中的其他进程组则为后台进程组。
* 无论何时键入终端的中断键（常常是DELETE或Ctrl+C），就会将中断信号发送给前台进程组的所有进程。
* 无论何时键入终端的退出键（常常是Ctrl+\），就会将退出信号发送给前台进程组中的所有进程。
* 如果终端接口检测到调制解调器（或网络）已经断开连接，则将挂断信号发送给控制进程（会话首进程）。

这些特性示于下图中。

|  |
| --- |
|  |
| 显不控制终端的进程组和会话 |

通常，我们不必关心控制终端，登录时，将自动建立控制终端。

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

经典的示例是用于读口令的getpass(3)函数（当然，终端回送被关闭）。这一函数由crypt(l)程序调用，而此程序则可用于管道中。例如：

crypt < salaries | lpr

将文件salaries解密，然后经由管道将输出送至打印假脱机程序。因为crypt从其标准输入读输入文件，所以标准输入不能用于输入口令。但是，crypt的一个设计特征是每次运行此程序时，都应输入加密口令，这样也就阻止了用户将口令存放在文件中（这可能是一个安全性漏洞）。

## 9.7 tcgetpgrp、tcsetpgrp和tcgetsid函数

需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处见上图。

|  |
| --- |
| #include <umstd.h>  pid\_t tcgetpgrp (int filedes);  返回值：若成功则返回前台进程组的进程组ID，若出错则返回-1  int tcsetpgrp (int filedes, pid\_t pgrpid);  返回值：若成功则返回0，若出错则返回-1 |

函数tcgetpgrp返回前台进程组的进程组ID，该前台进程组与在filedes上打开的终端相关联。

如果进程有一个控制终端，则该进程可以调用tcsetpgrp将前台进程组ID设置为pgrpid。pgrpid的值应当是在同一会话中的一个进程组的ID。必须引用该会话的控制终端。

大多数应用程序并不直接调用这两个函数。它们通常由作业控制shell调用。

tcgetsid用来获得获得会话首进程的进程组ID。

|  |
| --- |
| #include <termios.h>  pid\_t tcgetsid (int filedes);  返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1 |

需要管理控制终端的应用程序可以调用tcgetsid函数识别出控制终端的会话首进程的会 话ID （它等价于会话首进程的进程组ID）。

## 9.8作业控制

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

(1) 支持作业控制的shell。

(2) 内核中的终端驱动程序必须支持作业控制。

(3) 内核必须提供对某些作业控制信号的支持。

从shell使用作业控制功能角度讲，用户可以在前台或后台启动一个作业。一个作业只是几个进程的集合，通常是一个进程的管道。例如：

vi main.c

在前台启动了只有一个进程组成的一个作业。命令

pr \*.c | lpr &

make all &

在后台启动了两个作业。这两个后台作业调用的所有进程都在后台运行。

当启动一个后台作业时，shell赋予它一个作业标识，并打印一个或几个进程ID。下面的脚本显示了bash是如何处理这一点的：

|  |
| --- |
|  |

make是作业号1，启动的进程ID是1475。下一个管道线是作业号2，其第一个进程的进程ID是1490。当作业已完成而且我们键入回车时，shell通知我们作业已经完成。键入回车是为了让shell打印其提示符。shell并不在任何随意时刻打印后台作业的状态改变——它只在打印其提示符让用户输入新的命令行之前才这样做。如果不这样处理，则当我们正输入一行时，它也可能输出。

我们可以键入一个影响前台作业的特殊字符——挂起键（一般采用Ctrl+Z）与终端驱动程序进行交互。键入此字符使终端驱动程序将信号SIGTSTP送至前台进程组中的所有进程，后台进程组作业则不受影响。实际上有下面三个特殊字符可使终端驱动程序产生信号，并将它们送至前台进程组：

* 中断字符（一般采用DELETE或Ctrl+C）产生SIGINT。
* 退出字符（一般采用Ctri+\）产生SIGQUIT。
* 挂起字符（一般采用Ctrl+Z）产生SIGTSTP。

终端驱动程序必须处理与作业控制有关的另一种情况。我们可以有一个前台作业和若干个后台作业，这些作业中哪一个接收我们在终端上键入的字符呢？只有前台作业接收终端输入。如果后台作业试图读终端，那么这并不是一个错误，但是终端驱动程序将检测这种情况，并且向后台作业发送一个特定信号SIGTTIN。该信号通常会暂时停止此后台作业，而shell则向有关用户发出这种情况的通知，然后用户就可用shell命令将此作业转为前台作业运行，于是它就可读终端。下列操作过程演示了这一点：

|  |
| --- |
| 在后台启动，但将从标准输入读  $ cat > temp.foo &  [1] 1681  $  [1] + Stopped (SIGTTIN)  $ fg %1  cat > temp.foo  hello, world  "D  $ cat temp.foo  hello, world  键入回车  cat > temp.foo&  使i号作业成为前台作业  shell告诉我们现在哪一个作业在前台  输入1行  键入文件结束符  检査该行已送入文件 |

shell在后台启动cat进程，但是当cat试图读其标准输入（控制终端）时，终端驱动程序知道它是个后台作业，于是将SIGTTIN信号送至该后台作业。shell检测到其子进程的状态改变（前面章节中中对wait和waitpid的讨论），并通知我们该作业已被停止。然后，我们用shell的fg命令将此停止的作业送入前台运行（关于作业控制命令，例如fg和bg的详细情况，以及标识不同作业的各种方法，请参阅有关shell的手册页）。这样做可以使shell将此作业置入前台进程组（tcsetpgrp），并将继续信号（SIGCONT）送给该进程组。因为该作业现在位于前台进程组中，所以它可以读控制终端。

如果后台作业输出到控制终端又将发生什么呢？这是一个我们可以允许或禁止的选项。通常，可以用stty(l)命令改变这一选项。下面显示了这种工作方式：

|  |
| --- |
| $ cat temp.foo &  [1] 1719  $ hello, world  [1] + Done cat  $ stty tostop $ cat tamp.foo &  [1] 1721  $  [1] + Stopped(SIGTTOU) $ fg %X cat temp.foo hello, world  在后台执行  提示符后出现后台作业的输出 键入回车 temp.foo &  禁止后台作业输出至控制终端 在后台再试一次  键入回车，发现作业已停止 cat temp.foo &  在前台恢复停止的作业  shell告诉我们现在哪一个作业在前台  这是该作业的输出 |

在用户禁止后台作业写到控制终端时，cat命令将阻止该作业试图写到其标准输出，因为终端驱动程序将该写操作标识为来自于后台进程，于是向该作业发送SIGTTOU信号。与上面的例子一样，当用户使用shell的fg命令将该作业带至前台时，该作业继续执行直至完成。

下图总结了前面已说明的作业控制的某些功能。

|  |
| --- |
|  |
| 具有前台、后台作业以及终端驱动程序的作业控制功能总结 |

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

是否需要作业控制是一个有争议的话题。作业控制是在窗口终端广泛得到应用之前设计和实现的。很多人认为设计得好的窗口系统已经免除了对作业控制的需要。某些人抱怨作业控制的实现要求得到内核、终端驱动程序、shell以及某些应用程序的支持，是吃力不讨好的事情。某些人在窗口系统中使用作业控制，

## 9.9 shell执行程序

让我们研究下shell是如何执行程序的，以及这与进程组、控制终端和会话等概念的关系。为此，再次使用ps命令。

首先使用不支持作业控制的经典Bourne shell。如果执行

ps -o pid,ppid,pgid,sid,comm

则其输出为

|  |
| --- |
| PID PPID PGID SID COMMAND 949 947 949 949 sh 1774 949 949 94 9 ps |

ps命令的父进程是shell，这正是我们所期望的。shell和ps命令两者位于同一会话和前台进程组（949）中。因为我们使用一个不支持作业控制的shell执行命令时得到该值的，所以称949为前台进程组。

某些平台支持一个选项，它使ps(l)命令打印与会话控制终端相关联的进程组ID。该值显示在 TPGID列中。不幸的是，ps命令的输出在各个UNIX版本之间有所不同。例如，Solaris 9不支持该选项。

如果在后台执行该命令：

ps -o pid,ppid,pgid,sid,comm &

则唯一改变的值是命令的进程ID。

|  |
| --- |
| PID PPID PGID SID COMMAND 949 947 949 949 sh 1812 949 949 949 ps |

因为这种shell不知道作业控制，所以后台作业没有构成另一个进程组，也没有从后台作业处取走控制终端。

现在看一看Bourne shell如何处理管道。执行命令

ps -o pid,ppid,pgid,sid,comm | catl

时其输出是

|  |
| --- |
| PID PPID PGID SID COMMAND  949 947 949 949 sh  1823 949 949 949 catl  1824 1823 949 949 ps |

程序catl只是标准cat程序的一个副本，但名字不同。本节还将使用cat的另一个名为cat2的副本。在一个管道线中使用两个cat副本时，不同的名字可使我们将它们区分开来。注意，管道中的最后一个进程是shell的子进程，该管道线中的第一个进程则是最后一个进程的子进程。从中可以看出，shell fork一个它自身的副本，然后此副本再为管道线中的每条命令各fork一个进程。

如果在后台执行此管道：

ps -o pid,ppid,pgid,sid,comm | catl &

则只有进程ID改变了。因为shell并不处理作业控制，后台进程的进程组ID仍是949，如同会话的进程组ID—样。

如果一个后台进程试图读其控制终端，则会发生什么呢？例如，若执行：

cat > temp.foo &

在有作业控制时，后台作业被放在后台进程组中，如果后台作业试图读控制终端，则会产生信号SIGTTIN。在没有作业控制时，其处理方法是：如果该进程自己没有重定向标准输入，则shell自动将后台进程的标准输入重定向到/dev/null。读/dev/null则产生一个文件结束。这就意味着后台cat进程立即读到文件尾，并正常结束。

上面说明了对后台进程通过其标准输入访问控制终端的适当的处理方法，但是，如果一个后台进程打开/dev/tty并且读该控制终端，又将怎样呢？对此问题的回答是“看情况”。但 是这很可能不是我们所期望的。例如：

crypt < salaries | lpr &

就是这样一条管道。我们在后台运行它，但是crypt程序打开/dev/tty，更改终端的特性（禁止回送），然后读该设备，最后重置该终端特性。当执行这条后台管道时，crypt在终端上打印提示符“Password:”，但是shell读取了我们所输人的加密口令，并试图执行以加密口令为名称的命令。我们输送给shell的下一行则被crypt进程取为口令行，于是不能正确对文件加密，结果将一堆无用信息送到了打印机。在这里，我们有了两个进程，它们试图同时读同一设备，其结果则依赖于系统。前面说明的作业控制以较好的方式处理一个终端在多个进程间的复用。

返回到shell实例，如果在一条管道线中执行三个进程，我们就可以检验shell 使用的进程控制方式：

ps -o pid,ppid,pgid,sid,comm | catl | cat2

其输出为

|  |
| --- |
| PID PPID PGID SID COMMAND  949 947 949 949 sh  1988 949 949 949 cat2  1989 1988 949 949 ps  1990 1988 949 949 catl |

如果在你的系统上，输出的命令名不正确，输出：

|  |
| --- |
| PID PPID PGID SID COMMAND  949 947 949 949 sh  1831 949 949 949 sh  1832 1831 949 949 ps  1833 1831 949 949 sh |

也是正常的。造成此种结果的原因是，ps进程与shell产生竞争条件，shell创建一个子进程并由它执行cat命令。在这 种情况下，当ps已经获得进程列表并打印时，shell尚未完成exec调用。

再重申一遍，该管道中的最后一个进程是shell的子进程，而执行管道中其他命令的进 程则是该最后一个进程的子进程。下图显示了所发生的情况。

|  |
| --- |
|  |
| shell调用管道线ps I catl I cat2时的进程 |

现在让我们用一个运行在Linux上的作业控制shell来检验同一个例子。这将显示这些shell处 理后台作业的方法。在本例中将使用Boume-again shell，用其他作业控制shell得到的结果几乎完全一样。

ps -o pid, ppid, pgrp, session, tpgid, corrrm

其输出为

|  |
| --- |
| PID PPID PGRP SESS TPGID COMMAND  2837 2818 2837 2837 5796 bash  5796 2837 5796 2837 5796 ps |

我们立即看到了与Bourne shell例子的区别。Boume-again shell将前台作业(ps)放入它自己的进程组(5796)中。ps命令是组长进程，并是该进 程组中的唯一进程。

进一步讲，此进程组具有控制终端，所以它是前台进程组。我们的登录shell在执行ps命令时是后台进程组。但需要注意的是，这两个进程组2837和5796都是同一会话的成员。事实上，在本节的各实例中会话决不会改变。

在后台执行此进程：

ps -o pid,ppid,pgrp,session,tpgid,comm &

其输出为

|  |
| --- |
| PID PPID PGRP SESS TPGID COMMAND  2837 2818 2837 2837 2837 bash  5797 2837 5797 2837 2837 ps |

再一次，PS命令被放入它自己的进程组中，但是此时进程组(5797)不再是前台进程组，而是 一个后台进程组。TPGID 2837指示前台进程组是用户的登录shell。

按下列方式在一个管道中执行两个进程：

ps -o pid,ppid,pgrp,session,tpgid,comm | catl

其输出为：

|  |
| --- |
| PID PPID PGRP SESS TPGID COMMAND  2837 2818 2837 2837 5799 bash  5799 2837 5799 2837 5799 ps  5800 2837 5799 2837 5799 catl |

两个进程ps和catl都在一个新进程组(5799)中，这是一个前台进程组。在本例和类似的Bourne shell实例之间能看到另一个区别。Bourne shell首先创建将执行管道线中最后一条命令的 进程，而此进程是第一个进程的父进程。在这里，Bourae-again Shell是两个进程的父进程。但 是，如果在后台执行此管道线：

ps -o pid,ppid#pgrp,session,tpgid,comm | catl &

其结果是类似的，但是ps和catl现在都处于同一后台进程组中。

|  |
| --- |
| PGRP  2837  5801  5801  PPXD  2818  2837  2837 |

注意，如果使用的Shell不同，那么它创建各个进程的顺序也可能不同。

## 9.10孤儿进程组

我们曾提及，一个其父进程已终止的进程称为孤儿进程（orphan process），这种进程由 init进程“收养”。现在我们要说明整个进程组也可成为“孤儿”，以及LINUX/UNIX如何处理它。

考虑一个进程，它fork了一个子进程然后终止。这在系统中是经常发生的，并无异常之处，但是在父进程终止时，如果该子进程停止（用作业控制），则会发生什么情况呢？子进程如何继续，以及子进程是否知道它已经是孤儿进程？下图显示了这种情形：

|  |
| --- |
|  |
| 将要成为孤儿的进程组实例 |

父进程已经fork了子进程，该子进程停止，父进程则将退出。下面的程序就会产生这种情形。

|  |
| --- |
| 程序清单9-1创建一个孤儿进程组  #include "apue.h"  #include <errno.h>  static void s ig\_hup(int signo)  { \_  printf("SIGHUP received, pid = %d\n", getpid())?  }  static void  pr 一ids(char \*name)  { \_  printf("%s： pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",  name, getpidO , getppidO • getpgrp () , tcgetpgrp (STDIN\_FXLENO))； fflush(stdout)/  int  main(void)  {  char c； pid一t pid;  pr—ids("parent")； if一((pid = fork() ) < 0) { err\_sys("fork error")；  •} else if (pid > 0) { /\* parent \*/  sleep(5)； /\* sleep to let child stop itself \*/  exit(0); /\* then parent exits \*/  } else { /\* child \*/  pr一ids("child");  signal(SIGHUP, sig—hup); /\* establish signal handler ★/ kill(getpid(), SIGTSTP); /\* stop ourself \*/ pr\_ids("child"); /\* prints only if we're continued ★/ if (read(STDIN-FILENO, &c# 1) != 1)  printf("read error from controlling TTY, errno = %d\n", errno)；  exit(0)； |

下面要说明该程序的某些新特征。这里，假定使用了一个作业控制shell。回忆前面所述，shell将前台进程放在它（指前台进程）自已的进程组（本例中是6099）中，shell则留在自己的进程组（2837）内。子进程继承其父进程（6099）的进程组。在fork之后：

* 父进程休眠5秒钟，这是一种让子进程在父进程终止之前运行的一种权宜之计。
* 子进程为挂断信号（SIGHUP）建立信号处理程序。这样就能观察到SIGHUP信号是否已发送到子进程。
* 子进程用kill函数向其自身发送停止信号（SIGTSTP）。这停止了子进程，类似于用终端挂起字符（Ctrl+Z）停止一个前台作业。
* 当父进程终止时，该子进程成为孤儿进程，所以其父进程ID成为1，也就是init进程ID。
* 现在，子进程成为一个孤儿进程组的成员。POSIX.1将孤儿进程组（orphaned process group）定义为：该组中每个成员的父进程要么是该组的一个成员，要么不是该组所属会话的成员。对孤儿进程组的另一种描述如下：一个进程组不是孤儿进程组的条件是，该组中有一个进程，其父进程在属于同一会话的另一个组中。如果进程组不是孤儿进程组，那么在属于同一会话的另一个组中的父进程就有机会重新启动该组中停止的进程。在这里，进程组中每一个进程的父进程（例如，进程6100的父进程是进程1）都属于另一个会 话。所以此进程组是孤儿进程组。
* 因为在父进程终止后，进程组成为孤儿进程组，POSIX.1要求向新的孤儿进程组中处于停止状态的每一个进程发送挂断信号（SIGHUP）接着又向其发送继续信号（SIGCONT）。
* 在处理了挂断信号后，子进程继续。对挂断信号的系统默认动作是终止该进程，为此必须提供一个信号处理程序以捕捉该信号。因此，我们期望sig\_hup函数中的printf会在pr\_ids函数中的printf之前执行。

下面程序输出：

|  |
| --- |
| $ ./a.out  parent： pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099 child： pid = 6100, ppid = 6099, pgrp - 6099, tpgrp = 6099 $ SIGHUP received, pid = 6100  child： pid = 6100, ppid - 1, pgrp = 6099, tpgrp = 2837 read error from controlling TTY, errno = 5 |

注意，因为两个进程即登录shell和子进程都写向终端，所以shell提示符和子进程的输出一起出现。正如我们所期望的那样，子进程的父进程ID变成1。

在子进程中调用pr\_ids后，程序试图读标准输入。正如前述，当后台进程组试图读控制终端时，则对该后台进程组产生SIGTTIN。但在这里，这是一个孤儿进程组，如果内核用此信号停止它，则此进程组中的进程就再也不会继续。POSIX.1规定，在这种情形下read返回出错，并将其errno设置为EIO （在作者所用的系统中其值是5）。

最后，要注意的是父进程终止时，子进程被置入后台进程组中，因为父进程是由shell作为前台作业执行的。

## 9.11 FreeBSD 实现

上面说明了进程、进程组、会话和控制终端的各种属性，值得观察一下所有这些是如何实现的。下面简要说明FreeBSD的实现。

|  |
| --- |
|  |
| 会话和进程组的FreeBSD实现 |

下面说明上图中标出的各个字段。从session结构开始。每个会话都分配一个session 结构（例如，每次调用setsid时）。

* s\_count是该会话中的进程组数。当此计数器减至0时，则可释放此结构。
* s\_leader是指向会话首进程proc结构的指针。
* s\_ttyvp是指向控制终端vnode结构的指针。
* s\_ttyp是指向控制终端tty结构的指针。
* s\_sid是会话ID。

在调用sets id时，在内核中分配一个新的会话结构。现在s\_count设置为1， s\_leader设置为调用进程proc结构的指针，s\_sid设置为进程ID，因为新会话没有控制终端，所以s\_ttyvp和s\_ttyp设置为空指针。

接着说明tty结构。每个终端设备和每个伪终端设备均在内核中分配这样一种结构（第19 章将对伪终端作更多说明）。

* t\_session指向将此终端作为控制终端的session结构（注意，tty结构指向session 结构，session结构也指向tty结构）。终端在失去载波信号时使用此指针将挂断信号送给会话首进程。
* t\_pgrp指向前台进程组的pgrp结构。终端驱动程序用此字段将信号送至前台进程组。由输入特殊字符（中断、退出和挂起）而产生的三个信号被送至前台进程组。
* t\_termios是包含所有这些特殊字符以及与该终端有关信息（例如，波特率、回送打开 或关闭等）的结构。
* t\_winsize是包含终端窗口当前尺寸的winsize结构。当终端窗口尺寸改变时，信号SIGWINCH被送至前台进程组。

注意，为了找到特定会话的前台进程组，内核不得不从session结构开始，然后用s\_ttyp得到控制终端的tty结构，接着用t\_pgrp得到前台进程组的pgrp结构。pgrp结构包含一特定进程组的信息。其中各相关字段是：

* pg\_id是进程组ID。
* pg\_session指向此进程组所属会话的session结构。
* pg\_members是指向作为此进程组成员的p roc结构列表的指针。proc结构中的p\_ pglist结构是一个双向链表项，它同时指向此组中的下一个进程和前一个进程，依此类推，直到进程组中最后一个进程的proc结构中遇到一个空指针。proc结构包含一个进程的所有信息。
* p\_pid包含进程ID。
* p\_pptr是指向父进程proc结构的指针。
* p\_pgrp指向本进程所属进程组的pgrp结构的指针。
* 如前所述，p\_pglist是一个结构，其中包含两个指针，分别指向进程组中的前一个进程和下一个进程。

最后还有一个vnode结构。在打开控制终端设备时分配此结构。进程对/dev/tty的所有引用都通过vnode结构。

## 9.12 小结

本章说明了进程组之间的关系——会话，它由若干个进程组组成。作业控制是当今很多LINUX/UNIX系统所支持的功能，本章说明了它是如何由支持作业控制的shell实现的。在这些进程关系中也涉及到了进程的控制终端/dev/tty。

所有这些进程的关系都使用了很多信号方面的功能。下一章将详细讨论LINUX/UNIX中的信号机制。

# 第10章 信号

## 10.1引言

信号是软件中断。很多比较重要的应用程序都需处理信号。信号提供了一种处理异步事件的方法，例如，终端用户键入中断键，则会通过信号机制停止一个程序，或及早终止管道中的下一个程序。

LINUX/UNIX的早期版本就已经提供了信号机制，但是这些系统所提供的信号模型并 不可靠。信号可能丢失，而且在执行临界区代码时，进程很难关闭所选择的信号。后来POSIX.1对可靠信号例程进行了标准化，LINUX/UNIX对这方面进行了很多有益的改进。这正是本章所要说明的。

本章先对信号机制进行综述，并说明每种信号的一般用法。然后分析早期实现的问题。在分析存在的问题之后再说明解决这些问题的方法，这种安排有助于加深对改进机制的理解。本章也包含了很多并非完全正确的实例，这样做的目的是为了对其不足之处进行讨论。

## 10.2信号概念

首先，每个信号都有一个名字。这些名字都以三个字符SIG开头。例如，SIGABRT是夭折信号，当进程调用abort函数时产生这种信号。SIGALRM是闹钟信号，当由alarm函数设置的计时器超时后产生此信号。不同的LINUX/UNIX实现所拥有的信号总量也不同，但是这些信号都被定义为正整数（信号编号）。

不存在编号为0的信号。在后面将会看到，kill函数对信号编号0有特殊的应用。 POSIX.1将此种信号编号值称为空信号。

很多条件可以产生信号：

* 当用户按某些终端键时，引发终端产生的信号。在终端上按DELETE键（或者很多系统中的Ctrl+C键）通常产生中断信号(SIGINT)。这是停止一个已失去控制的程序的方法。
* 硬件异常产生信号：除数为0、无效的内存引用等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效内存引用的进程产生SIGSEGV信号。
* 进程调用kill（2）函数可将信号发送给另一个进程或进程组。自然，对此有所限制：接收信号进程和发送信号进程的所有者必须相同，或者发送信号进程的所有者必须是超级用户。
* 用户可用kill(l)命令将信号发送给其他进程。此命令只是kill函数的接口。常用此命令终止一个失控的后台进程。
* 当检测到某种软件条件已经发生，并应将其通知有关进程时也产生信号。这里指的不是硬件产生的条件（如除以0），而是软件条件。例如SIGURG（在网络连接上传来带外数据时产生）、SIGPIPE （在管道的读进程已终止后，一个进程写此管道时产生），以及 SIGALRM （进程所设置的闹钟时钟超时时产生）。

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

可以要求内核在某个信号出现时按照下列三种方式之一进行处理，我们称之为信号的处理或者与信号相关的动作。

(1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。 它们是SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供了使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法内存引用或除以0），则进程的运行行为是未定义的。

(2) 捕捉信号。为了做到这一点，要通知内核在某种信号发生时调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。例如，若正在运行一个命令解释器，它将用户的输入解释为命令并执行之，当用户用键盘产生中断信号时，很可能希望该命令解释器返回到主循环，终止正在为该用户执行的命令。如果捕捉到SIGCHLD信号，则表示一个子进程已经终止，所以此信号的捕捉函数可以调用waitpid以取得该子进程的进程ID以及它的终止状态。又例如，如果进程创建了临时文件，那么可能要为SIGTERM信号编写一个信号捕捉函数以清除临时文件（SIGTERM是终止信号，kill命令传送的系统默认信号是终止信号）。注意，不能捕 捉SIGKILL 和 SIGSTOP 信号。

(3) 执行系统默认动作。常用的默认动作有如下两种：

终止进程：因为没有信号处理器，接到信号之后进程终止。

core：将进程的存储镜像作为文件保存在进程的工作目录下。

core文件是大多数LINUX/UNIX系统的实现特征。在不同的实现中，core文件的名字可能不同。大多数实现在相应进程的当前工作目录中存放core文件。

在下列条件下不产生core文件：

(a)进程是设置用户ID的，而且当前用户并非程序文件的所有者。

(b)进程是设置组ID的，而且当前用户并非该程序文件的组所有者。

(c)用户没有写当前工作目录的权限。

(d)文件已存在，而且用户对该文件设有写权限。

(e)文件太大（回忆进程环境中讨论的RLIMIT\_CORE限制）。core文件的权限（假定该文件在此之前并不存在）通常是用户读/写，但是有的实现权限不一样。

使用kill -l可以看到系统支持的信号列表，使用man signal可以看到系统支持信号较为详细的介绍。

下面较详细地逐一说明这些信号。

|  |  |
| --- | --- |
| 常用信号列表 | |
| SIGABRT | 调用abort函数时产生此信号。进程异常终止。 |
| SIGALRM | 在用alarm函数设置的计时器超时时，产生此信号。若由setitimer⑵函数设置的间隔时间超时时，也会产生此信号。 |
| SIGBUS | 指本一个实现定义的硬件故障。当出现某些类型的内存故障时常常产生此种信号。 |
| SIGCANCEL | 这是Solaris线程库内部使用的信号。它不供一般应用。 |
| SIGCHLD | 在一个进程终止或停止时，将SIGCHLD信号发送给其父进程。按系统默认，将忽略此信号。如果父进程希望被告知其子进程的这种状态改变，则应捕捉此信号。信号捕捉函数中通常要调用一种wait函数以取得子进程ID和其终止状态。 |
| SIGCONT | 此作业控制信号被发送给需要继续运行，但当前处干停止状态的进程。如果接收到此信号的进程处于停止状态，则系统默认动作是使该进程继续运行，否则默认动作是忽略此信号。例如，全屏幕编辑器在捕捉到此信号后，使用信号处理程序发出新绘制终端屏幕的通知。 |
| SIGEMT | 指示一个实现定义的硬件故障。 |
| SIGFPE | 此信号表示一个算术运算异常，例如除以0，浮点溢出等。 |
| SIGFREEZE | 此信号仅由Solaris定义。它用干通知进程在冻结系统状态之前需要采取特定 动作，例如当系统进入冬眠或挂起模式时可能需要执行这种处理。 |
| SIGHUP | 如果终端接口检测到一个连接断开，则将此信号发送给与该终端相关的控 制进程(会话首进程)。 |
| SIGILL | 此信号指示进程已执行一条非法硬件指令。 |
| SIGINFO | 这是一种BSD信号，当用户按状态键(一般采用Ctrl+T)时，终端驱动程序 产生此信号并送至前台进程组中的每一个进程(见图9-8)。此信号通常导致 在终端上显示前台进程组中各进程的状态信息。 |
| SIGINT | 当用户按中断键(一般采用DELETE或Ctrl+C)时，终端驱动程序产生此信 号并送至前台进程组中的每一个进程(见图9-8)。当一个进程在运行时失控, 特别是它正在屏幕上产生大量不需要的输出时，常用此信号终止它。 |
| SIGIO | 此信号指示一个异步I/O事件。 |
| SIGIOT | 这指示一个实现定义的硬件故障。 |
| SIGKILL | 这是两个不能被捕捉或忽略的信号之一。它向系统管理员提供了一种可以 杀死任一进程的可靠方法。 |
| SIGLWP | 此信号由Solaris线程库内部使用，并不作一般使用。 |
| SIGPIPE | 如果在写到管道时读进程已终止，则产生此信号。 |
| SIGPOLL | 当在一个可轮询设备上发生一特定事件时产生此信号。 |
| SIGPROF | 当setitimer⑵函数设置的梗概统计间隔计时器(profiling interval timer) 已到期时产生此信号。 |
| SIGPWR | 这是一种依赖干系统的信号。它主要用干具有不间断电源(UPS)的系统。如果电源失效，则UPS起作用，而且通常软件会接到通知。在这种情况下， 系统依靠蓄电池电源继续运行，所以无须任何处理。但是如果蓄电池也将不能支持工作，则软件通常会再次接到通知，此时，系统必项在15~30秒内使其各部分都停止运行。此时应当发送SIGPWR信号。在大多数系统中， 接到蓄电池电压过低信息的进程将信号SIGPWR发送给init进程，然后由 init处理停机操作。 |
| SIGQUIT | 当用户在终端上按退出键(一般采用Ctrl+\)时，产生此信号，并送至前台进程组中的所有进程。此信号不仅会终止前台进程组(如 SIGINT所做的那样)，同时还会产生一个core文件。 |
| SIGSEGV | 该信号指示进程进行了一次无效内存引用。名字SEGV表示“段违例(segmentation violation)”。 |
| SIGSTKFLT | 此信号仅由Linux定义。它出现在Linux的早期版本，旨在用干数学协处理器 的栈故障。该信号并非由内核产生，但仍保留以向后兼容。 |
| SIGSTOP | 这是一个作业控制信号，用于停止一个进程。它类似于交互停止信号 (SIGTSTP),但是SIGST0P不能被捕捉或忽略。 |
| SIGSYS | 该信号指示一个无效的系统调用。由于某种未知的原因，进程执行了一条 机器指令，内核认为这是一个系统调用，但该指令指示系统调用类型的参 数却是无效的。这种情况是可能发生的，例如，若用户编写了一道使用新系统调用的程序，然后尝试运行该程序的二进制可执行代码，而所用的操 作系统却是不支持该系统调用的较早版本，干是就会出现上述情况。 |
| SIGTERM | 这是由kill(l)命令发送的系统默认终止信号。 |
| SIGTHAW | 此信号仅由Solaris定义。当系统恢复运行被挂起的操作时，该信号用干通知 相关进程，它们需要采取特殊的动作。 |
| SIGTRAP | 指示一个实现定义的硬件故障。 |
| SIGTSTP | 交互式停止信号，当用户在终端上按挂起键(一般采用Ctrl+Z)时，终端驱动程序产生此信号。该信号送至前台进程组中的所有进程。不幸的是，停止(stop)这个术语具有不同的含义。当讨论作业控制和信号时，我们谈及停止和 继续执行作业。但是，终端驱动程序一直使用术语“停止"表示用Ctrl+S字符停止终端输出，为了继 续启动该终端输出，则用Ctrl+Q字符。为此，终端驱动程序称产生交互式停止信号的字符为挂起字符， 而非停止字符。 |
| SIGTTIN | 当一个后台进程组中的进程试图读其控制终端时，终端驱动程序产生此信号在下列特殊情形下不产生此信号：(a)读进程忽略或阻塞此信号，(b)读进程所属的进程组是孤儿进程组，此时读操作返回出错，并将errno设置为EIO。 |
| SIGTTOU | 当一个后台进程组中的进程试图写到其控制终端时产生此信号(见9.8节对 此主题的讨论)。与上面所述的SIGTTIN信号不同，一个进程可以选择允许后台进程写到控制终端。如果不允许后台进程写，则与SIGTTIN相似，也有两种特殊情况：(a) 写进程忽略或阻塞此信号，(b)写进程所属进程组是孤儿进程组。在这两种情况下不产生此信号，写操作返回出错，并将errno设置为EIO。  不论是否允许后台进程写，某些除写以外的下列终端操作也能产生此信号：tcsetattr、tcsendbreak、tcdrain、tcflush, tcf]ow以及 tcsetpgrp |
| SIGURG | 此信号通知进程已经发生一个紧急情况。在网络连接上接收到带外的数据时，可选择产生此信号。 |
| SIGUSR1 | 这是一个用户定义的信号，可用干应用程序。 |
| SIGUSR2 | 这是另一个用户定义的信号，与SIGUSR1相似，可用于应用程序。 |
| SIGVTALRM | 当一个由setitimer⑵函数设置的虚拟间隔时间到期时产生此信号。 |
| SIGWAITING | 此信号由Solaris线程库内部使用，不作它用。 |
| SIGWINCH | 内核维持与每个终端或伪终端相关联的窗口大小。进程可以用ioctl函数得到或设置窗口的大小。如果进程用ioctl的设置窗口大小命令更改了窗口大小，则内核产生SIGWINCH信号并将其送至前台进程组。 |
| SIGXCPU | 如果进程超过了其软CPU时间限制，则产生SIGXCPU信号。 |
| SIGXFSZ | 如果进程超过了其软文件长度限制，则产生此信号。 |
| SIGXRES | 此信号仅由Solaris定义。可选择使用此信号以通知进程超过了预配置的资源值。Solaris资源限制机制是一种通用设施，用于控制在独立应用程序集之间使用共享资源。 |

## 10.3 signal 函数

LINUX/UNIX系统的信号机制最简单的接口是signal函数。

|  |
| --- |
| #include <signal.h>  void (\*signal (int signo, void (\*func) (int))) (int);  返回值：若成功则返回信号以前的处理配置（见下），若出错则返回SIG\_ERR |

signo数是第一节中的信号名。func的值是常量SIG\_IGN、常量SIG\_DFL或当接到此信号后要调用的函数的地址。如果指定SIG\_IGN，则向内核表示忽略此信号（记住有两个信号 SIGKILL和SIGSTOP不能忽略）。如果指定SIG\_DFL，则表示接到此信号后的动作是系统默认动作。当指定函数地址时，则在信号发生时，调用该函数，我们称这种处理为“捕捉”该信号。称此函数为信号处理程序（signal handler）或信号捕捉函数（signal-catching function）。

signal函数原型说明此函数需要两个参数，返回一个函数指针，而该指针所指向的函数无返回值(void)。第一个参数signo是一个整数，第二个参数是函数指针，它所指向的函数需要一个整型参数，无返回值。signal的返回值是一个函数地址，该函数有一个整型参数（即 最后的(int)）。用自然语言来描述也就是要向信号处理程序传送一个整型参数，而它却无返 回值。当调用signal设置信号处理程序时，第二个参数是指向该函数（也就是信号处理程序）的指针。signal的返回值则是指向之前的信号处理程序的指针。

本节开头所示的signal函数原型太复杂了，如果使用下面的typedef，则可使其简单一些。

typedef void Sigfunc(int);

然后，可将signal函数原型写成

Sigfunc \*signal(int, Sigfunc \*)；

如果査看系统的头文件<signal .h>，则很可能会找到下列形式的声明：

#define SIG—ERR (void (\*)())-1

#define SIG\_DFL (void (\*)())0

#define SIG—IGN (void (\*)())1

这些常量可用于代替“指向函数的指针，该函数需要一个整型参数，而且无返回值”。signal的第二个参数及其返回值就可用它们表示。这些常量所使用的三个值不一定是-1，0和1。它们必须是三个值而决不能是任一可声明函数的地址。大多数LINUX/UNIX系统使用上面所示的值。

下面程序显示了一个简单的信号处理程序，它捕捉两个用户定义的信号并打印信号编 号。后面将将说明pause函数，它使调用进程在接到一个信号前挂起。

|  |
| --- |
|  |

我们在后台调用该程序，并且用ki 11(1)命令将信号传送给它。注意，在LINUX/UNIX中，杀死(kill)这个术语是不恰当的。kill(l)命令和kill(2)函数只是将一个信号送给一个进程或进程组。信号是否终止进程则取决于信号的类型，以及进程是否安排了捕捉该信号。

程序运行结果如下：

|  |
| --- |
|  |

因为执行程序的不捕捉SIGTERM信号，而针对该信号的系统默认动作是终止，所以当向该进程发送SIGTERM信号后，该进程就会终止。

**1. 程序启动**

当执行一个程序时，所有信号的状态都是系统默认或忽略。通常所有信号都被设置为它们的默认动作，除非调用exec的进程忽略该信号。确切地讲，exec函数将原先设置为要捕捉的信号都更改为它们的默认动作，其他信号的状态则不变（对于一个进程原先要捕捉的信号，当其执行一个新程序后，就自然不能再捕捉它了，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义）。

一个具体例子是一个交互式shell如何处理针对后台进程的中断和退出信号。对于一个非作业控制shell，当在后台执行一个进程时，例如：

cc main.c &

shell自动将后台进程对中断和退出信号的处理方式设置为忽略。于是，当按中断键时就不会影响到后台进程。如果没有执行这样的处理，那么当按中断键时，它不但会终止前台进程，还会终止所有后台进程。

很多捕捉这两个信号的交互式程序具有下列形式的代码：

void sig\_int(int), sig\_quit(int)；

if (signal(SIGINT, SIG\_IGN) != SIG\_IGN)

signal(SIGINT, sig\_int);

if (signal(SIGQUIT, SIG\_IGN) != SIG\_IGN)

signal(SIGQUIT, sig\_quit);

这样处理后，仅当信号当前未被忽略时，进程才会捕捉它们。

从signal的这两个调用中也可以看到这种函数的限制：不改变信号的处理方式就不能确定信号的当前处理方式。我们将在本章的稍后部分说明使用sigaction函数可以确定一个信号的处理方式，而无需改变它。

**2. 进程创建**

当一个进程调用fork时，其子进程继承父进程的信号处理方式。因为子进程在开始时复制了父进程的存储映像，所以信号捕捉函数的地址在子进程中是有意义的。

## 10.4不可靠的信号

在早期的LINUX/UNIX版本中，信号是不可靠的。不可靠在这里指的是，信号可能会丢失：一个信号发生了，但进程却可能一直不知道这一点。同时，进程对信号的控制能力也很差，它能捕捉信号或忽略它。有时用户希望通知内核阻塞一个信号：不要忽略该信号，在其发生时记住它，然后在进程做好准备时再通知它。这种阻塞信号的能力当时并不具备。

早期版本中的一个问题是在进程每次接到信号对其进行处理时，随即将该信号动作复位为默认值（在前面运行程序时，我们只捕捉每种信号一次，从而回避了这一点）。在描述这些早期系统的编程书籍中，有个经典实例，它与如何处理中断信号相关，其代码与下面所示的相似：

|  |
| --- |
|  |

由于早期的c语言版本不支持ISO C的void数据类型，所以将信号处理程序声明为int类型。

这段代码的一个问题是：从信号发生之后到在信号处理程序中调用signal函数之前这段时间中有一个时间窗口。在此段时间中，可能发生另一次中断信号。第二个信号会导致执行默认动作，而针对中断信号的默认动作是终止该进程。这种类型的程序段在大多数情况下会正常工作，使得我们认为它们是正确无误的，而实际上并非如此。

这些早期系统的另一个问题是：在进程不希望某种信号发生时，它不能关闭该信号。进程能做的一切就是忽略该信号。有时希望通知系统“阻止下列信号发生，如果它们确实产生了， 请记住它们。”能够显现这种缺陷的一个经典实例是下列程序段，它捕捉一个信号，然后设置一个表示该信号已发生的标志：

|  |
| --- |
|  |

其中，进程调用pause函数使自己休眠，直至捕捉到一个信号。当捕捉到信号时，信号处理程序将标志sig\_int\_flag设置为非0值。从信号处理程序返回后，内核自动将该进程唤醒，它检测到该标志为非0，然后执行它所需做的工作。但是这里也有一个时间窗口，在此窗口中操作可能失误。如果在测试sig\_int\_flag之后和调用pause之前发生信号，则此进程在调用 pause时人睡，并且长眠不醒（假定此信号不会再次产生）。于是，这次发生的信号也就丢失了。这是另一个例子，某段代码并不正确，但是大多数时间却能正常工作。要查找并排除这种类型的问题很困难。

## 10.5中断的系统调用

早期LINUX/UNIX系统的一个特性是：如果进程在执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行。该系统调用返回出错，其errno被设置为EINTR。这样处理的理由是：因为一个信号发生了，进程捕捉到了它，这意味着已经发生了某种事情，所以是个应当唤醒阻塞的系统调用的好机会。

在这里，我们必须区分系统调用和函数。当捕极到某个信号时，被中断的是内核中执行的系统调用。

为了支持这种特性，将系统调用分成两类：低速系统调用和其他系统调用。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括：

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

在这些低速系统调用中，一个值得注意的例外是与磁盘I/O有关的系统调用。虽然读、写一个磁盘文件可能暂时阻塞调用者（在磁盘驱动器将请求排入队列，然后在适当时间执行请求期间），但是除非发生硬件错误，I/O操作总会很快返回，并使调用者不再处于阻塞状态。

可以用中断系统调用这种方法来处理的一个例子是：一个进程启动了读终端操作，而使用该终端设备的用户却离开该终端很长时间。在这种情况下进程可能处于阻塞状态几个小时甚至数天，除非系统停机，否则一直如此。

与被中断的系统调用相关的问题是必须显式地处理出错返回。典型的代码序列（假定进行一个读操作，它被中断，我们希望重新启动它）可能如下所示：

|  |
| --- |
| again：  if ((n = read(fd, buf, BUFFSIZE)) < 0) {  if (errno -= EINTR)  goto again； /\* just an interrupted system call \*/  /\* handle other errors \*/  } |

为了帮助应用程序使其不必处理被中断的系统调用，LINUX/UNIX引入了某些被中断系统调用的自动重启动。自动重启动的系统调用包括ioctl、read、readv、write、writev、wait和 waitpid。正如前述，其中前5个函数只有对低速设备进行操作时才会被信号中断。而wait和 waitpid在捕捉到信号时总是被中断。因为这种自动重启动的处理方式也会带来问题，所以某些应用程序并不希望这些函数被中断后重启动。为此LINUX/UNIX允许进程基于每个信号禁用此功能。

引入自动重启动功能的一个理由是：有时用户并不知道所使用的输人、输出设备是否是低速设备。如果我们编写的程序可以用交互方式运行，则它可能读、写低速终端设备。如果在程序中捕捉信号，而且系统并不提供重启动功能，则对每次读、写系统调用都要进行是否 出错返回的测试，如果是被中断的，则再调用读、写系统调用。

## 10.6可重入函数

进程捕捉到信号并对其进行处理时，进程正在执行的指令序列就被信号处理程序临时中断，它首先执行该信号处理程序中的指令。如果从信号处理程序返回（例如没有调用exit或 longjmp），则继续执行在捕捉到信号时进程正在执行的正常指令序列（这类似于发生硬件中断时所做的）。但在信号处理程序中，不能判断捕捉到信号时进程在何处执行。如果进程正在执行malloc，在其堆中分配另外的存储空间，而此时由于捕捉到信号而插入执行该信号处理程序，其中又调用malloc，这时会发生什么？又例如若进程正在执行getpvmam这种将其结果存放在静态存储单元中的函数，其间插入执行信号处理程序，它又调用这样的函数，这时又会发生什么呢？在malloc例子中，可能会对进程造成破坏，因为malloc通常为它所分配的存储区维护一个链接表，而插入执行信号处理程序时，进程可能正在更改此链接表。 在getpwnam的例子中，返回给正常调用者的信息可能被返回给信号处理程序的信息覆盖。 LINUX/UNIX说明了保证可重入的函数。下表列出了这些可重入函数。

表10-3倍号处理程序可以调用的可重入函数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| accept | fchmod | lseek | sendto | stat |
| aio\_error | fcntl | mkdir | setpgid | sysconf |
| aio\_return | fdatasync | mkfifo | setsid | tcdrain |
| alarm | fpathconf | pathconf | setuid | tcflush |
| bind | fstat | pause | shutdown | tcgetattr |
| cfgetispeed | f sync | pipe | sigaction | tcgetpgrp |
| cfgetospeed | ftruncate | poll | sigaddset | tcsendbreak |
| cfsetispeed | getegid | posix\_trace\_event | sigdelset | tcsetattr |
| cfsetospeed | geteuid | pselect | sigemptyset | tcsetpgrp |
| chdir | getgid | raise | sigfillset | time |
| chmod | getgroups | read | sigismember | timer\_getoverrun |
| chown | getpeername | readlink | signal | timer\_gettime |
| clock\_gettime | getpgrp | recv | sigpause | timer\_\_settime |
| close | getpid | recvfrom | sigpending | times |
| connect | getppid | recvmsg | sigprocmask | umask |
| creat | getsockname | rename | siggueue | uname |
| dup | getsockopt | rmdir | sigset | unlink |
| dup2 | getuid | select | sigsuspend | utime |
| execle | kill | sem\_post | sleep | wait |
| execve | link | send | socket | waitpid |
| \_Exit \_exit | listen | sendmsg | socketpair | write |

没有列入上表中的大多数函数是不可重入的，其原因为：

(a)已知它们使用静态数据结构，

(b)它们调用malloc或free，或

(c)它们是标准I/O函数。

标准I/O库的很多实现都以不可重入方式使用全局数据结构。注意，即使在本书的某些实例中，信号处理程序也调用了printf函数，但这并不保证产生所期望的结果，信号处理程序可能中断主程序中的printf 函数调用。

应当了解即使信号处理程序调用的是列于上表中的函数，但是由于每个线程只有一个 errno变量，所以信号处理程序可能会修改其原先值。考虑一个信号处理程序，它恰好在main刚刚设置errno之后被调用。例如，如果该信号处理程序调用read这类函数，则它可能更改errno的值，从而取代了刚刚由main设置的值。因此，作为一个通用的规则，当在信号处理程序中调用上表中列出的函数时，应当在其前保存，在其后恢复errno。（应当了解，经常被捕捉到的信号是SIGCHLD，其信号处理程序通常要调用一种wait函数，而各种wait函数都能改变errno。）

注意，上表没有包括longjmp和siglongjmp。这是因为主例程以非可重入方式正在更新数据结构时可能产生信号。如果不是从信号处理程序返回而是调用 siglongjmp，那么该数据结构可能是部分更新的。如果应用程序将要做更新全局数据结构这样的事情，同时要捕捉某些信号，而这些信号的处理程序又会引起执行sigsetjmp，则在更新这种数据结构时要阻塞此类信号。

再下列程序中，信号处理程序my\_alarm调用不可重人函数getpwnam，而 my\_alarm每秒钟被调用一次。后面将说明alarm函数。在该程序中调用alarm函数使得每秒产生一次SIGALRM信号。

|  |
| --- |
|  |

运行该程序时，其结果具有随意性。通常，在信号处理程序经多次迭代返回时，该程序将由SIGSEGV信号终止。检査core文件，从中可以看到main函数已调用getpwnam，而且当信号处理程序调用此同一函数时，某些内部指针出了问题。偶然，此程序会运行若干秒，然后因产生SIGSEGV信号而终止。在捕捉到信号后，若main函数仍正确运行，其返回值却有时错误，有时正确。

从此实例中可以看出，若在信号处理程序中调用一个不可重人函数，则其结果是不可预 见的。

## 10.7可靠信号术语和语义

我们需要先定义一些在讨论信号时会用到的术语。首先，当引发信号的事件发生时，为进程产生一个信号（或向进程发送一个信号）。事件可以是硬件异常（例如，除以0）、软件条件（例如，alarm计时器超时）、终端产生的信号或调用kill函数。在产生了信号时，内核通常在进程表中设置一个某种形式的标志。

当对信号采取了这种动作时，我们说向进程递送了一个信号。在信号产生（generation）和递送（delivery）之间的时间间隔内，称信号是未决的（pending）。

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

如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么将如何呢？ POSIX.1允 许系统递送该信号一次或多次。如果递送该信号多次，则称对这些信号进行了排队。但是除非支持POSIX.1实时扩展，否则大多数LINUX/UNIX并不对信号排队。代之以LINUX/UNIX内核只递送这种信号一次。

如果有多个信号要递送给一个进程，那么将如何呢？ POSIX.1并没有规定这些信号的递送顺序。但是POSIX.1的Rationale建议：在其他信号之前递送与进程当前状态有关的信号，例如 SIGSEGV。

每个进程都有一个信号屏蔽字（signal mask），它规定了当前要阻塞递送到该进程的信号集。对于每种可能的信号，该屏蔽字中都有一位与之对应。对于某种信号，若其对应位已设置，则它当前是被阻塞的。进程可以调用sigprocmask（后面将要讨论）来检测和更改其当前信号屏蔽字。

信号数量可能会超过整型所包含的二进制位数，因此POSIX.1定义了一个新数据类型 sigset\_t，用于保存一个信号集。例如，信号屏蔽字就存放在这些信号集的一个中。后面将说明对信号集进行操作的5个函数。

## 10.8 kill和raise函数

kill函数将信号发送给进程或进程组。raise函数则允许进程向自身发送信号。

|  |
| --- |
| #include <signal.h>  int kill (pid\_t pid, int signo);  int raise (int signo);  两个函数返回值：若成功则返回0，若出错则返回-1 |

调用

raise(signo);

等价于调用

kill(getpid(), signo);

kill的pid参数有4种不同的情况：

* pid > 0 将该信号发送给进程ID为pid的进程。
* pid==0 将该信号发送给与发送进程属于同一进程组的所有进程（这些进程的进程组ID 等于发送进程的进程组ID），而且发送进程具有向这些进程发送信号的权限。注意，这里用的术语“所有进程”不包括实现定义的系统进程集。对于大多数 LINUX/UNIX系统，系统进程集包括内核进程以及init (pid 1)。
* pid<0 将该信号发送给其进程组ID等于的绝对值，而且发送进程具有向其发送信号的权限。如上所述，“所有进程集”并不包括某些系统进程。
* pid==-1将该信号发送给发送进程有权限向它们发送信号的系统上的所有进程。如上所述，“进程集”不包括某些系统进程。

上面曾提及，进程将信号发送给其他进程需要权限。超级用户可将信号发送给任一进程。对于非超级用户，其基本规则是发送者的实际或有效用户ID必须等于接收者的实际或有效用户ID。在对权限进行测试时也有一个特例：如果被发送的信号是SIGCONT，则进程可将它发送给属于同一会话的任何其他进程。

POSIX.1将编号为0的信号定义为空信号。如果signo参数是0，则kill仍执行正常的错误检査，但不发送信号。这常被用来确定一个特定进程是否仍旧存在。如果向一个并不存在的进程发送空信号，则kill返回-1，并将errno设置为ESRCH。但是，应当了解，LINUX/UNIX系统在经过一段时间后会重新使用进程ID，所以一个现有的具有所给定进程ID的进程并不一定就是你想要的进程。

还应理解的是，对于进程是否存在的这种测试不是原子操作。在kill向调用者返回测试结果时，原来存在的被测试进程此时可能已经终止，所以这种测试并无多大价值。

如果调用kill为调用进程产生信号，而且此信号是不被阻塞的，那么在kill返回之前，就会将signo或者某个其他未决的非阻塞信号传送至该进程。

## 10.9 alarm和pause函数

使用alarm函数可以设置一个计时器，在将来某个指定的时间该计时器会超时。当计时器超时时，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止调用该alarm函数的进程。

|  |
| --- |
| #include <unistd.h>  unsigned int alarm (unsigned int seconds);  返回值：0或以前设置的闹钟时间的余留秒数 |

其中，参数second的值是秒数，经过了指定的seconds秒后会产生信号SIGALRM。要了解的是，经过了指定的秒数后，信号由内核产生，由于进程调度的延迟，所以进程得到控制从而能够处理该信号还需一些时间。

每个进程只能有一个闹钟时钟。如果在调用alarm时，以前已为该进程设置过闹钟时钟，而且它还没有超时，则将该闹钟时钟的余留值作为本次alarm函数调用的值返回。以前登记的闹钟时钟则被新值代替。

如果有以前为进程登记的尚未超过的闹钟时钟，而且本次调用的seconds值是0，则取消以前的闹钟时钟，其余留值仍作为alarm函数的返回值。

虽然SIGALRM的默认动作是终止进程，但是大多数使用闹钟的进程会捕捉此信号。如果此时进程要终止，则在终止之前它可以执行所需的清理操作。如果我们想捕捉SIGALRM信号，则必须在调用alarm之前设置该信号的处理程序。如果我们先调用alarm，然后在我们能够设置SIGALRM处理程序之前已接收到该信号，那么进程将终止。

pause函数使调用进程挂起直至捕捉到一个信号。

|  |
| --- |
| #include <unistd.h>  int pause(void);  返回值：-1，并将errno设置为EINTR |

只有执行了一个信号处理程序并从其返回时，pause才返回。在这种情况下，pause返回-1，并将errno设置为EINTR。

使用alarm和pause,进程可使自己休眠一段指定的时间。下面程序中的的sleep1函数提供这种功能（但是它有一些问题，我们很快就会看到）。

|  |
| --- |
|  |

程序中的sleepl函数看起来LINUX/UNIX的系统调用sleep函数类似，但这种简单实现有下列三个问题：

(1)如果在调用sleepl之前，调用者已设置了闹钟，则它会被sleepl函数中的第一次 alarm调用擦除。可用下列方法更正这一点：检查第一次调用alarm的返回值，如其小于本次调用alarm的参数值，则只应等到上次设置的闹钟超时。如果上次设置闹钟的超时时间晚于本次设置值，则在sleepl函数返回之前，复位此闹钟，使其在上次闹钟的设定时间再次发生超时。

(2)该程序中修改了对SIGALRM的配置。如果编写了一个函数供其他函数调用，则在该函数被调用时先要保存原配置，在该函数返回前再恢复原配置。更正这一点的方法是：保存signal函数的返回值，在返回前复位原配置。

(3)在第一次调用alarm和调用pause之间有一个竞争条件。在一个繁忙的系统中，可能alarm在调用pause之前超时，并调用了信号处理程序。如果发生这种情况，则在调用pause 后，如果没有捕捉到其他信号，则调用者将永远被挂起。

sleep的早期实现与上述问题类似，但更正了问题(1)和(2)。有两种方法可以更正问题(3)。第一种方法是使用setjmp,下一个实例将说明这种方法。另一种方法是使用 sigprocmask 和 sigsuspend，后面将讨论这种做法。

使用setjmp的代码如下：

|  |
| --- |
|  |

在此函数中，sleep1具有的竞争条件已被避免。即使pause从未执行，在发生SIGALRM时，sleep2函数也会返回。但是，sleep2函数中却有另一个难以察觉的问题，它涉及与其他信号的交互。如果 SIGALRM中断了某个其他信号处理程序，则调用longjmp会提早终止该信号处理程序。下面程序显示了这种情况。

|  |
| --- |
|  |

SIGINT处理程序中包含了for循环语句，它在作者所用系统上的执行时间超过5秒钟，也就是大于sleep2的参数值，这正是我们想要的。将整型变量k声明为volatile, 这样就阻止了优化编译器丢弃循环语句。

上述程序的执行结果如下：

|  |
| --- |
|  |

从中可见sleep2函数所引起的longjmp使另一个信号处理程序sig\_int提早终止，即使它未完成也会如此。

有关sleepl和sleep2函数的这两个实例的目的是告诉我们在涉及信号时需要有精细而周到的考虑。下面几节将说明解决这些问题的方法，使我们能够可靠地、在不影响其他代码段的情况下处理信号。

除了用来实现sleep函数外，alarm还常用于对可能阻塞的操作设置时间上限值。例如，程序中有一个读低速设备的可能阻塞的操作，我们希望超过一定时间量后就停止执行该操作。下面的程序实现了这一点，它从标准输入读一行，然后将其写到标准输出上。

|  |
| --- |
|  |

这种代码序列在很多LINUX/UNIX应用程序中都能见到，但是这种程序有两个问题：

(1)在第一次alarm调用和read调用之间有一个竞争条件。如果内核在这两个函数调用之间使进程阻塞，而其时间长度又超过闹钟时间，则read可能永远阻塞。大多数这种类型的操作使用较长的闹钟时间，例如1分钟或更长一点，使这种问题不会发生，但无论如何这是一个竞争条件。

(2)如果系统调用是自动重启动的，则当从SIGALRM信号处理程序返回时，read并不被中断。在这种情形下，设置时间限制不起作用。

在这里我们确实需要中断低速系统调用。POSIX.1并未提供一种可移植的方法来实现这一点，但是，Single UNIX Specification的XSI扩展却做到了这一点。后面将讨论这个问题。

让我们用longjmp重新实现前面的实例。使用这种方法则无需担心一个慢速的系统调用是否被中断。

|  |
| --- |
|  |

不管系统是否重新启动中断的系统调用，该程序都会如所预期的那样工作。但是要知道, 该程序仍旧有与其他信号处理程序交互的问题。

如果要对I/O操作设置时间限制，则如上所示可以使用longjmp，当然也要清楚它可能有与其他信号处理程序交互的问题。另一种选择是使用select或poll函数，后面章节将对它们进行说明。

## 10.10信号集

我们需要有一个能表示多个信号——信号集(signal set)的数据类型。我们将在诸如

sigprocmask （下一节中说明）类的函数中使用这种数据类型，以便告诉内核不允许发生该信号集中的信号。如前所述，信号种类数目可能超过一个整型量所包含的位数，所以一般而言，不能用整型量中的一位代表一种信号，也就是不能用一个整型量表示信号集。POSIX.1定义了数据类型sigset\_t以包含一个信号集，并且定义了下列五个处理信号集的函数。

|  |
| --- |
| #include <signal.h>  int sigemptyset (sigset\_t \*set) ;  int sigfillset (sigset\_t \*set);  int sigaddset (sigset\_t \*$et, int signo);  int sigdelset (sigset\_t \*set, int signo);  四个函数的返回值：若成功则返回0，若出错则返回-1  int sigismember (const sigset\_t \*set, int signo);  返回值\*.若真则返回1，若假则返回0，若出错则返回-1 |

函数sigemptyset初始化由如指向的信号集，清除其中所有信号。函数sigfillset初始化由set指向的信号集，使其包括所有信号。所有应用程序在使用信号集前，要对该信号集调 用sigemptyset或sigfillset—次。这是因为C编译器将把未陚初值的外部和静态变量都初始化为0，而这是否与给定系统上信号集的实现相对应却并不清楚。

一且已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号。函数sigaddset将一个信号添加到现有集中，sigdelset则从信号集中删除一个信号。对所有以信号集作为参数的函数，我们总是以信号集地址作为向其传送的参数。

如果实现的信号数目少于一个整型量所包含的位数，则可用一位代表一个信号的方法实现信号集。例如，在本书的后续部分，我们都假定一种实现有31种信号和32位整型量。 sigemptyset函数将整型量设置为0，sigfillset函数则将整型量中的各个位都设置为1。 这两个函数可以在<signal .h>头文件中实现为宏：

#define sigemptyset(ptr) (\*(ptr) = 0)

#define sigfillset(ptr) (\*(ptr) ＝~(sigset\_t)0, 0)

注意，除了设置信号集中各位为1外，sigfillset必须返回0，所以使用C语言的逗号运算符，它将逗号运算符后的值作为表达式的值返回。

使用这种实现，sigaddset打开一位（将该位设置为1）, sigdelset则关闭一位（将该位设置为0）, sigismember测试一指定位。因为没有编号为0的信号，所以从信号编号中减去1以得到要处理位的位编号数。下面给出这些函数的实现。

|  |
| --- |
|  |

也可将这三个函数在<signal.h>中实现为单行宏，但是POSIX. 1要求检查信号编号参数的有效性，如果无效则设置errno。在宏中实现这一点比在函数中要困难。

## 10.11 sigprocmask 函数

前面提及一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集。调用函数sigprocmask可以检测或更改其信号屏蔽字，或者在一个步骤中同时执行这两个操作。

|  |
| --- |
| #include <signal.h>  int sigprocmask (int how, const sigset\_t \*restrict set, sigset\_t \*restrict oset);  返回值：若成功则返回0，若出错则返回-1 |

首先，若oset是非空指针，那么进程的当前信号屏蔽字通过oset返回。

其次，若set是一个非空指针，则参数指示如何修改当前信号屏蔽字。下表说明了可选用的值。SIG\_BLOCK是“或”操作，而SIG\_SETMASK则是赋值操作。注意，不能阻塞

SIGKILL 和 SIGSTOP 信号。

sigprocmask更改当前信号屏蔽字的方法

|  |  |
| --- | --- |
| how | 说 明 |
| SIG\_BLOCK | 该进程新的信号屏蔽字是其当前信号屏蔽字和set指向信号集的并集。set包含了我们希望阻塞的附加信号 |
| SIG\_UNBLOCK | 该进程新的信号屏蔽字是其当前信号屏蔽字和set所指向信号集补集的交集。set包含了我们希望解除阻塞的信号 |
| SIG\_SETMASK | 该进程新的信号屏蔽字将被指向的信号集的值代替 |

如果set是空指针，则不改变该进程的信号屏蔽字，how的值也无意义。

在调用sigprocmask后如果有任何未决的、不再阻塞的信号，则在sigprocmask返回前,至少会将其中一个信号递送给该进程。

sigprocmask是仅为单线程的进程定义的。为处理多线程的进程中信号的屏蔽，提供了另一个单独的函数。我们将在线程相关章节中对此进行讨论。

下面函数打印调用进程的信号屏蔽字中信号的名称。后面的程序将调用这个函数。

|  |
| --- |
|  |

为了节省空间，上述代码没有对每一种信号测试该屏蔽字。

## 10.12 sigpending 函数

sigpending函数返回信号集，其中的各个信号对于调用进程是阻塞的而不能递送，因而也一定是当前未决的。该信号集通过参数返回。

|  |
| --- |
| #include <signal.h>  int sigpending(sigset\_t \*srf);  返回值：若成功则返回0，若出错则返回-1 |

下面程序使用了很多前面说明过的信号功能。

|  |
| --- |
|  |

进程阻塞SIGQUIT信号，保存了当前信号屏蔽字（以便以后恢复），然后休眠5秒钟。在此期间所产生的退出信号SIGQUIT都会被阻塞，而不递送至该进程，直到该信号不再被阻塞。在5秒钟休眠结束后，检査该信号是否是未决的，然后将SIGQUIT设置为不再阻塞。

注意，在设置SIGQUIT为阻塞时，我们保存了旧屏蔽字。为了解除对该信号的阻塞，用旧屏蔽字重新设置了进程信号屏蔽字（SIG\_SETMASK）。另一种方法是用SIG\_UNBLOCK使阻塞的信号不再被阻塞。但是，应当了解如果编写一个可能由其他人使用的函数，而且需要在函数中阻塞一个信号，则不能用SIG\_UNBLOCK简单地解除对此信号的阻塞，这是因为此函数的调用者在调用本函数之前可能也阻塞了此信号。在这种情况下必须使用SIG\_SETMASK将信号屏蔽字复位为原先值，这样也就能继续阻塞该信号。后面的systern函数部分有这样一个例子。

在休眠期间如果产生了退出信号，那么此时该信号是未决的，但是不再受阻塞，所以在sigprocmask返回之前，它被递送到调用进程。从程序的输出中可以看出：SIGQUIT处理程序(sig\_quit)中的printf语句先执行，然后再执行sigpromask之后的printf语句。

然后该进程再休眠5秒钟。如果在此期间再产生退出信号，那么因为在上次捕捉到该信号时，已将其处理方式设置为默认动作，所以这一次它就会使该进程终止。在下列输出中，当我们在终端上键入退出字符Ctrl+\时，终端打印、（终端退出字符）：

|  |
| --- |
| $ ./a.out  SIGQUIT pending caught SIGQUIT SIGQUIT unblocked "AQult (coredump)  $ ./a.out  SIGQUIT pending caught SIGQUIT SIGQUIT unblocked AQuit (coredump)  产生信号一次(在5秒钟之内)  从sleep返回后  在信号处理程序中  从sigprocmask返回后  再次产生信号  [323]  产生信号10次(在5秒钟之内)  只产生信号一次  再次产生信号 |

在shell发现其子进程异常终止时，会输出"Quit (coredump)"消息。注意，第二次运行该程序时，在进程休眠期间我们使SIGQUIT信号产生了 10次，但是解除了对该信号的阻塞后，只会向进程传送一次SIGQUIT。从中可以看出在此系统上没有对信号进行排队。

## 10.13 sigaction函数

sigaction函数的功能是检査或修改与指定信号相关联的处理动作（或同时执行这两种操作）。此函数取代了LINUX/UNIX早期版本使用的signal函数。在本节末尾用sigaction函数实现signal。

|  |
| --- |
| #include <signal.h>  int sigaction (int signo, const struct sigaction \* restrict act, struct sigaction \*restrict oact);  返回值：若成功则返回0，若出错则返回-1 |

其中，参数是要检测或修改其具体动作的信号编号。若指针非空，则要修改其动作。如果指针非空，则系统经由指针返回该信号的上一个动作。此函数使用下列结构：

|  |
| --- |
|  |

当更改信号动作时，如果sa\_handler字段包含一个信号捕捉函数的地址（与常量 SIG\_IGN或SIG\_DFL相对），则sa\_mask字段说明了一个信号集，在调用该信号捕捉函数之前，这一信号集要加到进程的信号屏蔽字中。仅当从信号捕捉函数返回时再将进程的信号屏蔽字复位为原先值。这样，在调用信号处理程序时就能阻塞某些信号。在信号处理程序被调用时，操作系统建立的新信号屏蔽字包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。回忆前面章节，若同一种信号多次发生，通常并不将它们排队，所以如果在某种信号被阻塞时它发生了五次，那么对这种信号解除阻塞后，其信号处理函数通常只会被调用一次。

一旦对给定的信号设置了一个动作，那么在调用sigaction显式地改变它之前，该设置就一直有效。这种处理方式与早期的不可靠信号机制不同，而符合了POSIX.1的要求。

act结构的sa\_ flags字段指定对信号进行处理的各个选项。下表详细列出了这些选项的 意义。

|  |  |
| --- | --- |
| 处理每个倍号的选项标志(sa\_flags) | |
| SA\_INTERRUPT | 由此信号中断的系统调用不会自动重启动 |
| SA\_NOCLDSTOP | 若signo是SIGCHLD，当子进程停止时（作业控制），不产生此信号。当子进程终止时，仍旧产生此信号。 |
| SA\_NOCLDWAIT | 若signo是SIGCHLD，则当调用进程的子进程终止时，不创建僵死进程。若调用进程在后面调用则调用进程阻塞，直到其所有子进程都终止，此时返回-1，并 将errno设置为ECHILD |
| SA\_NODEFER | 当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号（除非sa\_mask包括了此信号）。注意，此种类型的操作对应于早期的不可靠信号 |
| SA\_ONSTACK | 若用sinalstack(2)声明了一替换栈，则将此信号递送给替换栈上的进程 |
| SA\_RESETHAND | 在此信号捕捉函数的入口处，将此信号的处理方式复位为SIG\_DFL，并清除SA\_SIGINFO标志。注意，此种类型的信号对应于早期的不可靠信号。但是，不能自动复位SIGILL和SIGTRAP这两个信号的配置。设置此标志使sigaction的行为如同SA\_NODEFER标志也设置了一样 |
| SA\_RESTART | 由此信号中断的系统调用会自动重启动 |
| SA\_SIGINFO | 此选项对信号处理程序提供了附加信息：一个指向siginfo结构的指针以及一个指向进程上下文标识符的指针 |

sa\_sigaction字段是一个替代的信号处理程序，当在sigaction结构中使用了SA\_SIGINFO标志时，使用该信号处理程序。对于sa\_sigaction字段和sa\_handler字段这两者，其实现可能使用同一存储区，所以应用程序只能一次使用这两个字段中的一个。

通常，按下列方式调用信号处理程序：

void handler (int signo);

但是，如果设置了SA\_SIGINFO标志，那么按下列方式调用信号处理程序：

void handler (int signo f siginfo\_t \*info, void \* context);

siginfo\_t结构包含了信号产生原因的有关信息。该结构的大致样式如下所示。

|  |
| --- |
|  |

下表列出了各种信号的si\_code值，注意，实现可定义附加的代码值。

|  |
| --- |
| siginfo\_t代码值  表 10-6 siginfo\_t 代码值  信号 代码 原因  ILL—ILLOPC 非法操作码  ILL一ILLOPN 非法操作数  ILL\_ILLADR 非法地址模式  SIGILL ILL\_ILLTRP 非法陷入  ILL\_PRVOPC 特权操作码  ILL一PRVREG 特权寄存器  ILL\_COPROC 协处理器出错  ILL—BADSTK 内部栈出错  FPE\_INTDIV 整数除以0  FPE一INTOVF 整数溢出  FPE\_FLTDIV 浮点除以0  SIGFPE FPE\_FLTOVF 浮点上溢  FPE—FLTUND 浮点下溢  FPE-FLTRES 浮点不精确结果  FPE\_FLTINV 无效的浮点运算  FPE一FLTSUB 下标越界  SIGSEGV SEGV\_MAPERR 地址未映射到对象  SEGV 一 ACCERR 对于映射对象的无效权限  BUS一ADRALN 无效的地址对齐  SIGBUS BUS-ADRERR 不存在的物理地址  BUS一OBJERR 对象特有的硬件出错  SIGTRAP TRAP—BRKPT 进程断点陷入  TRAP-TRACE 进程跟踪陷入  CLD-EXITED 子进程已终止  CLD一KILLED 子进程已异常终止(无core)  SIGCHLD CLD\_DUMPED 子进程已异常终止(有core)  CLD一TRAPPED 被跟踪的子进程已陷入  CLD\_STOPPED 子进程已停止  CLD\_CONTINUED 停止的子进程已继续  POLL-IN 数据可读  POLL\_OUT 数据可写  SIGPOLL POLL-MSG 输入消息可用  POLL\_ERR i/o出错  POLL一PRI 高优先级消息可用  POLL—HUP 设备断开连接  SI\_USER kill发送的信号  SI\_QUEUE sigqueue发送的信号(实时扩展)  Any SI\_TIMER timer\_settime设置的计时器超时(实时扩展)  SI\_ASYNCIO 异步i/o请求完成(实时扩展)  SI\_MESGQ 一条消息到达消息队列(实时扩展) |

若信号是SIGCHLD，则将设置si\_pid、si\_status和si\_uid字段。若信号是SIGILL或SIGSEGV，则si\_addr包含造成故障的根源地址，尽管该地址可能并不准确。若信号是SIGPOLL，那么si\_band字段将包含STREAMS消息的优先级段(priority band)，该消息产生POLL\_IN、POLL\_OUT或POLL\_MSG事件。si\_errno字段包含错误编号，它对应于引发信号产生的条件，并由实现定义。

信号处理程序的contert参数是无类型指针，它可被强制转换为ucntext\_t结构类型，用于标识信号传递时进程的上下文。

现在用sigaction实现signal函数。很多平台都是这样做的。另一方面，有些系统支持旧的不可靠信号语义signal函数。其目的是实现二进制向后兼容，除非明确要求旧的不可靠语义（为了向后兼容），否则应当使用下面的signal实现，或者直接调用sigaction （可以在调用sigaction时指定SA\_RESETHAND 和SA\_NODEFER选项以实现旧语义的signal函数）。

下面是signal函数的源代码。

|  |
| --- |
|  |

注意，必须用sigemptyset函数初始化act结构的sa\_mask成员。不能保证：

act.sa\_mask =0;

会做同样的事情。

对除SIGALRM以外的所有信号，我们都有意尝试设置SA\_RESTART标志，于是被这些信号中断的系统调用都能重启动。不希望重启动由SIGALRM信号中断的系统调用的原因是：我们希望对I/O操作可以设置时间限制。

某些早期系统定义了SA\_INTERRUPT标志。这些系统的默认方式是重新启动被中断的系统调用，而指定此标志则使系统调用被中断后不再重启动。Linux定义SA\_INTERRUPT标志，以便与使用该标志的应用程序兼容。但是，如若信号处理程序是用sigaction设置的，那么其默认方式是不重新启动系统调用。Single UNIX Specification的XSI扩展规定，除非说明了SA\_RESTART标志，否则sigaction函数不再重启动被中断的系统调用。

下面是signal函数的另一种版本，它力图阻止任何被中断的系统调用重启动。

|  |
| --- |
|  |

如果系统定义了SA\_INTERRUPT标志，那么为了提高可移植性，我们在sa\_flags中增加该标志，这样也就阻止了被中断的系统调用重启动。

## 10.14 sigset jmp和siglongjmp 函数

前面说明了用于非局部转移的set jmp和longjmp函数。在信号处理程序中经常调用longjmp函数以返回到程序的主循环中，而不是从该处理程序返回。前面的程序中中已经出现了这种情况。

但是，调用longjmp有一个问题。当捕捉到一个信号时，进入信号捕捉函数，此时当前信号被自动地加到进程的信号屏蔽字中。这阻止了后来产生的这种信号中断该信号处理程序。如果用long jmp跳出信号处理程序，那么，对此进程的信号屏蔽字会发生什么呢？

为了允许两种形式的行为并存，POSIX.1并没有说明setjmp和longjmp对信号屏蔽字的作用，而是定义了两个新函数sigsetjmp和siglongjmp。在信号处理程序中进行非局部转移时应当使用这两个函数。

|  |
| --- |
| # include <setjmp.h>  int sigsetjmp (sigjmp—buf env, int savemask);  返回值：若直接调用则返回0，若从siglongjmp调用返冋则返回非0值  void siglongjmp (sigjmp\_buf env, int val); |

这两个函数与set jmp和long jmp之间的唯一区别是sigset jmp增加了一个参数。如果savemask非0，则sigsetjmp在env中保存进程的当前信号屏蔽字。调用siglongjmp时，如果带非0 savemask从的sigsetjmp调用已经保存env，则siglongjmp从其中恢复保存的信号屏蔽字。

下面程序演示了在信号处理程序被调用时，系统所设置的信号屏蔽字如何自动地包括刚被捕捉到的信号。该程序也通过实例说明了如何使用sigsetjmp和siglongjmp函数。

|  |
| --- |
|  |

此程序演示了另一种技术，只要在信号处理程序中调用siglongjmp，就应使用这种技术。仅在调用sigsetjmp之后才将变量canjump设置为非0值。在信号处理程序中检测此变量，仅当它为非0值时才调用siglongjmp。这提供了一种保护机制，使得在jmpbuf（跳转缓冲）尚未由sigset jmp初始化时，防止调用信号处理程序（在本程序中，调用siglongjmp之后程序很快就结束，但是在较大的程序中，在调用siglongjmp之后的一段较长时间内，信号处理程序可能仍旧被设置）。在一般的C代码中（不是信号处理程序），对于longjmp并不需要这种保护措施。但是，因为信号可能在任何时候发生，所以在信号处理程序中，需要这种保护措施。

在程序中使用了数据类型sig\_atomic\_t，这是由ISO C标准定义的变量类型，在写这种类型的变量时不会被中断。它意味着在具有虚拟存储器的系统上这种变量不会跨越页边界，可以用一条机器指令对其进行访问。这种类型的变量总是包括ISO类型修饰符volatile，其原因是：该变量将由两个不同的控制线程——main函数和异步执行的信号处理程序访问。

下图显示了此程序的执行时间顺序。

|  |
| --- |
|  |
| 处理两个信号的实例程序的时间顺序 |

可将图10-1分成三部分：左面部分(对应于main)、中间部分(sig\_usrl)和右面部分(sig\_alrm)。在进程执行左面部分时，信号屏蔽字是0 （没有信号是阻塞的）。而执行中间部

分时，其信号屏蔽字是SIGUSR1。执行右面部分时，信号屏蔽字是SIGUSR1 | SIGALRM。

执行上面的程序，得到下面的输出：

|  |
| --- |
| $ ./a.out &  starting main：  [1] 531  $ kill -USR1 531  starting sig一 usrl: SIGUSR1  $ in sig\_alrm： SIGUSR1 SIGALRM  finishing sig\_usrl： SIGUSR1  ending main：  [1] + Done ./a.out & |

该输出与我们所期望的相同：当调用一个信号处理程序时，被捕捉到的信号加到进程的当前信号屏蔽字中。当从信号处理程序返回时，恢复原来的屏蔽字。另外，siglongjmp恢复了由sigset jmp保存的信号屏蔽字。

如果在Linux 中将上面程序中的sigsetjmp和siglongjmp分别替换成setjmp 和longjmp则最后一行输出变成：

ending main： SIGUSR1

这意味着在调之后执行main函数时，其SIGUSR1是阻塞的。这多半不是我们所希望的。

## 10.15 sigsuspend函数

上面已经说明，更改进程的信号屏蔽字可以阻塞所选择的信号，或解除对它们的阻塞。使用这种技术可以保护不希望由信号中断的代码临界区。如果希望对一个信号解除阻塞，然后pause以等待以前被阻塞的信号发生，则又将如何呢？假定信号是SIGINT,实现这一点的一种不正确的方法是：

|  |
| --- |
|  |

如果在信号阻塞时将其发送给进程，那么该信号的传递就被推迟直到对它解除了阻塞。对应用程序而言，该信号好像发生在解除对SIGINT的阻塞和pause之间（取决于内核如何实现信号。）。如果发生了这种情况，或者如果在解除阻塞时刻和pause之间确实发生了信号，那么就产生了问题。因为我们可能不会再见到该信号，所以从这种意义上而言，在此时间窗口中发生的信号丢失了，这样就使得pause永远阻塞。这是早期的不可靠信号机制的另一个问题。

为了纠正此问题，需要在一个原子操作中先恢复信号屏蔽字，然后使进程休眠。这种功能是由sigsuspend函数提供的。

|  |
| --- |
| #include <signal.h>  int sigsuspend (const sigset\_t \*sigmask)；  返回值：-1，并将errno设置为EINTR |

将进程的信号屏蔽字设置为由sigmask指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则sigsuspend返回，并且将该进程的信号屏蔽字设置为调用sigsuspend之前的值。

注意，此函数没有成功返回值。如果它返回到调用者，则总是返回-1，并将errno设置为 EINTR （表示一个被中断的系统调用）。

下面程序显示了保护临界区，使其不被特定信号中断的正确方法。

|  |
| --- |
|  |

注意，当sigsuspend返回时，它将信号屏蔽字设置为调用它之前的值。在本例中，SIGINT信号将被阻塞。因此将信号屏蔽字复位为早先保存的值(oldmask)。运行程序得到下面的输出：

|  |
| --- |
|  |

在调用sigsuspend时，将SIGUSR1信号加到了进程信号屏蔽字中，所以当运行该信号处理程序时，我们得知信号屏蔽字已经改变了。从中可见，在sigsuspend返回时，它将信号屏蔽字恢复为调用它之前的值。

sigsuspend的另一种应用是等待一个信号处理程序设置一个全局变量。下面程序用于捕捉中断信号和退出信号，但是希望仅当捕捉到退出信号时，才唤醒主例程。

|  |
| --- |
|  |

此程序的示例输出是：

|  |
| --- |
|  |

可以用信号实现父、子进程之间的同步，这是信号应用的另一个实例。下面程序包含了前面提到的五个函数的实现，它们是：TELL\_WAIT、TELL\_PARENT、TELL\_CHILD、

WAIT\_PARENT 和 WAIT\_CHILD。

|  |
| --- |
|  |

其中使用了两个用户定义的信号：SIGUSR1由父进程发送给子进程，SIGUSR2由子进程发送给父进程。后面还会给出使用管道的这五个函数的另一种实现。

如果在等待信号发生时希望去休眠，则使用sigsuspend函数是非常适当的（正如前面两个例子中所示），但是如果在等待信号期间希望调用其他系统函数，那么将会怎样呢？不幸的是，在单线程环境下对此问题没有妥善的解决方法。如果可以使用多线程，则可专门安排一个线程处理信号。

如果不使用线程，那么我们能尽力做到最好的是，当信号发生时，在信号捕捉程序中对一个全局变量置1。例如，若我们捕捉SIGINT和SIGALRM这两种信号，并用signal\_intr函数设置这两个信号的处理程序，使得它们中断任一被阻塞的低速系统调用。当进程阻塞在select函数调用等待低速设备输入时，很可能发生这两种信号（如果设置闹钟以阻止永远等待输入，那么对于SIGALRM信号，这是特别真实的）。处理这种问题的代码类似于下面所示：

|  |
| --- |
|  |

在调用select之前测试各全局标志，如果select返回一个中断的系统调用错误，则再次进行测试。如果在前两个if语句和后随的select调用之间捕捉到两个信号中的任意一个，则问题就发生了。正如代码中的注释所指出的，在此处发生的信号丢失了。调用信号处理程序，它们设置了相应的全局变量，但是select决不会返回(除非某些数据已准备好可读)。

我们希望按顺序执行下列操作步骤：

(1)阻塞SIGINT和SIGALRM。

(2)测试两个全局变量以判别是否发生了一个信号，如果已发生则对此进行处理。

(3)调用select （或任何其他系统调用，例如read）并解除对这两个信号的阻塞，这两个操作应当是一个原子操作。

仅当第(3)步是pause操作时，sigsuspend函数才能帮助我们。

## 10.16 abort函数

前面已提及abort函数的功能是使异常程序终止。

|  |
| --- |
| #include <stdlib.h>  void abort(void);  此函数不返回 |

此函数将SIGABRT信号发送给调用进程（进程不应忽略此信号）。ISO C规定，调用abort将向主机环境递送一个未成功终止的通知，其方法是调用raise (SIGABRT)函数。

ISO C要求若捕捉到此信号而且相应信号处理程序返回，abort仍不会返回到其调用者。如果捕捉到此信号，则信号处理程序不能返回的唯一方法是它调用exit、\_exit、\_Exit、 longjmp或siglongjmp。POSIX.1也说明abort并不理会进程对此信号的阻塞和忽略。

让进程捕捉SIGABRT的意图是：在进程终止之前由其执行所需的清理操作。如果进程并不在信号处理程序中终止自己，POSIX.1声明当信号处理程序返回时，abort终止该进程。

ISO C针对此函数的规范将下列问题留由实现决定：是否要冲洗输出流以及是否要删除临时文件？POSIX.1的要求则更进一步，它要求如果abort调用终止进程，则它对所有打开标准I/O流的效果应当与进程终止前对每个流调用fclose相同。

下面的abort函数是按POSIX.1说明实现的。

|  |
| --- |
|  |

首先查看是否将执行默认动作，若是则冲洗所有标准I/O流。这并不等价于对所有打开的流调用fclose（因为只冲洗，并不关闭它们），但是当进程终止时，系统会关闭所有打开的文件。如果进程捕捉此信号并返回，那么因为进程可能产生了更多的输出，所以再一次冲洗所有的流。不进行冲洗处理的唯一条件是如果进程捕捉此信号，然后调用\_exit或\_Exit。在这种情况下，内存中任何未冲洗的标准I/O缓冲区都被丢弃。我们假定捕捉此信号，而且\_exit或\_Exit的调用者并不想要冲洗缓冲区。

前面说过，如果调用kill使其为调用者产生信号，并且如果该信号是不被阻塞的，则在kill返回前该信号（或某个未决、未阻塞的信号）就被传送给了该进程。我们阻塞除SIGABRT之外的所有信号，这样就可知如果对kill的调用返回了，则该进程一定巳捕捉到该信号，并且也从该信号处理程序返回。

## 10.17 system 函数

前面已经有了一个system函数的实现，但是该版本并不执行任何信号处理。POSIX.1要求system忽略SIGINT和SIGQUIT，阻塞SIGCHLD。在给出一个正确地处理这些信号的一个版本之前，先说明为什么要考虑信号处理。

下面程序使用前面章节中的system版本，用其调用ed(l)编辑器。ed很久以来就是LINUX/UNIX的组成部分。在这里使用它的原因是：它是捕捉中断和退出信号的交互式程序。若从shell调用ed，并键入中断字符，则它捕捉中断信号并打印问号。它还将对退出符的处理方式设置为忽略。

|  |
| --- |
|  |

上面程序用来捕捉SIGCHLD和SIGINT信号，若调用它则可得到下面结果：

|  |
| --- |
|  |

当编辑器终止时，系统向父进程（a. out进程）发送SIGCHLD信号。父进程捕捉它，然后从信号处理程序返回。但是若父进程正在捕捉SIGCHLD信号（因为它创建了子进程，所以应当这样做以便了解它的子进程在何时终止），那么正在执行system函数时，应当阻塞对父进程递送SIGCHLD信号。实际上，这就是POSIX.1所说明的。否则，当system创建的子进程结束时，systern的调用者可能错误地认为，它自己的一个子进程结束了。于是，调用者将会调用一种wait函数以获得子进程的终止状态，这样就阻止了system函数获得子进程的终止状态，并将其作为它的返回值。

如果再次执行该程序，在这次运行时将一个中断信号传送给编辑器，则可得：

|  |
| --- |
|  |

回忆进程关系可知，键入中断字符可使中断信号传送给前台进程组中的所有进程。下图显示了编辑程序正在运行时的各个进程的关系。

|  |
| --- |
|  |
| 程序运行时的前台和后台进程组 |

在这一实例中，SIGINT被送给三个前台进程（shell进程忽略此信号）。从输出中可见，a .out进程和ed进程捕捉该信号。但是，当用sys tem运行另一个程序（例如ed）时，不应使父、子进程两者都捕捉终端产生的两个信号：中断和退出。这两个信号只应送给正在运行的程序：子进程。因为由system执行的命令可能是交互式命令（如本例中的ed程序），以及因为system的调用者在程序执行时放弃了控制，等待该执行程序的结束，所以system的调用者就不应接收这两个终端产生的信号。这就是为什么POSIX.1规定system的调用者应当忽略这两个信号的原因。

下面是system函数的另一个实现，它进行了所要求的信号处理。

|  |
| --- |
|  |

如果链接前面程序与system函数的这一实现，那么所产生的二进制代码与上一个有缺陷的程序相比较，存在如下差别：

(1) 当我们键入中断或退出字符时，不向调用进程发送信号。

(2) 当ed命令终止时，不向调用进程发送SIGCHLD信号。作为替代，在程序末尾的sigprocmask调用对SIGCHLD信号解除阻塞之前，SIGCHLD信号一直被阻塞。而对sigprocmask函数的这一次调用是在system函数调用waitpid取到了子进程的终止状态之后。

很多较早的书籍中使用下列程序段，它忽略中断和退出信号：

|  |
| --- |
|  |

这段代码的问题是：在fork之后不能保证父进程还是子进程先运行。如果子进程先运行，父进程在一段时间后再运行，那么在父进程将中断信号的配置更改为忽略之前，就可能产生这种信号。由于这种原因，上面程序在fork之前就改变对该信号的配置。

注意，子进程在调用execl之前要先恢复这两个信号的配置。如同进程操作中所说明的一样，这就允许在调用者配置的基础上，execl可将它们的配置更改为默认值。

**system的返回值**

注意system的返回值，它是shell的终止状态，但shell的终止状态并不总是执行命令字符串进程的终止状态。前面程序中有一些例子，其结果正是我们所期望的。如果执行一条如date那样的简单命令，其终止状态是0。执行shell命令exit 44,则得终止状态44。在信号方面又如何呢？

运行进程操作中的程序，并向正在执行的命令发送一些信号：

|  |
| --- |
|  |

当用中断信号终止sleep时，pr\_exit函数认为它正常终止。当用退出键杀死sleep进程时，会发生同样的事情。终止状态130、131又是怎样得到的呢？原来Bourne shell有一个在其文档中没有说清楚的特性，其终止状态是128加上一个信号编号，该信号终止了正在执行的命令。用交互方式使用shell可以看到这一点。

|  |
| --- |
|  |

在所使用的系统中，SIGINT的值为2，SIGQUIT的值为3，于是给出shell终止状态130、131。再试一个类似的例子，这一次将一个信号直接送给shell,然后观察system返回什么：

|  |
| --- |
|  |

从中可见仅当shell本身异常终止时，system的返回值才报告一个异常终止。

在编写使用system函数的程序时，一定要正确地解释返回值。如果直接调用fork、exec和wait，则终止状态与调用system是不同的。

## 10.18 sleep函数

在本书的很多例子中都已使用了sheep函数，再前面的程序中有两个有缺陷的实现。

|  |
| --- |
| #include <unistd.h>  unsigned int sleep (unsigned int seconds);  返回值：0或未休眠够的秒数 |

此函数使调用进程被挂起，直到满足以下条件之一：

(1)已经过了seconds所指定的墙上时钟时间。

(2)调用进程捕捉到一个信号并从信号处理程序返回。

如同alarm信号一样，由于其他系统活动，实际返回时间比所要求的会迟一些。

在第1种情形中，返回值是0。当由于捕捉到某个信号sleep提早返回时（第2种情形），返回值是未睡够的秒数（所要求的时间减去实际休眠时间）。

尽管sleep可以用alarm函数实现，但这并不是必需的。如果使用alarm，则这两个函数之间可能交互作用。POSIX.1标准对这些交互作用并未做任何说明。例如，若先调用alarm(l0)，过了3秒后又调用sleep(5)，那么将如何呢？sleep将在5秒后返回（假定在这段时间内没有捕捉到另一个信号），但是否在2秒后又产生另一个SIGALRM信号呢？这些细节依赖于实现。

下列程序是一个POSIX.1 sleep函数的实现。此函数是前面程序的修改版，它可靠地处理信号，避免了早期实现中的竞争条件，但是仍未处理与以前设置的闹钟的交互作用，正如前面提到的，POSIX.1并未显式地定义这些交互作用。

|  |
| --- |
|  |

与前面的程序相比，为了可靠地实现sleep，上面的程序代码比较长。程序中没有使用任何形式的非局部转移（如前面的程序为了避免在alarm和pause之间的竞争条件所做的那样），所以对处理SIGALRM信号期间可能执行的其他信号处理程序没有任何影响。

## 10.19 作业控制信号

再前面的信号列表中，POSIX.1认为有6个与作业控制有关：

SIGCHLD子进程已停止或终止。

SIGCONT如果进程已停止，则使其继续运行。

SIGSTOP停止信号(不能被捕捉或忽略)。

SIGTSTP交互式停止信号。

SIGTTIN后台进程组成员读控制终端。

SIGTTOU后台进程组成员写到控制终端。

除SIGCHLD以外，大多数应用程序并不处理这些信号：交互式Shell则通常做处理这些信号的所有工作。当键入挂起字符（通常是Ctrl+Z）时，SIGTSTP被送至前台进程组的所有进程。当我们通知shell在前台或后台恢复运行一个作业时，shell向该作业中的所有进程发送SIGCONT信号。与此类似，如果向一个进程递送了SIGTTIN或SIGTTOU信号，则根据系统默认的方式停止此进程，作业控制shell了解到这一点后就通知我们。

一个例外是管理终端的进程——例如，vi(l)编辑器。当用户要挂起它时，它需要能了解到这一点，这样就能将终端状态恢复到vi启动时的情况。另外，当在前台恢复它时，它需要将终端状态设置回它所希望的状态，并需要重新绘制终端屏幕。可以在下面的例子中观察到与vi类似的程序是如何处理这种情况的。

在作业控制信号间有某种交互作用。当对一个进程产生四种停止信号(SIGTSTP、SIGSTOP、SIGTTIN或SIGTT0U)中的任意一种时，对该进程的任一未决SIGCONT信号就会被丢弃。与此类似，当对一个进程产生SIGCONT信号时，对同一进程的任一未决停止信号将被丢弃。

注意，如果进程是停止的，SIGCONT的默认动作是继续运行该进程，否则忽略此信号。通常，对该信号无需做任何事情。当对一个停止的进程产生一个SIGC0NT信号时，该进程就继续运行，即使该信号是被阻塞或忽略的也是如此。

下面程序演示了当一个程序处理作业控制时所使用的规范代码序列。该程序只是将其标准输入复制到其标准输出，而在信号处理程序中以注释形式给出了管理屏幕的程序所执行的典型操作。

|  |
| --- |
|  |

当上面程序启动时，仅当SIGTSTP信号的配置是SIG\_DFL,它才安排捕捉该信号。其理由是：当此程序由不支持作业控制的shell（例如/bin/sh）启动时，此信号的配置应当设置为SIG\_IGN。实际上，shell并不显式地忽略此信号，而是由init将这三个作业控制信号SIGTSTP、SIGTTIN和SIGTTOU设置为SIG\_IGN。然后，这种配置由所有登录shell继承。只有作业控制shell才应将这三个信号复位为SIG\_DFL。

当键入挂起字符时，进程接到SIGTSTP信号，然后调用该信号处理程序。此时，应当进行与终端有关的处理：将光标移到左下角，恢复终端工作方式等等。在将SIGTSTP复位为默认值（停止该进程），并且解除了对此信号的阻塞之后，进程向自己发送同一信号SIGTSTP。因为正在处理SIGTSTP信号，而在捕捉该信号期间系统会自动阻塞它，所以应当解除对此信号的阻塞。到达这一点时，系统停止该进程。仅当某个进程（通常是正响应一个交互式fg命令的作业控制 shell）向该进程发送一个SIGC0NT信号时，该进程才继续。我们不捕捉SIGCONT信号。该信号的默认配置是继续运行停止的进程，当此发生时，此程序如同从kill函数返回一样继续运行。当此程序继续运行时，将SIGTSTP信号复位为捕捉，并且做我们所希望做的终端处理（例如重新绘制屏幕）。

## 10.20其他特征

本节介绍某些依赖于实现的信号的其他特征。

1. 信号名字

某些系统提供数组

|  |
| --- |
| extern char \*sys\_siglist[]; |

数组下标是信号编号，给出一个指向信号字符串名字的指针。

这些系统通常也提供函数psignal。

|  |
| --- |
| #include <signal.h>  void psignal (int signo, const char \*msg); |

字符串msg（通常是程序名）输出到标准出错文件，后接一个冒号和一个空格，再接着对该信号的说明，最后是一个换行符。该函数类似于perror。

另一个常用函数是strsignal。它类似于strerror。

|  |
| --- |
| #include <string.h>  char \* strsignal (int signo)；  返回值：指向描述该信号的字符串的指针 |

给出一个信号编号，strsignal将返回说明该信号的字符串。应用程序可用该字符串打印关于接收到信号的出错消息。

10.12小结

信号用于大多数复杂的应用程序中。理解进行信号处理的原因和方式对于髙级LINUX/UNIX编程极其重要。本章对LINUX/UNIX信号进行了详细而且比较深入的介绍。首先说明了早期信号实施的问题以及它们是如何显现出来的。然后介绍了POSIX.1的可靠信号概念以及所有相关的函数。在此基础上接着提供了abort、system和sleep函数的POSIX.1实现。最后以观察分析作业控制信号以及信号名和信号编号之间的转换结束。

# 第13章 守护进程

## 13.1引言

守护进程也称精灵进程（daemon）是生存期较长的一种进程。它们常常在系统自举时启动，仅在系统关闭时才终止。因为它们没有控制终端，所以说它们是在后台运行的。LINUX/UNIX系统有很多守护进程，它们执行日常事务活动。

本章说明守护进程的结构，以及如何编写守护进程程序。因为守护进程没有控制终端，我们需要了解在出现问题时，守护进程如何报告出错情况。

## 13.2守护进程的特征

先来查看一些常用的系统守护进程，以及它们怎样和进程组、控制终端和会话等概念相关联。ps(l)命令打印系统中各个进程的状态。为了解本节讨论中所需的信息，在LINUX系统下执行：

ps -axj

选项-a显示由其他用户所拥有的进程的状态。-X显示没有控制终端的进程状态。-j显示与作业有关的信息：会话ID、进程组ID、控制终端以及终端进程组ID。ps的输出大致是：

|  |
| --- |
|  |

其中，已移去了一些我们并无兴趣的列，如累计CPU时间。按照顺序，各列标题的意义是:

父进程ID、进程ID、进程组ID、会话ID、终端名称、终端进程组ID（与该控制终端相关的前台进程组）、用户ID以及命令字符串。

系统进程依赖于操作系统实现。父进程ID为0的各进程通常是内核进程，它们作为系统自举过程的一部分而启动。（init是此种进程的例外，它是内核在自举时启动的用户层命令。）内核进程是特殊的，通常存在于系统的整个生命期中。它们以超级用户特权运行，无控制终端，无命令行。

进程1通常是init，我们前面对此作过说明。它是一个系统守护进程，负责启动各运 行层次特定的系统服务。这些服务通常是在它们自己拥有的守护进程的帮助下实现的。

在Linux下，keventd守护进程为在内核中运行计划执行的函数提供进程上下文。kapmd守护进程对很多计算机系统中具有的高级电源管理提供支持。kswapd守护进程也称为页面调 出守护进程（pageout daemon）。它通过将脏页面以低速写到磁盘上从而使这些页面在需要时仍可回收使用，这种方式支持虚存子系统。

Linux内核使用两个守护进程bdflush和kupdated将高速缓存中的数据冲洗到磁盘上。当可用内存达到下限时，bdf lush守护进程将脏缓冲区从缓冲池（buffer cache）中冲洗到磁盘上。每隔一定时间间隔，kupdated守护进程将脏页面冲洗到磁盘上，以便在系统失效时减少丢失的数据。

端口映射守护进程portmap提供将RPC（Remote Procedure Call，远程过程调用）程序号映射为网络端口号的服务。sys1ogd守护进程可由帮助操作人员把系统消息记入日志的任何程序 使用。可以在一台实际的控制台上打印这些消息，也可将它们写到一个文件中。

前面已经说到inetd守护进程（xinetd）。它侦听系统网络接口，以便取得来自网络的对各种网络服务进程的请求。nfsd、lockd和rpciod守护进程提供对网络文件系统（Network File System, NFS）的支持。

cron守护进程（crond）在指定的日期和时间执行指定的命令。许多系统管理任务是由cron定期地执行相关程序而实现的。cupsd守护进程是打印假脱机进程，它处理对系统提出的所有打印请求。

注意，大多数守护进程都以超级用户（用户ID为0）特权运行。没有一个守护进程具有控制终端，其终端名设置为问号（？），终端前台进程组ID设置为-1。内核守护进程以无控制终端方式启动。用户层守护进程缺少控制终端可能是守护进程调用了setsid的结果。所有用户层守护进程都是进程组的组长进程以及会话的首进程，而且是这些进程组和会话中的唯一进程。 最后，应当引起注意的是大多数守护进程的父进程是init进程。

## 13.3编程规则

在编写守护进程程序时需遵循一些基本规则，以便防止产生并不需要的交互作用。下面先说明这些规则，然后给出一个按照这些规则编写的函数daemonize。

(1) 首先要做的是调用umask将文件模式创建屏蔽字设置为0。由继承得来的文件模式创建屏蔽字可能会拒绝设置某些权限。例如，若守护进程要创建一个组可读、写的文件，而继承的文件模式创建屏蔽字可能屏蔽了这两种权限，于是所要求的组可读、写就不能起作用。

(2) 调用fork，然后使父进程退出（exit）。这样做实现了下面几点：第一，如果该守护进程是作为一条简单shell命令启动的，那么父进程终止使得shell认为这条命令已经执行完毕，第二，子进程继承了父进程的进程组ID，但具有一个新的进程ID,这就保证了子进程不是一个进程组的组长进程。这对于下面就要做的sets id调用是必要的前提条件。

(3) 调用setsid以创建一个新会话。于是执行三个操作，使调用进程：(a)成为新会话的首进

(4) 将当前工作目录更改为根目录。从父进程处继承过来的当前工作目录可能在一个装配文件系统中。因为守护进程通常在系统再引导之前是一直存在的，所以如果守护进程的当前工作目录在一个装配文件系统中，那么该文件系统就不能被卸载。这与装配文件系统的原意不符。

另外，某些守护进程可能会把当前工作目录更改到某个指定位置，在那里做它们的工作。例如，行式打印机假脱机守护进程常常将其工作目录更改到它们的spool目录上。

(5) 关闭不再需要的文件描述符。这使守护进程不再持有从其父进程继承来的某些文件描述符（父进程可能是shell进程，或某个其他进程)。可以使用getrlimit函数来判定最高文件描述符值，并关闭直到该值的所有描述符。

(6) 某些守护进程打开/dev/null使其具有文件描述符0、1和2，这样，任何一个试图读标准输入、写标准输出或标准出错的库例程都不会产生任何效果。因为守护进程并不与终端设备相关联，所以不能在终端设备上显示其输出，也无处从交互式用户那里接收输入。即使守护进程是从交互式会话启动的，但因为守护进程是在后台运行的，所以登录会话的终止并不影响守护进程。如果其他用户在同一终端设备上登录，我们也不会在该终端上见到守护进程的输出，用户也不可期望他们在终端上的输入会由守护进程读取。

下面的函数，可由想初始化成为一个守护进程的程序调用。

|  |
| --- |
|  |

若daemonize函数由main程序调用，然后main程序进入休眠状态，那么可以用ps命令检査该守护进程的状态：

|  |
| --- |
|  |

也可用ps命令验证，没有一个活动进程的ID是3345。这意味着，我们的守护进程在一个孤儿进程组中，它不是一个会话首进程，于是没有机会分配到一个控制终端。这是在daemonize中执行第二个fork造成的。由此可以见到，此守护进程已经被正确地初始化了。

## 13.4 出错纪录

与守护进程有关的一个问题是如何处理出错消息。因为它没有控制终端，所以不能只是简单地写到标准出错上。在很多工作站上，控制台设备运行一个窗口系统，所以我们不希望所有守护进程都写到控制台设备上。我们也不希望每个守护进程将它自己的出错消息写到一个单独的文件中。对系统管理人员而言，如果要关心哪一个守护进程写到哪个记录文件中，并定期地检査这些文件，那么一定会使他感到头痛。所以，需要有一个集中的守护进程出错记录设施。

在LINUX/UNIX中syslog设施得到了广泛应用。大多数守护进程使用这一设施。图 13-1显示了syslog设施的详细组织结构。

|  |
| --- |
|  |
| syslog整体结构 |

有三种方法产生日志消息：

(1) 内核例程可以调用log函数。任何一个用户进程通过打开（open）然后读（read）/dev/klog设备就可以读取这些消息。因为我们无意编写内核例程，所以不再进一步说明此函数。

(2) 大多数用户进程（守护进程）调用syslog⑶函数以产生日志消息。我们将在下面说明其调用序列。这使消息发送至LINUX/UNIX域数据报套接字/dev/log。

(3) 在此主机上的一个用户进程，或通过TCP/IP网络连接到此主机的其他主机上的一个用户进程可将日志消息发向UDP端口514。注意，syslog函数并不产生这些UDP数据报，而是要求产生此日志消息的进程进行显式的网络编程。

通常，syslogd守护进程读取三种格式的日志消息。此守护进程在启动时读一个配置文件，一般其文件名为/etc/syslog.conf，该文件决定了不同种类的消息应送向何处。例如，紧急消息可被送向系统管理员（若已登录），并在控制台上显示，而警告消息则可记录到一个文件中。

syslog函数的原型如下。

|  |
| --- |
| #include <syslog.h>  void openlog (const char \*identf int option, int facility);  void syslog (int priority, const char \* format,...);  void closelog(void);  int setlogmask (int maskpri);  返回值：前日志记录优先级屏蔽值 |

调用openlog是可选择的。如果不调用openlog，则在第一次调用syslog时，自动调用openlog。调用closelog也是可选择的它只是关闭曾被用于与syslogd守护进程通信的描述符。

调用openlog使我们可以指定一个以ident，将它加至每则日志消息中。ident—般是程序的名称（例如，cron、inetd等）。参数是指定许多选项的位屏蔽。下表说明了可用的option (选项)。

openlog的参数

|  |  |
| --- | --- |
| option | 说明 |
| LOG\_CONS | 若日志消息不能通过UNIX域数据报送至syslogd,则将该消息写至控制台 |
| LOG\_NDELAY | 立即打开至syslogd守护进程的UNIX域数据报套接字；不要等到记录了第一条消息。通常，在记录第一条消息之前，不打开该套接字 |
| LOG\_NOWAIT | 不等待在将消息记入日志过程中可能已创建的子进程。因为在syslog调用wait时，应用程序可能已获得子进程的状态，这种处理阻止了与捕捉SIGCHLD信号的应用程序的冲突 |
| LOG\_ODELAY | 在记录第一条消息之前延迟打开至syslogd守护进程的连接 |
| LOG\_PERROR | 除将日志消息发送给syslogd外，还将它写至标准出错 |
| LOG\_PID | 每条消息都包含进程1D。此选项可供对每个请求都调用fork产生一个子进程的守护进程使用（与从不调用fork的守护进程比如syslogd对比） |

openlog的参数facility可以选下表中列举的值。设置facility参数的目的是可以让配置文件说明，来自不同设施的消息将以不同的方式进行处理。如果不调用openlog，或者以为0来调用它，那么在调用syslog时，可将设施作为priority参数的一个部分进行说明。

|  |  |
| --- | --- |
| openlog的facility参数 | |
| facility | XSI |
| LOG\_AUTH | 授权程序：login, su, getty等 |
| LOG\_AUTHPRIV | 与LOG\_AUTH相同，但写日志文件时具有权限限制 |
| LOG\_CRON | cron 和 at |
| LOG\_DAEMON | 系统守护进程：inetd, routed等 |
| LOG\_FTP | FTP守护进程（ftpd） |
| LOG\_KERN | 内核产生的消息 |
| LOG\_LOCAL | 保留由本地使用 |
| LOG\_LOCAL1 | 保留由本地使用 |
| LOG\_LOCAL2 | 保留由本地使用 |
| LOG\_LOCAL3 | 保留由本地使用 |
| LOG\_LOCAL4 | 保留由本地使用 |
| LOG\_LOCAL5 | 保留由本地使用 |
| LOG\_LOCAL6 | 保留由本地使用 |
| LOG\_LOCAL7 | 保留由本地使用 |
| LOG\_LPR | 行打印系统：ipd, lpc等 |
| LOG\_MAIL | 邮件系统 |
| LOG\_NEWS | Usenet网络新闻系统 |
| LOG\_SYSLOG | syslogd守护进程本身 |
| LOG\_USER | 来自其他用户进程的消息（默认） |

调用syslog产生一个日志消息。其priority参数facility是level的组合，它们可选取的值分别列于上表和下表中。level值按优先级从最髙到最低按序排列。

|  |  |
| --- | --- |
| syslog中的level（按序排列） | |
| level | 说明 |
| LOG\_EMERG | 紧急状态（系统不可使用）（最髙优先级） |
| LOG\_ALERT | 必须立即修复的状态 |
| LOG\_CRIT | 严重状态（例如，硬设备出错） |
| LOG\_ERR | 出错状态 |
| LOG\_WARNING | 警告状态 |
| LOG—NOTICE | 正常，但重要的状态 |
| LOG\_INFO | 信息性消息 |
| LOG\_DEBUG | 调试消息（最低优先级） |

format参数以及其他参数传至vsprintf函数以便进行格式化。在format中，每个%m都先被代换成对应于errno值的出错消息字符串（strerror）。

setlogmask函数用于设置进程的记录优先级屏蔽字。它返回调用它之前的屏蔽字。当设置了记录优先级屏蔽字时，除非消息的优先级已在记录优先级屏蔽字中设置，否则消息不被记录。注意，试图将该屏蔽字设置为0并不产生任何作用。

很多系统也提供logger(l)程序，以其作为向syslog设施发送日志消息的方法。logger命令本是为了用于以非交互方式运行但又要产生日志消息的shell脚本的。

在一个（假定的）行式打印机假脱机守护进程中，可能包含有下面的调用序列：

openlog(lpd", LOG\_PID, LOG\_LPR)；

syslog(LOG\_ERR, "open error for %s ： %m'', filename);

第一个调用将ident字符串设置为程序名，指定打印该进程ID,并且将系统默认的facility设定为行式打印机系统。对syslog的调用指定一个出错状态和一个消息字符串。如若不调用 openlog，则第二个调用的形式可能是：

syslog(LOG\_ERR | LOG\_LPR, "open error for %s： tm", filename);

其中，将prioirity参数指定为level和facility的组合。

除了syslog,很多平台还提供它的一种变体处理可变参数列表。

|  |
| --- |
| #include <syslog.h>  #include <optarg.h>  void vsyslog(int priority, const char \* format, va\_list arg); |

大多数syslog实现将使消息短时间处于队列中。如果在此段时间中到达了重复消息，那么syslog守护进程将不把它写到日记记录中，而是打印输出一条消息，类似于“上一条消窗重复了N次”。

## 13.5单实例守护进程

为了正常运作，某些守护进程实现为单实例的，也就是在任一时刻只运行该守护进程的一个副本。例如，该守护进程可能需要排它地访问一个设备。在cron守护进程情况下，如果同时有多个实例运行，那么每个副本都可能试图开始某个预定的操作，于是造成该操作的重复执行，这很可能导致出错。

如果守护进程需要访问一设备，而该设备驱动程序将阻止多次打开在/dev目录下的相应设备节点，那么这就达到了任何时刻只运行守护进程一个副本的要求。但是如果没有这种设备可供使用，那么我们就需要自行处理。

文件锁和记录锁机制是一种方法的基础，该方法用来保证一个守护进程只有一个副本在运行。（后面章节中讨论文件和记录锁。）如果每一个守护进程创建一个文件，并且在整个文 件上加上一把写锁，那就只允许创建一把这样的写锁，所以在此之后如试图再创建一把这样的写锁就将失败，以此向后续守护进程副本指明已有一个副本正在运行。

文件和记录锁提供了一种方便的互斥机制。如果守护进程在整个文件上得到一把写锁，那么在该守护进程终止时，这把锁将被自动刪除。这就简化了复原所需的处理，去除了对以前的守护进程实例需要进行清理的有关操作。

下面的函数说明了如何使用文件和记录锁以保证只运行某守护进程的一个副本。

|  |
| --- |
|  |

守护进程的毎个副本都将试图创建一个文件，并将其进程ID写到该文件中。这使管理人员易于标识该进程。如果该文件已经加了锁，那么lockfile函数将失败，errno 设置为 EACCES或EAGAIN，函数返回1，这表明该守护进程已在运行。否则将文件长度截短为0，将进程ID写入该文件，函数返回0。

我们需要将文件长度截短为0，其原因是以前守护进程实例的进程ID字符串可能长于调用此函数的当前进程的进程ID字符串。例如，若以前的守护进程的进程ID是12345，而新实例的进程ID是9999，那么将此进程ID写入文件后，在文件中留下的是99995。将文件长度截短为0就解决了此问题。

## 13.6守护进程的惯例

在LINUX/UNIX系统中，守护进程遵循下列公共惯例：

若守护进程使用锁文件，那么该文件通常存放在/var/run目录中。注意，守护进程可能需具有超级用户权限才能在此目录下创建文件。锁文件的名字通常是name.pid，其中，name是该守护进程或服务的名字。例如，cron守护进程锁文件的名字是/var/run/ crond.pid。

若守护进程支持配置选项，那么配置文件通常存放在/etc目录中。配置文件的名字通常为name.conf, 其中，name是该守护进程或服务的名字。例如，syslogd守护进程的配置文件是/etc/syslog. conf。

守护进程可用命令行启动，但通常它们是由系统初始化脚本之一（/etc/rc\*或 /etc/init.d/\*）启动的。如果在守护进程终止时，应当自动地重新启动它，则我们可在/etc/inittab中为该守护进程包括\_respawn记录项，这样，init就将重启动该守护进程。

若一守护进程有一配置文件，那么当该守护进程启动时，它读该文件，但在此之后一般就不会再査看它。若一管理员更改了配置文件，那么该守护进程可能需要被停止，然后再启动，以使配置文件的更改生效。为避免此种麻烦，某些守护进程将捕捉SIGHUP信号,当它们接收到该信号时，重读配置文件。因为守护进程并不与终端相结合，它们或者是无控制终端的会话首进程，或者是孤儿进程组的成员，所以守护进程并不期望接收SIGHUP于是，它们可以安全地重复使用它。

下面程序说明了守护进程可以重读其配置文件的一种方法。该程序使用sigwait以及多线程。

|  |
| --- |
|  |

该程序调用本章开始的daemonize以初始化守护进程。从该函数返回后，调用前面的already\_running函数以确保该守护进程只有一个副本在运行。到达这一点时，SIGHUP信号仍被忽略，所以需恢复对该信号的系统默认处理方式?否则调用sigwait的线程决不会见到该信号。

如同对多线程程序所推荐的那样，我们阻塞所有信号，然后创建一线程，由它来处理信号。该线程的唯一工作是等待SIGHUP和SIGTERM。当接收到SIGHUP信号时，该线程调用reread函数重读它的配置文件。当它接收到SIGTERM信号时，记录一消息，然后终止。

对于SIGHUP和SIGTERM的默认动作是终止进程。因为我们阻塞了这些信号，所以当对进程产生这些信号时，守护进程不会消亡，而是调用sigwait的线程在返回时将指示已接收到该信号。

Linux线程对于信号的处理方式与众不同。由于这一点，在下面程序中对信号标识合适的进程是困难的。另外，由于实现的差别，不能保证守护进程将按所期望的那样做出反应。

下面程序说明守护进程无需使用多线程也可以捕捉SIGHUP并重读其配置文件。

|  |
| --- |
|  |

在初始化守护进程后，我们为SIGHUP和SIGTERM配置信号处理程序。我们可以将重读逻辑放在信号处理程序中，也可以只在其中设置一个标志，由守护进程的主线程做所有所需的工作。

## 13.7客户进程-服务器进程模型

守护进程常常用作服务器进程。确实，我们可以称本章开始的syslogd进程为服务器进程，用户进程（客户进程）用LINUX/UNIX域数据报套接字向其发送消息。

一般而言，服务器是等待客户进程与其联系的一个进程，客户进程向它提出某种类型的服务要求。在开始的syslog结构图中，由syslogd服务器进程提供的服务是将出错消息记录到日志文件中。

客户进程和服务器进程之间的通信是单向的。客户进程向服务器进程发送服务请求，服务器进程则不向客户进程回送任何消息。在接下来有关进程通信的几章中，我们将见到大量客户进程和服务器进程之间双向通信的实例。客户进程向服务器进程发送请求，服务器进程则向客户进程回送应答。

## 13.8小结

在大多数LINUX/UNIX系统中，守护进程是一直运行的。为了初始化我们自己的守护进程，需要审慎思索并深入理解前面章节中说明过的进程之间的关系。本章开发了一个可由守护进程调用的、对其自身正确地进行初始化的函数。

因为守护进程通常没有控制终端，所以本章还讨论了守护进程记录出错消息的几种方法。我们讨论了在大多数LINUX/UNIX系统中，守护进程遵循的若干惯例，给出了几个如何实现某些惯例的实例。