Linux进程编程介绍

东东 整理

第一章　进程的定义

摘要：本节将介绍进程的定义。进程作为构成系统的基本细胞，不仅是系统内部独立运行的实体，而且是独立竞争资源的基本实体。了解进程的本质，对于理解、描述和设计操作系统有着极为重要的意义。了解进程的活动、状态，也有利于编制复杂程序。

1.进程的基本概念

首先我们先看看进程的定义，进程是一个具有独立功能的程序关于某个数据集合的一次可以并发执行的运行活动，是处于活动状态的计算机程序。进程作为构成系统的基本细胞，不仅是系统内部独立运行的实体，而且是独立竞争资源的基本实体。了解进程的本质，对于理解、描述和设计操作系统有着极为重要的意义。了解进程的活动、状态，也有利于编制复杂程序。

1.1 进程状态和状态转换

现在我们来看看，进程在生存周期中的各种状态及状态的转换。下面是 LINUX 系统的进程状态模型的各种状态：

1) 用户状态：进程在用户状态下运行的状态。
2) 内核状态[①]：进程在内核状态下运行的状态。
3) 内存中就绪：进程没有执行，但处于就绪状态，只要内核调度它，就可以执行。
4) 内存中睡眠：进程正在睡眠并且进程存储在内存中，没有被交换到 SWAP 设备。
5) 就绪且换出：进程处于就绪状态，但是必须把它换入内存，内核才能再次调度它进行运行。
6) 睡眠且换出：进程正在睡眠，且被换出内存。
7) 被抢先：进程从内核状态返回用户状态时，内核抢先于它，做了上下文切换，调度了另一个进程。原先这个进程就处于被抢先状态。
8) 创建状态：进程刚被创建。该进程存在，但既不是就绪状态，也不是睡眠状态。这个状态是除了进程 0 以外的所有进程的最初状态。
9) 僵死状态（zombie）：进程调用 exit 结束，进程不再存在，但在进程表项中仍有纪录，该纪录可由父进程收集。

现在我们从进程的创建到退出来看看进程的状态转化。需要说明的是，进程在它的生命周期里并不一定要经历所有的状态。

首先父进程通过系统调用 fork 来创建子进程，调用 fork 时，子进程首先处于创建态，fork 调用为子进程配置好内核数据结构和子进程私有数据结构后，子进程就要进入就绪态 3 或 5，即在内存中就绪，或者因为内存不够，而导致在 SWAP 设备中就绪。

假设进程在内存中就绪，这时子进程就可以被内核调度程序调度上 CPU 运行。内核调度该进程进入内核状态[②]，再由内核状态返回用户状态执行。该进程在用户状态运行一定时间后，又会被调度程序所调

[①] 关于内核态与用户态，请参见附录 I。
[②] 内核对外的接口是系统调用，内核外的程序都必须经由系统调用才能获得操作系统的服务。进程调度程序属系统调用，因此无论进程获得或让出 CPU 都要执行进程调度程序，从而在这段时间内运行于内核态。

度而进入内核状态，由此转入就绪态[①]。有时进程在用户状态运行时，也会因为需要内核服务，使用系统调用而进入内核状态，服务完毕，会由内核状态转回用户状态。要注意的是，进程在从内核状态向用户状态返回时可能被抢占，进入状态 7，这是由于有优先级更高的进程急需使用 CPU，不能等到下一次调度时机，从而造成抢占。

进程还会因为请求的资源不能得到满足，进入睡眠状态，直到它请求的资源被释放，才会被内核唤醒而进入就绪态。如果进程在内存中睡眠时，内存不足，当进程睡眠时间达到一个阀值，进程会被 SWAP 出内存，使得进程在 SWAP 设备上睡眠。这种状况同样可能发生在就绪的进程上。

进程调用 exit 系统调用，将使得进程进入内核状态，执行 exit 调用，进入僵死状态而结束。以上就是进程状态转换的简单描述。

进程的上下文[②]是由用户级上下文、寄存器上下文以及系统级上下文组成。主要内容是该进程用户空间内容、寄存器内容以及与该进程有关的内核数据结构。当系统收到一个中断、执行系统调用或内核做上下文切换时，就会保存进程的上下文。一个进程是在它的上下文中运行的，若要调度进程，就要进行上下文切换。内核在四种情况下允许发生上下文切换：

1) 当进程自己进入睡眠时；
2) 当进程执行完系统调用要返回用户状态，但发现该进程不是最有资格运行的进程时；
3) 当内核完成中断处理后要返回用户状态，但发现该进程不是最有资格运行的进程时；
4) 当进程退出（执行系统调用 exit 后）时。

有时内核要求必须终止当前进程的执行，立即从先前保存的上下文处执行。这可由 setjmp 和 longjmp 实现，setjmp 将保存的上下文存入进程自身的数据空间（u 区）中，并继续在当前的上下文中执行，一旦碰到了 longjmp，内核就从该进程的 u 区，取出先前保存的上下文，并恢复该进程的上下文为原先保存的。这时内核将使得进程从 setjmp 处执行，并给 setjmp 返回 1[③]。

进程因等待资源或其他原因，进入睡眠态是通过内核的 sleep 算法。该算法与本章后面要讲到的 sleep 函数是两个概念。算法 sleep 记录进程原先的处理机优先级，置进程为睡眠态，将进程放入睡眠队列，记录睡眠的原因，给该进程进行上下文切换。内核通过算法 wakeup 来唤醒进程，如某资源被释放，则唤醒所有因等待该资源而进入睡眠的进程。如果进程睡眠在一个可以接收软中断信号（signal）的级别上，则进程的睡眠可由软中断信号的到来而被唤醒。

1.2 进程控制

现在我们开始讲述一下进程的控制，主要介绍内核对 fork、exec、wait、exit 的处理过程，为下一节学习这些调用打下概念上的基础，并介绍系统启动（boot）的过程以及进程 init 的作用。

在 Linux 系统中，用户创建一个进程的唯一方法就是使用系统调用 fork。内核为完成系统调用 fork 要进行几步操作：

---

① 同上页②。关于进程调度的详细介绍请参见本章第 3 节或附录 II。
② 关于进程上下文，请参见附录 I
③ 该过程类似于对当前进程上下文进行备份，一旦进程出现了问题，则从备份点重新运行。

第一步，为新进程在进程表中分配一个表项。系统对一个用户可以同时运行的进程数是有限制的，对超级用户没有该限制，但也不能超过进程表的最大表项的数目。

第二步，给子进程一个唯一的进程标识号（PID）。该进程标识号其实就是该表项在进程表中的索引号。

第三步，复制一个父进程的进程表项的副本给子进程。内核初始化子进程的进程表项时，是从父进程处拷贝的。所以子进程拥有与父进程一样的 uid、euid、gid、用于计算优先权的 nice 值、当前目录、当前根、用户文件描述符表等。

第四步，把与父进程相连的文件表和索引节点表[①]的引用数加 1。这些文件自动地与该子进程相连。

第五步，内核为子进程创建用户级上下文。内核为子进程的 u 区及辅助页表分配内存，并复制父进程的 u 区内容。这样生成的是进程的静态部分。

第六步，生成进程的动态部分，内核复制父进程的上下文的第一层，即寄存器上下文和内核栈，内核再为子进程虚设一个上下文层，这是为了子进程能“恢复”它的上下文。这时，该调用会对父进程返回子进程的 pid，对子进程返回 0。

Linux 系统的系统调用 exit，是进程用来终止执行时调用的。进程发出该调用，内核就会释放该进程所占的资源，释放进程上下文所占的内存空间，保留进程表项，将进程表项中纪录进程状态的字段设为僵死状态。内核在进程收到不可捕捉[②]的信号时，会从内核内部调用 exit，使得进程退出。父进程通过 wait 得到其子进程的进程表项中记录的计时数据，并释放进程表项。最后，内核使得进程 1（init 进程）接收终止执行的进程的所有子进程。如果有子进程僵死，就向 init 进程发出一个 SIGCHLD 的软中断信号.

一个进程通过调用 wait 来与它的子进程同步，如果发出调用的进程没有子进程则返回一个错误，如果找到一个僵死的子进程就取子进程的 PID 及退出时提供给父进程的参数。如果有子进程，但没有僵死的子进程，发出调用的进程就睡眠在一个可中断的级别上，直到收到一个子进程僵死（SIGCLD[③]）的信号或其他信号。

进程控制的另一个主要内容就是对其他程序引用。该功能是通过系统调用 exec 来实现的，该调用将一个可执行的程序文件读入，代替发出调用的进程执行。内核读入程序文件的正文，清除原先进程的数据区，清除原先用户软中断信号处理函数的地址，当 exec 调用返回时，进程执行新的正文。

一个系统启动的过程，也称作是自举的过程。该过程因机器的不同而有所差异。但该过程的目的对所有机器都相同：将操作系统装入内存并开始执行。计算机先由硬件将引导块的内容读到内存并执行，自举块[④]的程序将内核从文件系统中装入内存，并将控制转入内核的入口，内核开始运行。内核首先初始化它的数据结构，并将根文件系统安装到根“/”，为进程 0 形成执行环境。设置好进程 0 的环境后，内核便作为进程 0 开始执行，并调用系统调用 fork。因为这时进程 0 运行在内核状态，所以新的进程也运行在内核状态。新的进程（进程 1）创建自己的用户级上下文，设置并保存好用户寄存器上下文。这时，进程 1 就从内核状态返回用户状态执行从内核拷贝的代码（exec），并调用 exec 执行/sbin/init 程序。进程 1 通常称为初始化进程，它负责初始化新的进程。

进程 init 除了产生新的进程外，还负责一些使用户在系统上注册的进程。例如，进程 init 一般要产生一些 getty 的子进程来监视终端。如果一个终端被打开，getty 子进程就要求在这个终端上执行一个注册的过程，当成功注册后，执行一个 shell 程序，来使得用户与系统交互。同时，进程 init 执行系统调用 wait

① 关于索引结点表，请参见附录 III。
② 应该是指进程收到了信号，但并没有设置屏蔽（如用 sigprocmask）或注册相应的处理程序（如用 sigaction）。
③ SIGCHLD 是 POSIX 标准中定义的,SIGCLD 是 SysV 的事实标准,两个信号想表达的意思一样，只是出处不同，而今大多数 UNIX 都把两个信号 define 成同一个值。推荐用 SIGCHLD
④ 就是引导块（BootBlock）。

来监视子进程的死亡，以及由于父进程的退出而产生的孤儿进程的移交。以上是系统启动和进程 init 的一个粗略的模型[①]。

1.3 进程调度的概念

Linux 系统是一个分时系统，内核给每个进程分一个时间片，该进程的时间片用完就会调度另一个进程执行。LINUX 系统上的调度程序属于多级反馈循环调度。该调度方法是，给一个进程分一个时间片，抢先一个运行超过时间片的进程，并把进程反馈到若干优先级队列中的一个队列。进程在执行完之前，要经过这样多次反馈循环。

进程调度分成两个部分，一个是调度的时机，即什么时候调度；一个是调度的算法，即如何调度和调度哪个进程。我们先来看看调度的算法，假设目前内核要求进行调度，调度程序从“在内存中就绪”和“被抢先”状态的进程中选择一个优先权最高的进程，如果有若干优先权一样高的进程，则在其中选择等待时间最长的进程。切换进程上下文，继续执行该进程。如果没有选择到进程，则不做操作，等待下一次调度时机的到来。

每一个进程都有一个用于调度的优先权域。进程的优先权由低到高粗略地分为用户优先权和内核优先权。每种优先权有若干优先权值[②]（优先数）与其对应。每个优先权都有一个逻辑上与其相连的进程队列。进程从内核状态返回用户状态时被抢先，从而得到用户优先权。进程在内核算法 sleep 中得到内核优先权。内核优先权高于用户优先权，即内核优先权和用户优先权之间存在一个阀值，所有用户优先权低于该阀值，而内核优先权高于该阀值。内核优先权中又划分为可中断和不可中断，即进程在收到一个软中断信号时，低内核优先权的进程可被唤醒，而有高内核优先权的进程继续睡眠。

计算一个进程优先权的时机是：内核将一个优先权值赋给一个将进入睡眠的进程，这个优先权值是固定的，且与睡眠原因相联系；另一个时机是，时钟处理程序每隔一定时间（如每隔 1 秒）调整用户状态下的所有进程的优先权，并使内核运行调度算法。时钟处理程序还根据一个衰减函数，每秒一次的调整每个进程的最近 CPU 使用时间。例如可按如下公式调整：

decay(CPU) = CPU/2;

再根据公式重新计算在“就绪”和“被抢先”状态下的每个进程的优先权值。

Priority = (“recent CPU usage”/constant) + (base priority) + (nice value);

其中 constant 是个系统常量（一般取值为“2”）。base priority 值也是系统的一个常量，一般 base priority 取值为 60。最后，nice 的值是由进程发出 nice 调用时给出的值，这样就可以使得用户通过降低优先权而让出一些执行时间。只有超级用户才能指定提高优先权的 nice 值。

① 关于 Linux 系统启动的详细介绍，请参见附录 IV。
② 进程的优先权值（优先数）越小，优先权越高。

第二章　进程的基本操作

摘要：本节先介绍一些关于进程的基本操作，通过本节，我们将了解如何产生子进程，进程如何改变它的执行映像，父子进程的同步等操作。由此也了解到一些并行程序的基本概念与如何编制简单的并行程序。

2. 进程的一般操作

上一节介绍了一些有关进程的基本概念，从这一节开始要结合一些例子来阐述一些有关进程的系统调用。本节先介绍一些关于进程的基本操作，通过本节，我们将了解如何产生子进程，进程如何改变它的执行映像，父子进程的同步等操作。由此也了解到一些并行程序的基本概念与如何编制简单的并行程序。

2.1 fork 系统调用

系统调用 fork 是用来创建一个子进程。创建的过程前面一节已经介绍过。现在，再介绍一个系统调用 vfork，这个调用的产生是因为认识到创建子进程时对父进程的所有页不进行拷贝能带来性能上的改善。该调用假定进行 vfork 调用后，将立即调用 exec，这样就不需要拷贝父进程的所有页表。因为它不拷贝页表，所以比 fork 调用快[①]。有些系统的 fork 也采用了其他方法来提高性能，比较典型的一种是增加“写时拷贝”。这种 fork 调用，产生子进程时，并不拷贝父进程的所有页面，而是置父进程所有页面的写时拷贝位，子进程共享父进程的所有页面。直到父进程或子进程写某个页面时，就会发生一个保护性错误，并拷贝该页面。这样不仅提高了内核的性能，而且改善了内存的利用。

系统调用 fork 和 vfork 的声明格式如下：

```
pid_t fork(void);
pid_t vfork(void);
```

在使用该系统调用的程序中要加入以下头文件：

```
#include <unistd.h>
```

当调用执行成功时，该调用对父进程返回子进程的 PID，对子进程返回 0。调用失败时，给父进程返回-1，没有子进程创建。

下面是发生错误时，可能设置的错误代码 errno：

EAGAIN：系统调用 fork 不能得到足够的内存来拷贝父进程页表。或用户是超级用户但进程表满，或者用户不是超级用户但达到单个用户能执行的最大进程数。

ENOMEM：对创建新进程来说没有足够的空间，该错误是指没有足够的空间分配给必要的内核结构。

[①] 事实上，fork 与 vfork 的基本区别在于，当使用 vfork 创建新进程时，父进程将被暂时阻塞，而子进程则可以借用父进程的地址空间。这个奇特状态将持续直到子进程要么退出，要么调用 exec，至此父进程才继续执行。这意味著一个有 vfork 创建的子进程必须小心以免出乎意料的改变父进程的变量，也一定不要从包含 vfork 调用的函数中返回或调用 exit。

下面我们看一个 fork 调用的简单的例子。该例子产生一个子进程，父进程打印出自己和子进程的 PID，子进程打印出自己的 PID 和父进程的 PID。

注意：父进程打开了一个文件。父子进程都可以对该文件操作，该程序父子进程都向文件中写入了一行。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
extern int errno;
int main()
{
   char buf[100];
   pid_t cld_pid;
   int fd;
   int status;

   if ((fd=open("temp",O_CREAT|O_TRUNC|O_RDWR,S_IRWXU)) == -1)
   {
      printf("open error %d",errno);
      exit(1);
   }
   strcpy(buf,"This is parent process write");

   if ((cld_pid=fork()) == 0)
   { /*  这里是子进程执行的代码  */
      strcpy(buf,"This is child process write");
      printf("This is child process");
      printf("My PID(child) is %d",getpid()); /*打印出本进程的 ID*/
      printf("My parent PID is %d",getppid()); /*打印出父进程的 ID*/
      write(fd,buf,strlen(buf));
      close(fd);
      exit(0);
   }
   else
   { /*  这里是父进程执行的代码  */
      printf("This is parent process");
      printf("My PID(parent) is %d",getpid()); /*打印出本进程的 ID */
      printf("My child PID is %d",cld_pid); /*打印出子进程的 ID*/
```

```
        write(fd,buf,strlen(buf));
        close(fd);
    }
    wait(&status);   /* 如果此处没有这一句会如何？*/
    return 0;
}
```

下面我们看一下，程序运行的结果，假设源文件命名为 fork.c：

```
[root@wapgw /root]# gcc -o fork fork.c
[root@wapgw /root]# ./fork
This is parent process
This is child process
My PID(child) is 5258
My parent PID is 5257
My PID(parent) is 5257
My child PID is 5258
[root@wapgw /root]#
```

从上面的运行结果可以看出进程的调度，父进程打印出第一行后，CPU 调度子进程，打印出后续的三行，子进程结束，调度父进程执行（其中可能还有其他的进程被调度）,父进程执行完，将控制返还给 shell 程序，最后一行是 shell 程序输出的提示符。

看看 temp 文件里有什么内容

```
[root@wapgw /root]# more temp
This is child process write
This is parent process write
[root@wapgw /root]#
```

现在我们将程序稍作修改。将 wait 调用注释掉，我们看看会有什么样的结果。因为调度的原因，多执行几次，你会看到如下的结果：

```
[root@wapgw /root]#vi fork.c //将 wait 调用注释掉
[root@wapgw /root]# gcc -o fork fork.c
[root@wapgw /root]# ./fork
This is parent process
This is child process
My PID(parent) is 5282
My child PID is 5283
[root@wapgw /root]# My PID(child) is 5283
My parent PID is 1
[root@wapgw /root]#
```

第一行是父进程的输出，第二行是子进程的输出，第三、四行是父进程的输出，这时父进程由于没有 wait 调用，不等待子进程而结束。下面一行中的 “[root@wapgw /root]#”是父进程结束，将控制返回给 shell 时，shell 输出的提示符。然后 CPU 调用子进程，输出子进程的 PID 是 5283。注意，下面子进程输出其父进程的 ID 是 1，因为它的父进程结束了，内核将它交给了进程 1（进程 init）来管理，这个过程见前面一节。这里要提一下的是，输出结果的顺序和进程调度的顺序有关，自己试验的结果与例子中的顺序很可能不同，请自行分析。(从我的系统给出的结果来看，加不加 wait 都一样，都先执行完子进程，后执行父进程，不过用管道从父进程向子进程传消息，子进程也可以正常收到，看来现在内核调度比较智能，具体调度顺序有待于进一步研究)

2.2 exec 系统调用

系统调用 exec 是用来执行一个可执行文件来代替当前进程的执行映像。需要注意的是，该调用并没有生成新的进程，而是在原有进程的基础上，替换原有进程的正文，调用前后是同一个进程，进程号 PID 不变。但执行的程序变了（执行的指令序列改变了）。它有六种调用的形式，随着系统的不同并不完全与以下介绍的相同。它们的声明格式如下：

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char* const envp[]);
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[]);
```

在使用这些系统调用的程序中要加入以下头文件和外部变量：

```
#include <unistd.h>
extern char **environ;
```

下面我们先详细讲述其中的一个，然后再给出它们之间的区别。在系统调用 execve 中，参数 path 是将要执行的文件，参数 argv 是要传递给文件的参数，参数 envp 是要传递给文件的环境变量。当参数 path 所指的文件替换原进程的执行映像后，文件 path 开始执行，参数 argv 和 envp 便传递给进程。下面我们举一个简单的例子。

在讲述系统调用 execve 的例子之前，我们先来看看环境变量。为了使用户方便和灵活地使用 Shell，LINUX 引入了环境的概念。环境是一些数据，用户可以改变这些数据，增加新的数据或删除一些数据。这些数据称为环境变量。因为它们定义了用户的工作环境，同时又可以被修改。每个用户都可以有自己不同的环境变量，用户可以用 env 命令（不带参数）浏览环境变量，输出的格式和变量名随着 Shell 的不同和系统配置的不同而不同。下面这个例子打印出传递给该进程的所有参数和环境变量：

```
#include <stdio.h>
#include <unistd.h>
extern char **environ;
int main(int argc,char* argv[])
{
```

```
    int i;
    printf("Argument：\n");
    for (i=0;i<=argc;i++)
        printf("Arg%d is：  %s\n",i,argv[i]);
    printf("Environment：\n");
    for (i=0;environ[i]!=NULL;i++)
        printf("%s\n",environ[i]);
}
```

下面是执行时的屏幕拷贝：

```
[root@wapgw /root]# gcc -o example example.c
[root@wapgw /root]# ./example test
Argument：
Arg0 is ./example
Arg1 is test

Environment：
PWD=/root
REMOTEHOST=cjm
HOSTNAME=wapgw
HOME=/root
……
SSH_ASKPASS=/usr/libexec/ssh/gnome-ssh-askpass
PATH=/usr/local/sbin：/usr/sbin：/sbin：/usr/local/sbin：/usr/local/bin：
/sbin：/bin：/usr/sbin：/usr/bin：/usr/X11R6/bin：/root/bin
[root@wapgw /root]#
```

其中 Environment 后的都是环境变量及其取值。下面我们来看看 execve 的一个简单的例子：

```
#include <unistd.h>
#include <stdio.h>
extern char **environ;
int main(int argc,char* argv[])
{
    printf("Will replace by another image");
    execve("example",argv,environ); /*  用上面的例子 example 替换进程执行映像  */
    printf("process never go to here"); /*  进程永远不会执行到这里  */
}
```

该程序用自己的参数 argv 和环境变量传递给新的执行映像。执行结果的屏幕拷贝如下：

```
[root@wapgw /root]# gcc -o execve execve.c
[root@wapgw /root]# ./execve these args will dend to example
```

```
Will replace by another image
Argument：
Arg0 is ./execve
Arg1 is these
Arg2 is args
Arg3 is will
Arg4 is dend
Arg5 is to
Arg6 is example
Environment：
PWD=/root
REMOTEHOST=cjm
HOSTNAME=wapgw
HOME=/root
……
SSH_ASKPASS=/usr/libexec/ssh/gnome-ssh-askpass
PATH=/usr/local/sbin：/usr/sbin：/sbin：/usr/local/sbin：/usr/local/bin：
/sbin：/bin：/usr/sbin：/usr/bin：/usr/X11R6/bin：/root/bin
[root@wapgw /root]#
```

这里要注意的是，如果你用 execve some args > screen 时，会发现输出重定向到一个文件后，丢失了数据（即少了输出的第一行 Will replace …）。这是因为将输出重定向到一个文件后，进程的第一行是输出到文件，所以被缓冲还没有真正写入文件，进程的第二行替换进程的执行映像，也覆盖了文件的缓冲。这个问题可以通过 fflush(stdout)刷新 stdout 的缓冲区或者用 setbuf(stdout,NULL)设置 stdout 的缓冲为空来解决。

如果对某个文件描述符 fd 设置了 close-on-exec[①]标志，那么在 exec 调用后，该文件描述符被关闭。下面我们看一个简单的例子：

这里有一个程序 pp.c：

```
#include <stdio.h>
int main()
{
      printf("test");
}
```

它是用来替换进程执行图像的。再来看看下面的程序：

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
extern char **environ;
```

[①] 标志参数为 FD_CLOEXEC。很多现有的涉及文件描述符标志的程序并不使用常量 FD_CLOEXEC，而是将此标志设置为 0（系统默认，在 exec 时不关闭）或 1（在 exec 时关闭）。

```
int main(int argc,char* argv[])
{
   printf("close-on-exec is %d",fcntl(1,F_GETFD));
   fcntl(1,F_SETFD,16);
   printf("close-on-exec is %d",fcntl(1,F_GETFD));
   execve("pp",argv,environ);
   printf("AH!!!!!");
}
```

该程序的执行结果为：

```
[root@wapgw /root]# ./fcntl
close-on-exec is 0
close-on-exec is 0
test
[root@wapgw /root]#
```

这是没有设置 close-on-exec 标志的结果，将 fcntl[①]语句改为

```
fcntl(1,F_SETFD,25);
```

对于最后一个参数，只要保证该参数的最低位（二进制）是 1 就可以。这时的执行结果为：

```
[root@wapgw /root]# ./fcntl
close-on-exec is 0
close-on-exec is 1
[root@wapgw /root]#
```

这时，系统调用 execve 用 pp 替换原进程的执行图像，但由于文件描述符 1（stdout[②]）被关闭，所以执行完 execve 调用后无输出。

系统调用 execve 可以执行二进制的可执行文件（如 a.out）。也可以执行 shell 程序，该 shell 程序必须以下面所示的格式开头，即第一行为：#! interpreter [arg]。其中 interpreter 可以是 shell 或其他解释器，例如：#!/bin/bsh 或#!/usr/bin/perl。其中的 arg 是传递给解释器的参数。

该系统调用成功时，不会返回任何值（因为进程的执行映像已经被替换，没有接收返回值的地方了）。如果有任何返回值（一般是-1），就代表有错误发生，内核将设置相应的错误代码 errno，下面是一些可能设置的错误代码：

EACCES：指向的文件或脚本文件不是普通文件；指向的文件或脚本文件没有设置可执行位；文件系统被安装成 noexec；指向的文件或脚本文件所处的路径中有目录不能搜索（没有 execute 权）。

E2BIG：传递的参数清单太大。

① 关于 fctnl 系统调用的详细介绍，请参见附录 V。

② stdin, stdout 和 stderr 的文件描述符分别是 0，1，和 2

ENOEXEC：指定的文件确实有执行权限位，但是为即不可识别的执行文件格式。
ETXTBUSY：指定文件被一个或多个进程以可写入的方式打开。
EIO：从文件系统读入时发生 I/O 错误。

现在我们来看看这一族系统调用。在系统调用 execl、execlp、execle 中，参数是以 arg0、arg1、arg2、…的方式传递的。按照惯例，arg0 应该是要执行的程序名。在调用 execl、execlp 中环境变量的值是自动传递的，即不用象调用 execve、execle 那样在调用中指定参数 envp。在调用 execve、execv、execvp 中参数是以数组的方式传递的。另一个区别是，调用 execlp、execvp 可以在环境变量 PATH 定义的路径中查找执行程序。例如，PATH 定义为“/bin:/usr/bin:/usr/sbin”，如果调用指定执行文件名为 test，那么这两个调用会在 PATH 定义的三个目录中查找名为 test 的可执行文件。

系统调用 exec 和 fork 经常结合使用，父进程 fork 一个子进程，在子进程中调用 exec 来替换子进程的执行映像，并发的执行一些操作。

2.3 exit 系统调用

系统调用 exit 的功能是终止发出调用的进程。它的声明格式如下：

```
void _exit(int status);
```

在使用这个系统调用的程序中要加入以下头文件：

```
#include <unistd.h>
```

系统调用_exit[①]立即终止发出调用的进程。所有属于该进程的文件描述符都关闭。该进程的所有子进程由进程 1（进程 init）接收，并对该进程的父进程发出一个 SIGCHLD（子进程僵死）的信号。参数 status 作为退出的状态值返回父进程，该值可以通过系统调用 wait 来收集。返回状态码 status 只有最低一个字节有效。如果进程是一个控制终端进程，则 SIGHUP 信号将被送往该控制终端的前台进程。系统调用_exit 从不返回任何值给发出调用的进程；也不刷新 I/O 缓冲，如果要自动完成刷新，可以用函数调用 exit。

2.4 wait 系统调用

系统调用 wait 的功能是发出调用的进程只要有子进程，就睡眠直到它们中的一个终止为止。该调用声明的格式如下：

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

在使用这些系统调用的程序中要加入以下头文件：

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

① exit()其实调用的也是_exit()。_exit()立即中止当前工作，exit()则要做一些结束工作然后再调用_exit()回到系统内核。‘exit()’与‘_exit()’的基本区别在于前一个函数实施与调用库里用户状态结构有关的清除工作，而且调用用户自定义的清除程序；后一个函数只为进程实施内核清除工作。在由‘fork()’创建的子进程分支里，正常情况下使用‘exit()’是不正确的，这是因为使用它会导致标准输入输出的缓冲区被清空两次，而且临时文件被出乎意料的删除。

#include <sys/wait.h>

发出 wait 调用的进程进入睡眠直到它的一个子进程退出时或收到一个不能被忽略的信号时被唤醒。如果调用发出时，已经有退出的子进程（这时子进程的状态是僵死状态），该调用立即返回。其中调用返回时参数 status 中包含子进程退出时的状态信息。

调用 waitpid 与调用 wait 的区别是 waitpid 等待由参数 pid 指定的子进程退出。其中参数 pid 的含义与取值方法如下：

1) 参数 pid < -1 时，当退出的子进程满足下面条件时结束等待：该子进程的进程组 ID(process group)等于绝对值的 pid 这个条件。
2) 参数 pid = 0 时，等待任何满足下面条件的子进程退出：该子进程的进程组 ID 等于发出调用进程的进程组 ID。
3) 参数 pid > 0 时，等待进程 ID 等于参数 pid 的子进程退出。
4) 参数 pid = -1 时，等待任何子进程退出，相当于调用 wait。

对于调用 waitpid 中的参数 options 的取值及其含义如下：

WNOHANG：该选项要求如果没有子进程退出就立即返回。
WUNTRACED：对已经停止但未报告状态的子进程，该调用也从等待中返回和报告状态。

如果 status 不是空，调用将使 status 指向该信息。下面的宏可以用来检查子进程的返回状态。前面三个用来判断退出的原因，后面三个是对应不同的原因返回状态值：

1) WIFEXITED(status)：如果进程通过系统调用_exit 或函数调用 exit 正常退出，该宏的值为真。
2) WIFSIGNALED(status)：如果子进程由于得到的信号（signal）没有被捕捉而导致退出时，该宏的值为真。
3) WIFSTOPPED(status)：如果子进程没有终止，但停止了并可以重新执行时，该宏返回真。这种情况仅出现在 waitpid 调用中使用了 WUNTRACED 选项。
4) WEXITSTATUS(status)：如果 WIFEXITED(status)返回真，该宏返回由子进程调用_exit(status)或 exit(status)时设置的调用参数 status 值。
5) WTERMSIG(status)：如果 WIFSIGNALED(status)返回为真，该宏返回导致子进程退出的信号（signal）的值。
6) WSTOPSIG(status)：如果 WIFSTOPPED(status)返回真，该宏返回导致子进程停止的信号（signal）值。
该调用返回退出的子进程的 PID；或者发生错误时返回-1；或者设置了 WNOHANG 选项没有子进程退出就返回 0；发生错误时，可能设置的错误代码如下：

ECHILD：该调用指定的子进程 pid 不存在，或者不是发出调用进程的子进程。
EINVAL：参数 options 无效。
ERESTARTSYS：WNOHANG 没有设置并且捕获到 SIGCHLD 或其它未屏蔽信号。

关于 wait 调用的例子，前面在介绍 fork 调用时，就有了简单的应用。此处不再举例。

注意：子进程退出（SIGCHLD）信号设置不同的处理方法，会导致该调用不同的行为，详细情况见 Linux 信号处理机制。

2.5 sleep 函数调用

函数调用 sleep[①]可以用来使进程挂起指定的秒数。该函数调用的声明格式如下：

unsigned int sleep(unsigned int seconds)

在使用这个函数调用的程序中加上以下的头文件：

#include <unistd.h>

该函数调用使得进程挂起一个指定的时间，直到指定时间用完或者收到信号。系统的活动对指定的时间有一定的影响。Linux 系统是用 SIGALRM 实现的，在 Linux 系统里，sleep 函数不能和 alarm()调用混用。

如果指定挂起的时间到了，该调用返回 0；如果该函数调用被信号所打断，则返回剩余挂起的时间数（指定的时间减去已经挂起的时间）。

[①] 更精确的休眠时间使用 usleep，单位是微秒。

第三章　进程的特殊操作

摘要：本节要介绍一些有关进程的特殊操作。有了这些操作，就使得进程的编程更加完善，能编制更为实用的程序。主要的内容有得到关于进程的各种 ID、对进程的设置用户 ID、改变进程的工作目录、改变进程的根、改变进程的优先权值等操作。

3.进程的特殊操作

上一节介绍了有关进程的一些基本操作，如进程的产生（fork）、进程的终止（exit）、进程执行映像的改变（exec）、等待子进程终止（wait）等。本节要介绍一些有关进程的特殊操作。有了这些操作，就使得进程的编程更加完善，能编制更为实用的程序。

主要的内容有得到关于进程的各种 ID、对进程的设置用户 ID、改变进程的工作目录、改变进程的根、改变进程的优先权值等操作。

3.1 获得进程相关的 ID

与进程相关的 ID 有：

1) 真正用户标识号（UID）：该标识号负责标识运行进程的用户。
2) 有效用户标识号（EUID）：该标识号负责标识以什么用户身份来给新创建的进程赋所有权、检查文件的存取权限和检查通过系统调用 kill 向进程发送软中断信号的许可权限。
3) 真正用户组标识号(GID)：负责标识运行进程的用户所属的组 ID。
4) 有效用户组标识号(EGID)：用来标识目前进程所属的用户组。可能因为执行文件设置 set-gid 位而与 gid 不同。
5) 进程标识号(PID)：用来标识进程。
6) 进程组标识号(process group ID)：一个进程可以属于某个进程组。可以发送信号给一组进程。注意，它不同与 gid。前面的系统调用 wait 中指定参数 pid 时，就用到了进程组的概念。

如果要获得进程的用户标识号，用 getuid 调用。调用 geteuid 是用来获得进程的有效用户标识号。有效用户 ID 与真正用户 ID 的不同是由于执行文件设置 set-uid 位引起的。这两个调用的格式如下：

```
uid_t getuid(void);
uid_t geteuid(void);
```

在使用这两个调用的程序中加入下列头文件：

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

如果要获得运行进程的用户组 ID，使用 getgid 调用来获得真正的用户组 ID，用 getegid 获得有效的用户组 ID。标识 gid 与 egid 的不同是由于执行文件设置 set-gid 位引起的。这两个调用的格式如下：

```
gid_t getgid(void);
gid_t getegid(void);
```

在使用这两个调用的程序中加入下列头文件：

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

如果要获得进程的 ID，使用 getpid 调用；要获得进程的父进程的 ID，使用 getppid 调用。这两个调用的格式如下：

```
pid_t getpid(void);
pid_t getppid(void);
```

在使用这两个调用的程序中加入下列头文件：

```
#include <unistd.h>
```

如果要获得进程所属组的 ID，使用 getpgrp 调用；若要获得指定 PID 进程所属组的 ID 用 getpgid 调用。这两个调用的格式如下：

```
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

在使用这两个调用的程序中加入下列头文件：

```
#include <unistd.h>
```

注意一下 gid 和 pgrp 的区别，一般执行该进程的用户的组 ID 就是该进程的 gid，如果该执行文件设置了 set_gid 位，则文件所属的组 ID 就是该进程的 gid。对于进程组 ID，一般来说，一个进程在 shell 下执行，shell 程序就将该进程的 PID 赋给该进程的进程组 ID，从该进程派生的子进程都拥有父进程所属的进程组 ID，除非父进程将子进程的所属组 ID 设置成与该子进程的 PID 一样。由于这几个调用使用很简单，这里就不再举例。

3.2 setuid 和 setgid 系统调用

前面讲述了如何得到 uid 和 gid，现在来看看如何设置它们。在讲述这两个调用以前我们先来看看对文件设置 set_uid 位会有什么作用。我们先编了一个小程序来做试验。这个程序的作用是，打印出进程的 uid 和 euid，然后打开一个名为 tty.c 的文件。如果打不开，就显示错误代码；如果打开了，就显示打开成功。假设该程序名叫 uid_test.c：

```
/* uid_test.c */
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
extern int errno;

int main()
{
    int fd;
    printf("This process's uid = %d, euid = %d ",getuid(),geteuid());
    if ((fd = open("tty.c",O_RDONLY))==-1)
    {
        printf("Open error, errno is %d ",errno);
        exit(1);
    }
    else
    {
        printf("Open success ");
    }
}
```

下面列出这几个文件的目录，可以看到文件 tty.c 的存取许可权仅为属主 root 可读写。

```
[wap@wapgw /tmp]$ ls -l
total 3
-rw------- 1 root root 0 May 31 16：15 tty.c
-rwxr-xr-x 1 root root 14121 May 31 16：15 uid_test
-rw-r--r-- 1 root root 390 May 31 16：15 uid_test.c
[wap@wapgw /tmp]$
```

在该系统的用户中有个用户 wap（500），以 root 用户身份执行程序：

```
[root@wapgw /tmp]# ./uid_test
This process's uid = 0, euid = 0
Open success
[root@wapgw /tmp]#
```

下面使用 su 命令，转到用户 wap 下，执行程序

```
[root@wapgw /tmp]#su wap
[wap@wapgw /tmp]$ ./uid_test
This process's uid = 500, euid = 500
Open error, errno is 13
[wap@wapgw /tmp]$
```

这是由于进程的 uid 是 500(wap)，对文件 tty.c 没有存取权，所以出错。

给程序文件设置 set-uid 位

[root@wapgw /tmp]# chmod 4755[①] uid_test

再转到用户 wap 下，执行程序 uid_test。

```
[wap@wapgw /tmp]$ ./uid_test
This process's uid = 500, euid = 0
Open success
[wap@wapgw /tmp]$
```

从上面我们看到，进程打印出的 euid 是 0（root），而运行该进程的用户是 500（wap）。由于进程的 euid 是 root，所以成功打开了文件 tty.c。

上面的例子说明了两个事实：第一，内核对进程存取文件的许可权的检查，是通过检查进程的有效用户 ID 来实现的；第二，执行一个设置 set_uid 位的程序时，内核将进程表项中和 u 区[②]中的有效用户 ID 设置为文件属主的 ID。为了区别进程表项中的 euid 和 u 区中的 euid，我们将进程表项中的 euid 域称为保存用户标识号（saved user ID）。

下面我们来看看这两个调用。调用的声明格式如下：

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

在使用这两个调用的程序中加入下面的头文件：

```
#include <unistd.h>
```

调用 setuid 为当前发出调用的进程设置真正和有效用户 ID。参数 uid 是新的用户标识号（该标识号应该在/etc/passwd 文件中存在）。如果发出调用的进程的有效用户 ID（u 区中的）是超级用户，内核将进程表中和 u 区中的真正用户标识号和有效用户标识号置为参数 uid。如果发出调用的进程的有效用户 ID 不是超级用户，那么内核将根据指定的参数 uid 来执行，如果这时指定的参数 uid 的值是真正用户标识号或者是保存用户标识号(saved user ID)，则内核将 u 区中的有效用户标识号改为参数 uid，否则，该调用返回错误。该调用成功时，返回值为 0；发生错误时，返回-1，并设置相应的错误代码 errno，下面是经常可能发生的错误代码：

EPERM：用户不是超级用户，并且指定的参数 uid 与发出调用的进程的真正用户 ID 或保存用户 ID 不匹配。

---

① 使可执行文件具有与被操作文件属主同样的权力。关于更改文件权限的命令 chmod 的详细介绍，请参见附录 VI。

② u 区（u Area），用于存放进程表项的一些扩充信息。每一个进程都有一个私用的 u 区，其中含有：进程表项指针、真正用户标识符 u-ruid(real user ID)、有效用户标识符 u-euid(effective user ID)、用户文件描述符表、计时器、内部 I/O 参数、限制字段、差错字段、返回值、信号处理数组。

调用 setgid 设置当前发出调用的进程的真正、有效用户组 ID。该调用允许进程指定进程的用户组 ID 为参数 gid，如果进程的有效用户 ID 不是超级用户，该参数 gid 必须等于真正用户组 ID、有效用户组 ID 中的一个。如果进程的有效用户 ID 是超级用户，可以指定任何存在的用户组 ID（在 /etc/group 文件中存在）。

注意： 对于 setuid 程序尤其要小心，当进程的 euid 是超级用户时，如果将进程 setuid 到其他用户，就无法再得到超级用户的权力。我们可能这样用这个调用，某个程序，开始需要 root 权力才能完成开始的工作，但后续的工作不需要 root 的权力，所以，我们将程序的执行文件设置 set_uid 位，并使得执行文件的属主是 root，这样进程开始时，就具有了 root 的权限，在不再需要 root 权限的地方，用 setuid(getuid)恢复进程的 uid、 euid。对于可执行文件设置 set_uid 位，一定要注意，尤其是对那些属主是 root 的更要注意。因为 LINUX 系统中 root 拥有任何权力。使用不当，会对系统安全有极大的损害。

3.3 setpgrp 和 setpgid 系统调用

这两个调用是用来设置进程组 ID 的，其声明格式如下：

```
int setpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

在使用这两个调用的程序中加入下面的头文件：

```
#include <unistd.h>
```

调用 setpgrp 用来将发出调用的进程的进程组 ID 设置成与该进程的 PID 相等。注意，以后由这个进程派生的子进程都拥有该进程组 ID（除非修改子进程的进程组 ID）。

调用 setpgid 用来将进程号为参数 pid 的进程的进程组 ID 设定为参数 pgid。如果参数 pid 为 0，则修改发出调用进程的进程组 ID。如果参数 pgid 为 0，将进程号为 pid 的进程改为与发出调用的进程同组。如果不是超级用户发出的调用，那么被指定的进程必须与发出调用的进程有相同的 EUID，或者被指定的进程是发出调用进程的子进程。

进程组可用于信号的发送，或者终端输入的仲裁（与终端控制进程有相同的进程组 ID 且在前台可以读取终端，其他进程在企图读的时候被阻塞并发送信号给该进程）。

该调用成功时，返回值为 0；如果请求失败，返回-1，并设置全局变量 errno 为对应的值。下面是可能遇到的错误代码：

ESRCH：参数 pid 指定的进程不存在。
EINVAL：参数 pgid 小于 0。
EPERM：指定进程的 EUID 与发出调用进程的 euid 不同，且指定进程不是发出调用进程的子进程。

3.4 chdir 系统调用

chdir 是用来将进程的当前工作目录改为由参数指定的目录。该调用的声明格式如下：

int chdir(const char *path);

在使用该调用的程序中加入下面的头文件：

#include <unistd.h>

使用该调用时要注意，发出该调用的进程必须对参数 path 指定的目录有搜索(execute)的权力。调用成功时，返回值为 0；错误时，返回-1，并设置相应的错误代码。

3.5 chroot 系统调用

系统调用 chroot 用来改变发出调用进程的根（“/”）目录。该调用声明的格式如下：

int chroot(const char *path);

在使用该调用的程序中加入下面的头文件：

#include <unistd.h>

调用 chroot 将进程的根目录改到由参数 path 所指定的地方。以后该进程中以“/”（根）开始的路径，都从指定目录处开始查找。发出调用进程的子进程都继承这个根目录的位置。该调用只能由超级用户（root）发出。注意，该调用并不改变当前工作目录，所以有可能当前工作目录“.”在根目录“/” 之外。调用成功时，返回值为 0；错误时，返回-1，并设置相应的错误代码。

注意： 如果用 chroot 调用改变根后，不能由调用 chroot(“/”)来返回真正的根，因为调用中的参数“/”会被理解成新设置的根。该调用一般可以用在 login 程序中，或者现在国内常见的 BBS 系统等应用程序中，用户登录后执行系统的一个程序，该程序将根改变成用户登录的目录（例如 /home/bbs）。这样使用的好处是，利于调试和安装；也利于安全。

3.6、nice 系统调用

系统调用 nice 用来改变进程的优先权。该调用的声明格式如下：

int nice(int inc);

在使用该调用的程序中加入下面的头文件：

#include <unistd.h>

调用 nice 将发出调用进程的优先权值增加 inc 大小。只有超级用户才有权指定一个负的增加量 inc。发出调用的进程的子进程都继承该优先权。注意，进程的优先权值越低，优先权越高，即优先权值越低，调度上 CPU 的机会越大。所以只有 root 才能指定负值，一般用户只能指定正值，该值降低了进程的优先权，

使得进程使用更少的 CPU 时间。

该调用成功返回时，返回值为 0；错误时，返回-1，并设相应的错误代码：

EPERM：非超级用户指定参数 inc 为负值，企图增加进程的优先权。

这个调用适用于你的程序需要长时间运行，你不希望它对别的进程影响过大。而且执行的快慢对你来说并不十分重要这种情况。

第四章　进程的通信

摘要：这一节，我们来看一种比较简单的数据传送的方法，即通过管道传送数据

4. 进程间使用管道通信

前几节中我们讲述了有关进程的操作，我们已经学会产生一个新的进程，改变进程的执行映像等操作。然而，子进程与父进程，子进程与子进程之间，还缺少数据交换的方法。这一节，我们就来看一种比较简单的数据传送的方法，即通过管道传送数据。

管道允许在进程之间按先进先出的方式传送数据，管道也能使进程同步执行。管道传统的实现方法是通过文件系统作为存储数据的地方。有两种类型的管道：一种是无名管道，简称为管道；另一种是有名管道，也称为 FIFO。进程使用系统调用 open 来打开有名管道，使用系统调用 pipe 来建立无名管道。使用无名管道通讯的进程，必须是发出 pipe 调用的进程及其子进程。使用有名管道通讯的进程没有上述限制。在以后的叙述中，所有无名管道都简称为管道。下面来看一下系统调用 pipe。

4.1 pipe 系统调用

系统调用 pipe 是用来建立管道的。该调用的声明格式如下：

int pipe(int filedes[2]);

在使用该调用的程序中加入下面的头文件：

#include <unistd.h>

一个管道拥有两个文件描述符用来通信，它们指向一个管道的索引节点，该调用将这两文件描述符放在参数 filedes 中返回。现在的许多系统中管道允许数据双向流动，但一般习惯上，文件描述符 filedes[0] 用来读数据，filedes[1]用来写数据。如果要求程序的可移植性好，就按照习惯的用法来编程。调用成功时，返回值为 0；错误时，返回-1，并设置错误代码 errno：

EMFILE：进程使用了过多的文件描述符。
ENFILE：系统文件表满。
EFAULT：参数 filedes 无效。
下面介绍管道的操作的情况：

对于写管道：
1) 写入管道的数据按到达次序排列。如果管道满，则对管道的写被阻塞，直到管道的数据被读操作读取。对于写操作，如果一次 write 调用写的数据量小于管道容量，则写必须一次完成，即如果管道所剩余的容量不够，write 被阻塞直到管道的剩余容量可以一次写完为止。如果 write 调用写的数据量大于管道容量，则写操作分多次完成。如果用 fcntl 设置管道写端口为非阻塞方式，则管道满不会阻塞写，而只是对写返回 0。

2) 对于读管道：

读操作按数据到达的顺序读取数据。已经被读取的数据在管道内不再存在，这意味着数据在管道中不能重复利用。如果管道为空，且管道的写端口是打开状态，则读操作被阻塞直到有数据写入为止。一次 read 调用，如果管道中的数据量不够 read 指定的数量，则按实际的数量读取，并对 read 返回实际数量值。如果读端口使用 fcntl 设置了非阻塞方式，则当管道为空时，read 调用返回 0。

3) 对于管道的关闭：

如果管道的读端口关闭，那么在该管道上的发出写操作调用的进程将接收到一个 SIGPIPE 信号。关闭写端口是给读端口一个文件结束符的唯一方法。对于写端口关闭后，在该管道上的 read 调用将返回 0。下面再来看看，系统调用 pipe 的例子。在下面的例子中，父进程通过管道向子进程发送了一个字符串。子进程将它显示出来：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
int main()
{
    int fd[2],cld_pid,status;
    char buf[200], len;

    if (pipe(fd) == -1)
    {
        printf("creat pipe error ");
        exit(1);
    }
    if ((cld_pid=fork()) == 0)
    {
        close(fd[1]);
        len = read(fd[0],buf,sizeof(buf));
        buf[len]=0;
        printf("Child read pipe is -- %s ",buf);
        exit(0);
    }
    else
    {
        close(fd[0]);
        sprintf(buf,"Parent creat this buff for cld %d",cld_pid);
        write(fd[1],buf, strlen(buf));
        exit(0);
    }
}
```

该程序执行过程的屏幕拷贝：

```
[root@wapgw /tmp]# ./pipe
[root@wapgw /tmp]# Child read pipe is -- Parent creat this buff for cld 5954
[root@wapgw /tmp]#
```

4.2 dup 系统调用

系统调用 dup 是用来复制一个文件描述符，也就是将进程 u 区的文件描述符表中的一项复制一份，使得这两项同时指向系统文件表的同一表项。该调用的声明格式如下：

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

在使用该调用的程序中加入下面的头文件：

```
#include <unistd.h>
```

系统调用 dup 复制由参数 oldfd 指定的文件描述到进程文件描述符表的第一个空表项处。而系统调用 dup2 复制由参数 oldfd 指定的文件描述到参数 newfd 指定的文件描述符表项处。老的文件描述符和新复制的文件描述符可以互换使用。它们共享锁、文件指针和文件状态。例如，对其中一个文件描述符使用系统调用 lseek 修改文件指针的位置，对另一文件描述符来说文件指针也改变了，其实我们了解了内核的工作原理，这一点很容易理解。因为我们知道，文件指针是放在系统文件表中的。但这两个文件描述符具有不同的 close-on-exec 标志，因为该标志是存放在文件描述符表中的[①]。

该调用成功时，返回值为新的描述符；错误时，返回-1，并设置相应的错误代码 errno：

EBADF：参数 oldfd 不是一个已经打开的文件描述符；或者参数 newfd 超出允许的文件描述符的取值范围。

EMFILE：进程打开的文件描述符数量已经到达最大值，但仍然企图打开新的文件描述符。

下面我们来看一个简单的例子。在这个例子中，我们将标准输出（文件描述符为 1）关闭，并将一个打开了普通文件“output”的文件描述符复制到标准输出上，因为刚关闭了文件描述符 1，所以，文件描述符表的第一个空表项是 1。所以，程序以后的 printf 等向标准输出写的内容都写到了文件 output 中。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
    int fd;
```

[①] 文件子系统一共有 3 个结构表——用户文件描述符表，文件表和索引结点表(inode table)，用这 3 种结构表中的表项来维护文件的状态以及用户对它的存取。

```
    if ((fd=open("output",O_CREAT|O_RDWR,0644))==-1)
    {
        printf("cannot open output file ");
        exit(1);
    }
    close(1); /*  关闭标准输出  */
    dup(fd); /*  复制 fd 到文件描述符 1 上  */
    close(fd); /*  即时关闭不用的文件描述符是一个好习惯  */
    printf("This line will write to file ");
    exit(0);
}
```

该程序执行过程的屏幕拷贝：

```
[wap@wapgw /tmp]$ gcc -o dup_test dup_test.c
[wap@wapgw /tmp]$ ./dup_test
[wap@wapgw /tmp]$ more output
This line will write to file
[wap@wapgw /tmp]$
```

附录 I：内核态和用户态

1. 内核态和用户态

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0 级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3 级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。

简言之：

1、用系统调用时进入核心态。Linux 对硬件的操作只能在核心态，这可以通过写驱动程序来控制。在用户态操作硬件会造成 core dump。

2、要注意区分系统调用和一般的函数。系统调用由内核提供，如 read()、write()、open()等。而一般的函数由软件包中的函数库提供，如 sin()、cos()等。在语法上两者没有区别。

3、一般情况：系统调用运行在核心态，函数运行在用户态。但也有一些函数在内部使用了系统调用（如 fopen），这样的函数在调用系统调用是进入核心态，其他时候运行在用户态。

2. 进程上下文和中断上下文

处理器总处于以下状态中的一种：

1、内核态，运行于进程上下文，内核代表进程运行于内核空间；

2、内核态，运行于中断上下文，内核代表硬件运行于内核空间；

3、用户态，运行于用户空间。

用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存器值、变量等。所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。

硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。所谓的“中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。

附录 II：进程调度

首先了解一下用户模式和内核模式。

一般说来，进程既可在用户模式下运行，又可在内核模式下运行。内核模式的权限高于用户模式的权限。进程每次调用一个系统调用时，进程的运行方式都发生变化：从用户模式切换到内核模式，然后继续执行。

可见：一个进程在 CPU 上运行可以有两种运行模式（进程状态）：用户模式和内核模式。如果当前运行的是用户程序（用户代码），那么对应进程就处于用户模式（用户态），如果出现系统调用或者发生中断，那么对应进程就处于内核模式（核心态）。

在 UNIX 系统上，内核对外的接口是系统调用，系统调用以 C 函数的形式出现，所以内核外的程序都必须经由系统调用才能获得操作系统的服务。那么系统调用在核心模式运行和用户代码在用户模式运行有什么区别呢？

由于系统调用能直接进入内核执行，所以其执行效率很高。系统调用包含一种特殊的程序段，这些程序段称为原语，其特点是必须作为整体执行，不允许被中断，也不允许并发执行。你说效率高不高？

任何进程要想占有 CPU，从而真正处于执行状态，就必须经由进程调度。进程调度机制主要涉及到调度方式、调度时机和调度策略。

1．调度方式

Linux 内核的调度方式基本上采用“抢占式优先级”方式，即当进程在用户模式下运行时，不管是否自愿，在一定条件下(如时间片用完或等待 I/O)，核心就可以暂时剥夺其运行而调度其它进程进入运行。但是，一旦进程切换到内核模式下运行，就不受以上限制而一直运行下去，直至又回到用户模式之前才会发生进程调度。

Linux 系统中的调度策略基本上继承了 Unix 的以优先级为基础的调度。就是说，核心为系统中每个进程计算出一个优先权，该优先权反映了一个进程获得 CPU 使用权的资格，即高优先权的进程优先得到运行。

核心从进程就绪队列中挑选一个优先权最高的进程，为其分配一个 CPU 时间片，令其投入运行。在运行过程中，当前进程的优先权随时间递减，这样就实现了“负反馈”作用：经过一段时间之后，原来级别较低的进程就相对“提升”了级别，从而有机会得到运行。当所有进程的优先权都变为 0 时，就重新计算一次所有进程的优先权。

2．调度策略

Linux 系统针对不同类别的进程提供了三种不同的调度策略，即 SCHED_FIFO、SCHED_RR 及 SCHED_OTHER。其中：

SCHED_FIFO 适合于实时进程，它们对时间性要求比较强，而每次运行所需的时间比较短，一旦这种

进程被调度开始运行后，就要一直运行到自愿让出 CPU，或者被优先权更高的进程抢占其执行权为止。

SCHED_RR 对应“时间片轮转法”，适合于每次运行需要较长时间的实时进程。一个运行进程分配一个时间片(如 200 毫秒)，当时间片用完后，CPU 被另外进程抢占，而该进程被送回相同优先级队列的末尾。

SCHED_OTHER 是传统的 Unix 调度策略，适合于交互式的分时进程。这类进程的优先权取决于两个因素，一个因素是进程剩余时间配额，如果进程用完了配给的时间，则相应优先权为 0；另一个是进程的优先数 nice，这是从 Unix 系统沿袭下来的方法，优先数越小，其优先级[①]越高。

nice 的取值范围是 19～-20。用户可以利用 nice 命令设定进程的 nice 值。但一般用户只能设定正值，从而主动降低其优先级；只有特权用户才能把 nice 的值置为负数。进程的优先权就是以上二者之和。核心动态调整用户态进程的优先级。

这样，一个进程从创建到完成任务后终止，需要经历多次反馈循环。当进程再次被调度运行时，它就从上次断点处开始继续执行。

对于实时进程，其优先权的值是(1000＋设定的正值)，因此，至少是 1000。所以，实时进程的优先权高于其它类型进程的优先权。

另外，时间配额及 nice 值与实时进程的优先权无关。如果系统中有实时进程处于就绪状态，则非实时进程就不能被调度运行，直至所有实时进程都完成了，非实时进程才有机会占用 CPU。

后台命令（在命令末尾有&符号，如 gcc f1.c&)对应后台进程（又称后台作业），后台进程的优先级低于任何交互（前台）进程的优先级。所以，只有当系统中当前不存在可运行的交互进程时，才调度后台进程运行。后台进程往往按批处理方式调度运行。

3．调度时机

核心进行进程调度的时机有以下几种情况：(1)当前进程调用系统调用 nanosleep( )或 pause( )使自己进入睡眠状态，主动让出一段时间的 CPU 使用权；(2)进程终止，永久地放弃对 CPU 的使用；(3)在时钟中断处理程序执行过程中，发现当前进程连续运行的时间过长；(4)当唤醒一个睡眠进程时，发现被唤醒的进程比当前进程更有资格运行；(5)一个进程通过执行系统调用来改变调度策略或降低自身的优先权(如 nice 命令)，从而引起立即调度。

4．调度算法

进程调度的算法应该比较简单，以便减少频繁调度时的系统开销。Linux 执行进程调度时，首先查找所有在就绪队列中的进程，从中选出优先级最高且在内存的一个进程。如果队列中有实时进程，那么实时进程将优先运行。

如果最需要运行的进程不是当前进程，那么当前进程就被挂起，并且保存它的现场所涉及的一切机器状态，包括程序计数器和 CPU 寄存器等，然后为选中的进程恢复运行现场。

[①] “优先权”等价于“优先级”，“优先权值”等价于“优先数”。优先数与优先权的比例关系因操作系统而异，正常情况下，应该成正比（即越大越大），但对于 Unix（Linux）则是成反比（即越小越大）。

附录 III：索引结点表

1. 索引结点表

索引结点（index node，简称 inode）是文件系统的核心内容，也是内核最重要的数据结构。unix 中一个文件只有一个 inode 与其相对应，它储存在磁盘的索引结点表里。索引结点表是一个线性数组。索引结点的各个域的内容如下：

1. 文件所有者的标识号
2. 文件类型
3. 文件的存取权限
4. 文件的存取时间
5. 文件的联结数[①]
6. 文件在磁盘上的位置
7. 文件大小

（注意：索引结点不表明该文件的路径）

当索引结点被某个进程引用是，它被调入内存中。在内存中存在另外一个索引结点表（in-core inode list）和一个空闲索引结点表。当索引结点被调入内存时（在内核中使用 iget 算法），将索引结点放入内存索引结点表里，并将其引用数加 1。

内存索引结点里的数据域除了磁盘索引结点的内容以外，还有以下新的域：

1. 内存索引结点的状态
2. 含有该文件的文件系统的逻辑设备号
3. 索引结点号（索引结点在磁盘上数组里的索引）
4. 指向其他内存索引结点的指针
5. 引用数

内核用文件系统和索引结点号来标识特定的索引结点，在高层的系统调用请求是使用 iget 算法来分配一个索引结点的内存拷贝，即内存索引结点。内核独立的操纵索引结点锁和引用数。在系统调用执行期间将起锁住，防止其他的进程在此期间使用此结点，调用结束后开锁。在两次系统调用之间结点是不上锁的。

而 iget 算法主要完成的任务：索引结点号为参数，如果索引结点位于内存索引结点表中则引用数加 1，返回索引结点，如果索引结点位于空闲索引结点表中（此表中索引结点的引用数为 0），则从该表中取出该索引结点，放入在内存索引结点表中，并将其引用数加 1，如果在两个表中都没有该索引结点，则在空闲表中移出一个新的索引结点，在磁盘索引结点表中寻找到该结点，读入内存索引结点表中，然后返回该索引结点。

当内核释放索引结点时，引用数减 1。如果引用数降为 0，而且文件的内存拷贝与磁盘拷贝不同，则执行写磁盘操作，并把索引结点放入空闲表里。这里用到的算法是 iput。如果文件的联结数为 0，则内核释放该文件的所有数据块，并释放该文件的磁盘索引结点。

[①] 文件联结数，或称文件链接数，相当于 windows 下的快捷方式，连接有软硬之分。链接数指的是有多少个链接文件指向这个文件具体的请 man ln。

2. 正规的文件结构

索引结点包含着文件在磁盘上位置的明细表。磁盘的每个块都编了号。在 unix 系统 V 中，块的大小是 1k，（在 windows 里为 4k），索引结点里包含指向各个数据块的指针，其中有 10 个直接块，1 个 1 次间接块，1 个 2 次间接快，1 个 3 次间接块，如果使用到 3 次间接的话，单个文件的大小可以最大为 16G，存取文件时，内核调用 bmap 算法将逻辑文件的字节偏移量映射到文件系统的块上。

3. 目录

目录是使文件系统具有树型结构的那些文件。目录是文件，它只是数据是一些目录表项的文件，每个目录表项由他们的索引结点号和此目录下的文件名构成，路径就是由分割成的各个独立分量构成的。

目录的读权限为允许进程读目录，写权限为允许进程创建子目录或删除子目录，执行权限为允许进程寻找文件而搜索整个目录。

注意：读权限与执行权限的区别。

附录 IV Linux 启动过程

PC 硬件启动的过程大概是硬件首先加电自检，然后启动 BIOS，通过 BIOS 把操作系统的内核加载到内存中。

Linux 也是一样，在启动 BIOS 后，BIOS 会去查找 MBR（Master Boot Record)即是主引导区，主引导区共 512 字节，它是第一个硬盘的第一个磁道的第一个扇区，其中 64 byte 记录分区表，余下的 446 字节对于 windows 来说就是一个很简单的引导程序。如果是 windows 和 linux 的双系统的话在该 446 字节中管理多个操作系统的内核，并负责启动某个系统的内核。

然后 BIOS 根据 MBR 把内核装入内存,内核一般为压缩包，通过内核头部的一个解压缩程序解压缩，之后 BIOS 把控制权交给内核，内核开始初始化，之后内核调用 start_kernel 函数启动内核的接口，通过该接口创建 1#进程。1#进程实际上是个不完整的进程，也可以认为它是 0#进程，然后 0#进程创建若干个线程（例如收集内存、/etc/initab/等等）其中的一个线程/sbin/init/从次 1#进程变成一个普通的进程，开始以/etc/inittab 为基础进行系统初始化，/etc/rc.d/rc.sysinit(这是个 shell scripts 程序)以这个文件为基础开始启动系统服务，然后查看/etc/rc.d/rc 的系统运行级别$runlevel，（linux 运行级别共 0～6 级分别为：0-halt 终止停顿、1-单用户、2-多用户、3-多用户并启动 nfs、4-保留、5-运行 X windows、6-reboot）然后依据/etc/rc.d/rc.local →/sbin/mingetty(getty）启动用户登录进程。

然后就进入 shell 等待命令输入，用户输入命令后通过键盘监控进程将字符串与/sbin/的命令比较，然后调用内核工具完成操作。

==========

内容：

本文以 Redhat 6.0 Linux 2.2.19 for Alpha/AXP 为平台，描述了从开机到登录的 Linux 启动全过程。该文对 i386 平台同样适用。

一. Bootloader

在 Alpha/AXP 平台上引导 Linux 通常有两种方法，一种是由 MILO 及其他类似的引导程序引导，另一种是由 Firmware 直接引导。MILO 功能与 i386 平台的 LILO 相近，但内置有基本的磁盘驱动程序（如 IDE、SCSI 等），以及常见的文件系统驱动程序（如 ext2，iso9660 等）， firmware 有 ARC、SRM 两种形式，ARC 具有类 BIOS 界面，甚至还有多重引导的设置；而 SRM 则具有功能强大的命令行界面，用户可以在控制台上使用 boot 等命令引导系统。ARC 有分区（Partition）的概念，因此可以访问到分区的首扇区；而 SRM

只能将控制转给磁盘的首扇区。两种 firmware 都可以通过引导 MILO 来引导 Linux，也可以直接引导 Linux 的引导代码。

“arch/alpha/boot” 下就是制作 Linux Bootloader 的文件。“head.S”文件提供了对 OSF PAL/1 的调用入口，它将被编译后置于引导扇区（ARC 的分区首扇区或 SRM 的磁盘 0 扇区），得到控制后初始化一些数据结构，再将控制转给“main.c”中的 start_kernel()，start_kernel()向控制台输出一些提示，调用 pal_init()初始化 PAL 代码，调用 openboot() 打开引导设备（通过读取 Firmware 环境），调用 load()将核心代码加载到 START_ADDR（见 “include/asm-alpha/system.h”），再将 Firmware 中的核心引导参数加载到 ZERO_PAGE(0) 中，最后调用 runkernel()将控制转给 0x100000 的 kernel，bootloader 部分结束。

“arch/alpha/boot/bootp.c”以“main.c”为基础，可代替“main.c”与“head.S” 生成用于 BOOTP 协议网络引导的 Bootloader。

Bootloader 中使用的所有“srm_”函数在“arch/alpha/lib/”中定义。

以上这种 Boot 方式是一种最简单的方式，即不需其他工具就能引导 Kernel，前提是按照 Makefile 的指导，生成 bootimage 文件，内含以上提到的 bootloader 以及 vmlinux，然后将 bootimage 写入自磁盘引导扇区始的位置中。

当采用 MILO 这样的引导程序来引导 Linux 时，不需要上面所说的 Bootloader，而只需要 vmlinux 或 vmlinux.gz，引导程序会主动解压加载内核到 0x1000（小内核）或 0x100000（大内核），并直接进入内核引导部分，即本文的第二节。

对于 I386 平台

i386 系统中一般都有 BIOS 做最初的引导工作，那就是将四个主分区表中的第一个可引导 分区的第一个扇区加载到实模式地址 0x7c00 上，然后将控制转交给它。

在“arch/i386/boot” 目录下，bootsect.S 是生成引导扇区的汇编源码，它首先将自己拷贝到 0x90000 上，然后将紧接其后的 setup 部分（第二扇区）拷贝到 0x90200，将真正的内核代码拷贝到 0x100000。以上这些拷贝动作都是以 bootsect.S、setup.S 以及 vmlinux 在磁盘上连续存放为前提的，也就是说，我们的 bzImage 文件或者 zImage 文件是按照 bootsect，setup， vmlinux 这样的顺序组织，并存放于始于引导分区的首扇区的连续磁盘扇区之中。

bootsect.S 完成加载动作后，就直接跳转到 0x90200，这里正是 setup.S 的程序入口。 setup.S 的主要功能就是将系统参数（包括内存、磁盘等，由 BIOS 返回）拷贝到 0x90000-0x901FF 内存中，这个地方正是 bootsect.S 存放的地方，这时它将被系统参数覆盖。以后这些参数将由保护模式下的代码来读取。

除此之外，setup.S 还将 video.S 中的代码包含进来，检测和设置显示器和显示模式。最 后，setup.S 将系统转换到保护模式，并跳转到 0x100000（对于 bzImage 格式的大内核是 0x100000，对于 zImage 格式的是 0x1000）的内核引导代码，Bootloader 过程结束。

对于 2.4.x 版内核

没有什么变化。

二.Kernel 引导入口

在 arch/alpha/vmlinux.lds 的链接脚本控制下，链接程序将 vmlinux 的入口置于"arch/alpha/kernel/head.S"中的__start 上，因此当 Bootloader 跳转到 0x100000 时， __start 处的代码开始执行。__start 的代码很简单，只需要设置一下全局变量，然后就跳转到 start_kernel 去了。start_kernel()是"init/main.c"中的 asmlinkage 函数，至此，启动过程转入体系结构无关的通用 C 代码中。

对于 I386 平台

在 i386 体系结构中，因为 i386 本身的问题，在 "arch/alpha/kernel/head.S"中需要更多的设置，但最终也是通过 call SYMBOL_NAME(start_kernel)转到 start_kernel()这个体系结构无关的函数中去执行了。

所不同的是，在 i386 系统中，当内核以 bzImage 的形式压缩，即大内核方式（__BIG_KERNEL__）压缩时就需要预先处理 bootsect.S 和 setup.S，按照大核模式使用$(CPP) 处理生成 bbootsect.S 和 bsetup.S，然后再编译生成相应的.o 文件，并使用 "arch/i386/boot/compressed/build.c"生成的 build 工具，将实际的内核（未压缩的，含 kernel 中的 head.S 代码）与"arch/i386/boot/compressed"下的 head.S 和 misc.c 合成到一起，其中的 head.S 代替了"arch/i386/kernel/head.S"的位置，由 Bootloader 引导执行（startup_32 入口），然后它调用 misc.c 中定义的 decompress_kernel()函数，使用 "lib/inflate.c"中定义的 gunzip()将内核解压到 0x100000，再转到其上执行 "arch/i386/kernel/head.S"中的 startup_32 代码。

对于 2.4.x 版内核

没有变化。

三.核心数据结构初始化——内核引导第一部分

start_kernel()中调用了一系列初始化函数，以完成 kernel 本身的设置。 这些动作有的是公共的，有的则是需要配置的才会执行的。

在 start_kernel()函数中，

输出 Linux 版本信息（printk(linux_banner)）

设置与体系结构相关的环境（setup_arch()）

页表结构初始化（paging_init()）

使用"arch/alpha/kernel/entry.S"中的入口点设置系统自陷入口（trap_init()）

使用 alpha_mv 结构和 entry.S 入口初始化系统 IRQ（init_IRQ()）

核心进程调度器初始化（包括初始化几个缺省的 Bottom-half，sched_init()）

时间、定时器初始化（包括读取 CMOS 时钟、估测主频、初始化定时器中断等，time_init()）

提取并分析核心启动参数（从环境变量中读取参数，设置相应标志位等待处理，（parse_options()）

控制台初始化（为输出信息而先于 PCI 初始化，console_init()）

剖析器数据结构初始化（prof_buffer 和 prof_len 变量）

核心 Cache 初始化（描述 Cache 信息的 Cache，kmem_cache_init()）

延迟校准（获得时钟 jiffies 与 CPU 主频 ticks 的延迟，calibrate_delay()）

内存初始化（设置内存上下界和页表项初始值，mem_init()）

创建和设置内部及通用 cache（"slab_cache"，kmem_cache_sizes_init()）

创建 uid taskcount SLAB cache（"uid_cache"，uidcache_init()）

创建文件 cache（"files_cache"，filescache_init()）

创建目录 cache（"dentry_cache"，dcache_init()）

创建与虚存相关的 cache（"vm_area_struct"，"mm_struct"，vma_init()）

块设备读写缓冲区初始化（同时创建"buffer_head"cache 用户加速访问，buffer_init()）
创建页 cache（内存页 hash 表初始化，page_cache_init()）
创建信号队列 cache（"signal_queue"，signals_init()）
初始化内存 inode 表（inode_init()）
创建内存文件描述符表（"filp_cache"，file_table_init()）
检查体系结构漏洞（对于 alpha，此函数为空，check_bugs()）
SMP 机器其余 CPU（除当前引导 CPU）初始化（对于没有配置 SMP 的内核，此函数为空，smp_init()）
启动 init 过程（创建第一个核心线程，调用 init()函数，原执行序列调用 cpu_idle() 等待调度，init()）
至此 start_kernel()结束，基本的核心环境已经建立起来了。

对于 I386 平台
i386 平台上的内核启动过程与此基本相同，所不同的主要是实现方式。

对于 2.4.x 版内核
2.4.x 中变化比较大，但基本过程没变，变动的是各个数据结构的具体实现，比如 Cache。

四.外设初始化——内核引导第二部分

init()函数作为核心线程，首先锁定内核（仅对 SMP 机器有效），然后调用 do_basic_setup()完成外设及其驱动程序的加载和初始化。过程如下：

总线初始化（比如 pci_init()）
网络初始化（初始化网络数据结构，包括 sk_init()、skb_init()和 proto_init()三部分，在 proto_init()中，将调用 protocols 结构中包含的所有协议的初始化过程，sock_init()）
创建 bdflush 核心线程（bdflush()过程常驻核心空间，由核心唤醒来清理被写过的内存缓冲区，当 bdflush()由 kernel_thread()启动后，它将自己命名为 kflushd）
创建 kupdate 核心线程（kupdate()过程常驻核心空间，由核心按时调度执行，将内存缓冲区中的信息更新到磁盘中，更新的内容包括超级块和 inode 表）
设置并启动核心调页线程 kswapd（为了防止 kswapd 启动时将版本信息输出到其他信息中间，核心线调用 kswapd_setup()设置 kswapd 运行所要求的环境，然后再创建 kswapd 核心线程）
创建事件管理核心线程（start_context_thread()函数启动 context_thread()过程，并重命名为 keventd）
设备初始化（包括并口 parport_init()、字符设备 chr_dev_init()、块设备 blk_dev_init()、SCSI 设备 scsi_dev_init()、网络设备 net_dev_init()、磁盘初始化及分区检查等等， device_setup()）
执行文件格式设置（binfmt_setup()）
启动任何使用__initcall 标识的函数（方便核心开发者添加启动函数，do_initcalls()）
文件系统初始化（filesystem_setup()）
安装 root 文件系统（mount_root()）

至此 do_basic_setup()函数返回 init()，在释放启动内存段（free_initmem()）并给内核解锁以后，init()打开 /dev/console 设备，重定向 stdin、stdout 和 stderr 到控制台，最后，搜索文件系统中的 init 程序（或者由 init=命令行参数指定的程序），并使用 execve()系统调用加载执行 init 程序。

init()函数到此结束，内核的引导部分也到此结束了，这个由 start_kernel()创建的第一个线程已经成为一个用户模式下的进程了。此时系统中存在着六个运行实体：

start_kernel()本身所在的执行体，这其实是一个"手工"创建的线程，它在创建了 init()线程以后就进入 cpu_idle()循环了，它不会在进程（线程）列表中出现

init 线程，由 start_kernel()创建，当前处于用户态，加载了 init 程序

kflushd 核心线程，由 init 线程创建，在核心态运行 bdflush()函数

kupdate 核心线程，由 init 线程创建，在核心态运行 kupdate()函数

kswapd 核心线程，由 init 线程创建，在核心态运行 kswapd()函数

keventd 核心线程，由 init 线程创建，在核心态运行 context_thread()函数

对于 I386 平台

基本相同。

对于 2.4.x 版内核

这一部分的启动过程在 2.4.x 内核中简化了不少，缺省的独立初始化过程只剩下网络 （sock_init()）和创建事件管理核心线程，而其他所需要的初始化都使用__initcall()宏 包含在 do_initcalls()函数中启动执行。

五.init 进程和 inittab 引导指令

init 进程是系统所有进程的起点，内核在完成核内引导以后，即在本线程（进程）空 间内加载 init 程序，它的进程号是 1。

init 程序需要读取/etc/inittab 文件作为其行为指针，inittab 是以行为单位的描述性（非执行性）文本，每一个指令行都具有以下格式：

id:runlevel:action:process 其中 id 为入口标识符，runlevel 为运行级别，action 为动作代号，process 为具体的执行程序。

id 一般要求 4 个字符以内，对于 getty 或其他 login 程序项，要求 id 与 tty 的编号相同，否则 getty 程序将不能正常工作。

runlevel 是 init 所处于的运行级别的标识，一般使用 0－6 以及 S 或 s。0、1、6 运行级别被系统保留，0 作为 shutdown 动作，1 作为重启至单用户模式，6 为重启；S 和 s 意义相同，表示单用户模式，且无需 inittab 文件，因此也不在 inittab 中出现，实际上，进入单用户模式时，init 直接在控制台（/dev/console）上运行/sbin/sulogin。

在一般的系统实现中，都使用了 2、3、4、5 几个级别，在 Redhat 系统中，2 表示无 NFS 支持的多用户模式，3 表示完全多用户模式（也是最常用的级别），4 保留给用户自定义，5 表示 XDM 图形登录方式。7－ 9 级别也是可以使用的，传统的 Unix 系统没有定义这几个级别。runlevel 可以是并列的多个值，以匹配多个运行级别，对大多数 action 来说，仅当 runlevel 与当前运行级别匹配成功才会执行。

initdefault 是一个特殊的 action 值，用于标识缺省的启动级别；当 init 由核心激活 以后，它将读取 inittab 中的 initdefault 项，取得其中的 runlevel，并作为当前的运行级别。如果没有 inittab 文件，或者其中没有 initdefault 项，init 将在控制台上请求输入 runlevel。

sysinit、 boot、bootwait 等 action 将在系统启动时无条件运行，而忽略其中的 runlevel，其余的 action

（不含 initdefault）都与某个 runlevel 相关。各个 action 的定义在 inittab 的 man 手册中有详细的描述。

在 Redhat 系统中，一般情况下 inittab 都会有如下几项：

```
id:3:initdefault:
#表示当前缺省运行级别为 3--完全多任务模式；
si::sysinit:/etc/rc.d/rc.sysinit
#启动时自动执行/etc/rc.d/rc.sysinit 脚本
l3:3:wait:/etc/rc.d/rc 3
#当运行级别为 3 时，以 3 为参数运行/etc/rc.d/rc 脚本，init 将等待其返回
0:12345:respawn:/sbin/mingetty tty0
#在 1－5 各个级别上以 tty0 为参数执行/sbin/mingetty 程序，打开 tty0 终端用于
#用户登录，如果进程退出则再次运行 mingetty 程序
x:5:respawn:/usr/bin/X11/xdm -nodaemon
#在 5 级别上运行 xdm 程序，提供 xdm 图形方式登录界面，并在退出时重新执行.
```

六.rc 启动脚本

上一节已经提到 init 进程将启动运行 rc 脚本，这一节将介绍 rc 脚本具体的工作。

一般情况下，rc 启动脚本都位于/etc/rc.d 目录下，rc.sysinit 中最常见的动作就是激活交换分区，检查磁盘，加载硬件模块，这些动作无论哪个运行级别都是需要优先执行的。仅当 rc.sysinit 执行完以后 init 才会执行其他的 boot 或 bootwait 动作。

如果没有其他 boot、bootwait 动作，在运行级别 3 下，/etc/rc.d/rc 将会得到执行，命令行参数为 3，即执行 /etc/rc.d/rc3.d/目录下的所有文件。rc3.d 下的文件都是指向/etc/rc.d/init.d/目录下各个 Shell 脚本的符号连接，而这些脚本一般能接受 start、stop、restart、status 等参数。rc 脚本以 start 参数启动所有以 S 开头的脚本，在此之前，如果相应的脚本也存在 K 打头的链接，而且已经处于运行态了（以/var/lock/subsys/下的文件作为标志），则将首先启动 K 开头的脚本，以 stop 作为参数停止这些已经启动了的服务，然后再重新运行。显然，这样做的直接目的就是当 init 改变运行级别时，所有相关的服务都将重启，即使是同一个级别。

rc 程序执行完毕后，系统环境已经设置好了，下面就该用户登录系统了。

七.getty 和 login

在 rc 返回后，init 将得到控制，并启动 mingetty（见第五节）。mingetty 是 getty 的简化，不能处理串口操作。getty 的功能一般包括：

打开终端线，并设置模式
输出登录界面及提示，接受用户名的输入
以该用户名作为 login 的参数，加载 login 程序
缺省的登录提示记录在/etc/issue 文件中，但每次启动，一般都会由 rc.local 脚本根据系统环境重新生成。

注：用于远程登录的提示信息位于/etc/issue.net 中。

login 程序在 getty 的同一个进程空间中运行，接受 getty 传来的用户名参数作为登录的用户名。

如果用户名不是 root，且存在/etc/nologin 文件，login 将输出 nologin 文件的内容，然后退出。这通常用来系统维护时防止非 root 用户登录。

只有/etc/securetty 中登记了的终端才允许 root 用户登录，如果不存在这个文件，则 root 可以在任何终端上登录。/etc/usertty 文件用于对用户作出附加访问限制，如果不存在这个文件，则没有其他限制。

当用户登录通过了这些检查后，login 将搜索/etc/passwd 文件（必要时搜索 /etc/shadow 文件）用于匹配密码、设置主目录和加载 shell。如果没有指定主目录，将默认为根目录；如果没有指定 shell，将默认为/bin/sh。在将控制转交给 shell 以前， getty 将输出/var/log/lastlog 中记录的上次登录系统的信息，然后检查用户是否有新邮件（/usr/spool/mail/ {username}）。在设置好 shell 的 uid、gid，以及 TERM，PATH 等环境变量以后，进程加载 shell，login 的任务也就完成了。

八.bash

运行级别 3 下的用户 login 以后，将启动一个用户指定的 shell，以下以/bin/bash 为例继续我们的启动过程。

bash 是 Bourne Shell 的 GNU 扩展，除了继承了 sh 的所有特点以外，还增加了很多特性和功能。由 login 启动的 bash 是作为一个登录 shell 启动的，它继承了 getty 设置的 TERM、PATH 等环境变量，其中 PATH 对于普通用户为"/bin:/usr/bin:/usr/local/bin"，对于 root 为"/sbin:/bin:/usr/sbin:/usr/bin"。作为登录 shell，它将首先寻找/etc/profile 脚本文件，并执行它；然后如果存在~/.bash_profile，则执行它，否则执行~/.bash_login，如果该文件也不存在，则执行~/.profile 文件。然后 bash 将作为一个交互式 shell 执行~/.bashrc 文件（如果存在的话），很多系统中，~/.bashrc 都将启动 /etc/bashrc 作为系统范围内的配置文件。

当显示出命令行提示符的时候，整个启动过程就结束了。此时的系统，运行着内核，运行着几个核心线程，运行着 init 进程，运行着一批由 rc 启动脚本激活的守护进程（如 inetd 等），运行着一个 bash 作为用户的命令解释器。

附：XDM 方式登录

如果缺省运行级别设为 5，则系统中不光有 1-6 个 getty 监听着文本终端，还有启动了一个 XDM 的图形登录窗口。登录过程和文本方式差不多，也需要提供用户名和口令，XDM 的配置文件缺省为/usr/X11R6/lib/X11/xdm/xdm-config 文件，其中指定了 /usr/X11R6/lib/X11/xdm/xsession 作为 XDM 的会话描述脚本。登录成功后，XDM 将执行这个脚本以运行一个会话管理器，比如 gnome-session 等。

除了 XDM 以外，不同的窗口管理系统（如 KDE 和 GNOME）都提供了一个 XDM 的替代品，如 gdm 和 kdm，这些程序的功能和 XDM 都差不多。

附录 V fcntl 函数

fcntl 函数可以改变已打开的文件的性质。

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* int arg */ );
```

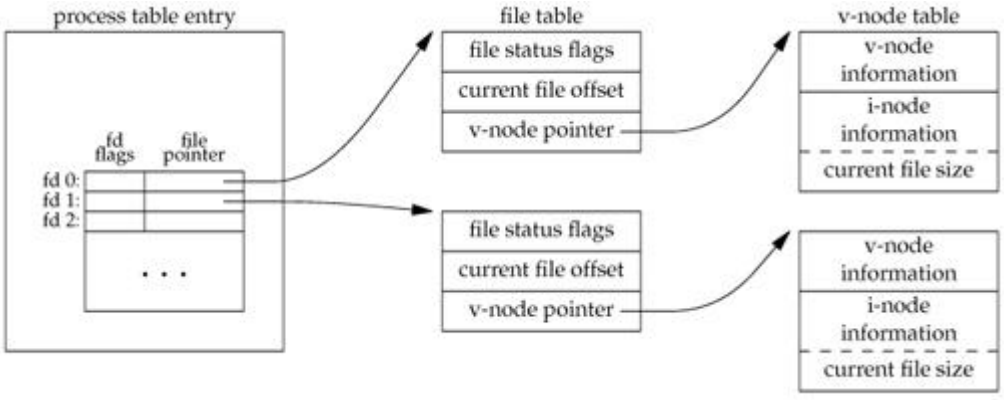
返回值：若成功则依赖于*cmd*（见下），若出错则返回−1

在本节的各实例中，第三个参数总是一个整数，与上面所示函数原型中的注释部分相对应。但是在 14.3 节说明记录锁时，第三个参数则是指向一个结构的指针。

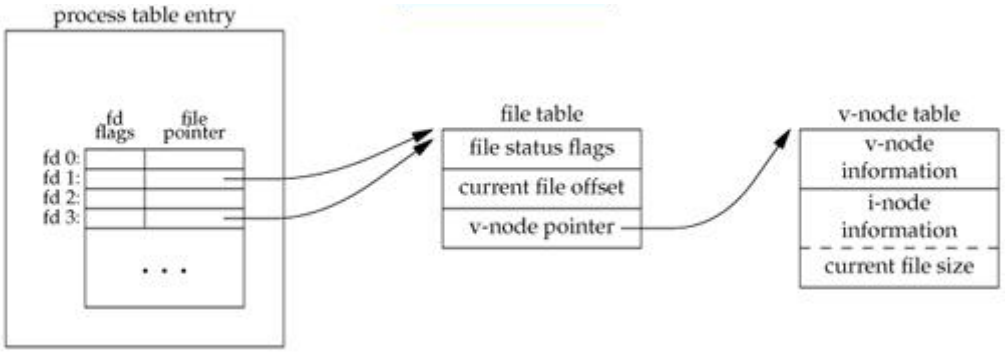
fcntl 函数有 5 种功能：

(1) 复制一个现有的描述符（cmd＝F_DUPFD）。
(2) 获得/设置文件描述符标记（cmd = F_GETFD 或 F_SETFD）。
(3) 获得/设置文件状态标志（cmd = F_GETFL 或 F_SETFL）。
(4) 获得/设置异步 I/O 所有权（cmd = F_GETOWN 或 F_SETOWN）。
(5) 获得/设置记录锁（cmd = F_GETLK、F_SETLK 或 F_SETLKW）。

我们先说明这 10 种 cmd 值中的前 7 种（14.3 节说明后 3 种，它们都与记录锁有关）。我们将涉及与进程表项中各文件描述符相关联的文件描述符标志，以及每个文件表项中的文件状态标志（见图 3-1）。



F_DUPFD：复制文件描述符 filedes。新文件描述符作为函数值返回。它是尚未打开的各描述符中大于或等于第三个参数值（取为整型值）中各值的最小值。新描述符与 filedes 共享同一文件表项（见图 3-3）。但是，新描述符有它自己的一套文件描述符标志，其 FD_CLOEXEC 文件描述符标志被清除（这表示该描述符在通过一个 exec 时仍保持有效，我们将在第 8 章对此进行讨论）。



F_GETFD：对应于 filedes 的文件描述符标志作为函数值返回。当前只定义了一个文件描述符标志 FD_CLOEXEC。

F_SETFD：对于 filedes 设置文件描述符标志。新标志值按第三个参数（取为整型值）设置。

应当了解很多现有的涉及文件描述符标志的程序并不使用常量 FD_CLOEXEC，而是将此标志设置为 0（系统默认，在 exec 时不关闭）或 1（在 exec 时关闭）。

F_GETFL：对应于 filedes 的文件状态标志作为函数值返回。在说明 open 函数时，已说明了文件状态标志。它们列于表 3-3 中。

表3-3 fcntl的文件状态标志

文件状态标志	说　明
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	为读、写打开
O_APPEND	每次写时追加
O_NONBLOCK	非阻塞模式
O_SYNC	等待写完成（数据和属性）
O_DSYNC	等待写完成（仅数据）
O_RSYNC	同步读、写
O_FSYNC	等待写完成（仅FreeBSD 和 Mac OS X）
O_ASYNC	异步I/O（仅FreeBSD和 Mac OS X）

不幸的是，三个访问方式标志（O_RDONLY、O_WRONLY 以及 O_RDWR）并不各占 1 位（正如前述，这三种标志的值分别是 0、1 和 2，由于历史原因。这三种值互斥—一个文件只能有这三种值之一）。因此首先必须用屏蔽字 O_ACCMODE 取得访问模式位，然后将结果与这三种值中的任一种作比较。

F_SETFL 将文件状态标志设置为第三个参数的值（取为整型值）。可以更改的几个标志是：O_APPEND、O_NONBLOCK、O_SYNC、O_DSYNC、O_RSYNC、 O_FSYNC 和 O_ASYNC。

F_GETOWN　　取当前接收 SIGIO 和 SIGURG 信号的进程 ID 或进程组 ID。14.6.2 节将论述这两种异步 I/O 信号。

F_SETOWN　　设置接收 SIGIO 和 SIGURG 信号的进程 ID 或进程组 ID。正的 arg 指定一个进程 ID，负的 arg 表示等于 arg 绝对值的一个进程组 ID。

fcntl 的返回值与命令有关。如果出错，所有命令都返回-1，如果成功则返回某个其他值。下列四个命令有特定返回值：F_DUPFD、F_GETFD、F_GETFL 以及 F_GETOWN。第一个返回新的文件描述符，接下来的两个返回相应标志，最后一个返回一个正的进程 ID 或负的进程组 ID。

实例

程序清单 3-4 中的程序的第一个参数指定文件描述符，并对于该描述符打印其所选择的文件标志说明。

程序清单3-4　对于指定的描述符打印文件标志

```
#include "apue.h"
#include <fcntl.h>

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

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
#if defined(O_SYNC)
    if (val & O_SYNC)
        printf(", synchronous writes");
#endif
#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

注意，我们使用了功能测试宏_POSIX_C_SOURCE，并且条件编译了 POSIX.1 中没有定义的文件访问标志。下面显示了从 bash（Bourne-again shell）调用该程序时的几种情况。当使用不同 shell 时，结果会发生变化。

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append


$ ./a.out 5 5<>temp.foo
read write
```

子句 5<>temp.foo 表示在文件描述符 5 上打开文件 temp.foo 以供读和写。

实例

在修改文件描述符标志或文件状态标志时必须谨慎，先要取得现有的标志值，然后根据需要修改它，最后设置新标志值。不能只是执行 F_SETFD 或 F_SETFL 命令，这样会关闭以前设置的标志位。

程序清单 3-5 显示了一个对一个文件描述符设置一个或多个文件状态标志的函数。

程序清单3-5 对一个文件描述符打开一个或多个文件状态标志

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

如果将中间的一条语句改为：

```
val &= ~flags;      /* turn flags off */
```

就构成另一个函数，我们称其为 clr_fl，并将在后面某个例子中用到它。此语句使当前文件状态标志值 val 与 flags 的补码进行逻辑“与”运算。

如果在程序清单 3-5 的开始处，加上下面一行以调用 set_fl，则打开了同步写标志。

```
set_fl(STDOUT_FILENO, O_SYNC);
```

这就使每次 write 都要等待，直至数据已写到磁盘上再返回。在 UNIX 系统中，通常 write 只是将数据排入队列，而实际的写磁盘操作则可能在以后的某个时刻进行。数据库系统很可能需要使用 O_SYNC，这样一来，当它从 write 返回时就知道数据已确实写到了磁盘上，以免在系统崩溃时产生数据丢失。

程序运行时，设置 O_SYNC 标志会增加时钟时间。为了测试这一点，运行程序清单 3-3，它从一个磁盘文件中将 98.5 MB 字节的数据复制到另一个文件。然后，在此程序中设置 O_SYNC 标志，使其完成上述同样的工作，以便将两者的结果进行比较。在使用 ext2 文件系统的 Linux 系统上执行上述操作，得到的结果见表 3-4。

```
#include "apue.h"

#define BUFFSIZE 4096

int
main(void)
{
    int    n;
    char   buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

表 3-4 中的 6 行都是在 BUFFSIZE 为 4096 的情况下测量得到的。表 3-2 中的结果所测量的情况是读一个磁盘文件，然后写到/dev/null，所以没有磁盘输出。表 3-4 中的第 2 行对应于读一个磁盘文件，然后写到另一个磁盘文件中。这就是为什么表 3-4 中第 1、2 行有差别的原因。在写磁盘文件时，系统时间增加了，其原因是内核需要从进程中复制数据，并将数据排入队列以便由磁盘驱动器将其写到磁盘上。当写至磁盘文件时，我们期望时钟时间也会增加，但在本测试中，它并未显著增加，这表明写操作将数据写到了系统高速缓存中，我们并没有测量将数据写到磁盘上的开销。

表3-4 用各种同步机制在Linux ext2环境中取得的计时结果

操　作	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）
取自表3-2中BUFFSIZE＝4096的读时间	0.03	0.16	6.86
正常写到磁盘文件	0.02	0.30	6.87
设置O_SYNC后写到磁盘文件	0.03	0.30	6.83
写到磁盘后接着调用fdatasync	0.03	0.42	18.28
写到磁盘后接着调用fsync	0.03	0.37	17.95
在设置O_SYNC的条件下写到磁盘，接着调用fsync	0.05	0.44	17.95

当支持同步写时，系统时间和时钟时间应当会显著增加。从第 3 行可见，同步写所用的时间与延迟写所用的时间几乎相同。这意味着 Linux ext2 文件系统并未真正实现 O_SYNC 标志功能。第 6 行的时间值支持了我们的这种怀疑，其中显示，实施同步写，然后跟随 fsync 调用，这一操作序列所用的时间与写文件（未设置同步标志），然后接着执行 fsync 调用这一序列所用的时间（第 5 行）几乎相同。在同步写一个文件后，我们期望 fsync 调用不会产生影响。

表 3-5 显示了在 Mac OS X 10.3 上运行同样的测试所得到的计时结果。注意该计时结果与我们的期望相符：同步写较延迟写所消耗的时间增加了很多，而且在同步写后再调用函数 fsync 并不会产生测量上的显著差别。还要引起注意的是，在延迟写后增加一个 fsync 函数调用也不产生可测量的差别。其原因很可能是，当写新数据到某个文件中时，操作系统将以前写的数据冲洗到了磁盘上，所以在调用函数 fsync 时几乎就没有什么工作要做了。

表3-5 用各种同步机制在Mac OS X环境中取得的计时结果

操 作	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）
写至/dev/null	0.06	0.79	4.33
正常写到磁盘文件	0.05	3.56	14.40
设置O_FSYNC后写到磁盘文件	0.13	9.53	22.48
写到磁盘后接着调用fsync	0.11	3.31	14.12
在设置O_FSYNC的条件下写到磁盘，接着调用fsync	0.17	9.14	22.12

比较 fsync 和 fdatasync 与 O_SYNC 标志，fsync 和 fdatasync 在我们需要时更新文件内容，O_SYNC 标志则在我们每次写至文件时更新文件内容。

在本例中，我们看到了 fcntl 的必要性。我们的程序在一个描述符（标准输出）上进行操作，但是根本不知道由 shell 打开的相应文件的文件名。因为这是 shell 打开的，于是不能在打开时，按我们的要求设置 O_SYNC 标志。fcntl 则允许仅知道打开文件描述符时可以修改其性质。在说明非阻塞管道时（15.2 节），我们还将了解到，由于我们对管道所具有的知识只是其描述符，所以也需要使用 fcntl 的功能。

附录 VI chmod 入门的一些常识

1. 基本用法

使用权限：所有使用者

使用方式：chmod [-cfvR] [--help] [--version] mode file...

说明：Linux/Unix 的档案存取权限分为三级：档案拥有者、群组、其他。利用 chmod 可以藉以控制档案如何被他人所存取。

mode：权限设定字串，格式如下：[ugoa...][[+-=][rwxX]...][,...]，其中 u 表示该档案的拥有者，g 表示与该档案的拥有者属于同一个群体(group)者，o 表示其他以外的人，a 表示这三者皆是。

+ 表示增加权限、- 表示取消权限、= 表示唯一设定权限。

r 表示可读取，w 表示可写入，x 表示可执行，X 表示只有当该档案是个子目录或者该档案已经被设定过为可执行。

-c：若该档案权限确实已经更改，才显示其更改动作

-f：若该档案权限无法被更改也不要显示错误讯息

-v：显示权限变更的详细资料

-R：对目前目录下的所有档案与子目录进行相同的权限变更(即以递回的方式逐个变更)

--help：显示辅助说明

--version：显示版本

范例 ：将档案 file1.txt 设为所有人皆可读取：

chmod ugo+r file1.txt

将档案 file1.txt 设为所有人皆可读取：

chmod a+r file1.txt

将档案 file1.txt 与 file2.txt 设为该档案拥有者，与其所属同一个群体者可写入，但其他以外的人则不可写入：

chmod ug+w,o-w file1.txt file2.txt

将 ex1.py 设定为只有该档案拥有者可以执行：

chmod u+x ex1.py

将目前目录下的所有档案与子目录皆设为任何人可读取：

chmod -R a+r *

此外 chmod 也可以用数字来表示权限如 chmod 777 file

语法为：chmod abc file

其中 a,b,c 各为一个数字，分别表示 User、Group、及 Other 的权限。

r=4，w=2，x=1

若要 rwx 属性则 4+2+1=7；

若要 rw-属性则 4+2=6；
若要 r-x 属性则 4+1=7。

范例：
chmod a=rwx file 和 chmod 777 file 效果相同

chmod ug=rwx,o=x file 和 chmod 771 file 效果相同

若用 chmod 4755 filename 可使此程式具有 root 的权限（见下文）

2. 档案的特殊权限：SUID/SGID/Sticky Bit

1．Set UID：当文件系统的“所有者权限组合”的可执行位(x)被 s(即 rws------)取代时，构成特殊权限规定 Set UID，简称 SUID。仅对系统中的二进制可执行文件设置有效，而且不可对 Shell Script 施加设置。
2．Set GID：当所有者所在的用户组(group)的权限组合中可执行位(x)被 s 所取代时(例如---rws---)，便构成 Set GID 的权限设置。SGID 可以针对二进制文件或目录进行设置。
3．Sticky Bit：当文件系统“其他(others)”的权限组合中可执行位(x)被 t 所取代时(例如------rwt)，便构成 Sticky Bit 的权限设置。它只对目录有效。

SUID 和 SGID，主要作用是用于当非某个文件的所有者(或组)执行(或操作目录)文件时，可以暂时获得该文件所有者的权限。

SBIT 的作用在于访问控制，当它对某个目录设置此属性后，该目录下的所有文件，即使其它人有 w 属性，都不得对其更名、移动、删除。

设置方法：
如果你已经掌握了用(八进制)数字来表示权限的规则，再结合 chmod 命令进行设置就很简单了。以下是 SUID/SGID/Sticky Bit 约定对应的八进制数值：
SUID ＝ 4
SGID ＝ 2
SBIT ＝ 1

设置时我们把表示特殊权限的数字放在其他三位数字权限的前面

例如：
SUID 的代表数字是 4，比如 4755 的结果是-rwsr-xr-x
SGID 的代表数字是 2，比如 6755 的结果是-rwsr-sr-x
SBIT 的代表数字是 1，比如 7755 的结果是-rwsr-sr-t
（当然 7755 这个 chmod 设置没多大意义，这里只是演示一下）

需要说明的是，四位数才是标准写法，我们通常写的 3 位是后三位，例如 755，系统会自己把你的 755 作为 0755 处理的

补充 1 Linux 下的多进程编程初步

摘要：多线程程序设计的概念早在六十年代就被提出，但直到八十年代中期，Unix 系统中才引入多线程机制，如今，由于自身的许多优点，多线程编程已经得到了广泛的应用。本文我们将介绍在 Linux 下编写多进程和多线程程序的一些初步知识。

1 引言

对于没有接触过 Unix/Linux 操作系统的人来说，fork 是最难理解的概念之一：它执行一次却返回两个值。fork 函数是 Unix 系统最杰出的成就之一，它是七十年代 UNIX 早期的开发者经过长期在理论和实践上的艰苦探索后取得的成果，一方面，它使操作系统在进程管理上付出了最小的代价，另一方面，又为程序员提供了一个简洁明了的多进程方法。与 DOS 和早期的 Windows 不同，Unix/Linux 系统是真正实现多任务操作的系统，可以说，不使用多进程编程，就不能算是真正的 Linux 环境下编程。

多线程程序设计的概念早在六十年代就被提出，但直到八十年代中期，Unix 系统中才引入多线程机制，如今，由于自身的许多优点，多线程编程已经得到了广泛的应用。

下面，我们将介绍在 Linux 下编写多进程和多线程程序的一些初步知识。

2 多进程编程

什么是一个进程？进程这个概念是针对系统而不是针对用户的，对用户来说，他面对的概念是程序。当用户敲入命令执行一个程序的时候，对系统而言，它将启动一个进程。但和程序不同的是，在这个进程中，系统可能需要再启动一个或多个进程来完成独立的多个任务。多进程编程的主要内容包括进程控制和进程间通信，在了解这些之前，我们先要简单知道进程的结构。

2.1 Linux 下进程的结构

Linux 下一个进程在内存里有三部分的数据，就是"代码段"、"堆栈段"和"数据段"。其实学过汇编语言的人一定知道，一般的 CPU 都有上述三种段寄存器，以方便操作系统的运行。这三个部分也是构成一个完整的执行序列的必要的部分。

"代码段"，顾名思义，就是存放了程序代码的数据，假如机器中有数个进程运行相同的一个程序，那么它们就可以使用相同的代码段。"堆栈段"存放的就是子程序的返回地址、子程序的参数以及程序的局部变量。而数据段则存放程序的全局变量，常数以及动态数据分配的数据空间（比如用 malloc 之类的函数取得的空间）。这其中有许多细节问题，这里限于篇幅就不多介绍了。系统如果同时运行数个相同的程序，它们之间就不能使用同一个堆栈段和数据段。

2.2 Linux 下的进程控制

在传统的 Unix 环境下，有两个基本的操作用于创建和修改进程：函数 fork( )用来创建一个新的进程，该进程几乎是当前进程的一个完全拷贝；函数族 exec( )用来启动另外的进程以取代当前运行的进程。Linux 的进程控制和传统的 Unix 进程控制基本一致，只在一些细节的地方有些区别，例如在 Linux 系统中调用 vfork 和 fork 完全相同，而在有些版本的 Unix 系统中，vfork 调用有不同的功能。由于这些差别几乎不影响我们大多数的编程，在这里我们不予考虑。

2.2.1 fork( )

fork 在英文中是"分叉"的意思。为什么取这个名字呢？因为一个进程在运行中，如果使用了 fork，就

产生了另一个进程，于是进程就"分叉"了，所以这个名字取得很形象。下面就看看如何具体使用 fork，这段程序演示了使用 fork 的基本框架：

```
void main()
{
   int i;
   if ( fork() == 0 )
   {
      /*  子进程程序  */
      for ( i = 1; i <1000; i ++ )
            printf("This is child process\n");
   }
   else
   {
      /*  父进程程序*/
      for ( i = 1; i <1000; i ++ )
            printf("This is process process\n");
   }
}
```

程序运行后，你就能看到屏幕上交替出现子进程与父进程各打印出的一千条信息了。如果程序还在运行中，你用 ps 命令就能看到系统中有两个它在运行了。

那么调用这个 fork 函数时发生了什么呢？fork 函数启动一个新的进程，前面我们说过，这个进程几乎是当前进程的一个拷贝：子进程和父进程使用相同的代码段；子进程复制父进程的堆栈段和数据段。这样，父进程的所有数据都可以留给子进程，但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上数据却已经分开，相互之间不再有影响了，也就是说，它们之间不再共享任何数据了。它们再要交互信息时，只有通过进程间通信来实现，这将是我们下面的内容。既然它们如此相象，系统如何来区分它们呢？这是由函数的返回值来决定的。对于父进程，fork 函数返回了子程序的进程号，而对于子程序，fork 函数则返回零。在操作系统中，我们用 ps 函数就可以看到不同的进程号，对父进程而言，它的进程号是由比它更低层的系统调用赋予的，而对于子进程而言，它的进程号即是 fork 函数对父进程的返回值。在程序设计中，父进程和子进程都要调用函数 fork（）下面的代码，而我们就是利用 fork（）函数对父子进程的不同返回值用 if...else...语句来实现让父子进程完成不同的功能，正如我们上面举的例子一样。我们看到，上面例子执行时两条信息是交互无规则的打印出来的，这是父子进程独立执行的结果，虽然我们的代码似乎和串行的代码没有什么区别。

读者也许会问，如果一个大程序在运行中，它的数据段和堆栈都很大，一次 fork 就要复制一次，那么 fork 的系统开销不是很大吗？其实 UNIX 自有其解决的办法，大家知道，一般 CPU 都是以"页"为单位来分配内存空间的，每一个页都是实际物理内存的一个映像，象 INTEL 的 CPU，其一页在通常情况下是 4086 字节大小，而无论是数据段还是堆栈段都是由许多"页"构成的，fork 函数复制这两个段，只是"逻辑"上的，并非"物理"上的，也就是说，实际执行 fork 时，物理空间上两个进程的数据段和堆栈段都还是共享着的，当有一个进程写了某个数据时，这时两个进程之间的数据才有了区别，系统就将有区别的"页"从物理上也分开。系统在空间上的开销就可以达到最小。

下面演示一个足以"搞死"Linux 的小程序，其源代码非常简单：

```
void main()
{
    for( ; ; ) fork();
}
```

这个程序什么也不做，就是死循环地 fork，其结果是程序不断产生进程，而这些进程又不断产生新的进程，很快，系统的进程就满了，系统就被这么多不断产生的进程"撑死了"。当然只要系统管理员预先给每个用户设置可运行的最大进程数，这个恶意的程序就完成不了企图了。

2.2.2 exec( )函数族

下面我们来看看一个进程如何来启动另一个程序的执行。在 Linux 中要使用 exec 函数族。系统调用 execve（）对当前进程进行替换，替换者为一个指定的程序，其参数包括文件名（filename）、参数列表（argv）以及环境变量（envp）。exec 函数族当然不止一个，但它们大致相同，在 Linux 中，它们分别是：execl，execlp，execle，execv，execve 和 execvp，下面我只以 execlp 为例，其它函数究竟与 execlp 有何区别，请通过 man exec 命令来了解它们的具体情况。

一个进程一旦调用 exec 类函数，它本身就"死亡"了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。（不过 exec 类函数中有的还允许继承环境变量之类的信息）

那么如果我的程序想启动另一程序的执行但自己仍想继续运行的话，怎么办呢？那就是结合 fork 与 exec 的使用。下面一段代码显示如何启动运行其它程序：

```
char command[256];
void main()
{
    int rtn; /*子进程的返回数值*/
    while(1)
    {
      /* 从终端读取要执行的命令 */
      printf( ">" );
      fgets( command, 256, stdin );
      command[strlen(command)-1] = 0;
      if ( fork() == 0 )
      {
        /* 子进程执行此命令 */
        execlp( command, command );
        /* 如果 exec 函数返回，表明没有正常执行命令，打印错误信息*/
        perror( command );
        exit( errorno );
      }
```

```
        else
        {
          /* 父进程，  等待子进程结束，并打印子进程的返回值 */
          wait ( &rtn );
          printf( " child process return %d\n",. rtn );
        }
      }
}
```

此程序从终端读入命令并执行之，执行完成后，父进程继续等待从终端读入命令。熟悉 DOS 和 WINDOWS 系统调用的朋友一定知道 DOS/WINDOWS 也有 exec 类函数，其使用方法是类似的，但 DOS/WINDOWS 还有 spawn 类函数，因为 DOS 是单任务的系统，它只能将"父进程"驻留在机器内再执行"子进程"，这就是 spawn 类的函数。WIN32 已经是多任务的系统了，但还保留了 spawn 类函数，WIN32 中实现 spawn 函数的方法同前述 UNIX 中的方法差不多，开设子进程后父进程等待子进程结束后才继续运行。UNIX 在其一开始就是多任务的系统，所以从核心角度上讲不需要 spawn 类函数。

在这一节里，我们还要讲讲 system（）和 popen（）函数。system（）函数先调用 fork（），然后再调用 exec（）来执行用户的登录 shell，通过它来查找可执行文件的命令并分析参数，最后它要使用 wait（）函数族之一来等待子进程的结束。函数 popen（）和函数 system（）相似，不同的是它调用 pipe（）函数创建一个管道，通过它来完成程序的标准输入和标准输出。这两个函数是为那些不太勤快的程序员设计的，在效率和安全方面都有相当的缺陷，在可能的情况下，应该尽量避免。

2.3 Linux 下的进程间通信

详细的讲述进程间通信在这里绝对是不可能的事情，而且笔者很难有信心说自己对这一部分内容的认识达到了什么样的地步，所以在这一节的开头首先向大家推荐著名作者 Richard Stevens 的著名作品：《Advanced Programming in the UNIX Environment》，它的中文译本《UNIX 环境高级编程》已有机械工业出版社出版，原文精彩，译文同样地道，如果你的确对在 Linux 下编程有浓厚的兴趣，那么赶紧将这本书摆到你的书桌上或计算机旁边来。说这么多实在是难抑心中的景仰之情，言归正传，在这一节里，我们将介绍进程间通信最最初步和最最简单的一些知识和概念。

首先，进程间通信至少可以通过传送打开文件来实现，不同的进程通过一个或多个文件来传递信息，事实上，在很多应用系统里，都使用了这种方法。但一般说来，进程间通信（IPC：InterProcess Communication）不包括这种似乎比较低级的通信方法。Unix 系统中实现进程间通信的方法很多，而且不幸的是，极少方法能在所有的 Unix 系统中进行移植（唯一一种是半双工的管道，这也是最原始的一种通信方式）。而 Linux 作为一种新兴的操作系统，几乎支持所有的 Unix 下常用的进程间通信方法：管道、消息队列、共享内存、信号量、套接口等等。下面我们将逐一介绍。

2.3.1 管道

管道是进程间通信中最古老的方式，它包括无名管道和有名管道两种，前者用于父进程和子进程间的通信，后者用于运行于同一台机器上的任意两个进程间的通信。

无名管道由 pipe（）函数创建：

```
#include <unistd.h>
```

int pipe(int filedis[2])；

参数 filedis 返回两个文件描述符：filedes[0]为读而打开，filedes[1]为写而打开。filedes[1]的输出是filedes[0]的输入。下面的例子示范了如何在父进程和子进程间实现通信。

```
#define INPUT 0
#define OUTPUT 1

void main()
{
   int file_descriptors[2];
   /*定义子进程号 */
   pid_t pid;
   char buf[256];
   int returned_count;
   /*创建无名管道*/
   pipe(file_descriptors);
   /*创建子进程*/
   if((pid = fork()) == -1)
   {
      printf("Error in fork\n");
      exit(1);
   }
   /*执行子进程*/
   if(pid == 0)
   {
      printf("in the spawned (child) process...\n");
      /*子进程向父进程写数据，关闭管道的读端*/
      close(file_descriptors[INPUT]);
      write(file_descriptors[OUTPUT], "test data", strlen("test data"));
      exit(0);
   }
   else
   {
      /*执行父进程*/
      printf("in the spawning (parent) process...\n");
      /*父进程从管道读取子进程写的数据，关闭管道的写端*/
      close(file_descriptors[OUTPUT]);
      returned_count = read(file_descriptors[INPUT], buf, sizeof(buf));
      printf("%d bytes of data received from spawned process: %s\n", returned_count, buf);
   }
}
```

在 Linux 系统下，有名管道可由两种方式创建：命令行方式 mknod 系统调用和函数 mkfifo。下面的两

种途径都在当前目录下生成了一个名为 myfifo 的有名管道：

方式一：mkfifo("myfifo","rw")[①];
方式二：mknod myfifo p

生成了有名管道后，就可以使用一般的文件 I/O 函数如 open、close、read、write 等来对它进行操作。下面即是一个简单的例子，假设我们已经创建了一个名为 myfifo 的有名管道。

```
/* 进程一：读有名管道*/
#include <stdio.h>
#include <unistd.h>
void main()
{
   FILE * in_file;
   int count = 1;
   char buf[80];
   in_file = fopen("mypipe", "r");
   if (in_file == NULL)
   {
      printf("Error in fdopen.\n");
      exit(1);
   }
   while ((count = fread(buf, 1, 80, in_file)) > 0)
      printf("received from pipe: %s\n", buf);
   fclose(in_file);
}

/* 进程二：写有名管道*/
#include <stdio.h>
#include <unistd.h>
void main()
{
   FILE * out_file;
   int count = 1;
   char buf[80];
   out_file = fopen("mypipe", "w");
   if (out_file == NULL)
   {
      printf("Error opening pipe.");
      exit(1);
   }
   sprintf(buf,"this is test data for the named pipe example\n");
   fwrite(buf, 1, 80, out_file);
```

[①] 可以看成是在内存中创建了一个名为 myfifo 的共享文件。

```
    fclose(out_file);
}
```

2.3.2 消息队列

消息队列用于运行于同一台机器上的进程间通信，它和管道很相似，事实上，它是一种正逐渐被淘汰的通信方式，我们可以用流管道或者套接口的方式来取代它，所以，我们对此方式也不再解释，也建议读者忽略这种方式。

2.3.3 共享内存

共享内存是运行在同一台机器上的进程间通信最快的方式，因为数据不需要在不同的进程间复制。通常由一个进程创建一块共享内存区，其余进程对这块内存区进行读写。得到共享内存有两种方式：映射/dev/mem 设备和内存映像文件。前一种方式不给系统带来额外的开销，但在现实中并不常用，因为它控制存取的将是实际的物理内存，在 Linux 系统下，这只有通过限制 Linux 系统存取的内存才可以做到，这当然不太实际。常用的方式是通过 shmXXX 函数族来实现利用共享内存进行存储的。

首先要用的函数是 shmget，它获得一个共享存储标识符。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flag);
```

这个函数有点类似大家熟悉的 malloc 函数，系统按照请求分配 size 大小的内存用作共享内存。Linux 系统内核中每个 IPC 结构都有的一个非负整数的标识符，这样对一个消息队列发送消息时只要引用标识符就可以了。这个标识符是内核由 IPC 结构的关键字得到的，这个关键字，就是上面第一个函数的 key。数据类型 key_t 是在头文件 sys/types.h 中定义的，它是一个长整形的数据。在我们后面的章节中，还会碰到这个关键字。

当共享内存创建后，其余进程可以调用 shmat（）将其连接到自身的地址空间中。

```
void *shmat(int shmid, void *addr, int flag);
```

shmid 为 shmget 函数返回的共享存储标识符，addr 和 flag 参数决定了以什么方式来确定连接的地址，函数的返回值即是该进程数据段所连接的实际地址，进程可以对此进程进行读写操作。

使用共享存储来实现进程间通信的注意点是对数据存取的同步，必须确保当一个进程去读取数据时，它所想要的数据已经写好了。通常，信号量被要求来实现对共享存储数据存取的同步，另外，可以通过使用 shmctl 函数设置共享存储内存的某些标志位如 SHM_LOCK、SHM_UNLOCK 等来实现。

2.3.4 信号量

信号量又称为信号灯，它是用来协调不同进程间的数据对象的，而最主要的应用是前一节的共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：

（1） 测试控制该资源的信号量。
（2） 若此信号量的值为正，则允许进行使用该资源。进程将进号量减 1。
（3） 若此信号量为 0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于 0，进程被唤醒，转入步骤（1）。
（4） 当进程不再使用一个信号量控制的资源时，信号量值加 1。如果此时有进程正在睡眠等待此信号量，则唤醒此进程。

维护信号量状态的是 Linux 内核操作系统而不是用户进程。我们可以从头文件/usr/src/linux/include /linux /sem.h 中看到内核用来维护信号量状态的各个结构的定义。信号量是一个数据集合，用户可以单独使用这一集合的每个元素。要调用的第一个函数是 semget，用以获得一个信号量 ID。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

key 是前面讲过的 IPC 结构的关键字，它将来决定是创建新的信号量集合，还是引用一个现有的信号量集合。nsems 是该集合中的信号量数。如果是创建新集合（一般在服务器中），则必须指定 nsems；如果是引用一个现有的信号量集合（一般在客户机中）则将 nsems 指定为 0。

semctl 函数用来对信号量进行操作。

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

不同的操作是通过 cmd 参数来实现的，在头文件 sem.h 中定义了 7 种不同的操作，实际编程时可以参照使用。

semop 函数自动执行信号量集合上的操作数组。

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

semoparray 是一个指针，它指向一个信号量操作数组。nops 规定该数组中操作的数量。

下面，我们看一个具体的例子，它创建一个特定的 IPC 结构的关键字和一个信号量，建立此信号量的索引，修改索引指向的信号量的值，最后我们清除信号量。在下面的代码中，函数 ftok 生成我们上文所说的唯一的 IPC 关键字。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>
void main()
{
   key_t unique_key; /*  定义一个 IPC 关键字*/
```

```
    int id;
    struct sembuf lock_it;
    union semun options;
    int i;

    unique_key = ftok(".", 'a'); /*  生成关键字，字符'a'是一个随机种子*/
    /*  创建一个新的信号量集合*/
    id = semget(unique_key, 1, IPC_CREAT | IPC_EXCL | 0666);
    printf("semaphore id=%d\n", id);
    options.val = 1; /*设置变量值*/
    semctl(id, 0, SETVAL, options); /*设置索引 0 的信号量*/

    /*打印出信号量的值*/
    i = semctl(id, 0, GETVAL, 0);
    printf("value of semaphore at index 0 is %d\n", i);

    /*下面重新设置信号量*/
    lock_it.sem_num = 0; /*设置哪个信号量*/
    lock_it.sem_op = -1; /*定义操作*/
    lock_it.sem_flg = IPC_NOWAIT; /*操作方式*/
    if (semop(id, &lock_it, 1) == -1)
    {
       printf("can not lock semaphore.\n");
       exit(1);
    }

    i = semctl(id, 0, GETVAL, 0);
    printf("value of semaphore at index 0 is %d\n", i);

    /*清除信号量*/
    semctl(id, 0, IPC_RMID, 0);
}
```

2.3.5 套接口

套接口（socket）编程是实现 Linux 系统和其他大多数操作系统中进程间通信的主要方式之一。我们熟知的 WWW 服务、FTP 服务、TELNET 服务等都是基于套接口编程来实现的。除了在异地的计算机进程间以外，套接口同样适用于本地同一台计算机内部的进程间通信。关于套接口的经典教材同样是 Richard Stevens 编著的《Unix 网络编程：联网的 API 和套接字》，清华大学出版社出版了该书的影印版。它同样是 Linux 程序员的必备书籍之一。

关于这一部分的内容，可以参照本文作者的另一篇文章《设计自己的网络蚂蚁》，那里由常用的几个套接口函数的介绍和示例程序。这一部分或许是 Linux 进程间通信编程中最须关注和最吸引人的一部分，毕竟，Internet 正在我们身边以不可思议的速度发展着，如果一个程序员在设计编写他下一个程序的时候，根本没有考虑到网络，考虑到 Internet，那么，可以说，他的设计很难成功。

3 Linux 的进程和 Win32 的进程/线程比较

熟悉 WIN32 编程的人一定知道，WIN32 的进程管理方式与 Linux 上有着很大区别，在 UNIX 里，只有进程的概念，但在 WIN32 里却还有一个"线程"的概念，那么 Linux 和 WIN32 在这里究竟有着什么区别呢？

WIN32 里的进程/线程是继承自 OS/2 的。在 WIN32 里，"进程"是指一个程序，而"线程"是一个"进程"里的一个执行"线索"。从核心上讲，WIN32 的多进程与 Linux 并无多大的区别，在 WIN32 里的线程才相当于 Linux 的进程，是一个实际正在执行的代码。但是，WIN32 里同一个进程里各个线程之间是共享数据段的。这才是与 Linux 的进程最大的不同。

下面这段程序显示了 WIN32 下一个进程如何启动一个线程。

```
int g;
DWORD WINAPI ChildProcess( LPVOID lpParameter )
{
   int i;
   for ( i = 1; i <1000; i ++)
   {
      g ++;
      printf( "This is Child Thread: %d\n", g );
   }
   ExitThread( 0 );
};

void main()
{
   int threadID;
   int i;
   g = 0;
   CreateThread( NULL, 0, ChildProcess, NULL, 0, &threadID );
   for ( i = 1; i <1000; i ++)
   {
      g ++;
      printf( "This is Parent Thread: %d\n", g );
   }
}
```

在 WIN32 下，使用 CreateThread 函数创建线程，与 Linux 下创建进程不同，WIN32 线程不是从创建处开始运行的，而是由 CreateThread 指定一个函数，线程就从那个函数处开始运行。此程序同前面的 UNIX 程序一样，由两个线程各打印 1000 条信息。threadID 是子线程的线程号，另外，全局变量 g 是子线程与父线程共享的，这就是与 Linux 最大的不同之处。大家可以看出，WIN32 的进程/线程要比 Linux 复杂，在 Linux 要实现类似 WIN32 的线程并不难，只要 fork 以后，让子进程调用 ThreadProc 函数，并且为全局变量开设共享数据区就行了，但在 WIN32 下就无法实现类似 fork 的功能了。所以现在 WIN32 下的 C 语言编

译器所提供的库函数虽然已经能兼容大多数 Linux/UNIX 的库函数，但却仍无法实现 fork。

对于多任务系统，共享数据区是必要的，但也是一个容易引起混乱的问题，在 WIN32 下，一个程序员很容易忘记线程之间的数据是共享的这一情况，一个线程修改过一个变量后，另一个线程却又修改了它，结果引起程序出问题。但在 Linux 下，由于变量本来并不共享，而由程序员来显式地指定要共享的数据，使程序变得更清晰与安全。

至于 WIN32 的"进程"概念，其含义则是"应用程序"，也就是相当于 UNIX 下的 exec 了。

Linux 也有自己的多线程函数 pthread，它既不同于 Linux 的进程，也不同于 WIN32 下的进程，关于 pthread 的介绍和如何在 Linux 环境下编写多线程程序我们将在另一篇文章《Linux 下的多线程编程》中讲述。

4 鸣谢

本文部分内容参照 www.lisoleg.org 内的《Linux 下的多进程编程》，原作者俞磊。

补充 2 Linux 系统调用跟我学

1. 引言

本文是 Linux 系统调用系列文章的第一篇，对 Linux 系统调用的定义、基本原理、使用方法和注意事项大概作了一个介绍，以便读者对 Linux 系统调用建立一个大致的印象。

什么是系统调用？

Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。从某种角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。二者在使用方式上也有相似之处，在下面将会提到。

随 Linux 核心还提供了一些 C 语言函数库，这些库对系统调用进行了一些包装和扩展，因为这些库函数与系统调用的关系非常紧密，所以习惯上把这些函数也称为系统调用。

Linux 中共有多少个系统调用？

这个问题可不太好回答，就算让 Linus Torvaldz 本人也不见得一下子就能说清楚。

在 2.4.4 版内核中，狭义上的系统调用共有 221 个，你可以在<内核源码目录>/include/asm-i386/unistd.h 中找到它们的原本，也可以通过命令"man 2 syscalls"察看它们的目录（man pages 的版本一般比较老，可能有很多最新的调用都没有包含在内）。广义上的系统调用，也就是以库函数的形式实现的那些，它们的个数从来没有人统计过，这是一件吃力不讨好的活，新内核不断地在推出，每一个新内核中函数数目的变化根本就没有人在乎，至少连内核的修改者本人都不在乎，因为他们从来没有发布过一个此类的声明。

随本文一起有一份经过整理的列表，它不可能非常全面，但常见的系统调用基本都已经包含在内，那里面只有不多的一部分是你平时用得到的，本专栏将会有选择的对它们进行介绍。

为什么要用系统调用？

实际上，很多已经被我们习以为常的 C 语言标准函数，在 Linux 平台上的实现都是靠系统调用完成的，所以如果想对系统底层的原理作深入的了解，掌握各种系统调用是初步的要求。进一步，若想成为一名 Linux 下编程高手，也就是我们常说的 Hacker，其标志之一也是能对各种系统调用有透彻的了解。

即使除去上面的原因，在平常的编程中你也会发现，在很多情况下，系统调用是实现你的想法的简洁有效的途径，所以有可能的话应该尽量多掌握一些系统调用，这会对你的程序设计过程带来意想不到的帮助。

系统调用是怎么工作的？

一般的，进程是不能访问内核的。它不能访问内核所占内存空间也不能调用内核函数。CPU 硬件决定

了这些（这就是为什么它被称作"保护模式"）。系统调用是这些规则的一个例外。其原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。在 Intel CPU 中，这个由中断 0x80 实现。硬件知道一旦你跳到这个位置，你就不是在限制模式下运行的用户，而是作为操作系统的内核——所以你就可以为所欲为。

进程可以跳转到的内核位置叫做 sysem_call。这个过程检查系统调用号，这个号码告诉内核进程请求哪种服务。然后，它查看系统调用表(sys_call_table)找到所调用的内核函数入口地址。接着，就调用函数，等返回后，做一些系统检查，最后返回到进程（或到其他进程，如果这个进程时间用尽）。如果你希望读这段代码，它在<内核源码目录>/kernel/entry.S，Entry(system_call)的下一行。

如何使用系统调用？

先来看一个例子：

```
#include<linux/unistd.h> /*定义宏_syscall1*/
#include<time.h> /*定义类型 time_t*/
_syscall1(time_t, time, time_t *, tloc) /*宏，展开后得到 time()函数的原型*/
main()
{
        time_t the_time;
        the_time=time((time_t *)0); /*调用 time 系统调用*/
        printf("The time is %ld\n", the_time);
}
```

系统调用 time 返回从格林尼治时间 1970 年 1 月 1 日 0:00 开始到现在的秒数。

这是最标准的系统调用的形式，宏_syscall1()展开来得到一个函数原型，稍后我会作详细解释。但事实上，如果把程序改成下面的样子，程序也可以运行得同样的结果。

```
#include<time.h>
main()
{
        time_t the_time;
        the_time=time((time_t *)0); /*调用 time 系统调用*/
        printf("The time is %ld\n", the_time);
}
```

这是因为在 time.h 中实际上已经用库函数的形式实现了 time 这个系统调用，替我们省掉了调用 _syscall1 宏展开得到函数原型这一步。

大多数系统调用都在各种 C 语言函数库中有所实现，所以在一般情况下，我们都可以像调用普通的库函数那样调用系统调用，只在极个别的情况下，我们才有机会用到_syscall*()这几个宏。

_syscall*()是什么？

在 unistd.h 里定义了 7 个宏，分别是

```
_syscall0(type, name)
_syscall1(type, name, type1, arg1)
_syscall2(type, name, type1, arg1, type2, arg2)
_syscall3(type, name, type1, arg1, type2, arg2, type3, arg3)
_syscall4(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)
_syscall5(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5)
_syscall6(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5, type6, arg6)
```

它们看起来似乎不太像宏，但其实质和

```
#define MAXSIZE 100
```

里面的 MAXSIZE 没有任何区别。

它们的作用是形成相应的系统调用函数原型，供我们在程序中调用。我们很容易就能发现规律，_syscall 后面的数字和 typeN, argN 的数目一样多。事实上，_syscall 后面跟的数字指明了展开后形成函数的参数的个数，让我们看一个实例，就是刚刚用过的 time 系统调用：

```
_syscall1(time_t, time, time_t *, tloc)
```

展开后的情形是这样：

```
time_t time(time_t *tloc)
{
    long __res;
    __asm__ volatile("int $0x80" : "=a" (__res) : "0" (13), "b" ((long)(tloc)));
    do {
        if ((unsigned long)(__res) >= (unsigned long)(-125)) {
            errno = -(__res);
            __res    = -1;
        }
        return (time_t) (__res);
    } while (0) ;
}
```

可以看出，_syscall1(time_t, time, time_t *, tloc)展开成一个名为 time 的函数，原参数 time_t 就是函数的返回类型，原参数 time_t *和 tloc 分别构成新函数的参数。事实上，程序中用到的 time 函数的原型就是它。

errno 是什么？

为防止和正常的返回值混淆，系统调用并不直接返回错误码，而是将错误码放入一个名为 errno 的全

局变量中。如果一个系统调用失败，你可以读出 errno 的值来确定问题所在。

errno 不同数值所代表的错误消息定义在 errno.h 中，你也可以通过命令"man 3 errno"来察看它们。

需要注意的是，errno 的值只在函数发生错误时设置，如果函数不发生错误，errno 的值就无定义，并不会被置为 0。另外，在处理 errno 前最好先把它的值存入另一个变量，因为在错误处理过程中，即使像 printf()这样的函数出错时也会改变 errno 的值。

系统调用兼容性好吗？

很遗憾，答案是——不好。但这决不意味着你的程序会三天两头的导致系统崩溃，因为系统调用是 Linux 的内核提供的，所以它们工作起来非常稳定，对于此点无需丝毫怀疑，在绝大多数的情况下，系统调用要比你自己编写的代码可靠而高效的多。

但是，在 Linux 的各版本内核之间，系统调用的兼容性表现得并不像想象那么好，这是由 Linux 本身的性质决定的。Linux 是一群程序设计高手利用业余时间开发出来的，他们中间的大部分人没有把 Linux 当成一个严肃的商业软件，（现在的情况有些不同了，随着 Linux 商业公司和以 Linux 为生的人的增长，不少人的脑筋发生了变化。）结果就是，如果新的方案在效率和兼容性上发生了矛盾，他们往往舍弃兼容性而追求效率，就这样，如果他们认为某个系统调用实现的比较糟糕，他们就会毫不犹豫的作出修改，有些时候甚至连接口也一起改掉了，更可怕的是，很多时候，他们对自己的修改连个招呼也不打，在任何文档里都找不到关于修改的提示。这样，每当新内核推出的时候，很可能都会悄悄的更新一些系统调用，用户编制的应用程序也会跟着出错。

说到这里，你是不是感觉前途一片昏暗呢？呵呵，不用太紧张，如前面所说，随着越来越多的人把 Linux 当成自己的饭碗，不兼容的情况也越来越罕见。从 2.2 版本以后的 Linux 内核已经非常稳定了，不过尽管如此，你还是有必要在每个新内核推出之后，对自己的应用程序进行兼容性测试，以防止意外的发生。

该如何学习使用 Linux 系统调用呢？

你可以用"man 2 系统调用名称"的命令来查看各条系统调用的介绍，但这首先要求你要有不错的英语基础，其次还得有一定的程序设计和系统编程的功底，man pages 不会涉及太多的应用细节，因为它只是一个手册而非教程。如果 man pages 所提供的东西不能使你感到非常满意，那就跟我来吧，本专栏将向你展示 Linux 系统调用编程的无穷魅力。

对读者的两点小小的要求：

1)读者必须有一定的 C 语言编程经验；

2)读者必须有一定的 Linux 使用经验。如果你能完全看懂本文从开头到这里所讲的东西，你就合格了。收拾好行囊，准备出发吧！

2．进程管理

本文介绍了 Linux 下的进程概念，并着重讲解了与 Linux 进程管理相关的 4 个重要系统调用 getpid, fork,

exit 和_exit，辅助一些例程说明了它们的特点和使用方法。

关于进程的一些必要知识

先看一下进程在大学课本里的标准定义：“进程是可并发执行的程序在一个数据集合上的运行过程。”这个定义非常严谨，而且难懂，如果你没有一下子理解这句话，就不妨看看笔者自己的并不严谨的解释。我们大家都知道，硬盘上的一个可执行文件经常被称作程序，在 Linux 系统中，当一个程序开始执行后，在开始执行到执行完毕退出这段时间里，它在内存中的部分就被称作一个进程。

当然，这个解释并不完善，但好处是容易理解，在以下的文章中，我们将会对进程作一些更全面的认识。

Linux 进程简介

Linux 是一个多任务的操作系统，也就是说，在同一个时间内，可以有多个进程同时执行。如果读者对计算机硬件体系有一定了解的话，会知道我们大家常用的单 CPU 计算机实际上在一个时间片断内只能执行一条指令，那么 Linux 是如何实现多进程同时执行的呢？原来 Linux 使用了一种称为“进程调度（process scheduling）”的手段，首先，为每个进程指派一定的运行时间，这个时间通常很短，短到以毫秒为单位，然后依照某种规则，从众多进程中挑选一个投入运行，其他的进程暂时等待，当正在运行的那个进程时间耗尽，或执行完毕退出，或因某种原因暂停，Linux 就会重新进行调度，挑选下一个进程投入运行。因为每个进程占用的时间片都很短，在我们使用者的角度来看，就好像多个进程同时运行一样了。

在 Linux 中，每个进程在创建时都会被分配一个数据结构，称为进程控制块（Process Control Block，简称 PCB）。PCB 中包含了很多重要的信息，供系统调度和进程本身执行使用，其中最重要的莫过于进程 ID（process ID）了，进程 ID 也被称作进程标识符，是一个非负的整数，在 Linux 操作系统中唯一地标志一个进程，在我们最常使用的 I386 架构（即 PC 使用的架构）上，一个非负的整数的变化范围是 0-32767，这也是我们所有可能取到的进程 ID。其实从进程 ID 的名字就可以看出，它就是进程的身份证号码，每个人的身份证号码都不会相同，每个进程的进程 ID 也不会相同。

一个或多个进程可以合起来构成一个进程组（process group），一个或多个进程组可以合起来构成一个会话（session）。这样我们就有了对进程进行批量操作的能力，比如通过向某个进程组发送信号来实现向该组中的每个进程发送信号。

最后，让我们通过 ps 命令亲眼看一看自己的系统中目前有多少进程在运行：

$ps -aux（以下是在我的计算机上的运行结果，你的结果很可能与这不同。）

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.4   1412   520 ?        S    May15   0:04 init [3]
root         2  0.0  0.0      0     0 ?        SW   May15   0:00 [keventd]
root         3  0.0  0.0      0     0 ?        SW   May15   0:00 [kapm-idled]
root         4  0.0  0.0      0     0 ?        SWN  May15   0:00 [ksoftirqd_CPU0]
root         5  0.0  0.0      0     0 ?        SW   May15   0:00 [kswapd]
root         6  0.0  0.0      0     0 ?        SW   May15   0:00 [kreclaimd]
root         7  0.0  0.0      0     0 ?        SW   May15   0:00 [bdflush]
root         8  0.0  0.0      0     0 ?        SW   May15   0:00 [kupdated]
root         9  0.0  0.0      0     0 ?        SW&lt;   May15   0:00 [mdrecoveryd]
root        13  0.0  0.0      0     0 ?        SW   May15   0:00 [kjournald]
root       132  0.0  0.0      0     0 ?        SW   May15   0:00 [kjournald]
root       673  0.0  0.4   1472   592 ?        S    May15   0:00 syslogd -m 0
root       678  0.0  0.8   2084  1116 ?        S    May15   0:00 klogd -2
rpc        698  0.0  0.4   1552   588 ?        S    May15   0:00 portmap
rpcuser    726  0.0  0.6   1596   764 ?        S    May15   0:00 rpc.statd
root       839  0.0  0.4   1396   524 ?        S    May15   0:00 /usr/sbin/apmd -p
root       908  0.0  0.7   2264  1000 ?        S    May15   0:00 xinetd -stayalive
root       948  0.0  1.5   5296  1984 ?        S    May15   0:00 sendmail: accepti
root       967  0.0  0.3   1440   484 ?        S    May15   0:00 gpm -t ps/2 -m /d
wnn        987  0.0  2.7   4732  3440 ?        S    May15   0:00 /usr/bin/cserver
root      1005  0.0  0.5   1584   660 ?        S    May15   0:00 crond
wnn       1025  0.0  1.9   3720  2488 ?        S    May15   0:00 /usr/bin/tserver
xfs       1079  0.0  2.5   4592  3216 ?        S    May15   0:00 xfs -droppriv -da
daemon    1115  0.0  0.4   1444   568 ?        S    May15   0:00 /usr/sbin/atd
root      1130  0.0  0.3   1384   448 tty1     S    May15   0:00 /sbin/mingetty tt
root      1131  0.0  0.3   1384   448 tty2     S    May15   0:00 /sbin/mingetty tt
root      1132  0.0  0.3   1384   448 tty3     S    May15   0:00 /sbin/mingetty tt
root      1133  0.0  0.3   1384   448 tty4     S    May15   0:00 /sbin/mingetty tt
root      1134  0.0  0.3   1384   448 tty5     S    May15   0:00 /sbin/mingetty tt
root      1135  0.0  0.3   1384   448 tty6     S    May15   0:00 /sbin/mingetty tt
root      8769  0.0  0.6   1744   812 ?        S    00:08   0:00 in.telnetd: 192.1
root      8770  0.0  0.9   2336  1184 pts/0    S    00:08   0:00 login -- lei
lei       8771  0.1  0.9   2432  1264 pts/0    S    00:08   0:00 -bash
lei       8809  0.0  0.6   2764   808 pts/0    R    00:09   0:00 ps -aux
```

以上除标题外，每一行都代表一个进程。在各列中，PID 一列代表了各进程的进程 ID，COMMAND 一列代表了进程的名称或在 Shell 中调用的命令行，对其他列的具体含义，我就不再作解释，有兴趣的读者可以去参考相关书籍。

getpid

在 2.4.4 版内核中，getpid 是第 20 号系统调用，其在 Linux 函数库中的原型是：

```
#include <sys/types.h> /*  提供类型 pid_t 的定义  */
#include <unistd.h> /*  提供函数的定义  */
pid_t getpid(void);
```

getpid 的作用很简单，就是返回当前进程的进程 ID，请大家看以下的例子：

```
/* getpid_test.c */
#include <unistd.h>
```

```
main()
{
    printf("The current process ID is %d\n", getpid());
}
```

细心的读者可能注意到了，这个程序的定义里并没有包含头文件 sys/types.h，这是因为我们在程序中没有用到 pid_t 类型，pid_t 类型即为进程 ID 的类型。事实上，在 i386 架构上（就是我们一般 PC 计算机的架构），pid_t 类型是和 int 类型完全兼容的，我们可以用处理整形数的方法去处理 pid_t 类型的数据，比如，用"%d"把它打印出来。

编译并运行程序 getpid_test.c：

```
$gcc getpid_test.c -o getpid_test
$./getpid_test
The current process ID is 1980
```

（你自己的运行结果很可能与这个数字不一样，这是很正常的。）

再运行一遍：

```
$./getpid_test
The current process ID is 1981
```

正如我们所见，尽管是同一个应用程序，每一次运行的时候，所分配的进程标识符都不相同。

fork

在 2.4.4 版内核中，fork 是第 2 号系统调用，其在 Linux 函数库中的原型是：

```
#include <sys/types.h> /*  提供类型 pid_t 的定义  */
#include <unistd.h> /*  提供函数的定义  */
pid_t fork(void);
```

只看 fork 的名字，可能难得有几个人可以猜到它是做什么用的。fork 系统调用的作用是复制一个进程。当一个进程调用它，完成后就出现两个几乎一模一样的进程，我们也由此得到了一个新进程。据说 fork 的名字就是来源于这个与叉子的形状颇有几分相似的工作流程。

在 Linux 中，创造新进程的方法只有一个，就是我们正在介绍的 fork。其他一些库函数，如 system()，看起来似乎它们也能创建新的进程，如果能看一下它们的源码就会明白，它们实际上也在内部调用了 fork。包括我们在命令行下运行应用程序，新的进程也是由 shell 调用 fork 制造出来的。fork 有一些很有意思的特征，下面就让我们通过一个小程序来对它有更多的了解。

```
/* fork_test.c */
#include <sys/types.h>
#inlcude <unistd.h>
```

```
main()
{
    pid_t pid;
    /*此时仅有一个进程*/
    pid=fork();
    /*此时已经有两个进程在同时运行*/
    if(pid <0)
        printf("error in fork!");
    else if(pid==0)
        printf("I am the child process, my process ID is %d\n", getpid());
    else
        printf("I am the parent process, my process ID is %d\n", getpid());
}
```

编译并运行：

```
$gcc fork_test.c -o fork_test
$./fork_test
I am the parent process, my process ID is 1991
I am the child process, my process ID is 1992
```

看这个程序的时候，头脑中必须首先了解一个概念：在语句 pid=fork()之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是 if(pid==0)……。

两个进程中，原先就存在的那个被称作“父进程”，新出现的那个被称作“子进程”。父子进程的区别除了进程标志符（process ID）不同外，变量 pid 的值也不相同，pid 存放的是 fork 的返回值。fork 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

在父进程中，fork 返回新创建子进程的进程 ID；

在子进程中，fork 返回 0；

如果出现错误，fork 返回一个负值；

fork 出错可能有两种原因：

（1）当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。（2）系统内存不足，这时 errno 的值被设置为 ENOMEM。（关于 errno 的意义，请参考本系列的第一篇文章。）

fork 系统调用出错的可能性很小，而且如果出错，一般都为第一种错误。如果出现第二种错误，说明系统已经没有可分配的内存，正处于崩溃的边缘，这种情况对 Linux 来说是很罕见的。

说到这里，聪明的读者可能已经完全看懂剩下的代码了，如果 pid 小于 0，说明出现了错误；pid==0，

就说明 fork 返回了 0，也就说明当前进程是子进程，就去执行 printf("I am the child!")，否则（else），当前进程就是父进程，执行 printf("I am the parent!")。完美主义者会觉得这很冗余，因为两个进程里都各有一条它们永远执行不到的语句。不必过于为此耿耿于怀，毕竟很多年以前，UNIX 的鼻祖们在当时内存小得无法想象的计算机上就是这样写程序的，以我们如今的“海量”内存，完全可以把这几个字节的顾虑抛到九霄云外。

说到这里，可能有些读者还有疑问：如果 fork 后子进程和父进程几乎完全一样，而系统中产生新进程唯一的方法就是 fork，那岂不是系统中所有的进程都要一模一样吗？那我们要执行新的应用程序时候怎么办呢？从对 Linux 系统的经验中，我们知道这种问题并不存在。至于采用了什么方法，我们把这个问题留到后面具体讨论。

exit

在 2.4.4 版内核中，exit 是第 1 号调用，其在 Linux 函数库中的原型是：

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

不像 fork 那么难理解，从 exit 的名字就能看出，这个系统调用是用来终止一个进程的。无论在程序中的什么位置，只要执行到 exit 系统调用，进程就会停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。请看下面的程序：

```
/* exit_test1.c */
#include <stdlib.h>
main()
{
    printf(“this process will exit!\n”);
    exit(0);
    printf(“never be displayed!\n”);
}
```

编译后运行：

```
$gcc exit_test1.c -o exit_test1
$./exit_test1
this process will exit!
```

我们可以看到，程序并没有打印后面的“never be displayed!\n”，因为在此之前，在执行到 exit(0)时，进程就已经终止了。

exit 系统调用带有一个整数类型的参数 status，我们可以利用这个参数传递进程结束时的状态，比如说，该进程是正常结束的，还是出现某种意外而结束的，一般来说，0 表示没有意外的正常结束；其他的数值表示出现了错误，进程非正常结束。我们在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理。关于 wait 的详细情况，我们将在以后的篇幅中进行介绍。

exit 和_exit

作为系统调用而言，_exit 和 exit 是一对孪生兄弟，它们究竟相似到什么程度，我们可以从 Linux 的源码中找到答案：

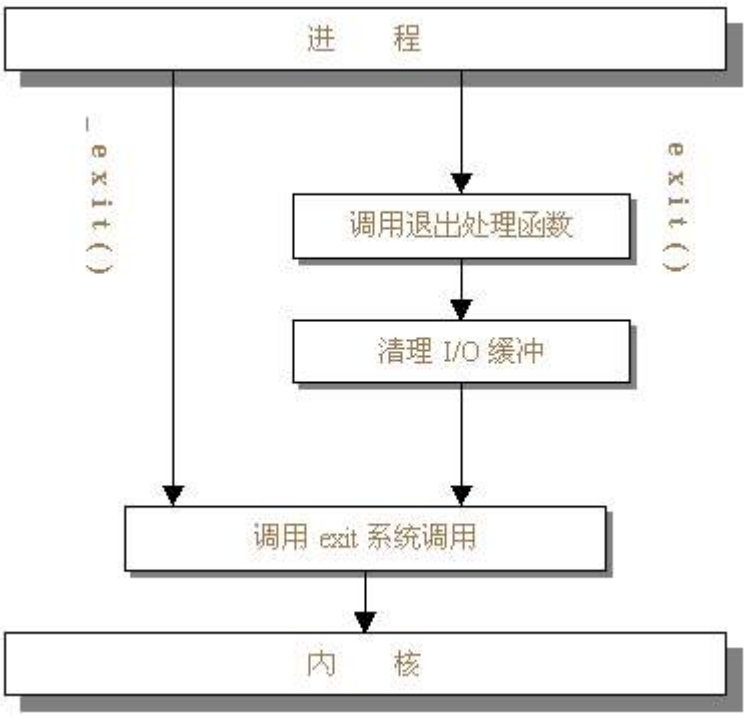
#define __NR__exit __NR_exit /* 摘自文件 include/asm-i386/unistd.h 第 334 行 */

“__NR_”是在 Linux 的源码中为每个系统调用加上的前缀，请注意第一个 exit 前有 2 条下划线，第二个 exit 前只有 1 条下划线。

这时随便一个懂得 C 语言并且头脑清醒的人都会说，_exit 和 exit 没有任何区别，但我们还要讲一下这两者之间的区别，这种区别主要体现在它们在函数库中的定义。_exit 在 Linux 函数库中的原型是：

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

和 exit 比较一下，exit()函数定义在 stdlib.h 中，而_exit()定义在 unistd.h 中，从名字上看，stdlib.h 似乎比 unistd.h 高级一点，那么，它们之间到底有什么区别呢？让我们先来看流程图，通过下图，我们会对这两个系统调用的执行过程产生一个较为直观的认识。



从图中可以看出，_exit()函数的作用最为简单：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；exit()函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序，也是因为这个原因，有些人认为 exit 已经不能算是纯粹的系统调用。

exit()函数与_exit()函数最大的区别就在于 exit()函数在调用 exit 系统调用之前要检查文件的打开情况，

把文件缓冲区中的内容写回文件，就是图中的“清理 I/O 缓冲”一项。

在 Linux 的标准函数库中，有一套称作“高级 I/O”的函数，我们熟知的 printf()、fopen()、fread()、fwrite()都在此列，它们也被称作“缓冲 I/O（buffered I/O）”，其特征是对应每一个打开的文件，在内存中都有一片缓冲区，每次读文件时，会多读出若干条记录，这样下次读文件时就可以直接从内存的缓冲区中读取，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件（达到一定数量，或遇到特定字符，如换行符\n 和文件结束符 EOF），再将缓冲区中的内容一次性写入文件，这样就大大增加了文件读写的速度，但也为我们编程带来了一点点麻烦。如果有一些数据，我们认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时我们用_exit()函数直接将进程关闭，缓冲区中的数据就会丢失，反之，如果想保证数据的完整性，就一定要使用 exit()函数。

请看以下例程：

```
/* exit2.c */
#include <stdlib.h>
main()
{
    printf(“output begin\n”);
    printf(“content in buffer”);
    exit(0);
}
```

编译并运行：

```
$gcc exit2.c -o exit2
$./exit2
output begin
content in buffer
```

```
/* _exit1.c */
#include <unistd.h>
main()
{
    printf(“output begin\n”);
    printf(“content in buffer”);
    _exit(0);
}
```

编译并运行：

```
$gcc _exit1.c -o _exit1
$./_exit1
output begin
```

在 Linux 中，标准输入和标准输出都是作为文件处理的，虽然是一类特殊的文件，但从程序员的角度来看，它们和硬盘上存储数据的普通文件并没有任何区别。与所有其他文件一样，它们在打开后也有自己的缓冲区。

请读者结合前面的叙述，思考一下为什么这两个程序会得出不同的结果。相信如果您理解了我前面所讲的内容，会很容易的得出结论。

在这篇文章中，我们对 Linux 的进程管理作了初步的了解，并在此基础上学习了 getpid、fork、exit 和 _exit 四个系统调用。在下一篇文章中，我们将学习与 Linux 进程管理相关的其他系统调用，并将作一些更深入的探讨。

3．僵尸进程

在前面的文章中，我们已经了解了父进程和子进程的概念，并已经掌握了系统调用 exit 的用法，但可能很少有人意识到，在一个进程调用了 exit 之后，该进程并非马上就消失掉，而是留下一个称为僵尸进程（Zombie）的数据结构。在 Linux 进程的 5 种状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。从这点来看，僵尸进程虽然有一个很酷的名字，但它的影响力远远抵不上那些真正的僵尸兄弟，真正的僵尸总能令人感到恐怖，而僵尸进程却除了留下一些供人凭吊的信息，对系统毫无作用。

也许读者们还对这个新概念比较好奇，那就让我们来看一眼 Linux 里的僵尸进程究竟长什么样子。

当一个进程已退出，但其父进程还没有调用系统调用 wait（稍后介绍）对其进行收集之前的这段时间里，它会一直保持僵尸状态，利用这个特点，我们来写一个简单的小程序：

```
/* zombie.c */
#include <sys/types.h>
#include <unistd.h>
main()
{
    pid_t pid;

    pid=fork();

    if(pid<0) /* 如果出错 */
        printf("error occurred!\n");
    else if(pid==0) /* 如果是子进程 */
        exit(0);
    else        /* 如果是父进程 */
        sleep(60);/* 休眠 60 秒，这段时间里，父进程什么也干不了 */
        wait(NULL);  /* 收集僵尸进程 */
}
```

sleep 的作用是让进程休眠指定的秒数，在这 60 秒内，子进程已经退出，而父进程正忙着睡觉，不可能对它进行收集，这样，我们就能保持子进程 60 秒的僵尸状态。

编译这个程序：

```
$ cc zombie.c -o zombie
```

后台运行程序，以使我们能够执行下一条命令：

```
$ ./zombie &
[1] 1577
```

列一下系统内的进程：

```
$ ps -ax
...   ...
 1177 pts/0      S      0:00 -bash
 1577 pts/0      S      0:00 ./zombie
 1578 pts/0      Z      0:00 [zombie <defunct>]
 1579 pts/0      R      0:00 ps -ax
```

看到中间的“Z”了吗？那就是僵尸进程的标志，它表示 1578 号进程现在就是一个僵尸进程。

我们已经学习了系统调用 exit，它的作用是使进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁。僵尸进程虽然对其他进程几乎没有什么影响，不占用 CPU 时间，消耗的内存也几乎可以忽略不计，但有它在那里呆着，还是让人觉得心里很不舒服。而且 Linux 系统中进程数目是有限制的，在一些特殊的情况下，如果存在太多的僵尸进程，也会影响到新进程的产生。那么，我们该如何来消灭这些僵尸进程呢？

先来了解一下僵尸进程的来由，我们知道，Linux 和 UNIX 总有着剪不断理还乱的亲缘关系，僵尸进程的概念也是从 UNIX 上继承来的，而 UNIX 的先驱们设计这个东西并非是因为闲来无聊想烦烦其他的程序员。僵尸进程中保存着很多对程序员和系统管理员非常重要的信息，首先，这个进程是怎么死亡的？是正常退出呢，还是出现了错误，还是被其它进程强迫退出的？其次，这个进程占用的总系统 CPU 时间和总用户 CPU 时间分别是多少？发生页错误的数目和收到信号的数目。这些信息都被存储在僵尸进程中，试想如果没有僵尸进程，进程一退出，所有与之相关的信息都立刻归于无形，而此时程序员或系统管理员需要用到，就只好干瞪眼了。

那么，我们如何收集这些信息，并终结这些僵尸进程呢？就要靠我们下面要讲到的 waitpid 调用和 wait 调用。这两者的作用都是收集僵尸进程留下的信息，同时使这个进程彻底消失。下面就对这两个调用分别作详细介绍。

wait

wait 的函数原型是：

```
#include <sys/types.h> /*  提供类型 pid_t 的定义  */
#include <sys/wait.h>
    pid_t wait(int *status)
```

进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。

参数 status 用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。但如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样想），我们就可以设定这个参数为 NULL，就象下面这样：

```
pid = wait(NULL);
```

如果成功，wait 会返回被收集的子进程的进程 ID，如果调用进程没有子进程，调用就会失败，此时 wait 返回-1，同时 errno 被置为 ECHILD。

实战

下面就让我们用一个例子来实战应用一下 wait 调用，程序中用到了系统调用 fork，如果你对此不大熟悉或已经忘记了，请参考上一篇文章进程管理相关的系统调用（1）。

```
/* wait1.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
    pid_t pc, pr;

    pc=fork();

    if(pc<0)        /*  如果出错  */
        printf("error ocurred!\n");
    else if(pc==0){ /*  如果是子进程  */
        printf("This is child process with pid of %d\n", getpid());
        sleep(10);/*  睡眠 10 秒钟  */
    }
    else{/*  如果是父进程  */
        pr=wait(NULL);    /*  在这里等待  */
        printf("I catched a child process with pid of %d\n"), pr);
    }
```

```
    exit(0);
}
```

编译并运行:

```
$ cc wait1.c -o wait1
$ ./wait1
This is child process with pid of 1508
I catched a child process with pid of 1508
```

可以明显注意到，在第 2 行结果打印出来前有 10 秒钟的等待时间，这就是我们设定的让子进程睡眠的时间，只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。其实这里我们不管设定子进程睡眠的时间有多长，父进程都会一直等待下去，读者如果有兴趣的话，可以试着自己修改一下这个数值，看看会出现怎样的结果。

参数 status

如果参数 status 的值不是 NULL，wait 就会把子进程退出时的状态取出并存入其中，这是一个整数值（int），指出了子进程是正常退出还是被非正常结束的（一个进程也可以被其他进程用信号结束，我们将在以后的文章中介绍），以及正常结束时的返回值，或被哪一个信号结束的等信息。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏（macro）来完成这项工作，下面我们来学习一下其中最常用的两个：

- WIFEXITED(status)

这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值。

（请注意，虽然名字一样，这里的参数 status 并不同于 wait 唯一的参数——指向整数的指针 status，而是那个指针所指向的整数，切记不要搞混了。）

- WEXITSTATUS(status)

当 WIFEXITED 返回非零值时，我们可以用这个宏来提取子进程的返回值，如果子进程调用 exit(5)退出，WEXITSTATUS(status)就会返回 5；如果子进程调用 exit(7)，WEXITSTATUS(status)就会返回 7。请注意，如果进程不是正常退出的，也就是说，WIFEXITED 返回 0，这个值就毫无意义。

下面通过例子来实战一下我们刚刚学到的内容：

```
/* wait2.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
main()
{
```

```
    int status;
    pid_t pc, pr;
    pc=fork();
    if(pc<0)  /* 如果出错 */
        printf("error ocurred!\n");
    else if(pc==0)
    { /* 子进程 */
        printf("This is child process with pid of %d.\n", getpid());
        exit(3);   /* 子进程返回 3 */
    }
    else
    {/* 父进程 */
        pr=wait(&status);

        if(WIFEXITED(status))
        { /* 如果 WIFEXITED 返回非零值 */
            printf("the child process %d exit normally.\n", pr);
            printf("the return code is %d.\n", WEXITSTATUS(status));
        }
        else /* 如果 WIFEXITED 返回零 */
            printf("the child process %d exit abnormally.\n", pr);
    }

}
```

编译并运行:

```
$ cc wait2.c -o wait2
$ ./wait2
This is child process with pid of 1538.
the child process 1538 exit normally.
the return code is 3.
```

父进程准确捕捉到了子进程的返回值 3，并把它打印了出来。

当然，处理进程退出状态的宏并不止这两个，但它们当中的绝大部分在平时的编程中很少用到，就也不在这里浪费篇幅介绍了，有兴趣的读者可以自己参阅 Linux man pages 去了解它们的用法。

进程同步

有时候，父进程要求子进程的运算结果进行下一步的运算，或者子进程的功能是为父进程提供了下一步执行的先决条件（如：子进程建立文件，而父进程写入数据），此时父进程就必须在某一个位置停下来，等待子进程运行结束，而如果父进程不等待而直接执行下去的话，可以想见，会出现极大的混乱。这种情况称为进程之间的同步，更准确地说，这是进程同步的一种特例。进程同步就是要协调好 2 个以上的进程，

使之以安排好地次序依次执行。解决进程同步问题有更通用的方法，我们将在以后介绍，但对于我们假设的这种情况，则完全可以用 wait 系统调用简单的予以解决。请看下面这段程序：

```
#include <sys/types.h>
#include <sys/wait.h>
main()
{
    pid_t pc, pr;
    int status;
    pc=fork();
    if(pc<0)
        printf("Error occured on forking.\n");
    else if(pc==0)
    {
        /* 子进程的工作 */
        exit(0);
    }
    else
    {
        /* 父进程的工作 */
        pr=wait(&status);
        /* 利用子进程的结果 */
    }
}
```

这段程序只是个例子，不能真正拿来执行，但它却说明了一些问题，首先，当 fork 调用成功后，父子进程各做各的事情，但当父进程的工作告一段落，需要用到子进程的结果时，它就停下来调用 wait，一直等到子进程运行结束，然后利用子进程的结果继续执行，这样就圆满地解决了我们提出的进程同步问题。

waitpid

waitpid 系统调用在 Linux 函数库中的原型是：

```
#include <sys/types.h> /* 提供类型 pid_t 的定义 */
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options)
```

从本质上讲，系统调用 waitpid 和 wait 的作用是完全相同的，但 waitpid 多出了两个可由用户控制的参数 pid 和 options，从而为我们编程提供了另一种更灵活的方式。下面我们就来详细介绍一下这两个参数：

- pid

从参数的名字 pid 和类型 pid_t 中就可以看出，这里需要的是一个进程 ID。但当 pid 取不同的值时，在这里有不同的意义。

pid>0 时，只等待进程 ID 等于 pid 的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就会一直等下去。

pid=-1 时，等待任何一个子进程退出，没有任何限制，此时 waitpid 和 wait 的作用一模一样。

pid=0 时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid 不会对它做任何理睬。

pid<-1 时，等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

- options

options 提供了一些额外的选项来控制 waitpid，目前在 Linux 中只支持 WNOHANG 和 WUNTRACED 两个选项，这是两个常数，可以用"|"运算符把它们连接起来使用，比如：

```
ret=waitpid(-1, NULL, WNOHANG | WUNTRACED);
```

如果我们不想使用它们，也可以把 options 设为 0，如：

```
ret=waitpid(-1, NULL, 0);
```

如果使用了 WNOHANG 参数调用 waitpid，即使没有子进程退出，它也会立即返回，不会像 wait 那样永远等下去。

而 WUNTRACED 参数，由于涉及到一些跟踪调试方面的知识，加之极少用到，这里就不多费笔墨了，有兴趣的读者可以自行查阅相关材料。

看到这里，聪明的读者可能已经看出端倪了——wait 不就是经过包装的 waitpid 吗？没错，察看<内核源码目录>/include/unistd.h 文件 349-352 行就会发现以下程序段：

```
static inline pid_t wait(int * wait_stat)
{
      return waitpid(-1, wait_stat, 0);
}
```

返回值和错误

waitpid 的返回值比 wait 稍微复杂一些，一共有 3 种情况：

- 当正常返回的时候，waitpid 返回收集到的子进程的进程 ID；
- 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0；

● 如果调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在；

当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid 就会出错返回，这时 errno 被设置为 ECHILD；

```
/* waitpid.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
main()
{
    pid_t pc, pr;
    pc=fork();
    if(pc<0)        /* 如果 fork 出错 */
        printf("Error occured on forking.\n");
    else if(pc==0)
    {/* 如果是子进程 */
        sleep(10);/* 睡眠 10 秒 */
        exit(0);
    }

    /* 如果是父进程 */
    do
    {
        pr=waitpid(pc, NULL, WNOHANG);  /* 使用了 WNOHANG 参数，waitpid 不会在这里等待 */
        if(pr==0)
        {/* 如果没有收集到子进程 */
            printf("No child exited\n");
            sleep(1);
        }
    }while(pr==0);          /* 没有收集到子进程，就回去继续尝试 */
    if(pr==pc)
        printf("successfully get child %d\n", pr);
    else
        printf("some error occured\n");
}
```

编译并运行：

```
$ cc waitpid.c -o waitpid
$ ./waitpid
No child exited
No child exited
No child exited
```

```
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 1526
```

父进程经过 10 次失败的尝试之后，终于收集到了退出的子进程。

因为这只是一个例子程序，不便写得太复杂，所以我们就让父进程和子进程分别睡眠了 10 秒钟和 1 秒钟，代表它们分别作了 10 秒钟和 1 秒钟的工作。父子进程都有工作要做，父进程利用工作的简短间歇察看子进程的是否退出，如退出就收集它。

4．exec 族

也许有不少读者从本系列文章一推出就开始读，一直到这里还有一个很大的疑惑：既然所有新进程都是由 fork 产生的，而且由 fork 产生的子进程和父进程几乎完全一样，那岂不是意味着系统中所有的进程都应该一模一样了吗？而且，就我们的常识来说，当我们执行一个程序的时候，新产生的进程的内容应就是程序的内容才对。是我们理解错了吗？显然不是，要解决这些疑惑，就必须提到我们下面要介绍的 exec 系统调用。

说是 exec 系统调用，实际上在 Linux 中，并不存在一个 exec()的函数形式，exec 指的是一组函数，一共有 6 个，分别是：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

其中只有 execve 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件。

与一般情况不同，exec 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样，颇有些神似"三十六计"中的"金蝉脱壳"。看上去还是旧的躯壳，却已经注入了新的灵魂。只有调用失败了，它们才会返回一个 -1，从原程序的调用点接着往下执行。

现在我们应该明白了，Linux 下是如何执行新程序的，每当有进程认为自己不能为系统和用户做出任何贡献了，他就可以发挥最后一点余热，调用任何一个 exec，让自己以新的面貌重生；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以 fork 出一个新进程，然后调用任何一个 exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

事实上第二种情况被应用得如此普遍，以至于 Linux 专门为其作了优化，我们已经知道，fork 会将调用进程的所有内容原封不动的拷贝到新产生的子进程中去，这些拷贝的动作很消耗时间，而如果 fork 完之后我们马上就调用 exec，这些辛辛苦苦拷贝来的东西又会被立刻抹掉，这看起来非常不划算，于是人们设计了一种"写时拷贝（copy-on-write）"技术，使得 fork 结束后并不立刻复制父进程的内容，而是到了真正实用的时候才复制，这样如果下一条语句是 exec，它就不会白白作无用功了，也就提高了效率。

稍稍深入

上面 6 条函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。在学习它们之前，先来了解一下我们习以为常的 main 函数。

下面这个 main 函数的形式可能有些出乎我们的意料：

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

它可能与绝大多数教科书上描述的都不一样，但实际上，这才是 main 函数真正完整的形式。

参数 argc 指出了运行该程序时命令行参数的个数，数组 argv 存放了所有的命令行参数，数组 envp 存放了所有的环境变量。环境变量指的是一组值，从用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，我们最常见的环境变量是 PATH，它指出了应到哪里去搜索应用程序，如/bin；HOME 也是比较常见的环境变量，它指出了我们在系统中的个人目录。环境变量一般以字符串"XXX=xxx"的形式存在，XXX 表示变量名，xxx 表示变量的值。

值得一提的是，argv 数组和 envp 数组存放的都是指向字符串的指针，这两个数组都以一个 NULL 元素表示数组的结尾。

我们可以通过以下这个程序来观看传到 argc、argv 和 envp 里的都是什么东西：

```
/* main.c */
int main(int argc, char *argv[], char *envp[])
{
    printf("\n### ARGC ###\n%d\n", argc);
    printf("\n### ARGV ###\n");
    while(*argv)
        printf("%s\n", *(argv++));
    printf("\n### ENVP ###\n");
    while(*envp)
        printf("%s\n", *(envp++));
```

```
    return 0;
}
```

编译它：

```
$ cc main.c -o main
```

运行时，我们故意加几个没有任何作用的命令行参数：

```
$ ./main -xx 000
### ARGC ###
3
### ARGV ###
./main
-xx
000
### ENVP ###
PWD=/home/lei
REMOTEHOST=dt.laser.com
HOSTNAME=localhost.localdomain
QTDIR=/usr/lib/qt-2.3.1
LESSOPEN=|/usr/bin/lesspipe.sh %s
KDEDIR=/usr
USER=lei
LS_COLORS=
MACHTYPE=i386-redhat-linux-gnu
MAIL=/var/spool/mail/lei
INPUTRC=/etc/inputrc
LANG=en_US
LOGNAME=lei
SHLVL=1
SHELL=/bin/bash
HOSTTYPE=i386
OSTYPE=linux-gnu
HISTSIZE=1000
TERM=ansi
HOME=/home/lei
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/lei/bin
_=./main
```

我们看到，程序将“./main”作为第 1 个命令行参数，所以我们一共有 3 个命令行参数。这可能与大家平时习惯的说法有些不同，小心不要搞错了。

现在回过头来看一下 exec 函数族，先把注意力集中在 execve 上：

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

对比一下 main 函数的完整形式，看出问题了吗？是的，这两个函数里的 argv 和 envp 是完全一一对应的关系。execve 第 1 个参数 path 是被执行应用程序的完整路径，第 2 个参数 argv 就是传给被执行应用程序的命令行参数，第 3 个参数 envp 是传给被执行应用程序的环境变量。

留心看一下这 6 个函数还可以发现，前 3 个函数都是以 execl 开头的，后 3 个都是以 execv 开头的，它们的区别在于，execv 开头的函数是以“char *argv[]”这样的形式传递命令行参数，而 execl 开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个 NULL 表示结束。这里的 NULL 的作用和 argv 数组里的 NULL 作用是一样的。

在全部 6 个函数中，只有 execle 和 execve 使用了 char *envp[]传递环境变量，其它的 4 个函数都没有这个参数，这并不意味着它们不传递环境变量，这 4 个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而 execle 和 execve 会用指定的环境变量去替代默认的那些。

还有 2 个以 p 结尾的函数 execlp 和 execvp，咋看起来，它们和 execl 与 execv 的差别很小，事实也确是如此，除 execlp 和 execvp 之外的 4 个函数都要求，它们的第 1 个参数 path 必须是一个完整的路径，如"/bin/ls"；而 execlp 和 execvp 的第 1 个参数 file 可以简单到仅仅是一个文件名，如"ls"，这两个函数可以自动到环境变量 PATH 制定的目录里去寻找。

实战

知识介绍得差不多了，接下来我们看看实际的应用：

```
/* exec.c */
#include <unistd.h>
main()
{
    char *envp[]={"PATH=/tmp",
                "USER=lei",
                "STATUS=testing",
                NULL};
    char *argv_execv[]={"echo", "excuted by execv", NULL};
    char *argv_execvp[]={"echo", "executed by execvp", NULL};
    char *argv_execve[]={"env", NULL};
    if(fork()==0)
        if(execl("/bin/echo", "echo", "executed by execl", NULL)<0)
            perror("Err on execl");
    if(fork()==0)
        if(execlp("echo", "echo", "executed by execlp", NULL)<0)
            perror("Err on execlp");
    if(fork()==0)
        if(execle("/usr/bin/env", "env", NULL, envp)<0)
```

```
            perror("Err on execle");
    if(fork()==0)
        if(execv("/bin/echo", argv_execv)<0)
            perror("Err on execv");
    if(fork()==0)
        if(execvp("echo", argv_execvp)<0)
            perror("Err on execvp");
    if(fork()==0)
        if(execve("/usr/bin/env", argv_execve, envp)<0)
            perror("Err on execve");
}
```

程序里调用了 2 个 Linux 常用的系统命令，echo 和 env。echo 会把后面跟的命令行参数原封不动的打印出来，env 用来列出所有环境变量。

由于各个子进程执行的顺序无法控制，所以有可能出现一个比较混乱的输出——各子进程打印的结果交杂在一起，而不是严格按照程序中列出的次序。

编译并运行：

```
$ cc exec.c -o exec
$ ./exec
executed by execl
PATH=/tmp
USER=lei
STATUS=testing
executed by execlp
excuted by execv
executed by execvp
PATH=/tmp
USER=lei
STATUS=testing
```

果然不出所料，execle 输出的结果跑到了 execlp 前面。

大家在平时的编程中，如果用到了 exec 函数族，一定记得要加错误判断语句。因为与其他系统调用比起来，exec 很容易受伤，被执行文件的位置，权限等很多因素都能导致该调用的失败。最常见的错误是：

找不到文件或路径，此时 errno 被设置为 ENOENT；

数组 argv 和 envp 忘记用 NULL 结束，此时 errno 被设置为 EFAULT；

没有对要执行文件的运行权限，此时 errno 被设置为 EACCES。

进程的一生

下面就让我用一些形象的比喻，来对进程短暂的一生作一个小小的总结：

随着一句 fork，一个新进程呱呱落地，但它这时只是老进程的一个克隆。

然后随着 exec，新进程脱胎换骨，离家独立，开始了为人民服务的职业生涯。

人有生老病死，进程也一样，它可以是自然死亡，即运行到 main 函数的最后一个"}"，从容地离我们而去；也可以是自杀，自杀有 2 种方式，一种是调用 exit 函数，一种是在 main 函数内使用 return，无论哪一种方式，它都可以留下遗书，放在返回值里保留下来；它还甚至能可被谋杀，被其它进程通过另外一些方式结束他的生命。

进程死掉以后，会留下一具僵尸，wait 和 waitpid 充当了殓尸工，把僵尸推去火化，使其最终归于无形。

这就是进程完整的一生。

5．小结

本文重点介绍了系统调用 wait、waitpid 和 exec 函数族，对与进程管理相关的系统调用的介绍就在这里告一段落，在下一篇文章，也是与进程管理相关的系统调用的最后一篇文章中，我们会通过两个很酷的实际例子，来重温一下最近学过的知识。