第 7 章　进程控制

知识点

- 进程的结构、类型
- 进程的创建
- 进程的终止
- 进程的等待
- 进程控制综合实例

进程是 Linux 编程中最重要的应用之一，因为 Linux 是多用户、多任务的操作系统，允许多个程序并发执行。正因为这种并发执行的特点，引入了进程的概念，需要让编写的程序以进程的形式来执行。

本章将学习关于 Linux 中进程的相关知识，包括进程的基本概念以及进程控制。进程控制又包括创建新进程、执行程序和进程终止等内容。另外还介绍了 system 函数。

7.1　Linux 进程概述

要理解进程的概念，重点是要先区别程序和进程。程序是存储到磁盘上包含可执行机器指令和数据的静态实体。进程是程序在计算机上的一次执行活动，当运行一个程序时，就启动了一个进程。显然程序是死的（静态的），而进程是活的（动态的）。

形象一点来讲，进程就像是一个大容器，当编写完程序运行它们时，就相当于把程序放进了容器里。同时，还可以往这个容器里加入其他的东西（如程序运行所需要的数据、需要应用的库文件等）。

进程中包含和程序一样的指令和数据，进程也包含了程序计数器和所有 CPU 寄存器的值，同时它的堆栈中还存放着子程序参数、返回值以及变量之类的临时数据。

7.1.1　进程简介

为了描述和控制进程的运行，操作系统为每个进程定义了一个数据结构——进程控制块，即 PCB（Process Control Block）。而由程序段、相关的数据段和 PCB 三部分才组成了所谓的进程实体。需要注意的是，一般对进程的创建/撤消操作，实质上只是针对进程实体中的 PCB。

PCB 中记录了操作系统所需的、用于描述进程的当前情况以及控制进程运行的全部信息。它主要包括进程标识符（进程 ID）、处理机状态、进程调度信息、进程控制信息 4 方面的信息。

一个程序在运行时，系统会跟踪到一些重要信息，如：

- 程序运行的当前位置，也就是程序的上下文。
- 程序正在访问的那个文件。
- 程序的权限（例如，与程序对应的进程是属于哪个用户或组的）。
- 程序运行在当前哪个目录下。

- 程序运行使用的内存和其他系统资源。

而这些重要信息都会记录在进程控制块（PCB）中。下面重点介绍与编程密切相关的几个进程的属性。

Linux 操作系统一般把进程分为 3 类，每种进程都有自己的特点和属性。

（1）交互进程：由一个 shell 启动的进程。交互进程既可以在前台运行，也可以在后台运行。

（2）批处理进程：这种进程和终端没有联系，是一个进程序列。

（3）精灵进程：Linux 系统启动时的进程，在后台运行。

7.1.2　进程标识

如上一小节所述，进程的一个重要属性是进程标识号（Process ID，PID）。PID 的值为一非负的整数。进程的标识号就是进程的身份证，它唯一标识一个进程。只要知道了进程的标识号，就能够控制这个进程。

Linux/UNIX 有一些专用的进程 ID：

- 进程 ID 为 0 的是调度进程，常被称为交换进程（Swapper）。该进程并不执行任何磁盘上的程序，它是内核的一部分，因此也被称为系统进程。
- 所有进程的祖先是一个进程 ID 为 1 的进程，这个进程叫做 init 进程，它是系统启动后第一个启动的进程。在 UNIX 的早期版本中，该进程的程序文件是/etc/init，在较新版本中是/sbin/init。此进程负责在内核自举后启动一个 UNIX 系统。init 通常读与系统有关的初始化文件（/etc/rc*文件），并将系统引导到一个状态（例如多用户）。init 进程绝不会终止。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。
- 进程 ID 为 2 的是页精灵进程（Pagedaemon）。在某些 UNIX 的虚存实现中，此进程负责支持虚存系统的请页操作。与交换进程一样，页精灵进程也是内核进程。

除了进程 ID，每个进程还有一些其他标识符。下列函数返回这些标识符。

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

pid_t getpid(void);          //返回：调用进程的进程 ID
pid_t getppid(void);         //返回：调用进程的父进程 ID
uid_t getuid(void);          //返回：调用进程的实际用户 ID
uid_t geteuid(void);         //返回：调用进程的有效用户 ID
gid_t getgid(void);          //返回：调用进程的实际组 ID
gid_t getegid(void);         //返回：调用进程的有效组 ID
```

7.2　创建进程

可以通过在一个程序内部启动另一个程序来达到创建新进程的目的，可以通过库函数 system 运行 ps 命令来完成，例如：

```
system("ps -ax");
```

system 函数很有用，上例中它将以"ps -ax"为参数使程序执行 ps 命令，这行代码需要等待由 system 函数启动的进程结束之后，即 ps 命令完成后才能返回。

7.2.1 fork 函数

也可以在一个现存进程中调用 fork 函数创建一个子进程以达到创建进程的目的。当一个进程（父进程）调用 fork 函数时，系统内核创建一个新的进程（子进程），并将父进程的内存映像复制到子进程的进程空间中。

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

pid_t fork(void);
                        若执行成功，子进程中返回 0，父进程中返回子进程 ID；若出错返回-1
```

由 fork 创建的新进程被称为子进程（Child Process）。该函数被调用一次，但返回两次。子进程的返回值是 0，而父进程的返回值是新子进程的进程 ID。将子进程 ID 返回给父进程的原因是：一个进程的子进程可以有多个，所以没有一个函数可以使进程获得其所有子进程的 ID。给子进程返回 0 的原因是：一个子进程只会有一个父进程，子进程可以使用 getppid 来获得其父进程的 PID（进程 ID 为 0 的是交换进程，所以一个子进程的 PID 不可能为 0）。

fork 创建的子进程是父进程的一个准确的副本，子进程中包含了父进程的 UID 和 GID、进程组和会话 ID、环境、资源限制、打开的文件等。

fork 的使用一般有两个目的：

- 父进程希望得到一个和自己一样的子进程，使得父、子进程同时执行不同的代码段，这是父进程的分身术。这种情况在网络服务进程中非常常见，提供服务的父进程一直等待服务请求，一旦这个请求到达，父进程就会调用 fork，创建一个自己的影子来处理这个请求。父进程自己会继续等待其他请求。
- 进程用来完成不同的任务。这种情况下，子进程在被 fork 创建后一般会立即调用 exec（7.5 节将介绍 exec）。

下面是一个创建子进程的例子 process_fork.c：

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

int nglobvar = 8;
char buf[] = "a test for I/O \n";

int main()
{
        int nvar;
        pid_t pid;

        nvar = 18;
        if(write(0,buf,sizeof(buf)-1)!= sizeof(buf)-1)
                write(2,"write error",11);
        printf("before fork\n");

        if((pid = fork())<0)
                fprintf(stderr,"fork error");
```

```
        else if(pid == 0){        //子进程
                nglobvar++;
                nvar++;
        }else
                sleep(2);         //保证子进程先运行
        printf("pid = %d, glob = %d, var = %d\n", getpid(),nglobvar, nvar);
        exit(0);
}
```

这个例子演示了 fork 函数的功能，执行此程序成功后，得到如下结果：

```
[root@localhost home test]# ./a.out
a test for I/O
before fork
pid = 7846, glob = 9, var = 19       //子进程的变量值改变了
pid = 7845, glob = 8, var = 18       //父进程的变量值没有改变
[root@localhost home test]# ./a.out > test
a test for I/O
[root@localhost home test]# cat test
before fork
pid = 8105, glob = 9, var = 19
before fork
pid = 8104, glob = 8, var = 18
```

一般来说，在 fork 之后父进程和子进程的执行顺序是不确定的，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。在上面的程序中，父进程使自己睡眠 2 秒钟，以此使子进程先执行。这里并不保证 2 秒钟已经足够。

注意以上程序中 fork 与 I/O 函数之间的关系。回忆第 6 章中所述，write 函数是不带缓存的。因为在 fork 之前调用 write，所以其数据写到标准输出一次。但是，标准 I/O 库函数是带缓存的。当以交互方式运行该程序时，只得到 printf 输出的行一次，其原因是标准输出缓存由新行符刷新。但是当将标准输出重新定向到一个文件时，却得到 printf 输出的行两次。其原因是，在 fork 之前调用了 printf 一次，但当调用 fork 时，该行数据仍在缓存中，然后在父进程数据空间复制到子进程中时，该缓存数据也被复制到子进程中。于是那时父、子进程各自有了带该行内容的缓存。在 exit 之前的第二个 printf 将其数据添加到现存的缓存中。当每个进程终止时，其缓存中的内容被写到相应文件中。

对上面的程序需要注意的另一点是：在重新定向父进程的标准输出时，子进程的标准输出也被重新定向。实际上，fork 的一个特性是所有由父进程打开的描述符都被复制到子进程中。

下面列举一些由子进程继承父进程的其他性质：

- 实际用户 ID、实际组 ID、有效用户 ID、有效组 ID。
- 进程组 ID。
- 控制终端。
- 设置－用户－ID 标志和设置－组－ID 标志。
- 当前工作目录。
- 根目录。
- 文件方式创建屏蔽字。
- 对任一打开文件描述符的在执行时关闭标志。
- 连接的共享存储段。

- 资源限制。

父、子进程之间的区别是：

- fork 的返回值。
- 进程 ID。
- 不同的父进程 ID。
- 子进程的 tms_utime、tms_stime、tms_cutime、tms_ustime 设置为 0。
- 父进程设置的锁，子进程不继承。

7.2.2 vfork 函数

由于 fork 创建子进程的过程是把父进程的全部内存空间映像复制给子进程，这决定了 fork 过程是一个缓慢的过程。为了能够快速地创建一个子进程，系统提供了另一个快速创建进程的系统调用函数 vfork。

fork 和 vfork 的区别在于：vfork 也是创建一个新进程，但它并不产生父进程的副本。在调用 exec 和 exit 之前，新进程运行在父进程的地址空间中，只有当它需要访问父进程的任何内存时，那部分内存才会复制给子进程。这个特点也被称为写时复制（copy-on-write）。另外，vfork 还有一个额外的特性，就是它保证子进程先运行，在调用了 exec 或 exit 后父进程才被调度运行。这样减少了竞争条件对子进程的威胁。

vfork 函数的语法：

```
#include <unistd>

pid_t vfork(void);
                        若执行成功，子进程中返回 0，父进程中返回子进程 ID；若出错返回-1
```

同 fork 一样，vfork 创建子进程成功时返回 0 到子进程，返回子进程的 PID 给父进程；创建失败则会返回-1。

7.3 终止进程

进程终止可以分为正常终止和异常终止。进程正常终止具体有 3 种方式：在 main 函数内执行 return 语句、调用 exit 函数、调用_exit 系统调用函数。异常终止由某个信号触发，如 SIGABORT 信号。

提示

信号是 Linux/UNIX 响应某些条件而产生的一个事件，类似于 Windows 编程中的消息，进程本身能产生或捕获到来自其他进程的信号。信号的名称定义在头文件 signal.h 中，它们都以 SIG 开头，例如进程异常终止时会产生 SIGABORT 信号、无效内存段访问时产生 SIGSEGV 信号、子进程退出时父进程会收到一个 SIGCHLD 信号、除数为 0 时产生异常信号 SIGPE 等。

不管进程如何终止，最终都会执行相同的操作，即相应进程关闭所有打开描述符，释放所使用的存储器等。

exit 和_exit 函数用于正常终止一个程序，它们都带有一个整型参数，以表示进程终止时的终止状态。_exit 函数属于系统调用函数，直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；而 exit 函数在此之前要执行一些清除处理，例如把文件缓

冲区中的内容写回文件，关闭所有的标准 I/O 流。

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

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

提示

exit 函数所带的整型参数以下称为退出码。例如无法启动 shell 来运行命令时，system 函数将返回退出码 127，其返回值与执行了 exit(127)一样。

在终止一个进程时，肯定都希望知道它的终止状态。在任意一种情况下，该终止进程的父进程都能用 wait 函数或 waitpid 函数（在下一节说明）取得其终止状态。

子进程会将其终止状态返回给父进程，但如果父进程在子进程之前终止，那么这时子进程的父进程都改变为 init 进程，称所有无父进程的子进程由 init 进程收养。其操作过程大致是：在一个进程终止时，内核会逐个检查所有活动进程，以找出终止进程的子进程，然后将其父进程 ID 更改为 1（init 进程的 ID）。这种处理方法保证了每个进程有一个父进程。

内核为每个终止子进程保存了一定量的信息，如果当终止进程的父进程调用 wait 或 waitpid 时，则可以得到其子进程的终止状态、进程 ID 等信息。虽然终止进程已经不再运行，但它仍然存在于系统中，其父进程还没有对其进行善后处理（获取终止子进程的有关信息、释放它仍占用的资源），这时这个终止的进程被称为僵尸进程（zombie）。如果编写一个长期运行的程序，它 fork 出了很多子进程，而父进程又未等到子进程的终止状态，那么这些子进程就会变成僵尸进程。

7.4　等待进程

很多时候都需要让一个进程等待另一个进程结束后才继续执行，以避免产生混乱的结果。wait 系统调用函数即可暂停父进程直到它的子进程结束（获得其终止状态）为止。

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid , int *status, int options);
                        若执行成功返回子进程 PID，若出错返回-1
```

waitpid 的作用和 wait 函数的作用是一样的。不过 wait 等待的是第一个结束的子进程，它只要截获了任意一个子进程结束的信息，便会立刻返回；waitpid 可以等待指定的子进程直到子进程结束。

7.4.1　wait 函数

wait 函数可以返回子进程的终止状态，终止状态信息将被写入 statloc 指针所指向的位置。而这些状态信息可以由一些 sys/wait.h 文件中的宏定义解释，如表 7-1 所示。宏可配合使用，可根据一个宏的值是否非零而选用其他宏来取得终止状态、信号编号等。

表 7-1 检查 wait 所返回的终止状态的宏

宏	说明
WIFEXITED(status)	若子进程为正常终止，它返回一个非零值
WEXITSTATUS(status)	若 WIFEXITED 非零，它返回子进程的退出码
WIFSIGNALED(status)	若子进程为接到一个未捕捉的信号而终止，它返回一个非零值
WTERMSIG(status)	若 WIFSIGNALED 非零，它返回一个子进程终止的信号编号
WIFSTOPPED(status)	若子进程为异常终止，它返回一个非零值
WSTOPSIG(status)	若 WIFSTOPPED(status)非零，它返回一个子进程终止的信号编号

如果一个子进程还在运行，那么父进程调用 wait 时将会被阻塞；如果一个子进程已终止，正等待父进程存取其终止状态，那么父进程调用 wait 时将立即返回并取得该子进程的终止状态；如果一个进程没有任何子进程，那么调用 wait 时出错，立即返回。

两个函数返回：若成功则为进程 ID，若出错则为-1。

这两个函数的区别是：

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

下面介绍一个利用 wait 函数实现让父进程等待并打印其终止状态的实例，请看下面的程序 process_wait.c：

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

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

        if((pid = fork()) < 0)
                fprintf(stderr,"fork error");
        else if(pid == 0)
                exit(3);

        if(wait(&status)!= pid)             //status 可返回其状态
                fprintf(stderr,"wait error");
        print_status(status);

        if((pid = fork()) < 0)
                fprintf(stderr,"fork error");
        else if(pid == 0)
                abort();
```

```
        if(wait(&status)!= pid)
            fprintf(stderr,"wait error");
        print_status(status);

        if((pid = fork()) < 0)
            fprintf(stderr,"fork error");
        else if(pid == 0)
            status /= 0;                //产生异常信号 SIGPE

        if(wait(&status)!= pid)
            fprintf(stderr,"wait error");
        print_status(status);

        exit(0);
}

void print_status(int s)
{
        if(WIFEXITED(s))
            printf("normal termination, exit status = %d\n", WEXITSTATUS(s));
        else if(WIFSIGNALED(s))
            printf("abnormal termination,signal number = %d\n", WTERMSIG(s));
        else if(WIFSTOPPED(s))
            printf("child stopped, signal number = %d\n",WSTOPSIG(s));
}
```

这个程序中的函数 print_status 使用表 7-1 中的宏以打印进程的终止状态。

例示终止状态的不同值，运行程序可得：

```
[root@localhost home test]#./a.out
normal termination, exit status=3
abnormal termination, signal number=6
abnormal termination, signal number=8
```

7.4.2　waitpid 函数

waitpid 函数的作用和 wait 函数的作用是一样的。不过 wait 等待的是第一个结束的子进程，它只要截获了任意一个子进程结束的信息，便会立刻返回；waitpid 可以等待指定的子进程直到子进程结束。

waitpid 函数的参数 pid 可以有 4 个取值：

- pid 取正值，waitpid 等待指定的进程，这个指定进程的 PID 为 pid。
- pid 为 0，waitpid 等待任何一个组 ID（PGID）和调用者的组 ID 相同的进程。
- pid 等于-1，waitpid 等待任何 PGID 等于 PID 的绝对值的子进程。
- pid 取值为 1，waitpid 等待任何子进程，作用等同于调用 wait。

status 参数的作用和 wait 的 status 的作用一样。options 参数可以决定父进程的状态，其值可以是 0，可以取另外两个宏：

- WNOHANG：没有子进程存在时立即返回。
- WUNTACHED：如果子进程进入暂停状态则马上返回，若为结束状态则不做处理。

waitpid 函数提供了 wait 函数没有提供的 3 个功能：

- waitpid 等待一个特定的进程（而 wait 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。
- waitpid 提供了一个 wait 的非阻塞版本。有时希望取得一个子进程的状态，但不想阻塞。
- waitpid 支持作业控制（以 WUNTRACED 选择项）。

process_waitpid.c 程序示例如下：

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

int main(void)
{
        pid_t pid;

        if((pid = fork()) < 0)
                fprintf(stderr,"fork error");
        else if(pid == 0){//第一次 fork();的子进程
                if((pid = fork()) < 0)
                        fprintf(stderr,"fork error");
                else if(pid > 0)    //第二次 fork();的父进程
                        exit(0);
                sleep(2);
                printf("second child, parent pid = %d\n",getppid());
                exit(0);
        }
        //等待第一个子进程
        if(waitpid(pid,NULL,0)!=pid)
                fprintf(stderr,"fork error");
        exit(0);
}
```

执行这个程序得到：

```
[root@localhost home test]#./a.out
[root@localhost home test]# second child, parent pid = 1
```

注意，当第一个子进程终止时，第二次 fork()的子进程被 init 进程收养，所以它的父 pid 为 1。而在第二个子进程打印其父进程 ID 时，第一个父进程等待的第一个子进程已经退出，所以 shell 打印其指示符后，第二个子进程打印其父进程 ID。

7.5 exec 函数

7.2 节曾提及父进程 fork 出子进程后，子进程通常要立即调用一种 exec 函数以执行另一个程序。有 6 种不同的 exec 函数可供使用，它们常被统称为 exec 系列函数。这 6 个函数在调用规则和用法上略有区别。当进程调用其中一种 exec 函数时，该进程被替换为一个全新的程序，新的程序从其 main 函数重新开始执行。exec 函数调用使用的是覆盖旧进程的方式，原来

程序的数据段、堆栈段和代码段都会被修改，但因为调用 exec 并不创建新进程，所以前后的进程 ID 并未改变。

和 fork 函数一样，它们也在头文件<unistd.h>中声明。它们的原型是：

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, .../*NULL*/);
int execle(const char *pathname, const char *arg0, .../*NULL*/, char *const envp[]);
int execlp(const char *filename, const char *arg0, .../*NULL*/);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *filename, char *const argv[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
                    若执行出错返回-1，若执行成功不返回
```

这 6 个函数比较复杂，参数较多而且不好记忆。这里可将其分为 3 类：execl、execlp 和 execle 的参数个数是可变的，参数以一个空指针结束；而 execv、execvp 函数的第二个参数是一个字符串数组；最后一个 execve 函数是内核的系统调用，另外 5 个函数只是库函数，它们都是通过调用 execve 实现的。

不管哪种情况，新的程序都由 pathname（路径名）或 filename（文件名）参数决定，而后面 argv 数组中给定的参数会在新程序启动时传递给 main 函数。以 filename 为参数时，函数通过 PATH 环境变量查找可执行文件的路径，若 PATH 变量中没有定义该可执行文件的路径，那么就需要给出该可执行文件的绝对路径。

提示

PATH 变量在 2.7.4 节中曾经提及，可以在该变量中设置想要搜寻的相关目录，目录之间用冒号（:）分隔。例如下列 name=value 环境字符串：

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

而 argv 数组中给定的参数会在新程序启动时传递给 main 函数。

执行 exec 函数后，进程 ID 不会改变。除此之外，执行新程序的进程还保持了原进程的下列特征：①进程 ID 和父进程 ID；②实际用户 ID 和实际组 ID；③添加组 ID、进程组 ID、对话期 ID；④控制终端；⑤闹钟尚余留的时间；⑥当前工作目录；⑦根目录；⑧文件方式创建屏蔽字；⑨文件锁；⑩进程信号屏蔽；⑪资源限制。

实例 process_exec.c 程序代码如下：

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

int main()
{
   pid_t pid;
   //环境变量
   char *arg_envir[] = { "EDITOR=/usr/bin/emacs", NULL };

   if ((pid = fork ()) < 0)
      fprintf (stderr, "fork error");
   else if (pid == 0)
      {
```

```
            if (execle("./echoarg", "echoarg", "arg 1", "arg 2", (char *) 0,arg_envir) < 0)
                fprintf (stderr, "execle error");
        }
    if (waitpid (pid, NULL, 0) < 0)
        fprintf (stderr, "wait error");

    if ((pid = fork ()) < 0)
        fprintf (stderr, "fork error");
    else if (pid == 0)
        {
            if (execlp ("echoarg", "echoarg", "only one arg", (char *) 0) < 0)
                fprintf (stderr, "execlp error");
        }
    exit (0);
}
```

在该程序中先调用 execle 函数，它要求一个指定的路径名，并给它指定一个环境变量 arg_envir。下一个调用的是 execlp 函数，它需要一个指定的文件名，并将调用者的环境变量（系统默认环境变量）传送给新程序 echoarg。新程序 echoarg 需要放到 PATH 指定的路径下 execlp 才能调用成功，例如把程序 echoarg 复制到/usr/bin 目录下。新程序 echoarg.c 代码如下：

```
#include <stdio.h>
#include <stdlib.h>

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

    for (i = 0; i < argc; i++)
        printf ("argv[%d]: %s\n", i, argv[i]);
    for (ptr = environ; *ptr != 0; ptr++)
        printf ("%s\n", *ptr);

    exit (0);
}
```

执行以上程序时得到：

```
# ./a.out
argv[0]: echoarg
argv[1]: arg 1
argv[2]: arg 2
EDITOR=/usr/bin/emacs
# argv[0]: echoarg
argv[1]: only one arg
ORBIT_SOCKETDIR=/tmp/orbit-john
…
…
```

注意，execlp 函数调用新程序 echoarg 时，因为没有指定环境变量，所以它将打印系统默

认的全部环境变量，上面的省略号部分即剩下的系统环境变量。而 shell 提示出现在第二个 exec 打印 argv [0]和 argv [1]之间。这是因为父进程并不等待该子进程结束。

7.6　system 函数

system 函数的作用是运行以字符串参数形式传递给它的命令并等待该命令的完成。

```
#include <stdlib.h>

int system(const char *cmdstring);
                        若执行命令失败返回退出码 127，若是其他错误返回-1
```

在 7.2 节开头说明了可以通过 system 函数实现在一个程序内部启动另一个程序，以达到创建新进程的目的。下面基于那行代码，把它补充完整，即实例 system1.c：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        printf("A test of system function \n");
        system("ps -ax");
        printf("OVER\n");
        exit(0);
}
```

编译并运行这个程序时，会看到如下所示的输出：

```
[root@localhost home test]# ./system1
A test of system function
PID TTY STAT TIME COMMAND
1?S 0:05 init
2?SW 0:00 [keventd]
...
1262 pts/1 S 0:00 /bin/bash
1273 pts/2 S 0:00 su -
1274 pts/2 S 0:00 -bash
1463 pts/1 S 0:00 oclock -transparent -geometry 135x135-10+40
1465 pts/1 S 0:01 emacs Makefile
1480 pts/1 S 0:00 ./system1
1481 pts/1 R 0:00 ps -ax
OVER.
```

这个例子中，程序以“ps -ax”字符串为参数调用 system 函数，从而在程序中调用 ps 命令，程序将一直等待，在 ps 命令执行完成从 system 调用中返回后才继续执行。这就是 system 函数的局限性，即在调用 system 函数时，在它启动的进程执行期间，不能立刻执行其他任务。尽管可以将 shell 程序放到后台去运行，以达到 system 函数在调用 shell 命令后立刻返回的效果，但一般来说，system 函数不是启动其他进程的理想手段，因为它必须用一个 shell 来启动预定的程序。因为必须先启动一个 shell，然后才能启动程序，所以这种办法的效率很低；对 shell 的安装情况和它所处的环境的依赖也很大。

将 shell 程序放到后台去运行的具体做法是 system1.c 中的函数调用语句修改为如下所示的样子：

```
system("ps -ax &");
```

7.7 综合实例

其实 system 函数的实现调用了 fork、exec 和 waitpid 这 3 个函数，本节以 system 函数的实现为例，综合分析上面介绍的几个进程控制原语。下面的示例是 system 函数的一种实现，同时也作为本章对进程控制原语的一个总结。

Linux 中的 system 函数实现源码如下：

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

int system(const char * cmdstring)
{
        pid_t pid;
        int status;

       if(cmdstring == NULL){

              return (1);
        }

        if((pid = fork())<0){
              status = -1;
        }
        else if(pid == 0){
              execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
              _exit(127);          //如果子进程正常执行则不会执行此语句
        }
        else{
              while(waitpid(pid, &status, 0) < 0){
                    if(errno != EINTR){ // errno = EINTR 表示中断错误
                          status = -1;
                          break;
                    }
              }
        }
        return status;
}
```

当 system 接受的命令为 NULL 时直接返回一个非零值，否则 fork 出一个子进程。如果 fork 失败则 system 返回-1，若成功 fork 函数在子进程中返回 0，在父进程中返回子进程的 pid。

在子进程中调用 execl 来启动一个新的程序代替自己，“/bin/sh”表示将要启动 shell 程序的路径，后面的都是其参数。这里的-c 选项表示 shell 程序将使用下一个参数作为命令输入，即这里的 cmdstring。最后一个参数为 NULL，shell 对以 NULL 字符终止的命令字符串进行语法分析，将它们分成分隔开的命令参数。execl 执行成功后，子进程就变成了一个 shell 进程。

如果 execl 执行成功，那么它不返回，等到 execl 执行结束，子进程结束；如果失败，即不能执行 shell，那么 execl 返回-1，执行 exit (127)。

在父进程中使用 waitpid 等待子进程结束。如果 waitpid 返回除 EINTR 之外的错误，则 system 返回-1，并在 errno 中设置错误类型。

如果 3 个函数（fork、exec 和 waitpid）都执行成功，那么 system 的返回值是 shell 的终止状态，其终止状态通过 waitpid 函数保存在 status 指针所指向的位置。

本章小结

在这一章中，看到了进程是如何成为 Linux 操作系统的一个基本组成部分的。学习了如何启动进程、终止进程和查看进程，如何用它们来解决程序设计问题。

习题七

1．编写一个程序，由主进程创建生成一个子进程，这两个进程都完成打印输出数字 1～100000 功能。运行看看输出的结果会是什么，为什么？

2．在第 1 题的程序中，将 getpid、getppid、getuid、geteid、getgid、getegid 这几个函数测试一次，看看会有什么结果？

类似：printf("getpid = %d \n", getpid);

3．在 Linux 下面创建新的进程有几种方法？都有哪些？

4．什么是僵尸进程？它与我们所说的一般进程有什么联系和区别？为什么会有僵尸进程？

5．wait 与 waitpid 有什么区别，它们各自有哪些优点和缺点？编写一个程序来测试一下 wait 和 waitpid。

6．编写一个程序，测试一下 exec 家族系列函数。