第 13 章　进　　程

进程是 Linux 操作系统中最重要的概念。要掌握好 Linux 编程，离不开对进程相关概念深入细致的理解。本章从进程的基本概念入手，逐步展开，重点介绍了进程的运行环境、进程的创建及结束等内容。通过本章的学习，读者应重点掌握如下内容。

- 理解进程的基本概念，特别是进程的各个属性的含义。
- 了解进程的运行环境，特别是要加深对命令行参数、环境变量等概念的理解。
- 掌握创建进程的若干方法，能够熟练应用这些方法进行多进程编程。

13.1　进程的基本概念

简单地说，进程是指处于运行状态的程序。一个源程序经过编译、链接后，成为一个可以运行的程序。当该可执行的程序被系统加载到内存空间运行时，就称为进程。程序是静态的保存在磁盘上的代码和数据的组合，而进程是动态的概念。

13.1.1　进程的属性

进程创建后，系统内核为其分配了一系列的数据结构。这些数据结构中保存了进程的相关属性。主要的进程属性包括以下几种。

- 进程的标识符：进程创建时，内核为每个进程分配一个惟一的进程标识符。进程的标识符是一个非负整数，取值范围从 0～32767。进程 ID 是由系统循环使用的。如果当前可用进程号超过了最大值，将从 0 选择可用的整数继续循环使用。
- 进程的父进程标识符：Linux 下的全部进程组成一棵进程树。其中树根进程是 0 号进程 swapper。除根进程外，每个进程都有其对应的父进程。
- 进程的用户标识：是指运行该程序的用户 ID。当一个程序被某个用户执行而变为进程时，该用户就成为进程的用户标识。
- 进程的组标识：是指运行该程序的用户所归属的组 ID。
- 进程的有效用户标识：是指该进程运行过程中有效的用户身份。在进行文件权限许可等检查时，以该有效用户标识为依据。
- 进程的有效组标识：是指当前进程的有效组标识。在进行文件权限许可等检查时，以该有效组标识为依据。进程的用户和组相关的 4 个标识主要用于检查对文件系统的访问权限。
- 进程的进程组标识符：一个进程可以属于某个进程组。通过设置进程组，可以实现向一组进程发送信号等进程控制操作。

- 进程的会话标识符：每个进程都属于惟一的会话。在进程的属性中包含了进程的会话 ID 信息。

进程的这些属性大部分可以通过执行命令 ps 查看得到，如图 13-1 所示。

图 13-1　ps 查看进程属性

【范例 13-1】通过执行 ps 命令查看到进程的属性信息。其实现过程如示例代码 13-1 所示。

示例代码 13-1

```
1    ps –alef|more                    /*输出系统内全部进程的信息*/
```

【运行结果】在 shell 下运行上述命令，其结果如下所示（有删节）。

```
1    F S UID          PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME
     CMD
2    4 S root        2977  2976  0  76   0    -   945 wait4  09:59 pts/2    00:00:00
     bash
3    4 R root        3008  2977  0  77   0    -   586 -      10:06 pts/2    00:00:00
     ps -alef
4    …
```

【代码解析】在 ps 命令的输出中，包含了进程的属性信息。输出信息的解释如下。

- 第 1 行：ps 命令输出的头部信息，用于标明后续列出的进程属性数据的含义。
- 第 2 行：命令 bash 正处于睡眠状态。这是一个 shell 进程。
- 第 3 行：命令 ps –alef 正处于运行状态。进程的标识符是 3008，父进程标识符是 2977。控制终端是 pts/2。可以看出，该进程是第 2 行输出的进程 bash 的子进程。

Linux 支持通过编程的方式获取进程的属性，包括获取进程 ID 的 getpid，获取进程父进程 ID 的 getppid，获取进程用户标识的 getuid，获取进程有效用户标识的 geteuid 等。这些系统调用的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
__pid_t getpid (void);                  //获取当前进程的进程 ID
__pid_t getppid (void);                 //获取当前进程的父进程 ID
__pid_t getpgrp (void);                 //获取当前进程的进程组 ID
__uid_t getuid (void);                  //获取当前进程的实际用户 ID
__uid_t geteuid (void);                 //获取当前进程的有效用户 ID
__gid_t getgid (void);                  //获取当前进程的实际用户组 ID
__gid_t getegid (void);                 //获取当前进程的有效用户组 ID
__pid_t getsid (__pid_t __pid);         //获取指定进程的会话 ID
```

返回值说明如下。

- −1：调用失败，查看 errno 获取详细错误信息。
- 其他：获取到的进程属性信息。

【范例 13-2】通过编程方式获取进程的属性信息。其实现过程如示例代码 13-2 所示。

示例代码 13-2

```
1    #include <stdio.h>                                                    /*头文件*/
2    #include <unistd.h>
3    main()                                                               /*主函数*/
4    {
5        printf("process id=%d\n",getpid());                              /*进程 ID*/
6        printf("parent process id=%d\n",getppid());                      /*进程的父进程 ID*/
7        printf("process group id=%d\n",getpgrp());                       /*进程的组 ID*/
8        printf("calling process's real user id=%d\n",getuid());          /*进程的用户 ID*/
9        printf("calling process's real group id=%d\n",getgid());         /*进程的用户组 ID*/
10       printf("calling process's effective user id=%d\n",geteuid());    /*进程的有效用户 ID*/
11       printf("calling process's effective group id=%d\n",getegid());   /*进程的有效用户组 ID*/
12   }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1    process id=3176
2    parent process id=3097
3    process group id=3176
4    calling process's real user id=1000
5    calling process's real group id=100
6    calling process's effective user id=1000
7    calling process's effective group id=100
```

【代码解析】在程序的输出中，包含了进程的属性信息。从输出可以看到，进程的 ID 与进程的进程组 ID 相同，这表示该进程是这个进程组的首进程。进程的真实用户 ID 与有效用户 ID 也是相同的。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 getpid 获取进程的 ID。
- 第 6 行：调用 getppid 获取进程的父进程 ID。
- 第 7 行：调用 getpgrp 获取进程的进程组 ID。
- 第 8 行：调用 getuid 获取进程的真实用户 ID。
- 第 9 行：调用 getgid 获取进程的真实用户组 ID。
- 第 10 行：调用 geteuid 获取进程的有效用户 ID。
- 第 11 行：调用 getegid 获取进程的有效用户组 ID。

提示：通常情况下，进程的用户标识与有效用户标识是相同的，但是对于 suid 程序来说，其有效用户 ID 与用户 ID 是不同的。以常用的/usr/bin/passwd 命令为例，该命令即是一个 suid 程序。运行 passwd 命令的用户，其用户 ID 是运行用户的 ID，而其有效用户 ID 则是超级用户 root。这也是为什么普通用户能够通过运行 passwd 命令修改口令文件/etc/passwd（或者/etc/shadow）文件的原因。

13.1.2　进程的内存映像

一个可执行程序被系统加载后，成为一个进程。在系统内存映像中，进程主要包括代码段、数据段、BSS 段、堆栈段等部分。其内存映像如图 13-2 所示。

其中各元素的含义如下。

❑ 代码段：代码段是用来存放可执行文件的指令，是可执行程序在内存中的映像。对代码段的访问有严格安全检查机制，以防止在运行时被非法修改。代码段是只读的。

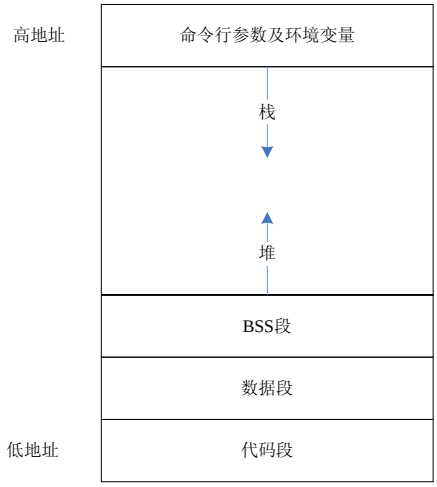

图 13-2　进程的内存映像

❑ 数据段：数据段用来存放程序中已初始化全局变量。

❑ BSS 段：BSS 段包含程序中未初始化的全局变量。

❑ 堆（heap）：堆是进程空间内的内存区，用于进程运行过程中动态分配内存。堆的大小不固定，可根据程序运行过程中的要求动态变化。当程序中调用 malloc 等函数申请内存时，新申请的内存被动态添加到堆内；当用 free 等函数释放内存时，被释放的内存从堆中被删除。

❑ 栈（stack）：栈是存放程序中的局部变量的内存区。另外，栈还被用于保存函数调用时的现场。在调用函数时，函数的参数被压入栈，由函数从栈中取出。在函数返回时，将返回值压入栈，由主调函数从栈中取出。栈是由系统自动分配的，用户程序不需要关心其分配及释放。

13.1.3　进程组

在 Linux 系统中，每个进程都惟一的归属于某个进程组。在 shell 环境中，一条 Linux 命令就形成一个进程组。这条命令可以只包含一个命令，也可以是通过管道符连接起来的若干命令。每个进程组都有一个组长进程。进程组的 ID 就是这个组长的 ID。当进程组内的所有进程都结束或者加入到其他进程组内时，该进程组就结束了。

可以通过系统调用 setpgid 修改某个进程的进程组。该函数位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int setpgid (__pid_t __pid, __pid_t __pgid);
```

（1）参数说明如下。

❑ __pid：输入参数，用于指定要修改的进程 ID。如果该参数为 0，则指当前进程 ID。

❑ __pgid：输入参数，用于指定新的进程组 ID。如果该参数为 0，则指当前进程 ID。

（2）返回值说明如下。

❑ 0：表明调用成功。

❑ −1：表明调用失败，查看 errno 可以获取详细的错误信息。

【范例 13-3】调用 setpgid 使本进程成为新进程组的组长。其实现过程如示例代码 13-3 所示。

示例代码 13-3

```
1    #include <stdio.h>                          /*头文件*/
2    #include <unistd.h>
3    main()                                      /*主函数*/
4    {
5        setpgid(0,0);                           /*设置当前进程为新的进程组的组长*/
6        sleep(10);                              /*休眠 10 秒，以供查看进程状态*/
7    }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在另外的 shell 中通过执行 ps -ao pid,pgrp,cmd 命令查看进程的进程组 ID，其结果如下所示。

```
1      PID   PGRP   CMD
2     3518   3518   ./example13_2
3     3519   3519   ps -ao pid,pgrp,cmd
```

【代码解析】从输出可以看到，进程./example13_2 的进程 ID（PID=3518）与进程的进程组 ID（PGRP=3518）相同，这表示该进程是这个进程组的首进程。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 setpgid 修改进程的进程组。该语句等价于 setpgrp()。
- 第 6 行：休眠 10 秒钟，以便于通过另外的 shell 查看进程信息。

提示：在 Linux 下还有另外一个函数 setpgrp 用于设置当前进程为进程组的组长。调用该函数后，将产生一个新的进程组，进程组的组 ID 为调用进程的 ID。也就是说，调用 setpgrp 将创建一个以调用进程为组长的新的进程组。该函数的功能可以用 setpgid(0,0)替代实现。

13.1.4　进程的会话

当用户登录一个新的 shell 环境时，一个新的会话就产生了。一个会话可以包括若干个进程组，但是这些进程组中只能有一个前台进程组，其他的为后台运行进程组。前台进程组通过其组长进程与控制终端相连接，接收来自控制终端的输入及信号。一个会话由会话 ID 来标识，会话 ID 是会话首进程的进程 ID。会话与进程组及进程的关系如图 13-3 所示。

Linux 提供了系统调用 setsid 用于产生一个新的会话。不过，调用 setsid 的进程应该保证不是某个进程组的组长进程。setsid 调用成功后，将生成一个新的会话。新会话的会话 ID 是调用进程的进程 ID。新会话中只包含一个进程组，该进程组内只包含一个进程：即调用 setsid 的进程，且该会话没有控制终端。setsid 系统调用的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
__pid_t   setsid(void);
```

返回值说明如下。

- −1：调用 setsid 失败，查看 errno 可以获取详细的错误信息。典型的错误是调用进程是某个进程组的组长，此时 setsid 将失败，错误码 errno 为 EPERM。
- 其他值：返回进程的进程组 ID。

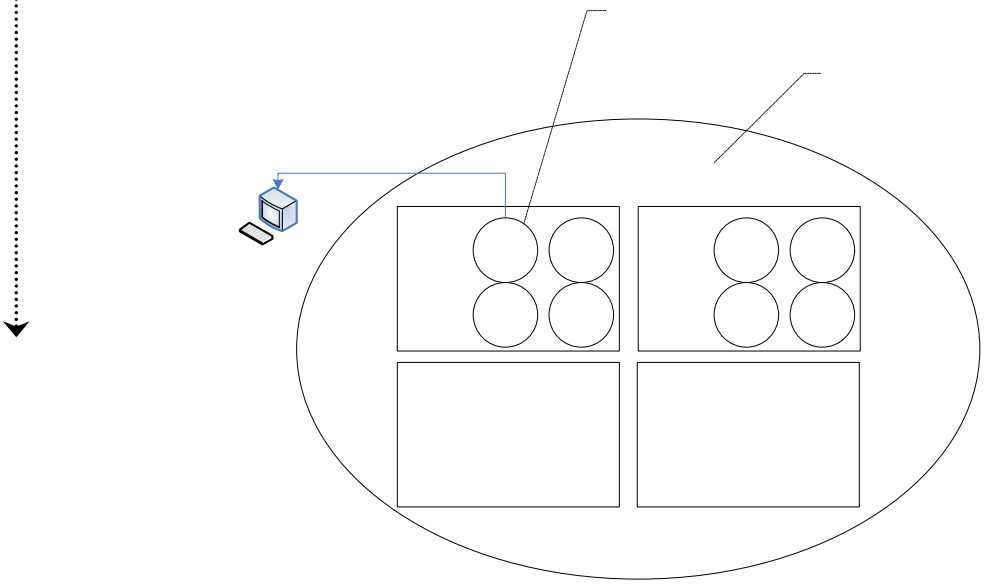

图 13-3　进程组与会话

【范例 13-4】调用 setsid 实现进程的后台运行。其实现过程如示例代码 13-4 所示。

示例代码 13-4

```
1    #include <stdio.h>                    /*头文件*/
2    #include <unistd.h>
3    main()                                /*主函数*/
4    {
5        int n;                            /*循环变量定义*/
6        __pid_t nPid;                     /*进程 ID 变量*/
7        __pid_t nGroupId;                 /*进程的组长进程 ID*/
8        if((nPid = fork()) < 0)           /*创建新的子进程*/
9        {
10           perror("fork");               /*创建子进程失败，错误处理*/
11           exit(0);
12       }
13       if(nPid != 0)                     /*父进程*/
14           exit(0);                      /*父进程退出*/
15       nGroupId = setsid();              /*产生新会话，返回新创建的进程组的组 ID*/
16       if(nGroupId == -1)                /*错误处理*/
17       {
18           perror("setsid");             /*输出错误信息*/
19           exit(0);
20       }
21       for(n=0;n<10;n++)                 /*循环休眠一段时间退出，供用户查看运行结果*/
22           sleep(3);                     /*休眠 3 秒*/
23   }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在另外的 shell 中通过执行 ps -ao pid,ppid,pgrp,session,tty,cmd 命令查看进程的信息，其结果如下所示。

```
1    3087      1  3087  3087  ?    ./example13_4
```

【代码解析】在本例的输出中，分别显示的是进程 ID、进程的父进程 ID、进程的进程组 ID、进程的会话 ID、进程的控制终端、进程的命令行。从输出可以看到：进程./example13_4 的进程 ID（PID=3087）与进程的进程组 ID（PGRP=3087）相同，这表示该进程是这个进程组的首进程；进程./example13_4 的进程 ID 与其会话 ID（SESSION=3087）相同，这表明进程 example13_4 是当前会话的首进程；进程./example13_4 无控制终端（TTY=?）。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 8～12 行：调用 fork 创建新的子进程。fork 返回值如果小于 0，表明创建子进程出错；fork 返回值如果不等于 0，表明是在父进程中，返回值是新生成的子进程的进程 ID；fork 返回值如果等于 0，表明当前是在子进程中。
- 第 13～14 行：父进程退出。一个程序执行时，将新生成一个进程组。在本句执行前，主进程是当前进程组的组长。创建子进程后，该进程组有两个进程：主进程和刚刚创建的子进程。此时，父进程退出后，进程组仍然存在（由于子进程还在）。通过调用本语句，确保子进程不是进程组的组长进程（调用 setsid 要求调用进程不能为进程组的组长进程）。
- 第 15～20 行：调用 setsid 生成新的会话。调用后，当前进程成为新会话的首进程。
- 第 21～22 行：循环调用 sleep，以方便通过另外的 shell 查看进程信息。

提示：Linux 系统的进程大体可以分为前台进程和后台进程。所谓前台进程就是运行过程中与控制终端相连接的进程，随时可以通过控制终端与前台进程进行交互，如用户登录时的 shell 进程就是前台进程。后台进程一般无控制终端，如 Linux 系统下各种守护进程就属于后台进程。

13.1.5　进程的控制终端

作为多用户、多任务的操作系统，Linux 支持多个用户同时从终端登录系统。Linux 终端类似于 Windows 环境下的远程桌面连接。用户通过终端输入请求，提交给主机运行并显示主机的运行结果。传统的 Linux 终端是由 RS232 串口通信协议的串口终端，终端与主机间的通讯通过主机的串口进行。这种串口终端数据传输速度较慢，并且传输距离有限，现在已逐渐为网络终端所代替。网络终端与主机间通过以太网相连接，数据传输速度大为提高。Linux 系统中多用户环境下终端的使用如图 13-4 所示。

Linux 系统中，每个终端设备都有一个设备文件与其相关联，这些终端设备称为 tty。在 shell 环境下，可以通过执行命令 tty 查看当前终端的名称。用户可以通过 telnet 远程登录到某个 Linux 系统，此时其实并没有真正的终端设备。这种情况下，Linux 系统将为用户自动分配一个称为“伪终端”的终端设备。伪终端的设备文件名称类似/dev/pts/???。

前面介绍的是 Linux 系统的终端环境。而在 Linux 的进程环境中，有一个称为“控制终端”的概念。所谓控制终端，就是指一个进程运行时，进程与用户进行交互的界面。一个进程从终端启动后，这个进程的运行过程就与控制终端密切相关。可以通过控制终端输入/输出，

也可以通过控制终端向进程发送信号（可以按<Ctrl>+<C>键中止程序运行）。当控制终端被关闭时，该控制终端所关联的进程将收到 SIGHUP 信号。系统对该信号的缺省处理方式就是中止进程。

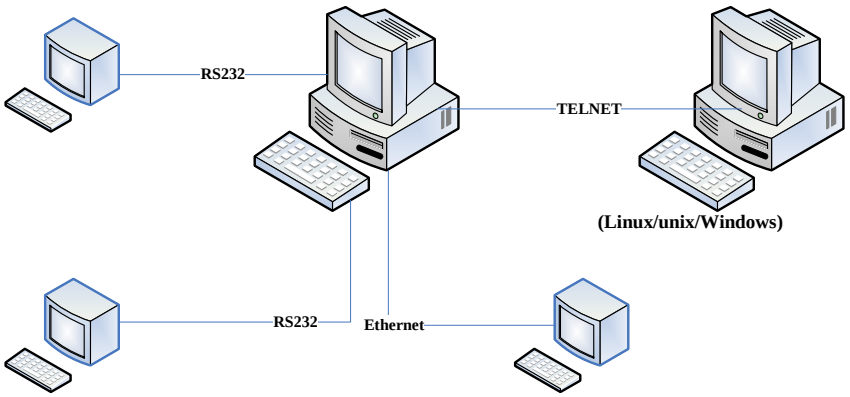

图 13-4　Linux 终端环境

提示：用户可以通过执行 shell 命令 ps –ax 查看进程的控制终端，如示例代码 13-1 所示。在 ps 的输出中，有一列名称为“TTY”的就是控制终端。如果该列中有值，表明进程是有控制终端的，否则表明进程没有控制终端。　串口终端

13.1.6　进程的状态

主机

Linux 的进程是由操作系统内核调度运行的。在调度过程中，进程的状态是不断发生变化的。这些状态主要包括可运行状态、等待状态（也称为睡眠状态）、暂停状态、僵尸状态、退出状态等，如图 13-5 所示。

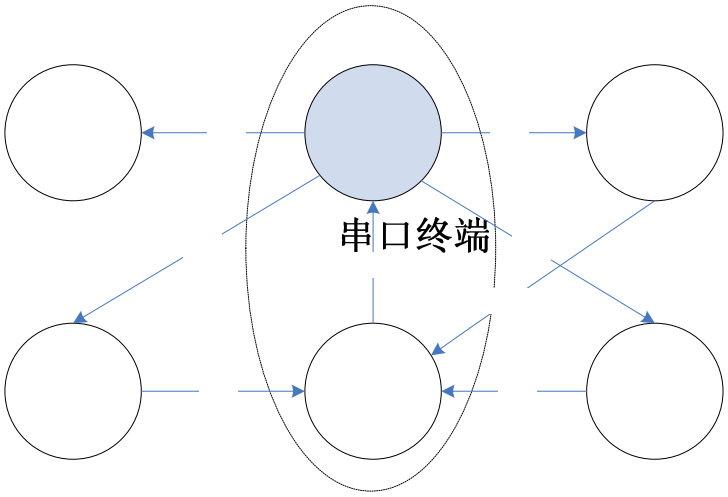

图 13-5　进程状态变化图

可运行状态又包括就绪状态和执行状态，只有处于执行状态的进程是真正占用 CPU 的进程。这些进程状态详细描述如下。

- 可运行状态（RUNNING）：该状态有两种情况，一是进程正在运行；二是处于就绪状态，只要得到 CPU 就可以立即投入运行。在第二种情况中，进程处于预备运行状态，在等待系统按照时间片轮转规则将 CPU 分配给它。
- 等待状态（SLEEPING）：表明进程正在等待某个事件发生或者等待某种资源。该状态可以分成两类：可中断的和不可中断的。处于可中断等待状态的进程，既可以被信号中断，也可以由于资源就续而被唤醒进入运行状态。而不可中断等待状态的进程在任何情况下都不可中断，只有在等待的资源准备好后方可被唤醒。
- 暂停状态（STOPPED）：进程接收到某个信号，暂时停止运行。大多数进程是由于处于调试中，才会出现该状态。
- 僵尸状态（ZOMBIE）：表示进程结束但尚未消亡的一种状态。一个进程结束运行退出时，就处于僵尸状态。进程会在退出前向其父进程发送 SIGCLD 信号（关于该信号的处理见第 14 章内容）。父进程应该调用 wait 为子进程的退出做最后的收尾工作。如果父进程未进行该工作，则子进程虽然已退出，但通过执行 ps 命令仍然可以看到该进程，其状态就是僵尸状态。在应用编程中，应尽量避免僵尸进程的出现。

在 Linux 系统下，除 ps 命令可以查看进程状态外，还有另外一个重要的进程查看工具 top。ps 命令输出的是静态的，是进程的某一时刻的信息，而 top 可以持续的动态的输出进程的信息。

【范例 13-5】通过 top 输出进程的信息。其实现过程如示例代码 13-5 所示。

示例代码 13-5

```
1    top
```

【运行结果】在 shell 下运行上述命令，其输出信息如下所示。

```
1    top - 11:02:52 up   9:15,   4 users,   load average: 0.01, 0.03, 0.00
2    Tasks:   55 total,    2 running,   53 sleeping,    0 stopped,    0 zombie
3    Cpu(s):   1.1% us,   2.3% sy,   0.0% ni, 91.7% id,   1.1% wa,   0.0% hi,   3.8% si
4    Mem:      256624k total,     163720k used,      92904k free,      43828k buffers
5    Swap:     128480k total,          0k used,     128480k free,      71432k cached
6      PID USER      PR  NI  VIRT  RES  SHR  S  %CPU %MEM    TIME+  COMMAND
7     2784 root      15   0 38000   17m   22m   S   2.0    6.9    6:59.71   X
8     2871 root      15   0  3512  1656  3276  S   1.0    0.6    3:42.27   xeyes
9     4031 root      16   0  1944  1020  1716  R   0.7    0.4    0:00.11   top
10       1 root      16   0   588   244   444   S   0.0    0.1    0:05.77   init
```

【代码解析】top 命令的输出是动态的，每间隔一定时间（可以通过-d 参数指定刷新间隔时间），该信息就会刷新一次。在本例的输出中，输出信息的详细解释如下。

- 第 1 行：当前时间（11:02:52）、系统启动时间（9:15）、当前系统登录用户数目（4 users）、平均负载（load average: 0.01, 0.03, 0.00）
- 第 2 行：当前进程的汇总信息，分别为当前进程总数（55 total）、休眠进程数（53 sleeping）、运行进程数（2 running）、僵死进程数（0 zombie）、终止进程数（0 stopped）。
- 第 3 行：当前 CPU 的消息情况，分别是用户占用（1.1%）、系统占用（2.3%）、优先进程占用（0.0%）、闲置进程占用（91.7%）。

- 第 4 行：内存状态汇总信息，分别为平均可用内存（256624k）、已用内存（163720k）、空闲内存（92904k）、缓存使用内存（43828k）。
- 第 5 行：交换区状态，分别为平均可用交换容量（128480k）、已用容量（0k）、闲置容量（128480k）、高速缓存容量（71432k）。
- 第 6 行：进程明细信息的标题，分别是进程 ID（PID）、进程的所有者用户（USER）、进程的优先级别（PR）、进程的谦让度值（NI）、进程占用的虚拟内存值（VIRT）、进程占用的物理内存值（RES）、进程使用的共享内存值（SHM）、进程的状态（S）、进程占用的 CPU 使用率（%CPU）、进程占用的物理内存的百分比（%MEM）、进程启动后占用的总的 CPU 时间（TIME+）、进程启动的启动命令行（COMMAND）。
- 第 7～10 行：系统中各进程的详细信息。该信息每隔一定时间被刷新一次。

13.1.7 进程的优先级

作为一个多任务的操作系统，Linux 操作系统中允许多个进程同时运行。但是，在 CPU 数量少于同时运行的进程数量时，是不可能实现真正同时运行的。以单 CPU 的计算机为例，如果系统中同时存在两个进程同时运行，则实际上在同一时刻只可能有一个进程占用 CPU 运行。

为实现多任务的目标，Linux 使用了一种称为“时间片轮转”的进程调度方式，为每个进程指派一定的运行时间。这个时间片通常很短，以毫秒甚至更小的时间级别为单位。系统核心依照某种规则，从大量进程中选择一个进程投入运行，其余的进程暂时等待。当正在运行的那个进程时间片用完，或进程执行完毕退出，Linux 就会重新进行进程调度，挑选下一个可用进程投入运行。由于每一个进程运行时占用的时间片很短，在用户的角度来看，就如同多个进程同时运行一样。

进程的优先级定义了进程被调度的优先顺序。优先级的数值越低，进程越是优先被调度。优先级的是由进程的优先级别（PR）和进程的谦让值（NI）两个因素联合确定的。Linux 系统内核在调度进程时，将优先级别（PR 值）和谦让值（NI）相加以确定进程的真正优先级别。对于一个进程来说，其优先级别是由父进程继承而来的，用户进程不可更改。

为方便用户修改进程运行的优先级，Linux 提供了 nice 系统调用以修改进程的谦让值。进程的谦让值在进程被创建时置为缺省值为 0。系统允许的谦让值范围为最高优先级的−20 到最低优先级的 19。通过 nice 系统调用可以修改该值。

1．nice

nice 的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int nice (int __inc);
```

（1）参数说明如下。

- __inc：输入参数，指定新的谦让值。该参数取值范围为−20～19。

（2）返回值说明如下。

- 0：调用成功。
- −1：调用失败，可以查看 errno 获取错误信息。

（3）关于使用 nice 系统调用需要注意以下几点。

只有超级用户可以在调用 nice 时指定负的谦让值。也就是说只有超级用户才可以提高进程的调度优先级别。如果不是超级用户而指定负的谦让值，则 nice 调用返回失败，errno 为 EPERM。

2．setpriority

nice 系统调用只能修改进程自身的谦让值，而另外一个系统调用 setpriority 则可以修改其他进程甚至一个进程组的谦让值。该系统调用的声明位于头文件<sys/resource.h>中，其原型如下所示。

```
#include <sys/resource.h>
int setpriority (__priority_which_t __which, id_t __who, int __prio);
```

（1）参数说明如下。

- __which：输入参数，指定设置谦让值的目标类型。setpriority 可以对 3 种目标进行谦让值设置，分别是 PRIO_PROCESS 、PRIO_PGRP 和 PRIO_USER 。PRIO_PROCESS 是为某进程设置谦让值；PRIO_PGRP 是为某进程组设置谦让值；PRIO_USER 是为某个用户的所有进程设置谦让值。
- __who：输入参数，设置谦让值的目标。对于 PRIO_PROCESS 类型的目标，该参数为进程 ID；对于 PRIO_PGRP 类型的目标，该参数为进程组 ID；对于 PRIO_USER 类型的目标，该参数为用户 ID。如果该参数为 0，对于 3 种目标类型，分别表示当前进程、当前进程组、当前用户。
- __prio：输入参数，要设置的谦让值，输入范围为−20～19。只有超级用户可以用负值调用 setpriority。

（2）返回值说明如下。

- 0：调用成功。
- −1：调用失败，可以查看 errno 获取错误信息。

进程的谦让值被修改后，如何得到进程的谦让值呢？Linux 提供了系统调用 getpriority 实现这一目的。getpriority 是与 setpriority 相对应的系统调用，其声明位于头文件<sys/resource.h>中，其原型如下所示。

```
#include <sys/resource.h>
int getpriority (__priority_which_t __which, id_t __who);
```

返回值 getpriority 系统调用比较特殊，可能返回负值，所以无法直接根据返回值确定是否调用成功。建议调用 getpriority 前置 errno 为 0，如果调用后 errno 为 0，表明成功，否则表明调用失败。

【范例 13-6】修改进程的谦让值，调用完成后输出进程的谦让值。其实现过程如示例代码 13-6 所示。

示例代码 13-6

```
1    #include <unistd.h>                          /*头文件*/
2    #include <errno.h>
3    #include <sys/resource.h>
4    main()                                       /*主函数*/
```

```
5   {
6       int nPr;                                        /*整型变量定义*/
7       if(nice(3) == -1)                               /*进程的谦让值为 3，进程的优先级降低*/
8       {
9           perror("nice");                             /*错误处理*/
10          exit(0);
11      }
12      errno = 0;                                      /*设置全局错误变量为 0*/
13      nPr = getpriority(PRIO_PROCESS,getpid());       /*获得当前进程的谦让值*/
14      if(errno != 0)                                  /*错误处理*/
15      {
16          perror("getpriority");                      /*输出错误信息*/
17          exit(0);
18      }
19      printf("priority is %d\n",nPr);                 /*输出进程的谦让值*/
20  }
```

【运行结果】在 shell 下运行上述命令，其输出信息如下所示。

```
1   priority is 3
```

【代码解析】在本例中，首先调用 nice 降低当前进程的谦让值，然后调用 getpriority 得到进程的谦让值并输出。本例中，源代码的各行解释如下。

- 第 1～3 行：头文件信息。
- 第 7～11 行：调用 nice 修改进程的谦让值。此处以 3 为参数调用 nice，实际上是降低了进程的优先级。
- 第 12 行：设置系统变量 errno 为 0。此处是由于 getpriority 返回值的特殊性而添加的，详细说明见 getpriority 的返回值说明。
- 第 13～19 行：调用 getpriority 获取当前进程的谦让值并输出。

提示：除 nice 系统调用外，也可以通过执行 nice 和 rnice 命令来修改一个进程的谦让值。nice 可以在执行一个程序时，直接指定谦让值，而 rnice 命令则可以修改一个正在运行的进程的谦让值。

13.2 进程的运行环境

程序被加载到系统内存而成为一个进程是一个复杂的过程。Linux 系统为进程的运行提供了强大的进程运行环境。本节将主要介绍进程的入口函数及进程的环境变量等内容。

13.2.1 进程的入口函数

1．main 函数

众所周知，C 语言的入口函数是 main，在 Linux 下也是如此。进程开始执行时，都是从 main 函数开始的。所以，在一个可执行的 Linux 程序中，必须有包含 main 函数。main 函数的原型定义如下所示。

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

（1）参数说明如下。

- argc：表明程序执行时的命令行参数个数。注意：该参数个数包含了程序名称本身。也就是说，如果执行程序时，未在命令行输入任何参数，则该 argc 值为 0。该参数的值是由系统确定的，用户程序可以根据该值获得程序执行时命令行参数的个数。
- argv：命令行参数数组。其中每一个数组成员为一个命令行参数。程序名称是该数组的第一个成员 argv[0]。在 Linux 系统中，各命令行参数是以空格分隔的。
- env：环境变量数组，可以在程序中访问这些环境变量。关于环境变量的详细说明参见下一节。

（2）返回值说明如下。

main 函数的返回值可以在 shell 中获取到。

（3）关于 main 函数，需要注意以下几点。

- main 函数可以有多种格式。在上述介绍中，是格式全面的 main 函数。通常，如果不需要对命令行参数进行处理，可以直接使用 main()这种简单的方式。
- main 函数的 argc 所包含的命令行参数个数是包含程序名称自身的。命令行参数 argv[0]就是程序名称。

【范例 13-7】演示 main 函数。其实现过程如示例代码 13-7 所示。

示例代码 13-7

```
1   #include <stdio.h>                          /*头文件*/
2   int main(int argc,char *argv[],char *env[])  /*主函数*/
3   {
4       int i;                                  /*循环变量*/
5       for(i=0; i<argc; i++)                   /*循环输出全部命令行参数*/
6           printf("argv[%d]=%s\n",i,argv[i]);  /*输出命令行参数*/
7       for(i=0; env[i]!=NULL; i++)             /*循环输出全部环境变量*/
8           printf("env[%d]:%s\n",i,env[i]);    /*输出环境变量*/
9       return 5;                               /*返回 5*/
10  }
```

【运行结果】经过编译链接，在 shell 下执行程序：./example13_7 aa bb cc，其输出信息如下所示。

```
1   argv[0]=./example13_7
2   argv[1]=aa
3   argv[2]=bb
4   argv[3]=cc
5   env[0]:LESSKEY=/etc/lesskey.bin
6   env[1]:REMOTEHOST=192.168.1.5
7   env[2]:NNTPSERVER=news
8   env[3]:INFODIR=/usr/local/info:/usr/share/info:/usr/info
9   env[4]:MANPATH=/usr/local/man:/usr/share/man:/usr/X11R6/man:/opt/gnome/share/man
10  env[5]:HOSTNAME=Hubery
```

程序执行完毕后，通过 shell 执行命令 echo $?查看程序的返回值，如下所示。

```
11  5
```

【代码解析】在本例中，分别输出命令行参数及环境变量信息，最后程序返回，返回值为 5。本例中，源代码的各行解释如下。

- 第 1 行：头文件信息。
- 第 5～6 行：输出命令行参数信息。从输出信息的第一行可以看到，命令行参数数组的第一个值就是程序名称。
- 第 7～8 行：输出环境变量信息。环境变量也可以通过 shell 执行命令 env 获取。
- 第 9 行：程序返回，返回值为 5。

本例中，各输出信息的解释如下。

- 第 1～4 行：命令行参数信息。
- 第 5～10 行：环境变量信息。
- 第 11 行：程序的返回值。

提示：Linux 环境下，有多个 shell 系统变量，本例中用到的$?就是其中之一。主要的系统变量如下。

$#：命令行参数个数；$n：命令行参数，n 为非负整数，如$0 表示程序名称，$1 表示第一个命令行参数；$?：前一条命令的返回码；$$：本进程的进程 ID；$!：上一进程的进程 ID。

2．getopt

从上面的介绍可以知道，在编程过程中可以通过直接访问命令行参数数组获取命令行参数。另外，Linux 还提供了专门的系统调用 getopt 获取命令行参数。getopt 提供更为强大的获取命令行参数的方法。它不仅可以获取命令行参数，而且可以按照规则解析命令行参数。如执行下述程序：./test_program –i –s abc –t，在这种情况下，要获取到-s 选项的参数 abc 需要进行很复杂的判断。而通过 getopt 可以很简单地解决这一问题。

getopt 的声明位于头文件<getopt.h>中，其原型如下所示。

```
#include <getopt.h>
int getopt (int ___argc, char *const *___argv, const char *__shortopts);
int getopt_long (int ___argc, char *const *___argv,const char *__shortopts,const struct option *__longopts, int *__longind);
```

参数说明如下。

- ___argc：输入参数，即 main 函数的 argc。
- ___argv：输入参数，即 main 函数的 argv。
- __shortopts：输入参数，选项字符串。该参数指定了解析命令行参数的规则。getopt 认可的命令行选项参数是通过“-”进行的。例如，ls –l 中的“-l”。该参数中，如果某个选项有输入数据，则在该选项字符的后面应该包含“:”，如 ls –l ./a，在指定本参数时，应该用“l:”。对于该选项，有一点需要注意，如果该参数的第一个字符是“:”，则 getopt 发现无效参数后并不返回失败，而是返回“?”或者“:”。返回“？”表示选项无效，返回“:”表示需要输入选项值。

getopt 只能支持单字符的选项，如-l、-a 等。如果需要支持多字符的选项，如-file 等，就需要用到 getopt_long。getopt_long 通过指定一个 struct option 类型的结构数组，将多字符的选项映射为单个字符，从而实现了对多字符选项的支持。struct option 的结构为如下所示。

```
struct option
```

```
{
   const char *name;
   int has_arg;
   int *flag;
   int val;
};
```

（1）在该结构中，各结构成员的说明如下。

- name：定义了多字符的选项名称。
- has_arg：定义了是否有选项值，如果该值为 0，表示没有选项值；如果该值为 1，表明该选项有选项值；如果该值为 2，表示该选项的值是可有可无的。
- flag：如果该成员定义为 NULL，那么调用 getopt_long 的返回值为该结构 val 字段值；如果该成员不为 NULL，getopt_long 调用后，将在该参数所指向的变量中填入 val 值，并且 getopt_long 返回 0。通常该成员定义为 NULL 即可。
- val 是该长选项对应的短选项名称。

（2）与 getopt_long 相关的其他参数说明如下。

- __longind：输出参数，如果该参数没有设置为 NULL，那么它是一个指向整型变量的指针。在 getopt_long 运行时，该整型变量会被赋为获取到的选项在结构数组__longopts 中的索引值。

（3）返回值说明如下。

- “?”：表明 getopt 返回一个未在__shortopts 定义的选项。
- “:”：表明该选项需要选项值，则实际未输入选项值。
- -1：表明 getopt 解析完毕，后面已经没有选项。
- 0：在 getopt_long 的结构数组参数__longopts 中的成员 flag 定义了值。此时，getopt_long 返回 0，而选项的参数将存储在 flag 所指向的变量中。
- 其他：返回的选项字符。

（4）在使用 getopt 时，需要注意与该系统调用相关的全局变量的使用，详细说明如下。

- optind：整型变量，存放环境变量数组 argv 的当前索引值。当调用 getopt 循环取选项结束（getopt 返回-1）后，剩余的参数在 argv[optind]～argv[argc-1]中。
- optarg：字符串指针，当处理一个带有选项值的参数时，全局变量 optarg 将存放该选项的值。
- optopt：整型变量，当调用 getopt 发现无效的选项（getopt 返回?或者:）时，此时 optopt 包含了当前无效的选项。
- opterr：整型变量，如果调用 getopt 前设置该变量为 0，则 getopt 在发现错误时不输出任何信息。

【范例 13-8】利用 getopt_long 编程实现可以接受如下选项的程序。具体的程序信息如表 13-1 所示。

表 13-1　接收选项的程序

短选项	长选项	说明
-f	--flag	输入标志
-n	username	--name

其实现过程如示例代码 13-8 所示。

示例代码 13-8

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <getopt.h>                                        /*getopt 系列函数要包含本头文件*/
3   int save_flag_arg;                                         /*全局整型变量定义*/
4   char *opt_arg_value;                                       /*全局字符串变量定义*/
5   struct option longopts[] = {                               /*结构数组，用于定义每个参数的细节*/
6       { "flag", no_argument, &save_flag_arg,'f'},            /*选项无选项值*/
7       { "name", required_argument, NULL,'n'},                /*选项需要选项值*/
8       {       NULL,       0,       NULL,       0},           /*结构数组结束*/
9   };
10  int main(int argc, char *argv[])                           /*主函数*/
11  {
12      int i,c;                                               /*整型变量定义*/
13      while((c = getopt_long(argc, argv, ":n:f", longopts, NULL)) != -1) /*循环解析命令行参数*/
14      {
15          switch (c)                                         /*调用 switch 判断输入的选项*/
16          {
17          case 'n':                                          /*选项 n*/
18              opt_arg_value = optarg;                        /*获得选项值*/
19              printf("name is %s.\n", opt_arg_value); /*输出用户名称*/
20              break;
21          case 0:
22              if(save_flag_arg == 'f')                       /*结构成员中定义了 flag，输入值保存在该
    变量中，而 getopt 返回 0*/
23              {
24                  printf("flag argument found!\n");     /*输出错误信息*/
25              }
26              break;
27          case ':':                                          /*选项需要输入值，而实际未输入*/
28              printf("argument %c need value.\n",optopt);  /*输出提示信息*/
29              break;
30          case '?':                                          /*无效的选项*/
31              printf("Invalid argument %c!\n",optopt); /*输出错误信息*/
32              break;
33          }
34      }
35      return 0;
36  }
```

【运行结果】经过编译链接，在 shell 下执行程序：./example13_8 –name Hubery -flag。其输出信息如下所示。

```
1   name is Hubery.
2   flag argument found!
```

【代码解析】在本例中，循环调用 getopt_long 获取命令行选项。本例中，源代码的各行解释如下。

- 第 1 行：头文件信息。
- 第 3 行：全局整型变量声明。该整型变量用于存储调用 getopt_long 时返回的短选项值。
- 第 4 行：全局字符串指针声明。该指针用于保存系统全局变量中存储的当前选项的值。

- 第 5～9 行：定义结构数组，该结构数组定义了多字符选项与单字符选项的对应关系。可以看到，选项“name”需要输入参数，而选项“flag”不需要输入参数。注意：在此处如果指定了需要输入参数，则应在 getopt_long 中的__shortopts 参数中的短选项后输入“:”。
- 第 13～34 行：循环调用 getopt_long 解析输入选项。
- 第 17～24 行：输入选项为 n 或者 name 的处理。此时，optarg 中保存是的输入选项的值。
- 第 21～26 行：getopt_long 返回值为 0，表明结构数组中有成员指定了 flag。此时，flag 指针中存储的值是单字节选项。特别注意这一点，此时 getopt_long 返回值并不是选项。
- 第 27～29 行：getopt_long 返回值为“:”，表明该选项需要选项值而未输入。
- 第 30~32 行：getopt_long 返回值为“？”，表明该选项无效。

提示：使用 getopt_long 函数时，如果使用多字符格式的选项输入，则应该用“--”作为选项的前导符。如果使用单字符格式的选项输入，则可以使用“--”或者“-”格式的前导符。

13.2.2　进程的环境变量

对于每个 Linux 系统中的进程来说，都有与进程相关的环境变量。环境变量在编程中非常重要，可以用于保存一些重要的配置信息。如用户在编程过程中，需要保存某个配置项的值，一个方法是将其写入到文件中。而更好的办法是定义成环境变量。

当用户登录 shell 时，会从两个位置获得环境变量的定义。一是全局环境变量文件/etc/profile，另外一个是当前用户的环境变量文件。/etc/profile 中定义的环境变量是对所有的用户都有效的，而用户的环境变量文件只对该用户有效。用户的环境变量文件名称与用户的 shell 类型有关。如果用户的 shell 类型是 bsh，则该文件包括.profile 和.bashrc。这两个文件分别用于不同的登录方式，如果是最常使用的交互式的 shell，则使用.profile 文件。而如果是非交互式的 shell，则使用.bashrc。

可以直接在命令行增加环境变量，通过这种方式增加的环境变量只在本次会话中有效。会话一旦退出，该环境变量将会失效。如果要永久增加某个环境变量，可以在.profile（对于 bsh 来说）中进行。增加环境变量的语法如下所示。

```
export 环境变量名称=值
```

可以通过 env 变量查询当前定义的全部环境变量。如果要查询某个单独的环境变量，可以执行 echo 命令。其语法格式如下所示。

```
echo $环境变量名称
```

要删除某个环境变量的定义，可以执行 unset 命令，其语法如下所示。

```
unset $环境变量名称
```

除通过执行 shell 命令获取环境变量外，也可以通过编程的方式获取。Linux 系统提供了两个系统调用用于获取或者设置环境变量，这两个系统调用分别是 getenv 和 putenv。其声明

位于头文件<stdlib.h>中，原型如下所示。

```
#include <stdlib.h>
char *getenv (__const char *__name);
int putenv (char *__string);
```

（1）参数说明如下。

- ___ name：getenv 输入参数，环境变量的名称。
- __string：setenv 输入参数，要设置的环境变量串，其格式为“环境变量名称=值”。

（2）getenv 返回值说明如下。

- NULL：表明相关的环境变量未定义。
- 其他：环境变量的值。

（3）putenv 返回值说明如下。

- −1：调用失败。
- 0：调用成功。

【范例 13-9】编程实现设置环境变量 CONFIG_PATH 的值为/etc。其实现过程如示例代码 13-9 所示。

示例代码 13-9

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <stdlib.h>
3   int main()                                                  /*主函数*/
4   {
5      char *buffer;                                            /*字符串指针，用于保存环境变量*/
6      buffer = getenv ("CONFIG_PATH");                         /*获得环境变量 CONFIG_PATH*/
7      if(buffer==NULL)                                         /*如果环境变量为空，则调用 putenv 设置*/
8      {
9         putenv("CONFIG_PATH=/etc");                           /*设置环境变量*/
10     }
11     printf("CONFIG_PATH=%s\n",getenv("CONFIG_PATH"));    /*获得并输出环境变量的值*/
12     return 0;
13  }
```

【运行结果】经过编译链接，在 shell 下执行程序，其输出信息如下所示。

```
14  CONFIG_PATH=/etc
```

【代码解析】在本例中，首先调用 getenv 检查是否定义环境变量 CONFIG_PATH。如果未定义，则调用 putenv 设置该环境变量。最后输出环境变量的值。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 6 行：调用 getenv 获取环境变量的值。如果该值为空，表明未定义环境变量。
- 第 9 行：调用 putenv 设置环境变量。
- 第 11 行：输出环境变量的值。

提示：修改环境变量还可以通过另外的两个系统调用 setenv 和 unsetenv 进行，其实现的功能与 putenv 大同小异，读者有兴趣可以参照相关文档。

13.2.3　进程的内存分配

在编程过程中，根据程序的需要可能需要动态申请内存。通过前面对进程的内存映像的了解可以知道，程序中定义的局部变量是在进程的栈（stack）中分配空间的。其内存的分配与释放不需要用户关心，由系统自动完成。而如果在程序运行时需要动态分配的内存，将从系统可用内存中申请新的空间，并加入到进程的堆（heap）中。动态申请的内存在使用完毕后，应该由用户进行释放。

Linux 提供了专门的用于内存申请及释放的系统调用，分别是申请内存的 malloc、重新申请内存的 realloc 和释放内存的 free。这两个函数的声明位于头文件<stdlib.h>中，其原型如下所示。

```
#include <stdlib.h>
void *malloc (size_t __size);
void *realloc (void *__ptr, size_t __size);
void free (void *__ptr);
```

参数说明如下。

- __size：输入参数，内存缓存区的大小，以字节为单位。
- __ptr：realloc 的输入参数，已有的内存缓存区指针。
- __ptr：free 的输入参数，要释放的内存缓存区指针。

【范例 13-10】编程实现由键盘输入字符，保存于程序中动态分配的空间中。其实现过程如示例代码 13-10 所示。

示例代码 13-10

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
main()                                                      /*主函数*/
{
    int i;                                                  /*循环变量*/
    char c,*p;                                              /*定义字符变量*/
    p = (char *)malloc(10);                                 /*分配 10 个字节的缓存区*/
    for(i=0;;i++)
    {
        c = getchar();                                      /*从键盘读入单个字符数据*/
        if(i>9)                  /*如果输入字符的个数大于分配的缓冲区，则重新申请内存*/
        {
            p = (char *)realloc(p,1);                       /*重新增加申请一个字节的空间*/
        }
        if(c == '\n')                                       /*输入<Enter>键，退出循环*/
        {
            p[i] = '\0';                                    /*终结字符串*/
            break;
        }
        else
        {
            p[i] = c;                                       /*将输入的字符保存到分配的缓存区*/
        }
```

```
24          }
25          printf("%s\n",p);                          /*输出缓存区中的内容*/
26          free(p);                                   /*释放动态分配的内存*/
27      }
```

【运行结果】经过编译链接，在 shell 下执行程序。通过键盘输入若干数据，输入<Enter>键，程序结束退出。

```
1       ./example13_10
2       1234567890abcde
3       1234567890abcde
```

【代码解析】在本例中，首先调用 malloc 初次分配长度为 10 字节的内存缓存区，然后循环从键盘接收输入的字符。如果输入字符数量超过已申请缓存区大小，则调用 realloc 增加申请更多的内存空间。最后，程序将在接收到回车符后退出。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 7 行：调用 malloc 初次分配长度为 10 个字节的内存空间。
- 第 8～24 行：循环调用 getchar 从键盘获取输入数据，并保存至动态分配的空间中。
- 第 10 行：调用 getchar 从键盘获得输入的单个字符。
- 第 11～14 行：如果输入数据的长度已超过初始分配的空间，则调用 realloc 增加分配空间。
- 第 15～19 行：如果输入回车符号，则中断输入退出循环。注意：此处将最后一位字符置为空字符，以便于后续的字符串输出。
- 第 20～23 行：将输入的字符保存到分配的内存空间中。
- 第 25～26 行：将输入的数据输出并释放动态分配的空间。

本例中用于两处转义字符：“\0”表示空，其 ASCII 码值为 0；“\n”表示回车符号。所谓转义，是指用特定格式的字符串表示某个字符。转义主要用来输入不能直接输入的控制类字符。在 C 语言中，用“\”作为转义的前导符。在 Linux 下支持若干转义字符。

提示：如果知道要输入字符的 ASCII 码值，可以通过八进制转义或者十六进制转义方式直接使用。使用八进程表示时，用“\”作为前导，后面跟该字符的三位的八进制 ASCII 码值。使用十六进制时，则使用“\x”作为前导，后面跟两位的十六进制 ASCII 码值。如换行的 ASCII 码值为 10，可以使用\n 表示，也可以用\012（八进制）或者\x0a（十六进制）表示。

13.3 进程的创建

Linux 系统提供了多种创建新进程的方法。这些方法主要包括 fork 系统调用、exec 系列和 system 系统调用。在本节中将逐一进行介绍。

13.3.1 调用 fork 创建进程

创建进程的简单方法是调用 fork。调用完成后，将生成新的进程。此时，新生成的进程

称为子进程，而原来的调用进程称为父进程。fork 系统调用是非常特殊的一个系统调用。调用 fork 一次将返回两次，分别在父进程和子进程中返回。在父进程中，其返回值为子进程的进程标识符。在子进程中，其返回值为 0。其调用过程如图 13-6 所示。

fork 调用成功后，产生的子进程继承了父进程大部分的属性。这些属性主要包括以下几点。

- 进程的实际用户 ID、实际用户组 ID 和有效用户 ID、有效用户组 ID。
- 进程组 ID、会话 ID 及控制终端。
- 当前工作目录及根目录。
- 文件创建掩码 UMASK。
- 环境变量。

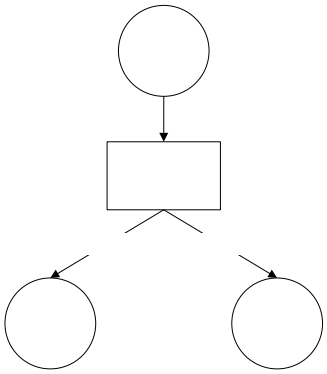

图 13-6　fork 调用过程

除此之外，也有一部分进程属性是不能直接从父进程继承的，主要包括以下几点。

- 进程号、子进程号不同于任何一个活动的进程组号。
- 子进程的用户时间和系统时间，这两个时间被初始化为 0。
- 子进程的超时时钟设置为 0，这个时钟是由 alarm 系统调用使用的。
- 子进程的信号处理函数指针组置为空。原来的父进程中的信号处理函数都将失效。
- 父进程的记录锁。

在调用 fork 时，应该注意以下几点。

- 父进程中已打开的文件描述符可以在子进程中直接使用。这些描述符不仅包括文件描述符，而且包括其他如套接口描述符等。在这种情况下，这些描述符的引用计数已经加一（每 fork 一次就加一）。因此，在关闭这些描述符时，要记住多次关闭直至描述符的引用计数为 0。
- 子进程复制了父进程的数据段，包括全局变量，但是父、子进程各有一份全局变量的拷贝。因此，不能通过全局变量在父子进程间通信，而要通过专门的进程间通信机制。

【范例 13-11】编程创建多个进程，每个进程输出当前时间。其实现过程如示例代码 13-11 所示。

示例代码 13-11

```
#include <stdio.h>                          /*头文件*/
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
int main()                                  /*主函数*/
{
    pid_t pid;                              /*进程 ID*/
    signal(SIGCLD, SIG_IGN);                /*信号处理，忽略 SIGCLD 信号，避免形成僵尸进程*/
    switch(pid=fork())                      /*创建子进程*/
    {
        case -1:                            /*创建子进程失败*/
            perror("fork");                 /*输出错误信息*/
```

```
13                  break;
14              case 0:                                 /*子进程*/
15                  printf("子进程:进程 ID=%d\n",getpid());      /*输出当前进程的进程 ID*/
16                  exit(0);
17                  break;
18              default:                                /*父进程*/
19                  printf("父进程:创建子进程%d 成功.\n", pid);  /*输出新创建的子进程的进程 ID*/
20                  sleep(5);                           /*休眠 5 秒*/
21                  break;
22          }
23  }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1   子进程:进程 ID=5101
2   父进程:创建子进程 5101 成功.
```

【代码解析】在本例中，首先调用 signal 忽略 SIGCLD 信号。然后调用 fork 创建子进程，并分别在父、子进程中输出相应提示信息。本例中，源代码的各行解释如下。

- 第 1～4 行：头文件信息。
- 第 8 行：调用 signal 忽略子进程的退出信号。关于 signal 及 SIGCLD 信号的详情，请参见第 14 章的相关内容。
- 第 9 行：调用 fork 创建子进程。
- 第 11～13 行：调用 fork 失败，输出错误信息后退出。
- 第 14～17 行：fork 的返回码为 0，表明是在子进程中。子进程输出提示信息后退出。
- 第 18～21 行：fork 的返回码非 0，表明是在父进程中，输出新创建的子进程 ID，休眠 5 秒后退出。

提示：Linux 系统中除 fork 外，还有另外一个系统调用 vfork。vfork 系统调用的目的是创建子进程，与 fork 不同的是，vfork 创建子进程的目的是调用 exec，并且 vfork 产生的子进程与父进程共享大多数进程空间。也就是说，vfork 调用成功后，父、子进程共享数据段。关于 vfork 的使用，有兴趣的读者可以查阅相关文档。

13.3.2 调用 exec 系列函数执行程序

exec 系统函数并不创建新进程，调用 exec 前后的进程 ID 是相同的。exec 系列函数的主要工作是清除父进程的可执行代码映像，用新程序的代码覆盖调用 exec 的进程代码。如果 exec 执行成功，进程将从新程序的 main 函数入口开始执行。调用 exec 函数后，除进程 ID 保持不变外，还有下列进程属性保持不变。

- 进程的父进程 ID。
- 实际用户 ID 和实际用户组 ID。
- 进程组 ID、会话 ID 和控制终端。
- 定时器剩余的时间。
- 当前工作目录及根目录。
- 文件创建掩码 UMASK。

- 进程的信号掩码。

exec 系列函数共有 6 种不同的形式，统称为 exec 函数。为讲解清晰，把这 6 个函数划分为两组：一组是 execl、execle、execlp；另一组是 execv、execve、execvp。这两组函数的不同在于 exec 后的第一个字符，第一组是 l，在这里称为 execl 系列；第二组是 v，在这里称为 execv 系列。这里的 l 是 list（列表）的意思，表示 execl 系列函数需要将每个命令行参数作为函数的参数进行传递。而 v 是 vector（矢量）的意思，表示 execv 系列函数将所有函数包装到一个矢量数组中传递即可。exec 函数的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int execv (__const char *__path, char *__const __argv[]);
int execve (__const char *__path, char *__const __argv[],char *__const __envp[]);
int execvp (__const char *__file, char *__const __argv[]);
int execl (__const char *__path, __const char *__arg, ...);
int execle (__const char *__path, __const char *__arg, ...);
int execlp (__const char *__file, __const char *__arg, ...);
```

（1）参数说明如下。

- __path：输入参数，要执行的程序路径。注意：这里是路径名，要求可以是绝对路径或者是相对路径。在 execv、execve、execl、execle 四个函数中，使用带路径名的文件名作为参数。
- __file：输入参数，要执行的程序名称。这里是指文件名。如果该参数中包含“/”字符，则视为路径名直接执行；否则视为单独的文件名，系统将根据 PATH 环境变量指定的路径顺序搜索指定的文件。
- __argv：输入参数，命令行参数的矢量数组。
- __envp：输入参数，带有该参数的 exec 函数，可以在调用 exec 系列函数时，指定一个环境变量数组。其他不带该参数的 exec 系列函数，则使用调用进程的环境变量。
- __arg：程序的第 0 个参数，即程序名自身，相当于__argv[0]。
- …：输入参数，命令行参数列表。调用相应程序时有多少命令行参数，就需要有多少个输入参数项。注意：在使用此类函数时，在所有命令行参数的最后，应该增加一个空的参数项，表明命令行参数结束。

（2）返回值说明如下。

- −1：表明调用 exec 失败，可以查看 errno 获取详细的错误信息。
- 无返回：表明调用成功。由于调用成功后，当前进程的代码空间被新进程覆盖，所以无返回。

【范例 13-12】编程实现调用执行 ls 命令输出当前目录的文件列表。其实现过程如示例代码 13-12 所示。

示例代码 13-12

```
1    #include <stdio.h>                                    /*头文件*/
2    #include <sys/types.h>
3    #include <unistd.h>
4    main()                                                /*主函数*/
5    {
6        pid_t pid;                                        /*进程标识变量*/
7        char *para[]={"ls","-a",NULL};                    /*定义参数数组，为 execv 所使用*/
```

```
8          if((pid = fork()) < 0)                                  /*创建新的子进程*/
9          {
10                perror("fork");                                  /*错误处理*/
11                exit(0);
12         }
13         if(pid == 0)                                            /*子进程*/
14         {
15               if(execl("/bin/ls","ls","-l",(char *)0) == -1)    /*执行 ls -l 命令*/
16               {
17                     perror("execl");                           /*错误处理*/
18                     exit(0);
19               }
20         }
21         if((pid = fork()) < 0)                                  /*创建新的子进程*/
22         {
23               perror("fork");                                  /*错误处理*/
24               exit(0);
25         }
26         if(pid == 0)                                            /*子进程*/
27         {
28               if(execv("/bin/ls",para) == -1)                   /*执行 ls –a 命令*/
29               {
30                     perror("execv");                           /*错误处理*/
31                     exit(0);
32               }
33         }
34         return;
35   }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1    .                    example13_2.c
2    ..                    example13_2
3    example13_1.c   example13_3.c
4    example13_1      example13_3
5    …
6    -rwxr-xr-x   1 develop users 9731 2008-09-25 02:03 example13_1
7    -rwxr-xr-x   1 develop users 9178 2008-09-28 10:07 example13_1.c
8    -rw-r--r--   1 develop users   407 2008-09-28 10:07 example13_2
9    -rwxr-xr-x   1 develop users 9393 2008-09-23 06:50 example13_2.c
10   …
```

【代码解析】在本例中，首先创建新的子进程，在子进程中调用 execl 执行程序 ls –l。然后创建子进程并调用 execv 执行/bin/ls –a。本例中，源代码的各行解释如下。

- 第 1～3 行：头文件信息。
- 第 7 行：定义并初始化调用 execv 所需要的矢量数组。数组中定义了调用 ls 的命令行参数。
- 第 8～12 行：调用 fork 创建新的子进程。
- 第 15～19 行：在子进程中调用 execl 执行程序 ls –l。
- 第 21～25 行：调用 fork 创建新的子进程。
- 第 28～32 行：在子进程中调用 execv 执行程序 ls –a。

13.3.3　调用 system 创建进程

为了方便地调用外部程序，Linux 提供了 system 系统调用。system 将加载外部的可执行程序，执行完毕后返回调用进程。system 的返回码就是加载的外部可执行程序的返回码。system 系统调用的声明位于头文件<stdlib.h>中，其原型如下所示。

```
#include <stdlib.h>
int system (__const char *__command);
```

（1）参数说明如下。

- __command：输入参数，要加载的外部程序的文件名。

（2）返回值说明如下。

- −1：执行 system 失败，可以从 errno 中获取详细的错误信息。
- 127：执行 system 失败。在 system 的内部实现中，system 首先 fork 子进程，然后调用 exec 执行新的 shell，在 shell 中执行要执行的程序。如果在调用 exec 时失败，system 将返回 127。由于要加载的外部程序也有可能返回 127，因此，在 system 返回 127 时，最好判断一下 errno。如果 errno 不为 0，表明调用 system 失败；否则，调用 system 成功，被加载的程序返回码是 127。
- 其他：执行 system 成功，返回值是调用的外部程序的返回码。

【范例 13-13】编程实现调用执行 ls 命令输出当前目录的文件列表。其实现过程如示例代码 13-13 所示。

示例代码 13-13

```
1    #include <stdio.h>                                          /*头文件*/
2    #include <stdlib.h>
3    main()                                                     /*主函数*/
4    {
5        printf("call ls return %d\n",system("ls -l"));         /*调用 system 执行 ls –l 并输出执行的
     返回值*/
6    }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1    -rwxr-xr-x   1 develop users 9731 2008-09-25 02:03 example13_1
2    -rwxr-xr-x   1 develop users 9178 2008-09-28 10:07 example13_1.c
3    -rw-r--r--   1 develop users   407 2008-09-28 10:07 example13_2
4    -rwxr-xr-x   1 develop users 9393 2008-09-23 06:50 example13_2.c
5    …
6    call ls return 0
```

【代码解析】在本例中，直接调用 system 执行命令 ls –l，并输出 system 的返回值。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 system 执行 ls –l 并输出 system 返回值。

13.4 进程的终止

进程执行完毕后，应该合理的终止，释放进程占用的资源。终止进程的方式有多种，可以是接收到其他进程发送的信号而被动终止进程，也可以是进程自己执行完毕后主动退出进程。本节将介绍用于主动退出进程的 exit 函数。从外部发送信号退出进程的过程将在第 14 章中进行讲述。

13.4.1 调用 exit 退出进程

进程要执行的功能执行完毕，或者执行过程中出错，需要调用 exit 退出进程。在 Linux 系统中，除调用 exit 可以结束进程外，还有另外一个函数_exit 也可以实现类似的功能。但是，由于_exit 函数在退出时并不刷新带缓冲 I/O 的缓冲区。所以，在使用带缓冲的 I/O 操作时，应该调用 exit 函数，而不是_exit。exit 函数的声明位于头文件<stdlib.h>中，而_exit 位于<unistd.h>中。其原型如下所示。

```
#include <stdlib.h>
void exit (int __status);
#include <unistd.h>
void _exit (int __status);
```

参数说明如下。

- __status：输入参数，程序退出时的返回码。该返回码可以在 shell 中通过$?系统变量取得，也可以通过 system 系统调用的返回值取得，还可以在父进程中通过调用 wait 函数获得。

13.4.2 调用 wait 等待进程退出

一个进程结束运行时，将向其父进程发送 SIGCLD 信号。这一点将在第 14 章中进行详细讲解。父进程在收到 SIGCLD 信号后，可以忽略该信号或者安装信号处理函数处理该信号。而处理该信号需要调用 wait 系列函数。wait 系列函数的作用是等待子进程的退出，并获取子进程的返回码。

通常情况下，父进程调用 wait 等待其子进程的退出。如果没有任何子进程退出，则 wait 在缺省状态下将进入阻塞状态，直到调用进程的某个子进程退出。wait 系列函数主要有两个：一个是 wait，另一个是 waitpid。这两个函数的声明位于头文件<sys/wait.h>中，其原型如下所示。

```
#include <sys/wait.h>
__pid_t wait (__WAIT_STATUS __stat_loc);
__pid_t waitpid (__pid_t __pid, int *__stat_loc, int __options);
```

（1）参数说明如下。

- __stat_loc：输出参数，用于保存子进程的结束状态。

- __pid：输入参数，用于 waitpid。该参数可以有若干输入方式，每种方式有其独特的含义。
- __options：输入参数，用于 waitpid。该参数指定了调用 waitpid 时的选项。

（2）Linux 提供了多个宏以便从该结束状态中获取特定信息，具体信息如下。

- WIFEXITED(__stat_loc)：如果子进程正常结束则为非 0 值。
- WEXITSTATUS(__stat_loc)：取得子进程 exit()返回的结束代码。通常情况下，应先用 WIFEXITED 来判断是否正常结束才能使用此宏。
- WIFSIGNALED(__stat_loc)：如果子进程是因为信号而结束则返回真。
- WTERMSIG(__stat_loc)：返回子进程因信号而中止的信号代码。通常应先用 WIFSIGNALED 来判断后才使用此宏。
- WIFSTOPPED(__stat_loc)：如果子进程处于暂停执行情况则此宏返回真。只有使用 WUNTRACED 选项时才会有此情况。
- WSTOPSIG(__stat_loc)：返回引发子进程暂停的信号代码。通常应先调用 WIFSTOPPED 来判断后才使用此宏。

（3）返回值说明如下。

- −1：调用失败。
- 其他：调用成功，返回值为退出的子进程 ID。

【范例 13-14】编写代码实现子进程退出。其实现过程如示例代码 13-14 所示。

示例代码 13-14

```
#include <stdio.h>                                          /*头文件*/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
void handle_sigcld(int signo)                               /*SIGCLD 信号处理函数*/
{
    pid_t pid;                                              /*保存退出进程的进程 ID*/
    int status;                                             /*保存进程的退出状态*/
    if((pid = wait(&status)) != -1)                         /*调用 wait 等待子进程退出*/
    {
        printf("子进程%d 退出\n",pid);                      /*输出提示信息*/
    }
    if(WIFEXITED(status))                                   /*判断子进程退出时是否有返回码*/
    {
        printf("子进程返回%d\n",WEXITSTATUS(status)); /*输出子进程的返回码*/
    }
    if(WIFSIGNALED(status))                                 /*判断子进程是否被信号中断而结束*/
    {
    printf("子进程被信号%d 结束\n",WTERMSIG(status));/*输出中断子进程的信号*/
    }
}
main()                                                      /*主函数*/
{
    pid_t pid;                                              /*定义 pid_t 类型变量，用于保存进程 ID*/
    signal(SIGCLD,handle_sigcld);                           /*安装 SIGCLD 信号*/
```

```
27          if((pid = fork()) < 0)                          /*创建子进程*/
28          {
29                perror("fork");                           /*错误处理*/
30                exit(0);
31          }
32          if(pid == 0)                                    /*子进程*/
33          {
34              exit(123);                                  /*子进程返回 123*/
35          }
36          sleep(5);                                       /*父进程休眠 5 秒,等待子进程退出*/
37    }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1     子进程 5497 退出
2     子进程返回 123
```

【代码解析】在本例中，调用 signal 安装 SIGCLD 信号处理函数，以便在子进程退出时可以捕获信号。在信号处理函数中，调用 wait 等待子进程退出，并输出子进程的返回码。然后创建子进程，子进程未进行任何操作，直接返回。从输出可以看到，父进程通过捕获 SIGCLD 信号，成功获取到子进程的退出代码是 123。本例中，源代码的各行解释如下。

- 第 1～5 行：头文件信息。
- 第 6～21 行：信号 SIGCLD 的信号处理函数。在函数中，调用 wait 等待某个子进程退出。输出子进程的进程 ID 及退出代码。
- 第 26 行：调用 signal 安装 SIGCLD 信号。关于信号安装的详细过程，参见第 14 章的相关内容。
- 第 27～31 行：调用 fork 生成新的子进程。
- 第 34 行：子进程代码。在本例中，子进程未进行任何操作，直接退出。返回代码为 123。
- 第 36 行：父进程睡眠，等待子进程退出。

13.5 小结

在 Linux 系统中，进程是十分重要的概念。当可执行的程序被系统加载到内存空间运行时，就成为一个进程。本章首先讲解了关于进程的重要概念，然后讲解了进程的运行环境，最后重点讲解了如何创建和中止进程。灵活创建和中止进程是 Linux 程序开发的重要技能。本章的部分内容涉及信号的处理，请读者在学习第 14 章时能与本章的内容进行结合，以加深理解。