第7章 进程控制与进程间通信

7.1 进 程 控 制

7.1.1 进程的基本概念

1. 进程与程序

- 所谓“进程”，就是程序的一次执行。从内核的角度看，进程是系统中的一个对象，它对应一个程序的执行流并且是一个资源分配（包括内存和文件等）的单位。
- 操作系统就以“进程”为单位，管理这个执行流和它占用的内存等资源，并负责做到多个进程之间互不影响。

- 所谓“程序”，是一个由CPU指令和数据构成的集合，这些指令和数据存放在磁盘的一个普通文件中。
- 程序是一个静态的概念，它用于在创建进程时初始化进程的用户数据段和指令段。初始化完成之后，进程开始执行。自此以后，进程和初始化它的程序之间就不再有联系了。
- 进程运行后，它对应的磁盘上的程序文件不可被删除。这是由于操作系统中虚拟内存管理功能的需要。
- 几个同时运行的进程可以由同一程序初始化得到，然而这些进程之间并没有什么联系。

2. 进程的组成部分

进程是操作系统管理系统活动的基本单位。一个进程包括四个部分内容：指令段，用户数据段，用户堆栈段和系统数据段。

- 指令段，存放程序的CPU指令代码。对于一个C语言程序来说，主程序和子程序编译后的可执行CPU指令代码，以及调用的其他库函数的代码，都被组织在该段内。
- 指令段大小是固定不变的，占用进程独立的逻辑地址空间的一部分，并被设置为只读内存段。

- 用户数据段，存放程序运行所需要的数据。具体来说，C语言中的全局变量，静态（static）变量，字符串常数，都存放在该区域。
- UNIX允许数据段增长和缩小，这样便实现了内存的动态分配。UNIX系统调用sbrk()允许编程调整数据段的大小，但通常用户使用C语言基于sbrk()调用编写的一组内存管理库函数申请或释放内存，如：malloc(), free()等。

- 用户堆栈段是用户程序执行所需要的堆栈空间，用于实现函数的嵌套调用。递归调用也算作嵌套调用的一种特殊形式。
- 用户堆栈用于保存子程序执行完毕之后返回主调程序的下条指令的地址，以及在函数和被调函数之间传递参数。

- 系统数据段，是在操作系统内核内的数据，每个进程对应一套数据，包括页表和进程控制块。
- **UNIX是分时系统，当进程的执行被临时打断时，CPU寄存器的现场被保存在PCB中。根据CPU寄存器的现场以及堆栈段中记录的堆栈的状态，就可以知道程序被打断的那个时刻已经执行到了哪条指令，这个位置是函数嵌套调用或递归调用的哪个层次。**
- **一个进程PCB中还含有许多进程的属性，比如：当前目录，记录当前目录的i-节点，已经打开的文件描述符表，umask值，进程PID，等等。**

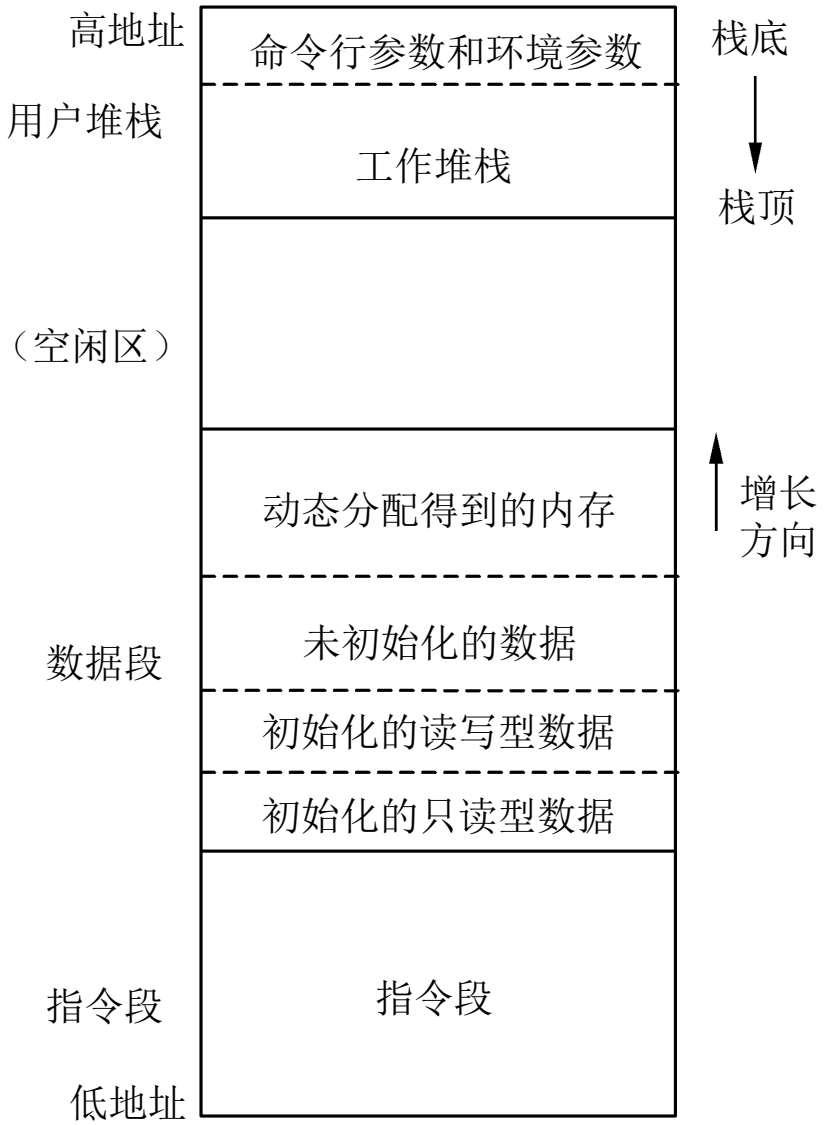

图7-1 进程逻辑地址空间的布局

UNIX的命令size可以列出程序文件或者编译产生的目标文件（文件名后缀一般为.o）中相应段的大小。用法为：

size *filename-list*

【例7-1】 使用size命令观察可执行程序文件的指令段和数据段大小。

```
$ cd /bin
$ size grep awk cat more
  text  data    bss     dec    hex  filename
 72315   660   2456   75431  126a7  grep
284569  6376  20328  311273  4bfe9 awk
 12412   468    328   13208   3398   cat
 24385   588  10128   35101   891d  more
```

3. 进程的状态

- 主要理解它的两种状态就可以：运行状态和阻塞状态。
- 阻塞状态，也叫睡眠状态，或者等待状态，挂起状态等。
- 运行状态的进程和阻塞状态的进程的最大区别是，系统总是按照优先级在分时处理运行状态的进程，而不顾那些处于阻塞状态的进程。
- 处于运行状态的进程，会因为等待某些事件发生而转为睡眠状态。处于睡眠状态的进程，在条件满足后，被唤醒，重新转为运行状态，被调度执行。

4. 进程状态的转换

这里通过一个具体的例子介绍程序员编写的C程序在执行时进程状态是怎样变化的。

【例7-2】 函数scanf执行期间进程状态的变化和系统的活动步骤。

设在应用程序中有语句scanf("%d", &n);并且可执行程序在一个RS-232串口终端上执行。执行时，操作员输入756然后按下Enter键，上述语句执行结束。然后继续执行后面的语句。图7-2示例出这段时间进程的活动。这段时间系统按照下列步骤进行处理，包括了进程的睡眠和唤醒。

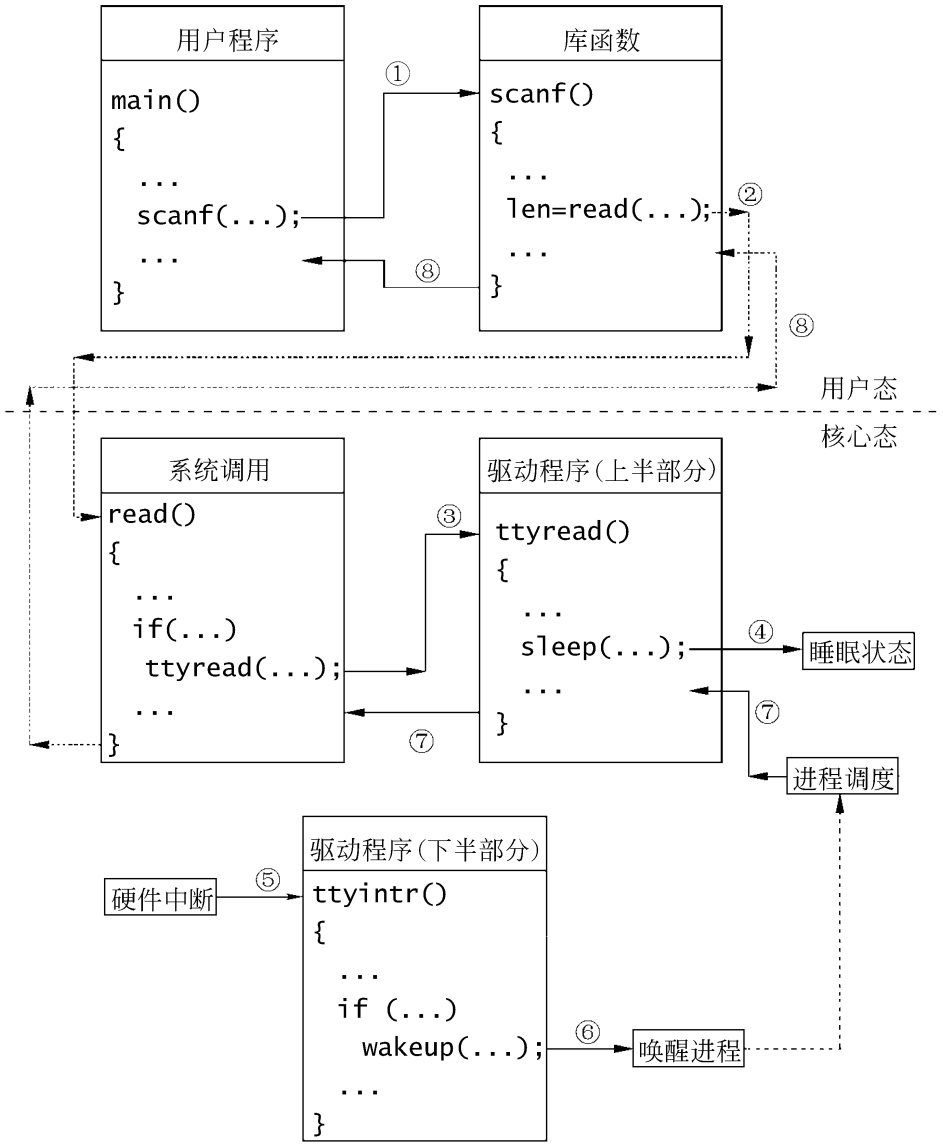

图7-2 执行函数scanf时进程的活动

① 进程调用scanf()，由于scanf是一个C语言库函数，转库函数处理。

② 库函数scanf进行一系列的处理，最终调用UNIX的系统调用：

len = read(0, buf, sizeof buf);

系统调用会使用类似int 80之类的软件中断指令。

③ 操作系统内核处理所有的软中断和硬件设备中断。从某种意义上说，操作系统内核就是系统中所有中断服务程序的集合。在中断服务程序中判定出当前进程期望read功能。执行read调用的内核代码，由于所访问的文件是字符设备文件，转设备文件的驱动程序入口ttyread()。

④ 执行ttyread()的函数体，进行一些处理之后，在驱动程序代码中调用内核的一个进程控制原语sleep()，将当前进程的状态改为睡眠状态，进程睡眠，让出CPU。系统转去执行其他的进程。

⑤ 操作员按下终端上的按键7，产生硬件中断，内核中的中断服务程序执行驱动程序事先在内核中注册好的终端设备驱动程序的ttyintr()函数。ttyintr()执行一系列处理，并将字符7暂存到缓冲区中。中断服务程序ttyintr()返回，进程维持睡眠。

⑥ 系统等待操作员按下5和6两个按键，重复⑤的操作。最后，操作员按下Enter键，数据输入就绪。在ttyintr()处理中，调用内核中的一个进程控制原语wakeup()将睡眠的那个进程唤醒，进程状态变为运行状态，进入调度队列。然后，中断处理ttyintr()结束。

⑦ 内核的进程调度在合适的时机，将进程投入运行，进程恢复运行，从第④步的sleep()睡眠之后的下一条语句开始继续运行，直到ttyread()函数运行完毕，返回read()调用的代码部分。最终，系统调用read()完成。

⑧ 返回到库函数中len = read(0, ……)之后的语句继续执行，库函数执行完毕后，返回到主调函数scanf()语句下面的其他语句执行。

5. time:进程执行的时间

前面提到了进程的用户时间和系统时间，使用time命令，可以打印出程序执行所花费的时间。这个时间包括三个值：实际时间，系统时间和用户时间。

- 实际时间是进程开始运行到停止所花费的真实时间。
- 系统时间是在进程运行期间，操作系统内核为进程的活动所花费的CPU时间。
- 用户时间是运行用户态程序代码所花费的CPU时间。
- 系统时间和用户时间之和，不一定会等于实际时间，一般来说会小于实际时间。

time命令的语法为：

time *command*

使用time测试命令*command*执行的时间，仅需要在原执行命令的前面增加一个前缀time。

- **系统中有个外部命令/usr/bin/time命令**
- **B-shell和C-shell都有个内部命令time，内部命令和外部命令执行时显示的信息格式会不同。**
- **要引用外部命令，就必须使用命令文件的路径名，否则，会被理解成内部命令。**

【例7-3】 使用time命令统计执行一个命令所占用的CPU时间。

/usr/bin/time find /usr -name '*.c' –print

命令执行完毕后，最终给出的时间报告是：

Real 6.06

User 0.36

System 2.13

如果在C-shell中直接使用time命令，那么使用的是C-shell的内部命令：

time find /usr -name '*.h' –print

命令执行完毕后，最终给出一个资源使用报告：

0.4u 6.2s 0:10 61% 4+28k 0+0io 0pf+0w

vmstat命令

打印出整个系统，包括所有进程，近期内占用CPU的情况。

【例7-4】 使用vmstat命令监视整个系统对CPU的占用情况。

```
$ vmstat 10
  procs                  memory      swap      io     system         cpu
 r  b  w   swpd   free   buff  cache   si   so    bi    bo   in   cs  us sy id
 0  0  0      0  55916   6128  38156    0    0   439   118  146  180   8 15 76
 0  0  0      0  55252   6160  38160    0    0     0    32  112   54  26  1 73
 0  0  0      0  46380   7384  38160    0    0   121    21  136   91  15  8 76
 0  0  0      0  46360   7400  38736    0    0    58    19  110   41   1  0 99
^C
```

时间有关系统调用

- 系统调用函数times()，程序员可以在自己的程序中使用这个系统调用获取当前进程的用户CPU时间，系统CPU时间。系统
- 调用clock()直接返回times()调用得到的四个CPU时间的总和。
- C语言的标准函数库中time()用来获得一个时间坐标，坐标的0是格林威治时间1970年1月1日零点，单位为s。
- 函数gettimeofday()也用来获得一个时间坐标，坐标的0是1970年1月1日零点，但是可以精确到微秒μs(10^{-6}s)。
- C语言还提供一些库函数，localtime()将坐标值转换为本地时区的年月日时分秒，mktime将年月日时分秒转换为坐标值，ctime()和asctime()分别将坐标值和年月日时分秒转换为可读字符串，等等。

6. 忙等待

在多任务系统中，一个“忙等待”的程序是不可取的。下面是一个忙等待的例子。

【例7-5】 监视一个文件，如果文件被其他进程追加了内容，就把追加的内容打印出来。

本例是一段程序，功能类似tail -f命令。

```
$ awk '{print NR, $0}' tailf0.c
 1 #include <fcntl.h>

 2 main(int argc, char **argv)
 3 {
 4     unsigned char buf[1024];
 5     int len, fd;
```

```
 6    if ( (fd = open(argv[1], O_RDONLY)) < 0) {
 7        perror("Open file");
 8        exit(1);
 9    }

10    lseek(fd, 0, SEEK_END);
11    for(;;) {
12        while ( (len = read(fd, buf, sizeof buf)) > 0)
13            write(1, buf, len);
14    }
15 }
```

$ cc tailf0.c -o tailf0

$ ./tailf0 /usr/adm/pppd.log

修改后的程序为：

```
$ awk '{print NR, $0}' tailf.c
 1 #include <fcntl.h>

 2 main(int argc, char **argv)
 3 {
 4     unsigned char buf[1024];
 5     int len, fd;

 6     if ( (fd = open(argv[1], O_RDONLY)) < 0) {
 7         perror("Open file");
 8         exit(1);
 9     }
10      lseek(fd, 0, SEEK_END);
11      for(;;) {
12          while ( (len = read(fd, buf, sizeof buf)) > 0)
13              write(1, buf, len);
14          sleep(1);
15      }
16 }

$ cc tailf.c -o tailf
$ ./tailf /usr/adm/pppd.log
```

7.1.2 fork:创建新进程

- 系统调用fork创建一个新进程，但新进程的指令段，用户数据段，用户堆栈段都是旧进程一模一样的复制。系统数据段也几乎全是旧进程的复制。这种子进程的初始状态复制父进程的方法，叫“继承”
- 原先的进程被称作“父进程”,新创建的进程被称作“子进程”。
- 在UNIX中，fork系统调用是创建新进程的惟一方式。fork调用惟一能出错的原因是系统中的资源耗尽。

【例7-6】 使用fork创建子进程。

$ cat fork1.c

```
int a;
int main(int argc, char **argv)
{
    int b;
    printf("[1] %s: BEGIN\n", argv[0]);
    a = 10;
    b = 20;
    printf("[2] a+b=%d\n", a + b);
    fork();
    a += 100;
    b += 100;
    printf("[3] a+b=%d\n", a + b);
    printf("[4] %s: END\n", argv[0]);
}
```

下面是上述程序的执行结果：

$ ./fork1

[1] ./fork1: BEGIN

[2] a+b=30

[3] a+b=230

[4] ./fork1: END

[3] a+b=230

[4] ./fork1: END

int p;

p = fork();

那么，这条语句可以分解为两个动作：

① 执行fork()系统调用；

② 给变量p赋值。

按照fork()系统调用的功能，执行完①后，新产生一个子进程，子进程是父进程的复制。接下来的操作②就会有两个进程分别在自己独立的地址空间内执行，各有自己独立的p变量。但是，操作系统对系统调用的处理导致函数调用fork()在父子进程中有不同的返回值。在对p赋值时，父进程的p得到一个正整数，而子进程的p得到0。程序据此区分父子进程，父进程得到的正整数是子进程的PID号。

【例7-7】 fork创建子进程，父子进程执行不同的程序段。

$ cat fork2.c

```
int main(int argc, char *argv[])
{
    int p;
    printf("[1] %s: BEGIN\n", argv[0]);
    p = fork();
    if (p > 0) {
        printf("[2] Parent, p=%d\n", p);
    } else if (p == 0) {
        printf("[3] Child, p=%d\n", p);
    } else {
        perror("Create new process");
    }
    printf("[4] My PID %d, Parent's PID %d\n", getpid(), getppid());
    printf("[5] %s: END\n", argv[0]);
}
```

上述程序的执行输出如下。这个例子中子进程先于父进程运行，也有可能父进程先于子进程运行。

```
$ ./fork2
[1] ./fork2: BEGIN
[3] Child, p=0
[4] My PID 20796, Parent's PID 23950
[5] ./fork2: END
[2] Parent, p=20796
[4] My PID 23950, Parent's PID 30320
[5] ./fork2: END
```

fork返回值大于0，是子进程的PID；子进程的fork返回值为0。内核在实现fork调用时,首先初始化创建一个新的proc结构，然后复制父进程环境（包括user结构和内存资源）给子进程。

7.1.3 exec:重新初始化进程

- exec系统调用，从一个指定的程序文件重新初始化一个进程。exec系统调用并没有创建新进程，进程还是旧的进程，只是exec调用将它重新初始化了指令段，用户数据段和堆栈段,系统数据段也几乎没有变化。应当说，exec系统调用是在旧进程中运行新程序。
- 在组成进程的四个要素中，系统数据段PCB基本不变。调用exec时需要提供一个程序文件的名字，从磁盘上读入这一程序文件的内容，用这个程序文件中的CPU指令，重新初始化当前进程的指令段，当前进程的原指令段被丢弃；使用程序文件中的数据段说明，重新初始化当前进程的数据段，并从程序文件中获取数据段初值，进程的原数据段被丢弃；同样，当前进程的原堆栈段也被丢弃，用exec提供的函数参数重新初始化新的堆栈段底部。

在执行main()函数之前，系统对进程用户堆栈段的初始化为堆栈段的栈底被放入了命令行参数和环境参数。通过main()函数的参数，可以访问到命令行参数，也可以访问到环境参数。

【例7-8】打印出命令行参数和环境参数。

下面的程序打印出了程序执行时的命令行参数和环境参数的内容，并打印出了这些指针的地址值。C语言printf格式字符串中的%p是以十六进制格式打印出指针的值。对于本例，研究一下这些地址值，可以发现argv指针数组，env指针数组，命令行参数字符串，环境字符串，这些都在内存中连续相继地存储在一起。

```
$ cat  argenv.c
int main(int argc, char *argv[], char *env[])
{
   int i;
   printf("&i=%p argv=%p env=%p\n", &i, argv, env);
   for ( i = 0; i < argc; i++)
      printf("ARG %d: %p [%s]\n", i, argv[i], argv[i]);
   for ( i = 0; env[i]; i++)
      printf("ENV %d: %p [%s]\n", i, env[i], env[i]);
}
$ cc argenv.c -o argenv
$ ./argenv  aa  bbbb  c  ddd
&i=2ff22bc0 argv=2ff22c1c env=2ff22c34
ARG 0: 2ff22c98 [./argenv]
ARG 1: 2ff22ca1 [aa]
ARG 2: 2ff22ca4 [bbbb]
ARG 3: 2ff22ca8 [c]
ARG 4: 2ff22caa [ddd]
```

ENV 0: 2ff22cae [AUTHSTATE=compat]

ENV 1: 2ff22cbf [CGI_DIRECTORY=/usr/HTTPServer/cgi-bin]

ENV 2: 2ff22ce5 [DEFAULT_BROWSER=netscape]

ENV 3: 2ff22cfe [DOCUMENT_DIRECTORY=/usr/HTTPServer/htdocs]

……

ENV 20: 2ff22f0c [TERM=vt100]

ENV 21: 2ff22f17 [TZ=BEIST-8BEIDT]

ENV 22: 2ff22f27 [USER=jiangy]

ENV 23: 2ff22f33 [NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat]

$

有6种格式的exec系统调用：

```
int execl(path, arg0, arg1, ..., 0);
    char *path, *arg0, *arg1, ..., *argn;

int execv(path, argv)
    char *path, **argv;

int execle(path, arg0, arg1, ..., argn, 0, envp)
    char *path, *arg0, *arg1, ..., *argn, **envp;

int execve(path, argv, envp)
    char *path, **argv, **envp;

int execlp(file, arg0, arg1, ..., argn, 0);
    char *file, *arg0, *arg1, ..., *argn;

int execvp(file, argv)
    char *file, **argv;
```

这些调用的参数有微小的差别。这些系统调用的命名中都有exec前缀，后跟1~2个字母，这里字母代表含义为，l(List)，v(Vector)，e(Environment)，p(Path)。

l与v：指定命令行参数的两种方式，l以表的形式在函数参数中列举出命令行参数，v要事先组织成一个数组。函数参数中的命令行参数表，参数数目可变，所以，最后一个要以0为结束标志。

e：用envp数组的内容初始化环境参数，否则，会使用与当前相同的环境参数environ。

p：函数第一个参数，是程序文件名，在环境变量PATH指定的多个查找目录中查寻可执行文件file。否则，不按PATH指定路径自动搜索。

【例7-9】 使用exec在子进程中执行ls命令。

下面的程序先执行fork创建子进程，子进程中执行exec系统调用启动新程序。

```
$ cat forkexec.c
int main(void)
{
    int pid;
    pid = fork();
    if (pid > 0) {
        printf("Child Process is %d\n", pid);
    } else if (pid == 0) {
        execlp("ls", "ls", "-l", "forkexec.c", 0);
        perror("execlp");
    }
    printf("Bye!\n");
```

```
}
$ cc forkexec.c -o forkexec
$ ./forkexec
-rw-r--r--   1 jiang   usr      235 Jun 21 14:12 forkexec.c
Child Process is 28592
Bye!
```

上述程序执行的第一行是子进程输出的，第二第三行是父进程输出的。由于父子进程独立并发运行，所以，第一行要出现在最后，也是可能的。这与进程的调度算法有关。

7.1.4 wait:等待子进程运行结束

- 父进程在创建了子进程之后，父子进程独立运行，互不影响。只有在子进程比父进程提前中止时，系统会通过一定的机制向父进程报告。
- 子进程中止后，会产生“僵尸”进程。僵尸进程是进程生命期结束时的一种特殊状态。系统已经释放了进程占用的包括内存在内的系统资源，但是，仍然在内核中保留进程的部分数据结构，记录进程的终止状态，等待父进程来“收尸”。父进程的“收尸”动作完成之后，“僵尸”进程不再存在。

- 系统调用wait就是等待当前进程的任一个子进程终止，获取子进程终止的状态。wait系统调用执行完之后，一个已死亡的僵尸子进程不再存在。
- 从字面上看wait是“等待”子进程终止，但wait系统调用的最重要功能是销毁子进程的僵尸，得到子进程终止的状态并回收僵尸子进程占用的系统资源。
- 如果执行wait时，还没有子进程终止，那么，父进程会进入睡眠状态等待，直到有一个子进程终止。如果执行wait时，已经有子进程终止，那么，wait会立刻返回。

wait的函数原型为：

#include <sys/wait.h>

pid = wait(int *_status_);

函数的返回值是已终止的子进程的PID号，_status_中会返回子进程终止的原因。如果进程有多个子进程，那么，只要有一个子进程终止，wait就返回。

子进程的终止有两种情况：

- **一种是进程自愿终止（自杀），在程序中调用函数exit()或者在main()函数中的return；**
- **另一种情况是异常终止（被杀），其他进程或者操作系统内核向进程发送信号将进程杀死。**

WIFEXITED(*status*)如果进程正常终止，则为真。可以用**WEXITSTATUS(*status*)**获得子进程的返回码。

WIFSIGNALED(*status*)　　如果进程异常终止，则为真。可以用**WTERMSIG(*status*)**获得子进程被杀的信号值。

【例7-10】 用wait系统调用获取子进程的终止状态。

```
$ cat forkexec2.c
#include <sys/wait.h>
int main(void)
{
    int pid;
    pid = fork();
    if (pid > 0) {
        int stat;
        printf("Child Process is %d\n", pid);
        pid = wait(&stat);
        printf("Child process %d terminated\n", pid);
        if (WIFEXITED(stat))
            printf("Exit code %d\n", WEXITSTATUS(stat));
        if (WIFSIGNALED(stat))
            printf("Killed by signal %d\n", WTERMSIG(stat));
```

```
    } else if (pid == 0) {
        execlp("ls", "ls", "-l", "/etc/passwd", 0);
        perror("execlp");
        exit(1);
    }
    printf("Bye!\n");
}
$ cc forkexec2.c -o forkexec2
$ ./forkexec2
-rw-r--r--   1 root     security       5975 Jun 18 13:44 /etc/passwd
Child Process is 31628
Child process 31628 terminated
Exit code 0
Bye!
```

- **waitpid()**调用可以指定等待到多个子进程中某个指定的子进程终止后才返回；另外，还允许一种非阻塞方式，允许没有任何子进程终止的时候，进程不要进入睡眠等待，而是立刻返回。
- **wait3()**在进程终止的时候，可以得到子进程存活期间使用的系统资源摘要，包括占用**CPU**时间、内存调页次数等。
- 父子进程独立运行，完全有可能父进程在子进程终止之前中止。这样，父进程终止后，子进程成为“孤儿”。所有的孤儿进程统统由**PID**号为**1**的进程领养，进程**1**成为它们的父进程。进程**1**是操作系统的系统进程**/etc/init**，该进程从**UNIX**开始时启动，永远不会终止。