# 第5章　多进程编程

进程（Process）是操作系统结构的基础。进程是一个具有独立功能的程序对某个数据集在处理机上的执行过程，进程也是作为资源分配的一个基本单位。Linux 作为一个多用户、多任务的操作系统，必定支持多进程。多进程是现代操作系统的基本特征。

## 5.1　进程的基本概念

进程是现代操作系统重要的特征之一。操作系统在裸机硬件层面之上提供了更为简单、可靠、安全、高效的功能，而操作系统的首要功能就是管理和协调各种计算机系统资源，包括物理的和虚拟的资源。为了提高计算机系统中各种资源的利用效率，现代操作系统广泛采用了多道程序技术，使多种硬件资源能够并行工作。因此，程序的并发执行以及多任务共享资源成为现代操作系统的重要特点。为了描述计算机程序的执行过程和作为资源分配的基本单位，便引进了“进程”这个概念。

从提出进程这一概念以来，人们已经对进程下过许多种定义，尽管侧重点不尽相同，但都注重这一点，就是进程是一个动态的执行过程。因此可以这样定义进程的概念：进程是一个具有独立功能的程序对某个数据集在处理机上的执行过程，进程也是资源分配的基本单位。为了更好地理解进程的概念，有必要将进程与程序的概念做一下比较。

（1）进程和程序是相辅相成的。程序是进程的组成部分之一，一个进程的运行目标是执行它所对应的程序，如果没有程序，进程就失去了其存在的意义。一个程序也可以由多个进程组成。

（2）进程是一个动态概念，而程序则是一个静态概念。程序是指令的有序集合，其本身没有任何运行的含义，是一个静态的概念。进程是程序在处理机上的一次执行过程，它是一个动态的概念，动态地产生、执行，然后消亡。因此进程的存在也是暂时的。

（3）进程具有并行性特征，而程序则没有。进程具有并行特征的两个方面：独立性和异步性。独立性是指，进程是一个相对完整的资源分配单位。异步性是指，每个进程按照各自独立的、不可预知的速度向前推进。显然程序不反映执行过程，所以不具有并行性。

## 5.2　进程的描述

从构成要素来看，进程由 3 部分组成，也就是进程控制块（Process Control Block，PCB）、有关的程序段以及操作的数据集。其中进程控制块主要包括进程的一些描述信息、资源信息以及控制

信息等。系统为每个进程设置一个 PCB，它是标识和描述进程存在及相关特性的数据块，是进程存在的唯一标识，是进程动态特征的集中反映。当创建一个进程时，系统首先创建其 PCB，然后根据 PCB 中的信息对进程实施有效的管理和控制。当一个进程完成其功能之后，系统则释放 PCB，进程也随之消亡。进程控制块的具体内容随操作系统的不同而有所区别，但主要都应当包括以下信息。

（1）进程标识。每个进程都有系统唯一的进程名称或标识号。在识别一个进程时，进程名或标识号就代表该进程。

（2）状态信息。指明进程当前所处的状态，作为进程调度、分配处理机的依据。进程在活动期间有 3 种基本的状态，可分为就绪状态、执行状态和等待状态。一个进程在任一时刻只能具有这三种状态中的一种。执行状态表示该进程当前占有处理机，正在处理机上调度执行；就绪状态表示该进程已经得到了除处理机之外的全部资源，准备占有处理机；等待状态则表示进程因某种原因（等待某事件发生）而暂时不能占有处理机。当然在具体的系统中，为了最大可能地提高资源的利用率，可能会引进或者进一步细分某些状态。

（3）进程的优先级。进程优先级是选取进程占有处理机的重要依据，一般根据进程的轻重缓急程度为进程指定一个优先级，包括静态或者动态的优先级。

（4）CPU 现场信息。当进程状态变化时（例如一个进程放弃使用处理机），它需要将当时的 CPU 现场保护到内存中，以便再次占用处理机时恢复正常运行。包括各种通用寄存器、程序计数器、程序状态字等。

（5）资源清单。每个进程在运行时，除了需要内存外，还需要其他资源，如 I/O 设备、外存、数据区等。

（6）队列指针。用于将处于同一状态或者具有家族关系的进程链接成一个队列，在该单元中存放下一进程 PCB 首地址。

（7）其他，如计时信息、记账信息、通信信息等。

Linux 中的每个进程都由一个 task_struct 数据结构来表示。task_struct 其实就是通常意义上的进程控制块，或者称为进程描述符，系统正是通过 task_stmct 结构来对进程进行有效管理和控制的。当系统创建一个进程时，Linux 为新的进程分配一个 task_stmct 结构，进程结束时，又收回其 task_struct 结构，进程也随之消亡。分配给进程的 task_truct 结构可以被内核中的许多模块（如调度程序、资源分配程序、中断处理程序等）访问，并常驻于内存。在最新发布的 Linux 4.14 内核中，Linux 为每个新创建的进程动态地分配一个 task_struct 结构，系统所能允许的最大进程数是由机器所拥有的物理内存的大小决定的，这是对以前版本的改进。

Linux 支持两种进程：普通进程和实时进程。实时进程具有一定程度上的紧迫性，应该有一个短的响应时间，更重要的是，这个响应时间应该有很小的变化；而普通进程则没有这种限制。因此，调度程序需要区别对待这两类进程。

由于 task_struct 结构包含进程的全部信息，因此有必要来详细分析 task_struct 结构中所包含的内容，task_struct 结构包含的数据比较庞大，按其功能主要可分为几大部分：进程标识符信息、进程调度信息、进程间通信信息、时间和定时器信息、进程链接信息、文件系统信息、虚拟内存信息、处理器特定信息及其他信息。

（1）进程标识符信息

进程标识符信息包括进程标识符、用户标识符、组标识符等一些信息。每个进程都有一个唯一的进程标识符（Process ID，PID），内核通过这个标识符来识别不同的进程，同时，进程标识符也是内核提供给用户程序的接口。PID 是 32 位的无符号整数，存放在进程描述符的 PID 域中，它被顺序编号，新创建进程的 PID 通常是前一个进程的 PID 加 1，为了与 16 位硬件平台的传统 UNIX 系统保持兼容，Linux 上允许的最大 PID 号是 32767。当内核在系统中创建第 32768 个进程时，就必须重新开始使用闲置的 PID 号。

此外，每个进程都属于某个用户和某个用户组。进程描述符中定义了多种类别的用户标识符和组标识符，比如用户标识符（uid）、有效用户标识符（euid）以及组标识符（gid）、有效组标识符（egid）等。这些也都是简单的数字，主要用于系统的安全控制。

（2）进程调度信息

调度程序利用这些信息来决定系统中哪个进程最迫切需要运行，并采用适当的策略来保证系统运转的公平性和高效性。这些信息主要包括调度标志、调度的策略、进程的类别、进程的优先级、进程状态。其中可能的进程状态有：可运行状态、可中断的等待状态、不可中断的等待状态、暂停状态和僵死状态。

（3）进程间通信信息

在多任务编程环境中，进程之间必然会发生多种多样的合作、协调等，因此进程之间就必须进行通信，来交换信息和交流数据。Linux 支持多种不同形式的进程间通信机制，如信号、管道，也支持 System V 进程间通信机制，如信号量、消息队列和共享内存等。进程描述符中主要有这些域与进程通信相关：sig，信号处理函数，包括自定义的和系统默认的处理函数；blocked，进程所能接收信号的位掩码；sigmask_lock，信号掩码的自旋锁；semundo，进程信号量的取消操作队列，进程每操作一次信号量，都生成一个对此次操作的取消操作，这些属于同一进程的取消操作组成一个链表，当进程异常终止时，内核就会执行取消操作；semsleeping，与信号量相关的等待队列，每一信号量集合对应一个等待队列。

（4）进程链接信息

Linux 系统中所有进程都是相互联系的。除了初始化进程 init 外，其他所有进程都有一个父进程。可以通过 fork 或 clone 系统调用来创建子进程，除了进程标识符（PID）等必要的信息外，子进程的 task_struct 结构中的绝大部分信息都是从父进程中复制过来的。每个进程对应的 task_struct 结构中都包含有指向其父进程和兄弟进程（具有相同父进程的进程）以及子进程的指针。有了这些指针，进程之间的通信、协作就更加方便了。进程的 ask_struct 结构中主要有下面这些域记录了进程间的各种关系。next_task、prev_task 用于链入进程双向链表的前后指针，系统的所有进程组成一个双向循环链表。p_opptr、p_pptr、p_cptr、p_ysptr、p_osptr 分别表示指向祖先进程、父进程、子进程、兄弟进程的指针。Pidhash_next、pidhash_pprev 用于链入进程哈希表的前后指针。

（5）时间和定时器信息

内核需要记录进程的创建时间以及在其生命周期中消耗的 CPU 时间。进程耗费的 CPU 时间由两部分组成：一是在用户态（用户模式）下耗费的时间，二是在内核态（内核模式）下耗费的时间。

每个时钟滴答，也就是每个时钟中断，内核都要更新当前进程耗费的时间。Linux 支持与进程相关的多种间隔定时器，包括实时定时器、虚拟定时器和概况定时器。进程可以通过系统调用来设定定时器，以便在定时器到期后向它发送信号。这些定时器可以是一次性的或者周期性的。

（6）文件系统信息

进程经常会访问文件系统资源，打开或者关闭文件，Linux 内核要对进程使用文件的情况进行记录。task_struct 结构中有两个数据结构用于描述进程与文件相关的信息。其中，fs 域是指向 fs_struct 结构的指针，fs_struct 结构中描述了两个 VFS 索引节点，这两个索引节点叫作 root 和 pwd，分别指向进程的可执行映像所对应的主目录和当前工作目录。files 域用来记录进程打开文件的文件描述符。

（7）虚拟内存信息

Linux 采用按需分页的策略来解决进程的内存需求，当物理内存不足时，Linux 内存管理系统需要把内存中的部分页面交换到外存。每个进程都有自己的虚拟地址空间（内核线程除外），用 mm_struct 来描述，其中包含一个指向若干个虚存块的虚存队列。另外，Linux 内核还引入了另一个域 active_mm，它指向活动地址空间，但这一空间并不为该进程所拥有，通常为内核线程所使用。内核线程与用户进程相比不需要 mm_struct 结构：当用户进程切换到内核线程时，内核线程可以直接借用进程的页表，无须重新加载独立的页表。内核线程用 active_mm 指针指向所借用进程的 mm_struct 结构。

（8）处理器特定信息

进程可以看作是系统当前执行状态的综合。进程运行时，它将使用处理器的寄存器以及堆栈等。进程被挂起时，进程的上下文，即所有与 CPU 相关的处理机状态必须保存在它的 task_struct 结构中。当进程被调度重新运行时，再从中恢复这些环境，重新设定上下文，也就是恢复这些寄存器和堆栈的值。

### 5.2.1 进程的标识符

进程标识符也称进程识别码（Process Identification，进程 ID，PID），可以用来唯一表示某个进程，就像我们每个人的身份证号一样，每人都不同。就算几个进程来自同一个程序，这些进程的 ID 也是不同的。PID 是进程运行时系统随机分配的，在进程运行时，PID 是不会改变的，进程终止后，PID 就会被系统回收，以后可能会被分配给新运行的进程。

进程 ID 在系统中其实就是一个无符号整型数值，类型是 pid_t，该类型定义在 /usr/include/sys/types.h 中，定义如下：

```
#ifndef __pid_t_defined
typedef __pid_t pid_t;
# define __pid_t_defined
#endif
```

可以看到 pid_t 其实就是__pid_t 类型。而__pid_t 在/usr/include/bits/types.h 中被定义为__PID_T_TYPE 类型。在文件/usr/include/bits/typesizes.h 中可以看到这样的定义：

```
#define ____PID_T_TYPE        ___S32_TYPE
```

可以看出__PID_T_TYPE 被定义为__S32_TYPE 类型。在文件/usr/include/bits/types.h 中，我们终于找到了这样的定义：

```
#define    ____S32_TYPE       int
```

pid_t 实际上就是一个 int 型。真是山穷水尽疑无路，柳暗花明又一村。

**【例 5.1】获取 pid_t 的字节长度**

（1）新建一个 test.cpp，输入代码如下：

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    pid_t pid;
    cout << sizeof(pid_t) << endl;
    return 0;
}
```

（2）在 Linux 下编译运行后，结果如下：

```
sizeof(pid_t)=4
```

可以看到，在 64 位的 Linux 下，pid_t 的字节长度是 4，就是 int 型的大小。

我们可以在终端下用命令 ps -e 来查看所有进程的 ID，比如：

```
[root@localhost ~]# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:26 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:11 ksoftirqd/0
    7 ?        00:00:00 migration/0
    8 ?        00:00:00 rcu_bh
    9 ?        00:00:00 rcuob/0
     ...
  995 ?        00:00:00 sedispatch
 1001 ?        00:00:04 rtkit-daemon
38136 ?        00:00:00 sshd
38140 pts/3    00:00:00 bash
38816 ?        00:00:00 sshd
42238 pts/0    00:00:00 bash
```

-e 表示显示所有进程，也可以用-A，含义一样。上面第一列的内容就是进程的 ID，即 PID。最后一列就是进程的名字，和所对应的程序名字相同，因此会出现重名（比如上面的 ssh 和 bash 进程），虽然重名了，但其 PID 是不同的，因此 PID 可以用来标识一个进程。

在开发中，我们可以用函数 getpid 来获取当前进程的 ID，该函数声明如下：

```
#include <unistd.h>
pid_t getpid(void);
```

【例 5.2】获取当前进程的 ID

（1）新建一个 test.cpp 文件，输入代码如下：

```
#include <iostream>
#include <unistd.h>
using namespace std;

int main(int argc, char *argv[])
{
    pid t pid = getpid();
    cout <<"pid="<<pid << endl;
    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
    pid=42518
```

结果打印的内容就是进程 test 的 ID，多次运行可以发现每次打印的值是不同的。

### 5.2.2 PID 文件

在 Linux 系统的/var/run 目录下，一般会看到很多*.pid 文件，而且往往新安装的程序在运行后也会在/var/run 目录下产生自己的 PID 文件。它的内容是什么呢？其实，PID 文件为文本文件，内容只有一行，记录了该进程的 ID。我们可以用 cat 命令来查看 PID 文件的内容。比如可以用 cat 命令查看/var/run 目录下的 sshd.pid 文件。

```
[root@localhost ~]# cd /var/run
[root@localhost run]# cat sshd.pid
1712
```

说明进程 sshd 的 PID 是 1712。可以用 ps 来查看一下进程 sshd 的 PID。

```
[root@localhost run]# ps -e|grep ssh
 1712 ?       00:00:02 sshd
```

那么这些 PID 文件有什么作用呢？PID 文件的作用是防止进程启动多个副本。只有获得相应 PID 文件写入权限的进程才能正常启动，并把自身的 PID 写入该文件中。PID 文件位于固定路径（/var/run），并且文件名也是固定的（进程名字为.pid）。

通常有两种方法配合 PID 文件来实现进程的重复启动。一种是文件加锁法，另一种是 PID 读写法。文件加锁法的基本思路是进程运行后会给.pid 文件加一个文件锁，只有获得该锁的进程才有写入权限（F_WRLCK），以后其他试图获得该锁的进程会自动退出。给文件加锁的函数是 fcntl，如果成功锁定，进程则继续往下执行，如果锁定不成功，说明已经有同样的进程在运行了，进程就

退出。我们在第 4 章对 fcntl 函数进行了详细阐述，这里就不赘述了。PID 读写法就是先启动的进程往 PID 文件中写入自己的进程 ID 号，然后其他进程判断该 PID 文件中是否有数据了，下面看一个小例子。

【例 5.3】通过 PID 文件判断进程是否运行

（1）打开 UE，输入代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

static  char* starter_pid_file_default =  "/var/run/test.pid";


static bool check_pid(char *pid_file)
{
    struct stat stb;
    FILE *pidfile;

    if (stat(pid_file, &stb) == 0)
    {
        pidfile = fopen(pid file, "r");
        if (pidfile)
        {
            char buf[64];
            pid_t pid = 0;
            memset(buf, 0, sizeof(buf));
            if (fread(buf, 1, sizeof(buf), pidfile))
            {
                buf[sizeof(buf) - 1] = '\0';
                pid = atoi(buf);
            }
            fclose(pidfile);
            if (pid && kill(pid, 0) == 0)//检查进程
            {   /* such a process is running */
                return 1;
            }
        }
        printf("removing pidfile '%s', process not running", pid file);
        unlink(pid file);
    }
    return 0;
}


int main()
{
```

```
        FILE *fd = fopen(starter_pid_file_default, "w");

        if (fd)
        {
            fprintf(fd, "%u\n", getpid());
            fclose(fd);
        }
        if (check pid(starter pid file default))
        {
            printf("test is already running (%s exists)",
starter pid file default);

        }
        else
            printf("test is NOT running (%s NOT exists)",
starter_pid_file_default);

        unlink(starter_pid_file_default);


        return 0;
    }
```

代码中，check_pid 是一个自定义函数，用来检查 PID 文件是否存在，继而判断进程是否运行，因为整个程序设计的思路是程序刚刚启动的时候，会创建一个/var/run/test.pid 文件，并把本进程的进程号写入该文件中。在 check_pid 中，用了 stat 函数判断文件是否存在，为了保险起见，又用 fopen 打开了一次。如果存在，就读取该文件中的进程号，然后通过 kill 函数检查一下该进程是否在运行。kill 函数的第二参数表示准备发送的信号代码，如果为零，则没有任何信号送出，但是系统会执行错误检查，通常会利用 sig 值为零来检验某个进程是否仍在执行。

程序结束的时候，也就是进程即将退出的时候，我们会删除 PID 文件。

（2）上传到 Linux，然后在命令行下编译运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
test is already running (/var/run/test.pid exists)
```

## 5.3 进程的创建

### 5.3.1 使用 fork 创建进程

Linux 可以通过执行系统调用函数 fork 来创建新进程。由 fork 创建的新进程被称为子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值是子进程的 PID。子进程和父进程继续执行 fork 之后的指令。父进程和子进程几乎是等同的——它们具有相同的变量值（但变量内存并不共享），打开的文件也都相同，还有其他一些相同属性。如果父进程改变了变量的值，子进程将不会看到这个变化。实际上，子进程是父进程的一个复制，但它

们并不共享内存。实际上，Linux 并不完全复制内存页，而是采用了写时复制（copy on write）的技术，这些内存区域由父、子进程共享，而且内核将它们的许可权限改为只读，当有进程试图修改这些区域时，内核就为相关部分做一下复制。系统调用函数 fork 的声明如下：

```
#include <unistd.h>
pid_t fork();
```

该函数将创建一个子进程。如果成功，在父进程的程序中将返回子进程的线程 ID，即 PID 值；在子进程中函数则返回 0。如果失败，则在父进程程序中返回-1，并且可以通过 errno 得到错误码。

一个进程成功调用 fork 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。下面我们来看一个小例子。

**【例 5.4】通过 fork 来创建子进程**

（1）打开 UE，输入代码如下：

```
#include <iostream>
using namespace std;

#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t fpid;
    int count = 0;
    fpid = fork();
    if (fpid < 0)   //如果函数返回负数，则出错了
        cout<<"failed to fork";
    else if (fpid == 0)  //如果fork返回0，则下面进入子程序
    {
        cout<<"I am the child process, my pid is  "<<getpid()<<endl;
        count++;
    }
    else  //如果fork返回值大于0，则依旧在父进程中执行
    {
        cout<<"I am the parent process, my pid is "<<getpid()<<endl;
        cout << "fpid =" << fpid << endl;
        count++;
    }
    printf("result: %d\n", count);
    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
I am the parent process, my pid is 32726
```

```
fpid =32727
count=1
I am the child process, my pid is  32727
count=1
```

我们可以看到父进程和子进程的 PID 是不同的，说明是两个不同的进程。在语句 fpid=fork 之前，只有一个进程（父进程）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，这两个进程几乎完全相同，将要执行的下一条语句都是 if(fpid<0)，父进程和子进程都会执行这条语句。

为什么这两个进程的 fpid 不同呢？这与 fork 函数的特性有关。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。父进程 fork 返回的是子进程的 PID，我们可以看到父进程中的打印“fpid =32727”和子进程中的打印“my pid is  32727”一样，都是 32727。另外，count 分别在父进程和子进程中执行了一次 count++，所以输出都是 count=1。

有些读者可能疑惑为什么不是从第一行#include 处开始复制代码，这是因为 fork 是把进程当前的情况复制一份，执行 fork 时，进程已经执行完了语句“int count=0;”，fork 只复制下一次要执行的代码到新的进程。

再次强调，在 fork 函数执行完毕后，如果新进程创建成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID。我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。创建新进程成功后，系统中出现两个基本相同的进程，这两个进程没有固定的先后执行顺序，哪个进程先执行要看操作系统的进程调度策略。

### 5.3.2 使用 exec 创建进程

exec 用被执行的程序（新的程序）替换调用它（调用 exec）的程序。相对于 fork 函数会创建一个新的进程，产生一个新的 PID，exec 会启动一个新的程序替换当前的进程，且 PID 不变。友情提醒，胆小的朋友可略过下面的内容。当我们看恐怖片时，经常会有这样的场景：当一个人被鬼上身后，这个人的身体表面上还和以前一样，但是他的灵魂和思想已经被这个鬼占有了，因此会控制这个人做它想做的事情，exec 创建的进程就如同这样，新创建的进程已经占据了原来的进程，而表面（PID）上看起来依旧不变。那么是如何实现的呢？现在我们来学习 exec()函数族。

```
#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 execvpe(const char *file, char *const argv[],  char *const envp[]);
```

一共有 6 个函数，我们来看一下常用的几个。

#### 1. execl 函数

函数 execl 函数声明如下：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
```

其中，参数 path 指向要执行的文件路径（可以是命令的全路径、执行程序的全路径或脚本文件的全路径）；后面的参数（arg 及其后面的省略号）代表执行该程序时传递的参数列表，并且第一个被认为是 argv[0]（即 path 后面的参数被认为是 argv[0]），第二个被认为是 argv[1]……相当于 main 函数中的 argv，我们知道 main 中的 argv[0]是程序的名称，程序所需的参数是从 argv[1]才开始获取的，execl 的 argv[0]也是按照此习惯来的，即 argv[1]才是传给 execl 要启动的程序的第一个参数，argv[0]可以只写个程序名（其实对于大多数命令程序来说没什么作用，随便写一个字符串也可以，大家可以从后面的例子看到，但不要写 NULL，写 NULL 就认为参数列表就此结束了。而对于自定义程序，则要视实际情况而定，有些自定义程序需要 argv[0]，此时就不能乱输了，大家要记住紧跟 path 后面的参数相当于 main 的 argv[0]），最后一个参数必须用空指针 NULL 结束。函数成功时不返回值，失败则返回-1，失败原因存于 errno 中，可通过 perror()打印。

另外要注意的是，对于系统命令程序，比如 pwd 命令，argv[0]是必须要有的，但其值可以是一个无意义的字符串。

### 【例 5.5】使用 execl 执行不带选项的命令程序 pwd

（1）打开 UE，输入代码如下：

```
//执行/bin/pwd
#include <unistd.h>

int main()
{
    //执行/bin目录下的pwd，注意argv[0]必须要有
    execl("/bin/pwd", "asdfaf",  NULL);
    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
/zww/test
```

程序运行后，打印了当前路径，这和执行 pwd 命令是一样的。虽然 pwd 命令不带选项，但用 execl 执行的时候，依然要有 argv[0]这个参数。大家可以试试把“asdfaf”去掉，那样就会报错。不过这样乱写似乎不好看，一般都是写命令的名称，比如 execl("/bin/pwd", "pwd",  NULL); 。

### 【例 5.6】使用 execl 执行带选项的命令程序 ls

（1）打开 UE，输入代码如下：

```
/*
 * execl函数使用实例1
 *功能:执行/bin/ls -al /etc/passwd
 * */
```

```
#include <unistd.h>

int main()
{
    /*执行/bin目录下的ls，注意，argv[0]传入的是程序名ls，argv[1]才传入-al，argv[2]传入的是要查看的文件/etc/passwd  */
    execl("/bin/ls", "ls","-al","/etc/passwd",NULL);
    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
-rw-r--r--. 1 root root 2727 12月 16 2016 /etc/passwd
```

passwd 是 etc 下的一个文件。我们可以在 shell 下直接用 ls 命令进行查看。

```
[root@localhost test]# ls -la /etc/passwd
-rw-r--r--. 1 root root 2727 12月 16 2016 /etc/passwd
```

可以发现命令行运行和程序运行结果是一样的。这个程序中，execl 的第二个参数（相当于 argv[0]）其实没什么用处，我们即使随便输入一个字符串，效果也是一样的，大家可以在例子中修改一下，比如：

```
execl("/bin/ls", "lsadfadfae", "-al", "/etc/passwd", NULL);
```

运行结果不变。说明对于 execl 函数，只要提供了程序的全路径和 argv[1]开始的参数信息，就可以了。

### 【例 5.7】使用 execl 执行我们的程序

（1）首先打开 UE，编写一个小程序，代码如下：

```
#include <string.h>
using namespace std;
#include <iostream>

int main(int argc, char* argv[])
{
    int i;
    cout <<"argc=" << argc << endl; //打印一下传进来的参数个数

    for(i=0;i<argc;i++)   //打印各个参数
        cout<<argv[i]<<endl;

    if (argc == 2&&strcmp(argv[1], "-p")==0)  //判断是否带了参数-p
        cout << "will print all" << endl;
    else
        cout << "will print little" << endl;
```

```
    cout << "my program over" << endl;
    return 0;
}
```

（2）保存文件为 mytest.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ mytest.cpp -o mytest
[root@localhost test]# ./mytest
argc=1
./mytest
will print little
my program over
```

（3）小程序编写完毕，然后把它复制到一个地方，比如/zww/test 下。下面用 execl 来执行它。继续打开 UE，输入代码如下：

```
#include <unistd.h>
using namespace std;
#include <iostream>

int main(int argc, char* argv[])
{
    execl("/zww/test/mytest", NULL); //不传任何参数给mytest
    cout << "------------------\n";//如果execl执行成功，这一句不会执行到的

    return 0;
}
```

（4）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
argc=0
will print little
my program over
```

在调用 execl 时，没有传任何参数给 mytest，因此 argc 打印了 0，这说明执行自己的程序的时候，可以不传 argv[0]，这一点和执行系统命令不一样，大家可以和前面的例子比较一下。下面传 2 个参数给 mytest，调用方式改为如下：

```
execl("/zww/test/mytest", "adsfadf","-p",NULL);
```

保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
argc=2
adsfadf
-p
```

```
    will print all
    my program over
```

可以看到，我们给 mytest 传了两个参数，分别是“adsfadf”和“-p”。大家还可以试试传一个参数给 mytest 的情况，比如：execl("/zww/test/mytest", "adsfadf" ,NULL);。

下面再看一下函数 execlp，其声明如下：

```
    int execlp(const char *file, const char *arg, ...);
```

其中，参数 file 指向要执行的程序，但不需要写出完整路径，函数会到环境变量 PATH 所给出的路径中去查找，找到后便执行；后面的参数同 execl，最后一个参数也必须用空指针 NULL 作为结束。如果函数执行成功，则不会返回，执行失败则直接返回-1，错误码存于 errno 中。

### 2. execlp 函数

execlp 函数会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个以后的参数当作该文件的 argv[0]、argv[1]……，最后一个参数必须用空指针（NULL）结束。execlp 函数声明如下：

```
    #include <unistd.h>
    int execlp(const char *file, const char *arg, ...);
```

如果执行成功，则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

### 【例 5.8】使用 execlp 执行不带选项的命令程序 pwd

（1）首先打开 UE，编写一个小程序，代码如下：

```
    #include <unistd.h>

    int main(int argc, char* argv[])
    {
        execlp("pwd", "",  NULL);
        return 0;
    }
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
    [root@localhost test]# g++ test.cpp -o test
    [root@localhost test]# ./test
    /zww/test
```

execlp 的第一个参数直接用 pwd 这个命令程序即可，而不需要写出其全路径，因为环境变量 PATH 中已经包含路径/usr/bin 了，而/usr/bin 下有 pwd 这个程序了，大家可以用 echo 看一下：

```
    /usr/lib64/qt-3.3/bin:/root/perl5/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/root/bin
    [root@localhost bin]# cd /usr/bin
    [root@localhost bin]# ls pwd
    pwd
```

至于 execlp 的第二个参数为什么是空字符串，这其实不重要，传任意字符串都可以，但必须要有，不能为 NULL，否则运行会报错。这只是针对创建系统命令程序的情况，我们自己的程序无须这样。

**【例 5.9】使用 execlp 执行我们的程序**

（1）首先打开 UE，编写一个小程序，代码如下：

```
#include <string.h>
using namespace std;
#include <iostream>

int main(int argc, char* argv[])
{
    int i;
    cout <<"argc=" << argc << endl; //打印一下传进来的参数个数

    for(i=0;i<argc;i++)   //打印各个参数
        cout<<argv[i]<<endl;
}
```

（2）保存文件为 mytest.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ mytest.cpp -o mytest
[root@localhost test]# ./mytest hello world
argc=3
./mytest
hello
world
```

（3）小程序编写完毕，然后把它复制到/usr/bin 下。下面用 execlp 来执行它。继续打开 UE，输入代码如下：

```
#include <unistd.h>
using namespace std;
#include <iostream>

int main(int argc, char* argv[])
{
    execl("mytest", NULL); //不传任何参数给mytest
    cout << "------------------\n";//如果execl执行成功，这一句不会执行到

    return 0;
}
```

（4）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
argc=0
```

我们的 mytest 执行成功了。

其实，只有 execvpe 是真正意义上的系统调用，其他都是在此基础上经过包装的函数。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（调用 exec 函数族的进程）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。

细看一下，这 6 个函数都是以 exec 开头（表示属于 exec 函数簇）的，前 3 个函数后面是字母 l，表示 list（列举参数）。后 3 个函数接着字母 v，表示 vector（参数向量表）。它们的区别在于，execv 开头的函数是以“char *argv[]”(vector)形式传递命令行参数的，而 execl 开头的函数采用罗列（list）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。

### 5.3.3 使用 system 创建进程

system 函数通过调用 shell 程序来执行所传入的命令（效率低），相当于先 fork()，再 execve()。该函数的特点是源进程和子进程各自运行，且源进程需要等子进程运行完后再继续。system()会调用 fork()产生子进程，然后由子进程来调用/bin/sh -c 执行 system 函数的参数 command 字符串所代表的命令，此命令执行完后随即返回原调用的进程。/bin/sh 一般是一个软链接，指向某个具体的 shell，比如 bash，-c 选项是告诉 shell 从字符串 command 中读取命令。在该 command 执行期间，SIGCHLD 信号会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略（关于信号后面章节会讲述）。该函数声明如下：

```
#include <stdlib.h>
int system(const char *command);
```

其中，command 是要执行的命令。如果 fork 失败，返回-1，如果 command 顺利执行完毕，则返回 command 通过 exit 或 return 返回的值。

为了更好地理解 system()函数的返回值，需要了解其执行过程，实际上 system()函数执行了 3 步操作：

（1）fork 一个子进程。

（2）在子进程中调用 exec 函数去执行 command。

（3）在父进程中调用 wait 等待子进程结束。如果 fork 失败，system()函数返回-1。如果 exec 执行成功，即 command 顺利执行完毕，则返回 command 通过 exit 或 return 返回的值（注意，command 顺利执行不代表执行成功，比如 command："rm debuglog.txt"，无论文件是否存在，该 command 都顺利执行了）。如果 exec 执行失败，即 command 没有顺利执行，比如被信号中断或者 command 命令根本不存在，system()函数返回 127。如果 command 为 NULL，则 system()函数返回非 0 值，一般为 1。

看完这 3 点，肯定有人对 system()函数的返回值还是不清楚，下面给出一个使用 system()函数的例子。

```
int system(const char * cmdstring)
```

```
    {
        pid_t pid;
        int status;
        if(cmdstring == NULL)
        {
            return (1); //如果cmdstring为空，返回非零值，一般为1
        }
        if((pid = fork())<0)
        {
            status = -1; //fork失败，返回-1
        }
        else if(pid == 0)
        {
            execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);//子进程调用execl执
行cmdstring
             exit(127); // exec执行失败返回127，注意exec只在失败时才返回现在的进程，成功
的话现在的进程就不存在
        }
        else //父进程
        {
            while(waitpid(pid, &status, 0) < 0)
            {
                if(errno != EINTR)
                {
                    status = -1; //如果waitpid被信号中断，则返回-1
                    break;
                }
            }
        }
        return status; //如果waitpid成功，则返回子进程的状态
    }
```

仔细看完这个 system()函数的实现，对该函数的返回值就清楚了，比如什么时候 system()函数返回 0 呢？只在 command 命令返回 0 时。

## 5.4 进程调度

进程调度也就是处理机调度。在多道程序设计环境中，进程数往往多于处理机数，这将导致多个进程对处理机资源的互相争夺。进程调度的任务是控制和协调进程对 CPU 的竞争，按照一定的调度算法使某一就绪进程取得 CPU 的控制权，从而转为运行状态。进程调度的功能主要包括：记录系统中所有进程的执行状况；根据一定的调度算法，从就绪队列中选出一个进程来准备把处理机分配给它；将处理机分配给进程，进行上下文切换，把选中进程的进程控制块内有关的现场信息（如程序状态字、通用寄存器等内容）送入处理器相应的寄存器中，从而让它占用处理机运行。

进程的调度一般可以在下述情况下发生：

（1）正在执行的进程运行完毕。

（2）正在执行的进程调用阻塞原语将自己阻塞起，并来进入等待状态。

（3）执行中的进程提出 I/O 请求后被阻塞。

（4）正在执行的进程调用了 P 原语操作，因资源得不到满足而被阻塞；或者调用 V 原语操作释放了资源，从而激活了等待相应资源的进程队列。

（5）在分时系统中，时间片已经用完。

（6）就绪队列中的某个进程的优先级变得高于当前运行进程的优先级，从而引起进程的调度。

进程调度的主要问题是采用某种算法合理有效地将处理机分配给进程，其调度算法应尽可能提高资源的利用率，减少处理机的空闲时间。衡量进程调度的算法的指标有：面向系统的吞吐量、处理机利用率、公平性以及资源分配的平衡性等，面向用户的作业周转时间、响应时间、可预测性等。而这些“合理的目标”往往是相互制约的，难以全部达到要求。实际系统中，往往综合考虑这些因素，根据具体情况区别对待或者进行某些取舍。常见的进程调度算法有以下 4 种。

（1）先来先服务法（FCFS）

将进程变为就绪状态的先后次序排成队列，并按照先来先服务的方式进行调度处理，这是一种最普遍也是最简单的方法。

（2）时间片轮转法（RR）

其基本思想是，将 CPU 的处理时间划分成一个个时间片，就绪队列中的各个进程轮流运行一个时间片，当时间片结束时，就强迫运行进程让出 CPU，该进程进入就绪队列等待下一次调度。而同时又去选择就绪队列中的一个进程，分配给它一个时间片，以投入运行。如此轮流调度，使得就绪队列中的所有进程在有限的时间内都可以依次轮流获得一个时间片的处理机时间。这主要是分时系统中采用的一种调度算法。

（3）优先级算法

进程调度每次将处理机分配给具有最高优先级的就绪进程。进程优先级的设置可以是静态的，也可以是动态的。静态优先级是在进程创建时根据进程初始状态或者用户要求而确定，在进程运行期间不再改变。动态优先级则是指在进程创建时先确定一个初始优先级，以后在进程运行中随着进程的不断推进，其优先级值也随着不断地改变。

（4）多级反馈队列法

在实际系统中，调度模式往往是几种调度算法的结合。多级队列反馈法就是综合了先来先服务法、时间片轮转法和优先级法的一种进程调度算法。系统按照优先级别的不同设置若干个就绪队列，对级别较高的队列分配较小的时间片，对级别较低的队列分配稍大一点的时间片。除了最低一级的队列采用时间片轮转法调度之外，其他各级队列均采用先来先服务法调度。系统总是先调度级别较高队列中的进程，仅当该队列为空时才去调度下一级队列中的进程。当执行进程用完其时间片时，便被剥夺并进入下一级就绪队列。当等待进程被唤醒时，它进入与其优先级对应的就绪队列，若其优先级高于当前执行进程，便抢占 CPU 执行。

Linux 采用的是基于优先级可抢占式的调度系统，并使用 schedule 函数来实现进程调度的功能。

Linux 所实现的可抢占还只是一定程度上的抢占，因为到目前为止，Linux 内核还不是抢占式的，因此这意味着进程只有在用户态运行时才能被抢占。如果一个进程变为 TASK_RUNNING 状态，内核则会检查它的动态优先级是否大于当前正在 CPU 上运行的进程优先级。如果是，当前执行进程将被中断，并使用调度程序选择另一个进程运行（通常是刚刚变为可运行状态的进程）。此外，进程在它的时间片到期时也可以被抢占。

### 1. 优先级

为了选择一个进程运行，Linux 调度程序必须考虑每个进程的优先级。实际上，Linux 采用了两种优先级：静态优先级和动态优先级。静态优先级只针对实时进程，它由用户赋给实时进程，范围为 1~99，以后调度程序不再改变它。动态优先级只应用于普通进程，实质上它是基本时间片与当前时期内的剩余时间片之和。其实，实时进程的静态优先级总是高于普通进程的动态优先级，因此只有在处于可运行状态的进程中且没有实时进程后，调度程序才开始运行普通进程。

### 2. 调度策略

Linux 对实时进程和普通进程区别对待。对于实时进程有两种调度策略：SCHED_FIFO 和 SCHED_RR。SCHED_FIFO 就是先进先出的算法，当调度程序将 CPU 分配给一个进程时，该进程的 task_struct 结构还保留在运行队列链表的当前位置。如果没有其他更高优先级的实时进程，这个进程就可以占用 CPU 直至运行完毕。SCHED_RR 就是采用循环轮转的方法，当调度程序将 CPU 分配给一个进程时，则将这个进程的 tasl_struct 放置在运行队列的末尾。这种策略确保了把 CPU 时间公平地分配给具有相同优先级的实时进程。对于普通的分时进程采用 SCHED_OTHER 策略。

Linux 的进程调度由内核函数 schedule 实现。Linux 在进程终止、进程睡眠或者某个进程变为可运行状态时都可能会发生进程调度；如果当前进程的时间片用完，或者进程从中断、异常及系统调用返回到用户态时，都可能会发生进程调度。Linux 进程的状态转换如图 5-1 所示。

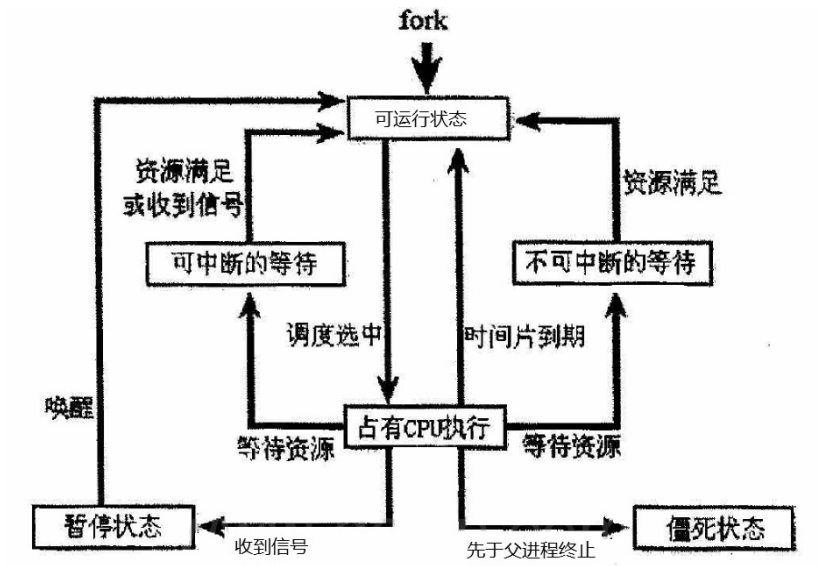


图 5-1

（1）可运行状态（TASK_RUNNING）

处于这种状态的进程，要么正在 CPU 上运行，要么准备运行。正在 CPU 上运行的进程就是当前进程（由 current 宏表示），而准备运行的进程只要得到 CPU 就可以立即投入运行，CPU 是这些进程唯一等待的系统资源。系统中有一个运行队列，用来容纳所有处于可运行状态的进程，调度程序执行时，从中选择一个进程投入运行。

（2）可中断的等待状态（TASK_INTERRUPTIBLE）

进程被挂起，直到一些条件满足为止，条件可能包括：产生一个硬件中断、释放进程正等待的系统资源或者传递一个信号，这些都有可能唤醒进程，让进程的状态返回到 TASK_RUNING 状态。

（3）不可中断的等待状态（TASK_UNINTERRUPTIBLE）

这种状态与前一种状态相似，但不同的是，传递信号给睡眠的进程并不能改变其状态。这种状态不太常见，但在一些特定的情况下是很有用的，例如进程必须等待，直到给定的事件发生，而其间不能被中断。

（4）暂停状态（TASK_STOPPED）

进程的执行已经被暂停，当进程接收到 SIGSTOP、SIGTSTP、SIGTTIN 或者 SIGTT0U 信号后，进入暂停状态。当一个进程正在被另一个进程监控时（例如调试程序执行 ptrace 系统调用来监控测试程序），每一个这样的信号都可以将这个进程置于 TASK_STOPPED 状态。

（5）僵死状态（TASK_ZOMBIE）

进程的执行已经终止，但是父进程还没有发布 wait 类系统调用来返回有关终止进程的信息。在父进程发布 wait 类系统调用之前，内核不能丢弃包含在终止进程 task_struct 结构中的数据，因为父进程可能还需要这些信息。

## 5.5 进程的分类

Linux 下，进程一般分为前台进程、后台进程和守护进程（Daemon）3 类。

### 5.5.1 前台进程

前台进程（也称普通进程）就是需要和用户交换的进程。默认情况下，启动一个进程都是在前台运行的，这时它就把 Shell 给占据了，我们无法进行其他操作，一直等到该进程终止。前面讲述的进程基本都是前台进程。查看普通进程的命令是 ps，可以根据需要加上不同的命令选项，比如通过进程名来查找进程号：

```
ps -e | grep 进程名
```

### 5.5.2 后台进程

对于那些不需要交互的进程，很多时候希望将其在后台启动，可以在启动的时候加一个“&”。

比如一个进程的名字叫 recv，我们希望它在后台运行，则可以输入：recv &。这样它就是一个后台进程了，而且不会占据 Shell，我们依然可以在 Shell 下做其他操作。但关闭 Shell 窗口的时候，后台进程也将随之退出。我们把切换到后台运行的进程称为 job。当一个进程以后台方式启动时（即启动时加上&），系统会输出该进程的相关 job 信息，会输出 job ID 和进程 ID。在后台运行的进程，可以用 ps 命令查看，或通过 jobs 命令只查看所有 job（后台进程）。如果想要终止某个后台进程，可使用命令 killall，比如终止所有名为 recv 的后台进程：killall recv。不过这种方法有点简单粗暴。

**【例 5.10】制作一个后台进程并查看**

（1）打开 UE，新建一个 test.cpp 文件，在 test.cpp 中输入代码如下：

```
#include <unistd.h>
#include <iostream>
using namespace std;

int main(void)
{
    cout << "hello,world" << endl;
    sleep(10000);
    cout << "byebye"<<endl;
}
```

（2）上传 test.cpp 到 Linux，在终端下输入命令：g++ -o test test.cpp，然后运行 test，运行结果如下：

```
[root@localhost test]# g++ -o test test.cpp
[root@localhost test]# ./test &
[1] 62096
[root@localhost test]# hello,world
```

其中，[1]表示 job 的 ID，62096 表示进程 test 的进程 ID。现在进程 test 以后台方式运行了，马上可以通过命令 jobs 来查看：

```
[root@localhost test]# jobs
[1]+  运行中              ./test &
```

显示一个后台进程 test 正在运行中。

## 5.6 守护进程

### 5.6.1 守护进程的概念

守护进程（Daemon Process）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程是一种很有用的进程。Linux 中大多数服务器都是用守护进程实现的，比如 Internet 服务器 inetd、Web 服务器 httpd 等，另外还有常见的守护进

程包括系统日志进程 syslogd、数据库服务器 mysqld 等。同时，守护进程完成许多系统任务，比如作业规划进程 crond、打印进程 lpd 等。

守护进程脱离终端运行，之所以脱离于终端，是为了避免进程被任何终端所产生的信息所打断，其在执行过程中的信息也不在任何终端上显示。Linux 中，每一个系统与用户进行交互的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭，因此守护进程要脱离终端运行，默默地在后台提供服务。

守护进程一般在系统启动时开始运行，除非强行终止，否则直到系统关机都保持运行。守护进程经常以超级用户（root）权限运行，因为它们要使用特殊的端口（1~1024）或访问某些特殊的资源。

一个守护进程的父进程是 init 进程，因为它真正的父进程在创建出子进程后就先于子进程退出了，所以它是一个由 init 继承的孤儿进程。守护进程是非交互式程序，没有控制终端，所以任何输出，无论是向标准输出设备 stdout 还是标准出错设备 stderr 的输出都需要特殊处理。守护进程的名称通常以 d 结尾，比如 sshd、xinetd、crond 等。

守护进程类似于 Windows 操作系统中的服务程序。它通常以超级用户启动，并且没有控制终端。

### 5.6.2 守护进程的特点

守护进程最重要的特点是后台运行。其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符、控制终端、会话和进程组、工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是 shell）中继承下来的。最后，守护进程的启动方式有其特殊之处。它可以在 Linux 系统启动时从启动脚本/etc/rc.d 中启动，可以由作业规划进程 crond 启动，还可以由用户终端（通常是 shell）执行。

总之，除了这些特殊性以外，守护进程与普通进程基本上没有什么区别。因此，编写守护进程实际上是把一个普通进程按照上述守护进程的特性改造成守护进程。如果对进程有比较深入的认识，就更容易理解和编程了。

守护进程有下面几个特点：

（1）守护进程都具有超级用户的权限。
（2）守护进程的父进程是 init 进程。
（3）守护进程都不用控制终端，其 TTY 列以“？”表示，TPGID 为-1。
（4）守护进程都是各自进程组合会话过程的唯一进程。

除了这几个特点之外，守护进程和普通进程基本没区别，但守护进程应该编写得更可靠、更健壮，这一点要注意。编写守护进程可以先写一个普通进程，然后按照一定的规则改造成守护进程。

### 5.6.3 查看守护进程

可以使用命令 ps x 或 ps axj 来查看当前运行着的守护进程。其中，a 表示不仅列出当前用户的进程，也列出所有其他用户的进程；x 表示不仅列有控制终端的进程，也列出所有无控制终端的进

程；j 表示列出与作业控制相关的信息。

比如我们在终端下输入 ps axj：

```
[root@localhost ~]# ps axj
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     2     0     0 ?         -1  S      0    0:00 [kthreadd]
    2     3     0     0 ?         -1  S      0    0:04 [ksoftirqd/0]
    2     7     0     0 ?         -1  S      0    0:00 [migration/0]
    2     8     0     0 ?         -1  S      0    0:00 [rcu_bh]
```

从上面的结果可以看出守护进程的特点，TTY 表示控制终端，可以看到这几个守护进程的控制终端为“？”，意思是这几个守护进程没有控制终端。UID 为 0，表示进程的启动者是超级进程。

### 5.6.4 守护进程的分类

根据守护进程的启动和管理方式，可以将守护进程分为独立启动守护进程和超级守护进程两类。

独立启动（stand_alone）守护进程：该类守护进程随系统启动，启动后就常驻内存，所以会一直占用系统资源。其最大的优点是它会一直启动，当外界有要求时响应速度较快，比如 httpd 等进程。此类守护进程通常保存在/etc/rc.d/init.d 目录下。

超级守护进程：系统启动时，由一个统一的守护进程 xinet 来负责管理一些进程，当响应请求到来时，需要通过 xinet 的转接才可以唤醒被 xinet 管理的进程。这种进程的优点是，最初只有 xinet 这一守护进程占有系统资源，其他的内部服务并不一直占有系统资源，只有数据包或其他请求到来时才会被 xinet 唤醒。并且还可以通过 xinet 对它所管理的进程设置一些访问权限，相当于多了一层管理机制。

我们可以用银行业务来形容这两类守护进程。

独立启动守护进程：银行里有一种单服务的窗口，像取钱、存钱等窗口，这种窗口边上始终会坐着一个人，如果有人来取钱或存钱，可以直接到相应的窗口去办理，这个处理单一服务的始终存在的人就是独立启动的守护进程。

超级守护进程：银行里还有一种窗口，提供综合服务，像汇款、转账、提款等业务，这种窗口附近也始终坐着一个人（xinet），他可能不提供具体的服务，提供具体服务的人在里面闲着聊天、喝茶，但是当有人来汇款时，他会通知里面的人，比如有人来汇款了，他会通知里面管汇款的工作人员，然后里面管汇款的工作人员会立马跑过来帮忙办完汇款业务。其他的人继续聊天、喝茶。这些负责具体业务的人就称为超级守护进程。当然，可能汇款时会有一些规则，比如不能往北京汇款，管汇款的工作人员就会提早告诉外面的管理员，当有人想往北京汇款时，管理员就直接告诉他不能办理，于是根本不会去喊汇款员，相当于提供了一层管理机制。这里需要注意的是，超级守护进程的管理员 xinet 也是一个守护进程，只不过它的任务就是传话，其实这也是一个很具体很艰巨的任务。

当然，每个守护进程都会监听一个端口（银行窗口），一些常用守护进程的监听端口是固定的，像 httpd 监听 80 端口、sshd 监听 22 端口等，我们可以将其理解为责任制，时刻等待，有求必应。具体的端口信息可以通过 cat /etc/services 来查看。

每个守护进程都会有一个脚本，可以理解成工作配置文件，守护进程的脚本需要放在指定位

置，独立启动守护进程的脚本放在/etc/init.d/目录下，当然也包括 xinet 的 shell 脚本；超级守护进程按照 xinet 中脚本的指示，所管理的守护进程位于/etc/xinetd.config 目录下。

### 5.6.5 守护进程的启动方式

守护进程一般是随着系统启动而自动激活的。它可以通过以下方式启动：

（1）在系统启动时由启动脚本启动，这些启动脚本通常放在 /etc/rc.d 目录下。
（2）利用 inetd 超级服务器启动，如 telnet 等。
（3）由 cron 定时启动，在终端用 nohup 启动的进程也是守护进程。

### 5.6.6 编写守护进程的步骤

在 Linux 或者 UNIX 操作系统中，在系统引导的时候会开启很多服务，这些服务就叫作守护进程。为了增加灵活性，root 可以选择系统开启的模式，这些模式叫作运行级别，每一种运行级别以一定的方式配置系统。守护进程是脱离于终端并且在后台运行的进程。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示，并且进程也不会被任何终端所产生的终端信息所打断。

在具体编写守护进程之前，我们先来了解一下守护进程编程的基本步骤。

（1）创建子进程，父进程退出

这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此完成第一步后就会在 Shell 终端里造成程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在 Shell 终端里则可以执行其他命令，从而在形式上做到与控制终端的脱离。

在 Linux 中，父进程先于子进程退出会造成子进程成为孤儿进程，而每当系统发现一个孤儿进程时，就会自动由 1 号进程（init）收养它，这样原先的子进程就会变成 init 进程的子进程。

（2）在子进程中创建新对话

这个步骤是创建守护进程中最重要的一步，虽然它的实现非常简单，但意义却非常重大。在这里使用的是系统函数 setsid，在具体介绍 setsid 之前，首先要了解两个概念：进程组和会话周期。

进程组：是一个或多个进程的集合。进程组由进程组 ID 来唯一标识。除了进程号（PID）之外，进程组 ID 也是一个进程的必备属性。每个进程组都有一个组长进程，其组长进程的进程号等于进程组 ID，且该进程组 ID 不会因组长进程的退出而受到影响。

会话周期：会话周期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出，在此期间，该用户运行的所有进程都属于这个会话周期。

接下来具体介绍 setsid 的相关内容。

setsid 函数用于创建一个新的会话，并担任该会话组的组长。调用 setsid 有下面 3 个作用。

- 让进程摆脱原会话的控制。
- 让进程摆脱原进程组的控制。
- 让进程摆脱原控制终端的控制。

那么，在创建守护进程时为什么要调用 setsid 函数呢？由于创建守护进程的第一步调用了 fork 函数来创建子进程，再将父进程退出。在调用 fork 函数时，子进程全盘复制了父进程的会话周期、进程组、控制终端等，虽然父进程退出了，但会话周期、进程组、控制终端等并没有改变，因此还不是真正意义上的独立开来，而 setsid 函数能够使进程完全独立出来，从而摆脱其他进程的控制。

（3）改变当前目录为根目录

这一步也是必要的步骤。使用 fork 创建的子进程继承了父进程当前的工作目录。由于在进程运行中，当前目录所在的文件系统（如“/mnt/usb”）是不能卸载的，这对以后的使用会造成诸多麻烦。因此，通常的做法是让“/”作为守护进程的当前工作目录，这样就可以避免上述的问题。当然，如果有特殊需要，也可以把当前工作目录换成其他的路径，如/tmp。改变工作目录的常见函数是 chdir。

（4）重设文件权限掩码

文件权限掩码是指屏蔽掉文件权限中的对应位。比如，有个文件权限掩码是 050，它就屏蔽了文件组拥有者的可读与可执行权限。由于使用 fork 函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多麻烦。因此，把文件权限掩码设置为 0 可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 umask。在这里，通常的使用方法为 umask(0)。

（5）关闭文件描述符

同文件权限掩码一样，用 fork 函数新建的子进程会从父进程那里继承一些已经打开的文件。这些被打开的文件可能永远不会被守护进程读写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸载。

由于守护进程是脱离控制终端的，因此从终端输入的字符不可能到达守护进程，守护进程中用常规方法（如 printf）输出的字符也不可能在终端上显示出来。所以，文件描述符为 0、1 和 2 的 3 个文件（常说的输入、输出和报错）已经失去了存在的价值，也应被关闭。通常按如下方式关闭文件描述符：

```
for(i=0;i<MAXFILE;i++)
    close(i);
```

（6）守护进程退出处理

当用户需要外部停止守护进程运行时，往往会使用 kill 命令停止该守护进程。所以，守护进程中需要编码来实现 kill 发出的 signal 信号处理，达到进程的正常退出。

```
signal(SIGTERM, sigterm_handler);
void sigterm_handler(int arg)
{
   _running = 0;
}
```

这样，一个简单的守护进程就建立起来了。下面我们来看一个具体实例。

【例 5.11】隔 10 秒在/tmp/dameon.log 中写入一句话

（1）打开 UE，输入代码如下：

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <fcntl.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <sys/wait.h>
    #include <signal.h>
    #include <sys/stat.h>

    #define MAXFILE 65535

    volatile sig_atomic_t _running = 1;

    void sigterm_handler(int arg)
    {
        running = 0;
    }

    int main()
    {
        pid_t pc;
        int i, fd, len;
        char *buf = "this is a Dameon\n";
        len = strlen(buf);
        pc = fork(); //第一步
        if(pc < 0)
        {
            printf("error fork\n");
            exit(1);
        }
        else if(pc > 0)
            exit(0);

        setsid(); //第二步
        chdir("/"); //第三步
        umask(0); //第四步
        for(i = 0 ; i < MAXFILE ; i++) //第五步
            close(i);
        signal(SIGTERM, sigterm_handler);
        while(_running)
        {
            if((fd = open("/tmp/dameon.log", O_CREAT | O_WRONLY | O_APPEND, 0600))
< 0) \
            {
```

```
                perror("open");
                exit(1);
            }
            write(fd, buf, len + 1);
            close(fd);
            usleep(10 * 1000); //10毫秒
        }

    }
```

（2）保存代码为 test.cpp，上传到 Linux，在命令行下编译生成 test 程序。然后重启 Linux 就可以开始运行了。

# 第6章　Linux进程间的通信

Linux 中的进程为了能在同一项任务上协调工作，它们彼此之间必须能够进行通信。对于一个操作系统来说，进程间的通信是不可或缺的。Linux 支持多种不同方式的进程间通信机制，如信号、管道、FIFO 和 System V IPC 机制，其中 System V IPC 机制包括：信号量、消息队列和共享内存 3 种机制。Linux 下的这些进程间通信机制基本上是从 UNIX 平台上的进程间通信机制继承和发展而来的。下面我们分别阐述这些进程通信方式。

本章将讲述 Linux 常用的 3 种进程通信方式：信号、管道和消息队列。效果差别不是很大，大家熟练一种，基本上就可以应对一般的一线开发场景，当然熟练 3 种就更好了。

## 6.1　信号

### 6.1.1　信号的基本概念

信号可以说是最早引入类 UNIX 系统中的进程间通信方式之一，Linux 同样支持这种通信方式。信号是很短的信息，可以被发送到一个进程或者一组进程，发送给进程的这个唯一信息通常是标识信号的一个数。信号可以从键盘中断中产生，进程在虚拟内存的非法访问等系统错误环境下也会有信号产生，信号也可以被 shell 程序用来向其子进程发送任务控制命令等。信号异步地发生，也就是说没有确定的时序。而收到信号的进程则采取某种行为或者将其忽略。大多数信号可以被阻塞，以便能够在稍后的时间里再采取行动。

信号机制可以说是在软件层次上对中断机制的模拟。Linux 使用信号主要有两个目的：一是让进程意识到已经发生了一个特定的事件；二是迫使进程执行包含在其自身代码中的信号处理程序。对于每一个信号，进程可以采取以下 3 种行为之一。

（1）忽略信号。进程将忽略这个信号的出现。但有两个信号不能被忽略：SIGKILL 和 SIGSTOP。

（2）执行与这个信号相关的默认操作。由内核预定义的这个操作依赖于信号的类型，默认操作可以是这些类型之一：忽略信号，内核将信号丢弃，信号对进程没有任何影响（进程永远不知道曾经出现过该信号）；终止（杀死）进程，有时是指进程异常终止，而不是进程因调用 exit 而发生的正常终止；产生核心转储文件，同时进程终止，核心转储文件包含对进程虚拟内存的镜像，可将其加载到调试器中以检查进程终止时的状态；停止（不是终止）进程，使进程暂停执行；执行之前被暂停的进程。

（3）调用相应的信号处理函数来捕获信号，进程可以事先登记特殊的信号处理函数，当进程收到信号时，信号处理函数被调用。当从信号处理函数返回后，被中断的进程将从其断点处重新开始执行。

Linux 支持 P0SIX 标准信号和实时信号，但内核不使用实时信号。我们可以用 kill 命令来显示 Linux 支持的信号列表，比如：

```
[root@localhost ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

上面的列表中，编号为 1～31 的信号为传统 UNIX 支持的信号，是不可靠信号（非实时的）；编号为 32～64 的信号是后来扩充的，称作可靠信号（实时信号）。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。下面我们对编号小于 SIGRTMIN 的信号进行讨论。

（1）SIGHUP

本信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一 Session 内的各个作业，这时它们与控制终端不再关联。登录 Linux 时，系统会分配给登录用户一个终端（Session）。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于这个 Session。当用户退出 Linux 登录时，前台进程组和后台进程组有对终端输出的进程将会收到 SIGHUP 信号。这个信号的默认操作为终止进程，因此前台进程组和后台进程组有终端输出的进程就会中止。不过可以捕获这个信号，比如 wget 能捕获 SIGHUP 信号，并忽略它，这样即使退出了 Linux 登录，wget 也能继续下载。此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。

（2）SIGINT

程序终止（interrupt）信号，在用户输入 INTR 字符（通常是 Ctrl+C）时发出，用于通知前台进程组终止进程。

（3）SIGQUIT

和 SIGINT 类似，但由 QUIT 字符来控制。进程因收到 SIGQUIT 退出时会产生 core 文件，在这个意义上类似于一个程序错误信号。

（4）SIGILL

执行了非法指令。通常是因为可执行文件本身出现错误，或者试图执行数据段。堆栈溢出时也有可能产生这个信号。

（5）SIGTRAP

由断点指令或其他 trap 指令产生，由 debugger 使用。

（6）SIGABRT

调用 abort 函数生成的信号。

（7）SIGBUS

非法地址，包括内存地址对齐（alignment）出错。比如访问一个 4 个字长的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于，后者是由于对合法存储地址的非法访问触发的（如访问不属于自己存储空间或只读存储空间的数据）。

（8）SIGFPE

在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术错误。

（9）SIGKILL

用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。

（10）SIGUSR1

留给用户使用。

（11）SIGSEGV

试图访问未分配给自己的内存，或试图往没有写权限的内存地址写数据。

（12）SIGUSR2

留给用户使用。

（13）SIGPIPE

管道破裂。这个信号通常在进程间通信产生，比如采用 FIFO（管道）通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到 SIGPIPE 信号。此外，用 Socket 通信的两个进程，写进程在写 Socket 的时候，读进程已经终止。

（14）SIGALRM

时钟定时信号，计算的是实际时间或时钟时间，alarm 函数使用该信号。

（15）SIGTERM

程序结束（terminate）信号，与 SIGKILL 不同的是，该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell 命令 kill 默认产生这个信号。如果进程终止不了，我们才会尝试

SIGKILL。

（16）SIGSTKFLT

Linux 专用，数学协处理器的栈异常。

（17）SIGCHLD

子进程结束时，父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待（wait）子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情况我们应该避免（父进程或者忽略 SIGCHILD 信号，或者捕捉它，或者等待它派生的子进程，或者父进程先终止，这时子进程的终止自动由 init 进程来接管）。

（18）SIGCONT

让一个停止（stopped）的进程继续执行。本信号不能被阻塞，可以用一个 handler 来让程序在由停止状态变为继续执行时完成特定的工作，例如重新显示提示符。

（19）SIGSTOP

停止进程的执行。注意它和 terminate 以及 interrupt 的区别：该进程还未结束, 只是暂停执行，本信号不能被阻塞、处理或忽略。

（20）SIGTSTP

停止进程的运行，但该信号可以被处理和忽略，用户输入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号。

（21）SIGTTIN

当后台作业要从用户终端读数据时，该作业中的所有进程会收到 SIGTTIN 信号。默认时这些进程会停止执行。

（22）SIGTTOU

类似于 SIGTTIN，但在写终端（或修改终端模式）时收到。

（23）SIGURG

有“紧急”数据或带外（out-of-band）数据到达 socket 时产生。

（24）SIGXCPU

超过 CPU 时间资源限制。这个限制可以由 getrlimit/setrlimit 来读取/改变。

（25）SIGXFSZ

进程企图扩大文件，以至于超过文件大小资源限制。

（26）SIGVTALRM

虚拟时钟信号。类似于 SIGALRM，但是计算的是该进程占用的 CPU 时间。

（27）SIGPROF

类似于 SIGALRM/SIGVTALRM，但包括该进程用的 CPU 时间以及系统调用的时间。

（28）SIGWINCH

窗口大小改变时发出。

（29）SIGIO

文件描述符准备就绪，可以开始进行输入/输出操作。

（30）SIGPWR

电源失败。

（31）SIGSYS

非法的系统调用。

在以上列出的信号中，程序不可捕获、阻塞或忽略的信号有：SIGKILL 和 SIGSTOP 。不能恢复至默认动作的信号有：SIGILL 和 SIGTRAP。默认会导致进程流产的信号有：SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGIOT、SIGQUIT、SIGSEGV、SIGTRAP、SIGXCPU、SIGXFSZ。默认会导致进程退出的信号有：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM。 默认会导致进程停止的信号有：SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU。默认进程忽略的信号有：SIGCHLD、SIGPWR、SIGURG、SIGWINCH。

总的来说，可以归纳为下列 5 种方式：

（1）硬件异常产生信号。比如无效的内存访问将产生 SIGSEGV 信号，而除数为 0 时将产生 SIGFPE 信号等。这些条件通常由硬件检测到，并将其通知给内核，内核为该条件发生时正在运行的进程产生适当的信号。

（2）软件条件触发信号。当内核检测到某种软件条件已经发生，并将其通知给有关进程时，也产生信号，比如进程所设置的定时器到期。

（3）用户按某些终端键时产生信号。比如用户在键盘终端上按 Ctrl+C 键将产生 S1GINT 信号。

（4）用户使用 kill 命令将信号发送给进程。kill 命令的语法是这样的：

```
kill[参数][进程号]
```

其中，参数通常取如下几项。

- -l: 使用 “-l” 参数会列出全部的信号名称。
- -a: 当处理当前进程时，不限制命令名和进程号的对应关系。
- -p: 指定 kill 命令只打印相关进程的进程号，而不发送任何信号。
- -s: 指定发送信号。
- -u: 指定用户。

一般可以用该命令终止一个失控的后台进程，比如希望尽快终止一个进程，可以使用命令：kill -9 pid。

（5）进程使用系统调用函数 kill 将信号发送给一个进程或一组进程。注意，这个系统调用 kill

不是杀死进程，而是一个进程发送信号给另一个进程。其中，要求接收信号进程和发送信号进程的所有者相同，或者发送信号进程的所有者是超级用户。

Linux 内核中并没有专门的机制来区分不同信号的相对优先级。也就是说，当有多个信号在同一时刻发出时，进程可能会以任意的顺序接收到信号并进行处理。此外，当进程调用信号处理函数处理某个信号时，一般会自动阻塞相同的信号，直到信号处理结束。Linux 通过存储在进程 task_struct 结构中的信息来实现信号，它维护着挂起的信号（已经产生但还没有被接收的信号）、阻塞信号的掩码以及进程处理每个可能信号的信息等。信号并非一产生就立刻交付给进程，而是必须等到进程再次运行时才交付给进程。进程在系统调用退出之前，它都会检查是否有可以立刻发送的非阻塞信号。当然，进程可以选择去等待信号，此时进程将一直处于可中断状态直到信号出现。

## 6.1.2 与信号相关的系统调用

通过系统调用，进程可以向其他进程发送信号，也可以更改默认的信号处理函数、阻塞信号的掩码以及检查是否有挂起的信号等。与信号相关的系统调用主要有 kill()、sigaction()、sigprocmask()、sigpending()、signal()等。

### 1. 使用 kill 发送信号

系统调用 kill()用来向一个进程或一个进程组发送一个信号，其中第一个参数决定信号发送的对象，该系统调用声明如下：

```
#include <sys/types. h>
#include <signal. h>
int kill(pid_t pid, int sig);
```

其中，pid 可能的选择有以下 4 种：

（1）当 pid>0 时，pid 是信号欲送往的进程的标识。
（2）当 pid=0 时，信号将送往所有与调用 kill()的那个进程属于同一个使用组的进程。
（3）当 pid=-1 时，信号将送往所有调用进程有权给其发送信号的进程，除了进程 1（init）外。
（4）当 pid<-1 时，信号将送往以-pid 为组标识的进程。

参数 sig 表示准备发送的信号代码，如果其值为零，则没有任何信号送出，但是系统会执行错误检查，通常会利用 sig 值为 0 来检验某个进程是否仍在执行。当函数成功执行时，返回 0，否则返回-1，此时 errno 可以得到错误码，错误码值 EINVAL 表示指定的信号码无效（参数 sig 不合法），错误码值 EPERM 表示权限不够，无法传送信号给指定进程，错误码值 ESRCH 表示参数 PID 所指定的进程或进程组不存在。

**【例 6.1】使用 kill 发送信号终止目标进程**

（1）打开 UE，输入代码如下：

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

```
#include <signal.h>
#include <unistd.h>
int main(void)
{
    pid_t childpid;
    int status;
    int retval;

    childpid = fork(); //创建子进程
    if (-1 == childpid) //判断是否创建失败
    {
        perror("fork()");
        exit(EXIT FAILURE);
    }
    else if (0 == childpid)
    {
        puts("In child process");
        sleep(100);//让子进程睡眠，以便查看父进程的行为
        exit(EXIT_SUCCESS);
    }
    else
    {
        if (0 == (waitpid(childpid,&status,WNOHANG)))//判断子进程是否已经退出
        {
            retval=kill(childpid,SIGKILL);//发送SIGKILL给子进程，要求其停止运行

            if (retval) //判断是否发生信号
            {
                puts("kill failed.");
                perror("kill");
                waitpid(childpid, &status, 0);
            }
            else
            {
                printf("%d killed\n", childpid);
            }

        }
    }

    exit(EXIT_SUCCESS);
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
39759 killed
```

上面例子的代码首先创建了一个子进程，然后让子进程休眠一会儿，在父进程中判断子进程

是否存在，如果存在，则发送 SIGKILL 信号给子进程，让其退出。其中，函数 waitpid 会暂时停止目前进程的执行，直到有信号来到或子进程结束，声明如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

其中，参数 pid 为欲等待的子进程识别码，不同的取值含义不同，具体如下：

- 当 pid<-1 时，等待进程组识别码为 pid 绝对值的任何子进程。
- 当 pid=-1 时，等待任何子进程，相当于 wait()。
- 当 pid=0 时，等待进程组识别码与目前进程相同的任何子进程。
- 当 pid>0 时，等待任何子进程识别码为 pid 的子进程。

参数 options 提供了一些额外的选项来控制 waitpid，常见的有 WNOHANG 或 WUNTRACED，WNOHANG 表示若 PID 指定的子进程没有结束，则 waitpid()函数返回 0，不予以等待，若结束，则返回该子进程的 ID。WUNTRACED 表示若子进程进入暂停状态，则马上返回，若子进程处于结束状态，则不予以理会。如果不想使用 options，可以把 options 设为 NULL。参数 status 用来存放子进程的结束状态。如果函数执行成功，则返回子进程识别码（PID），如果有错误发生，则返回值-1，失败原因存于 errno 中。

### 2. 使用 sigaction 查询或设置信号处理方式

系统调用 sigaction()可以用来查询或设置信号处理方式。函数声明如下：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,struct sigaction
*oldact);
```

参数 signum 表示要操作的信号，可以指定 SIGKILL 和 SIGSTOP 以外的所有信号；act 表示要设置的对信号的新处理方式，它是一个结构体指针；oldact 表示原来对信号的处理方式。如果函数执行成功就返回 0，否则返回-1。结构体 struct sigaction 用来描述对信号的处理，定义如下：

```
struct sigaction
{
   void     (*sa handler)(int);
   void     (*sa sigaction)(int, siginfo t *, void *);
   sigset_t  sa_mask;
   int       sa_flags;
   void     (*sa_restorer)(void);
};
```

在这个结构体中，成员 sa_handler 是一个函数指针，指向一个信号处理函数；成员 sa_sigaction 则是另一个信号处理函数，它有 3 个参数，可以获得关于信号的更详细的信息，当 sa_flags 成员的值包含 SA_SIGINFO 标志时，系统将使用 sa_sigaction 函数作为信号处理函数，否则使用 sa_handler 作为信号处理函数。在某些系统中，成员 sa_handler 与 sa_sigaction 被放在联合体中，

因此使用时不要同时设置。成员 sa_mask 用来指定在信号处理函数执行期间需要被屏蔽的信号，特别是当某个信号被处理时，它自身会被自动放入进程的信号掩码，因此在信号处理函数执行期间，这个信号不会再度发生。

sa_flags 成员用于指定信号处理的行为，它可以是以下值的“按位或”组合。

- SA_RESTART：使被信号打断的系统调用自动重新发起。
- SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号。
- SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也，不会成为僵尸进程。
- SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间，仍能发出这个信号。
- SA_RESETHAND：信号处理之后重新设置为默认的处理方式。
- SA_SIGINFO：使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数。

成员 re_restorer 则是一个已经废弃的数据域，不要使用。

如果希望能用相同方式处理某信号的多次出现，最好用 sigaction，因为它设置的响应函数设置后就一直有效，不会重置。

下面用一个小例子来说明 sigaction 函数的使用。

### 【例 6.2】系统调用 sigaction 函数的简单使用

（1）打开 UE，输入代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

static void sig_usr(int signum)
{
    if (signum == SIGUSR1)
    {
        printf("SIGUSR1 received\n");
    }
    else if (signum == SIGUSR2)
    {
        printf("SIGUSR2 received\n");
    }
    else
    {
        printf("signal %d received\n", signum);
    }
}
```

```
int main(void)
{
    char buf[512];
    int  n;
    struct sigaction sa_usr;
    sa_usr.sa_flags = 0;
    sa_usr.sa_handler = sig_usr;   //信号处理函数

    sigaction(SIGUSR1, &sa usr, NULL);  //设置信号处理方式
    sigaction(SIGUSR2, &sa usr, NULL); //设置信号处理方式
    printf("My PID is %d\n", getpid()); //打印当前进程的pid
    while (1)
    {
        if ((n = read(STDIN FILENO, buf, 511)) == -1)// 从标准输入读入字符
        {
            if (errno == EINTR)
            {
                printf("read is interrupted by signal\n");
            }
        }
        else
        {
            buf[n] = '\0';
            printf("%d bytes read: %s\n", n, buf);
        }
    }

    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
My PID is 58471
```

此时，我们可以另外打开一个终端，然后输入发送信号的命令：

```
[root@localhost ~]# kill -USR1 58471
```

这样程序就会收到信号，并打印信息了：

```
SIGUSR1 received
read is interrupted by signal
```

这说明用 sigaction 注册信号处理函数时，不会自动重新发起被信号打断的系统调用。如果需要自动重新发起，则要设置 SA_RESTART 标志，比如在上例中可以进行类似 sa_usr.sa_flags = SA_RESTART;的设置。

### 3. 使用 sigprocmask 检测或更改信号屏蔽字

系统调用 sigprocmask()可以检测或更改其信号屏蔽字。一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集。函数声明如下：

```
#include <signal.h>
int  sigprocmask(int  how, const sigset_t *set, sigset_t *oldset);
```

其中，参数 how 用于指定信号修改的方式，可能的选择有 3 种：

- SIG_BLOCK 表示加入信号到进程屏蔽。
- SIG_UNBLOCK 表示从进程屏蔽里将信号删除。
- SIG_SETMASK 表示将 set 的值设定为新的进程屏蔽。

参数 set 为指向信号集的指针，在此专指新设的信号集，如果仅想读取现在的屏蔽值，可将其设置为 NULL；参数 oldset 也是指向信号集的指针，在此存放原来的信号集。如果函数成功执行，返回 0，失败则返回-1，errno 被设为 EINVAL。

下面用一个小例子来说明函数 sigprocmask 的使用。

**【例 6.3】系统调用 sigprocmask 的使用**

（1）打开 UE，输入代码如下：

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

void handler(int sig) //SIGINT信号处理函数
{
    printf("Deal SIGINT");
}
int main()
{
    sigset_t newmask;
    sigset_t oldmask;
    sigset_t pendmask;

    struct sigaction act;
    act.sa_handler = handler;  //handler为信号处理函数首地址
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, 0);  //信号捕捉函数，捕捉Ctrl+C
    sigemptyset(&newmask);//初始化信号量集
    sigaddset(&newmask, SIGINT);//将SIGINT添加到信号量集中
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);//将newmask中的SIGINT阻塞掉，并保存当前信号屏蔽字到Oldmask
    sleep(5);//休眠5秒钟，说明:在5秒休眠期间，任何SIGINT信号都会被阻塞，如果在5秒内收到任何键盘的Ctrl+C信号，则此时会把这些信息存在内核的队列中，等待5秒结束后，可能要处理此信号
    sigpending(&pendmask);//检查信号是悬而未决的，这个函数后面会讲到
```

```
    //判断信号SIGINT是否悬而未决。所谓悬而未决，是指SIGINT被阻塞还没有被处理
        if (sigismember(&pendmask, SIGINT))
            printf(" SIGINT pending\n");
        sigprocmask(SIG_SETMASK, &oldmask, NULL);//恢复被屏蔽的信号SIGINT

        //此处开始可以处理信号
        printf("SIGINT unblocked\n");
        sleep(5);  //在这个时间段内，如果按下Ctrl+C,则会调用函数handler
        return (0);
    }
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
^C^C^C SIGINT pending
Deal SIGINTSIGINT unblocked
^CDeal SIGINT[root@localhost test]#
```

程序运行后，在开始的 5 秒内，如果按 Ctrl+C 键，则不会有反应，因为这个信号被我们屏蔽了，过了 5 秒后，再按下 Ctrl+C 键，将进入 SIGINT 信号的处理函数，即打印“Deal SIGINT”。这个例子演示了更改信号屏蔽字来屏蔽某个信号。

### 4. 使用 sigpending 检查是否有挂起的信号

系统调用 sigpending()用来检查进程是否有挂起的信号，也就是已经产生但被阻塞的信号。函数声明如下：

```
#include <signal.h>
int sigpending(sigset_t *set);
```

其中，信号集通过 set 参数返回。如果函数执行成功，返回 0，错误返回-1。

系统调用 sigpending 在上例实现过了，用法可以参考上例，这里不再赘述。

### 5. 使用 signal 设置信号处理程序

系统调用 signal()来为信号设置一个新的信号处理程序，可以将这个信号处理程序设置为一个用户指定的函数，或者设置为宏 SIG_ING 和 SIG_DFL。函数声明如下：

```
#include <signal. h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

参数 signum 是我们要处理的信号，指明了所要处理的信号类型，它可以取除了 SIGKILL 和 SIGSTOP 外的任何一种信号；参数 handler 描述了与信号关联的动作，它可以取以下 3 种值。

（1）SIG_ING

宏 SIG_ING 代表忽略信号，比如 signal(SIGINT,SIG_ING )表示忽略 SIGINT 信号，SIGINT 信号由 InterruptKey 产生，通常是用户按了 CTRL +C 键或者 DELETE 键产生。

（2）SIG_DFL

宏 SIG_DFL 表示恢复对信号的系统默认处理。比如 signal(SIGINT ,SIG_DFL );表示对信号 SIGINT 进行默认处理，即终止该进程。这种方式是否显式地写 signal 效果都一样。

（3）sighandler_t 类型的函数指针

此时，参数 handler 是 sighandler_t 类型的函数指针，指向一个我们自己定义的函数，用来响应对信号 signum 的处理。而且，这个自定义信号处理函数的参数是 signum。进程只要接收到类型为 signum 的信号，不管其正在执行程序的哪一部分，都立即执行 handler 函数。当 handler 函数执行结束后，控制权返回进程被中断的那一点继续执行。

如果函数执行成功，则返回该信号上一次的 handler 值，如果出错，则返回 SIG_ERR，此时可以通过错误码 errno 获得。

函数 signal 类似于 sigaction，不过两者是有区别的，首先 signal 是 ANSI C 标准的，而 sigaction 符合 P0SIX 标准。其次，signal 比 sigaction 使用简单，但要注意，如果在 C 语言中使用，并且 gcc 编译时加上-std=c99 时，signal 注册的信号在 sa_handler 被调用之前会把信号的 sa_handler 指针恢复，用 signal 函数注册的信号处理函数只会被调用一次，之后收到这个信号将按默认方式处理；如果编译时没有加上-std=c99，则 signal 注册的信号在处理信号时不会恢复 sa_handler 指针，下次依旧会使用 signal 定义的信号处理行为来处理。在 C++程序中，signal 注册的信号不会在 sa_handler 被调用之前把信号的 sa_handler 指针恢复。

而 sigaction 注册的信号在处理信号时不管编译时是否加上-std=c99，都不会恢复 sa_handler 指针，下次收到该信号时，依旧会根据 sigaction 注册的信号处理行为来处理。

### 【例 6.4】忽略 SIGINT 信号

（1）打开 UE，输入代码如下：

```
#include <stdio.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    signal(SIGINT, SIG_IGN);//忽略SIGINT信号
    while (1);
    return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
    ^C^C^C^C^C^C^C^C^C^C
```

可以看到，程序运行时，我们多次按下 Ctrl+C 键，但并没有使得程序退出，说明信号 SIGINT 被忽略了。

### 【例 6.5】自定义信号 SIGINT 的处理

（1）打开 UE，输入代码如下：

```
#include <stdio.h>
#include <signal.h>
typedef void (*signal_handler)(int);
void signal_handler_fun(int signum) {
   printf("catch signal %d\n", signum);
}
int main(int argc, char *argv[])
{
   signal(SIGINT, signal hander fun); //注册信号SIGINT的处理行为
   while(1);
   return 0;
}
```

（2）保存文件为 test.cpp，然后上传到 Linux 下，输入编译命令并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
^Ccatch signal 2
^Ccatch signal 2
^Ccatch signal 2
^Ccatch signal 2
```

可以看到，每次按下 Ctrl+C 键的时候，都会执行 signal_handler_fun 函数，而不是退出程序。

## 6.2 管道

### 6.2.1 管道的基本概念

所谓管道，是指用于连接读进程和写进程，以实现它们之间通信的共享文件，故又称管道文件。这种进程通信方式首创于 UNIX 系统，因它能传送大量的数据并且十分有效，很多操作系统都引入了这种通信方式，当然 Linux 也支持管道。

可以说管道是一种以先进先出的方式保存一定数量数据的特殊文件，而且管道一般是单向的。写进程将数据写入管道的一端，读进程从管道另一端读取数据，腾出空间以便写进程写入新的数据，所有的数据只能读取一次。Linux 下管道的大小有一定的限制。实际上，管道是一个固定大小的缓冲区。在 Linux 中，该缓冲区的大小为一个页面，即 4KB，因此管道的大小不像文件那样可以任意增长。

为了协调双方的通信，管道通信机制必须能够提供读写进程之间的同步机制。如果一个进程试图写入一个已满的管道，在默认情况下，系统会自动阻塞该进程，直到管道能够有空间接收数据。同样，如果试图读一个空的管道，进程也会被阻塞，直到管道有可读的数据。此外，如果一个进程以读方式打开一个管道，而没有另外的进程以写方式打开该管道，则同样会造成该进程阻塞（因为没有数据会写到这个管道里）。同样，当一个进程试图对没有读进程的管道进行写操作时，则会出现异常，导致进程终止。

管道是一个进程连接数据流到另一个进程的通道，它通常用作把一个进程的输出通过管道连

接到另一个进程的输入。在 Shell 下，可以通过符号“|”来使用管道。比如，当前路径下有一个文件夹 perl5，在 Shell 中输入命令：ls -l | grep per，我们知道 ls 命令（其实也是一个进程）会把当前目录中的文件或文件夹都列出来，现在把本来要输出到屏幕上的数据通过管道输出到 grep 进程中，作为 grep 进程的输入，然后 grep 进程对输入的信息进行筛选，把存在 per 字符串的那行（ls -l 是一行一行的字符串）打印在屏幕上：

```
[root@localhost ~]# ls -l|grep per
drwxr-xr-x. 2 root root    6 12月 17 2016 perl5
```

### 6.2.2 管道读写的特点

管道读写是通过标准的无缓冲的输入输出系统调用 read()和 write()实现的。系统调用 read ()将从一个由管道文件描述符所指的管道中读取指定的字节数到缓冲区中。如果调用成功，函数将返回实际所读的字节数。如果失败，将返回-1。当然由于管道的特殊性，管道的读取有其自身的特点：

（1）所有的读取操作总是从管道当前位置开始读，不支持文件指针的移动。

（2）如果管道没有被其他进程以写方式打开，那么 read()系统调用将返回 0，也就是遇到文件末端的条件。

（3）如果管道中没有数据，也就是管道为空，默认情况下 read()系统调用将会阻塞，直到有数据被写进该管道或者该管道被关闭。当然，也可以通过 fcuntl()系统调用对管道进行设置，如在管道为空的情况下让 read()系统调用立即返回。

数据通过系统调用 write()写入管道。write()系统调用将数据从缓冲区向管道文件描述符所指的管道中写入数据。如果该系统调用成功，将返回实际所写的字节数，否则返回-1。当然，由于管道的特殊性，管道的写操作也有其自身的特点：

（1）每一次的写请求操作总是附加在管道的末端。

（2）当有多个对同一管道的写请求发生时，系统保证小于或等于 4KB 大小的写请求操作不会交叉进行。

（3）如果试图对一个没有被任何进程以读方式打开的管道进行写操作，则将会产生 SIGPIPE 信号。默认情况下（假如 SIGP1PE 信号没有被捕获），该进程将会被系统终止。

（4）默认情况下，对管道的写操作请求会导致进程阻塞，因为如果设备处于忙状态，write()系统调用会被阻塞并且将被延迟写入，当然也可以通过 fcntl()系统调用对管道进行设置。

### 6.2.3 管道的局限性

管道有下列几个局限性：

（1）数据自己读却不能自己写。

（2）数据一旦被读走，便不在管道中存在，不可反复读取。

（3）由于管道采用半双工通信方式，因此数据只能在一个方向上流动。

（4）只能在有公共祖先的进程间使用管道。

### 6.2.4 创建管道函数 pipe

管道是一种基本的 IPC 机制，由 pipe 函数创建，该函数如下：

```
int pipe(int filedes[2]);
```

其中，参数 filedes 表示两个文件描述符，filedes[0]指向管道的读端，filedes[1]指向管道的写端。如果函数调用成功返回 0，调用失败返回-1。

调用 pipe 函数时，在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端和一个写端，然后通过 filedes 参数传出给用户程序两个文件描述符，filedes[0]指向管道的读端，filedes[1]指向管道的写端（很好记，就像 0 是标准输入，1 是标准输出一样）。所以管道在用户程序中看起来就像一个打开的文件，通过 read(filedes[0]);或者 write(filedes[1]);向这个文件读写数据其实是在读写内核缓冲区。

值得注意的是，管道创建时默认打开了文件描述符，且默认是以阻塞（block）模式打开的。

### 6.2.5 读写管道函数 read/write

读写管道的函数和读写文件的函数一样，都是 read 和 write。这两个函数我们在第 4 章讲过了，这里不再赘述。但有两点要注意：

（1）当没有数据可读时，read 调用阻塞，即进程暂停执行，一直等到有数据来到为止。

（2）当管道满的时候，write 调用阻塞，直到有进程读取数据。

### 6.2.6 等待子进程中断或结束的函数 wait

wait 函数用于等待子进程中断或结束，进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析当前进程的某个子进程是否已经退出，如果让它找到了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，函数声明如下：

```
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait (int * status);
```

子进程的结束状态值会由参数 status 返回，如果不在意结束状态值，则参数 status 可以设成 NULL。如果执行成功，则返回子进程识别码（PID），如果有错误发生，则返回-1。失败原因存于 errno 中。

管道创建成功以后，创建该管道的进程（父进程）同时掌握着管道的读端和写端。下面来看一个例子，实现父子进程间的通信，这是一个非常典型的通过管道的进程通信的例子。通常可以采用如图 6-1 所示的步骤。

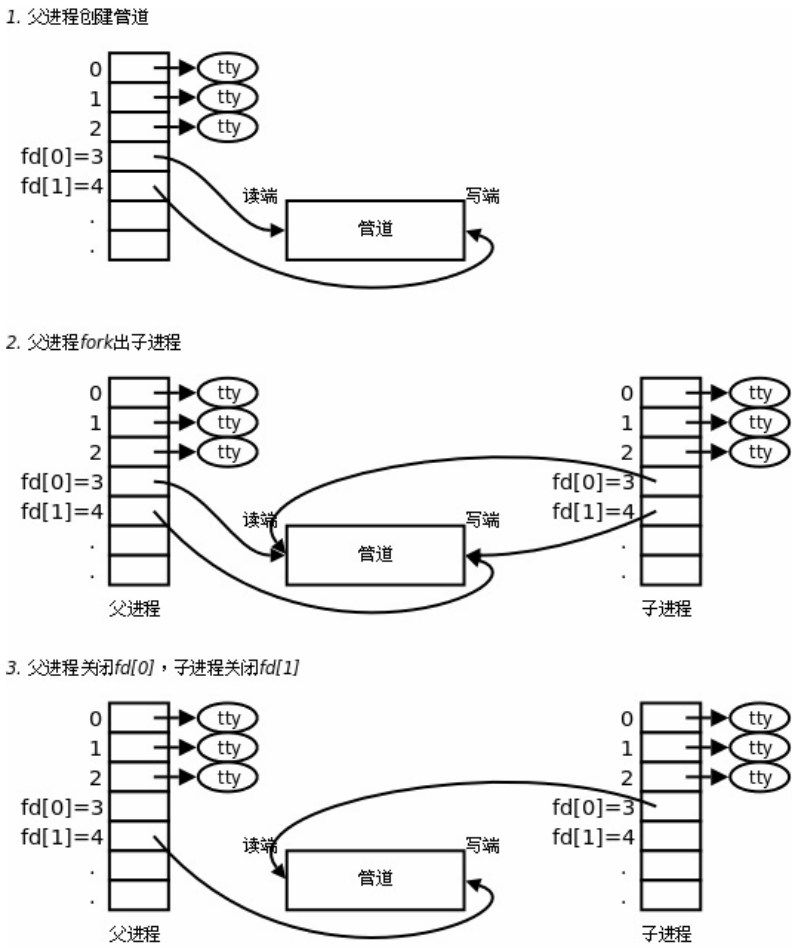


图 6-1

第一步，父进程调用 pipe 函数创建管道，得到两个文件描述符 fd[0]、fd[1]，分别指向管道的读端和写端。

第二步，父进程调用 fork 创建子进程，那么子进程也有两个文件描述符指向同一管道。

第三步，父进程关闭管道读端，子进程关闭管道写端。父进程可以向管道中写入数据，子进程将管道中的数据读出。由于管道是利用环形队列实现的，因此数据从写端流入管道，从读端流出，这样就实现了进程间的通信。

下面我们来看一个实例。

### 【例 6.6】父子进程使用管道通信

（1）打开 UE，输入代码如下：

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

```
void sys_err(const char *str)
{
    perror(str);
    exit(1);
}

int main(void)
{
    pid t pid;
    char buf[1024];
    int fd[2];
    char *p = "test for pipe\n";

    if (pipe(fd) == -1)   //创建管道
        sys err("pipe");

    pid = fork();  //创建子进程
    if (pid < 0) {
        sys_err("fork err");
    }
    else if (pid == 0) {
        close(fd[1]);  //关闭写描述符
        printf("child process wait to read:\n");
        int len = read(fd[0], buf, sizeof(buf));  //等待管道上的数据
        write(STDOUT_FILENO, buf, len);
        close(fd[0]);
    }
    else {
        close(fd[0]);   //关闭读描述符
        write(fd[1], p, strlen(p));  //向管道写入字符串数据
        wait(NULL);
        close(fd[1]);
    }

    return 0;
}
```

在代码中，首先创建一个管道，得到 fd[0]和 fd[1]两个读写描述符，然后用 fork 函数创建一个子进程，在子进程中，先关闭写描述符，然后开始读管道上的数据，而父进程中创建了子进程后，就关闭读描述符，并向管道写入字符串 p。

（2）保存代码为 test.cpp，然后上传到 Linux，在命令行下编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
child process wait to read:
test for pipe
```

【例 6.7】read 阻塞 10 秒后读数据

（1）打开 UE，输入代码如下：

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

int main(void)
{
    int fds[2];
    if (pipe(fds) == -1) {
        perror("pipe error");
        exit(EXIT_FAILURE);
    }
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        perror("fork error");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        close(fds[0]);//子进程关闭读端
        sleep(10);//睡眠10秒
        write(fds[1], "hello", 5);// 子进程写数据给管道
        exit(EXIT_SUCCESS);
    }

    close(fds[1]);//父进程关闭写端
    char buf[10] = { 0 };
    read(fds[0], buf, 10); //等待读数据
    printf("receive datas = %s\n", buf);
    return 0;
}
```

在代码中，我们让子进程先睡眠 10 秒，父进程因为没有数据从管道中读出，被阻塞了，直到子进程睡眠结束，向管道中写入数据后，父进程才读到数据。

（2）保存代码为 test.cpp，然后上传到 Linux，在命令行下编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
receive datas = hello
```

可以看到，运行 test 后，稍等 10 秒，父进程就会读到数据了，可见 read 在管道中没有数据可读的时候，的确阻塞了调用进程（这里是父进程）。

### 6.2.7 使用管道的特殊情况

使用管道需要注意以下 4 种特殊情况（假设都是阻塞 I/O 操作，没有设置 O_NONBLOCK 标志）：

（1）如果所有指向管道写端的文件描述符都关闭了（管道写端引用计数为 0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次 read 会返回 0，就像读到文件末尾一样。

（2）如果有指向管道写端的文件描述符没关闭（管道写端引用计数大于 0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次 read 会阻塞，直到管道中有数据可读了才读取数据并返回。

（3）如果所有指向管道读端的文件描述符都关闭了（管道读端引用计数为 0），这时有进程向管道的写端 write，那么该进程会收到信号 SIGPIPE，通常会导致进程异常终止。当然，也可以对 SIGPIPE 信号实施捕捉，不终止进程。

（4）如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于 0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时，再次 write 会阻塞，直到管道中有空位置了才写入数据并返回。

## 6.3 消息队列

现在我们来讨论另一种常用的进程间通信方式：消息队列。从许多方面来看，消息队列类似于有名管道，但是却没有与打开和关闭管道的复杂关联。然而，使用消息队列并没有解决我们使用有名管道所遇到的问题，例如管道上的阻塞。

消息队列提供了一种在两个不相关的进程之间传递数据的简单高效的方法。与有名管道比较起来，消息队列的优点在独立于发送与接收进程，这减少了在打开与关闭有名管道之间同步的困难。

消息队列提供了一种由一个进程向另一个进程发送块数据的方法。另外，每一个数据块被看作有一个类型，而接收进程可以独立接收具有不同类型的数据块。消息队列的好处在于我们几乎可以完全避免同步问题，并且可以通过发送消息屏蔽有名管道的问题。更好的是，我们可以使用某些紧急方式发送消息。坏处在于，与管道类似，在每一个数据块上有一个最大尺寸限制，同时在系统中所有消息队列的块尺寸上也有一个最大尺寸限制。

尽管有这些限制，但是 Linux 并没有定义这些限制的具体值，除了指出超过这些尺寸是某些消息队列功能失败的原因。Linux 系统有两个定义：MSGMAX 与 MSGMNB，分别用于定义单个消息与一个队列的最大尺寸。这些宏定义在其他系统上也许并不相同，甚至也许就不存在。

Linux 提供了一组消息队列函数让我们使用消息队列，消息队列函数定义如下：

```
    #include <sys/msg.h>
    int msgctl(int msqid, int cmd, struct msqid_ds *buf);
    int msgget(key_t key, int msgflg);
    int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int
msgflg);
```

```
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

与信息号和共享内存一样，头文件 sys/types.h 与 sys/ipc.h 通常也是需要的。函数不多，下面来一个一个了解。

### 6.3.1 创建和打开消息队列函数 msgget

函数 msgget 用于得到一个已存在的消息队列标识符或创建一个消息队列对象，声明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

其中，参数 key 表示消息队列的键值（有点类似数据库表中的键值概念），用于标识一个消息队列，函数将它与已有的消息队列对象的关键字进行比较，以此来判断消息队列对象是否已经创建，如果取宏 IPC_PRIVATE（数值为 0）表示创建一个私有队列，这在理论上只可以被当前进程所访问，key_t 是一个 32 位整型；参数 msgflg 表示创建或访问消息队列的具体方式，通常取值如下。

IPC_CREAT：如果消息队列对象不存在，则创建消息队列对象，否则进行打开操作。要创建一个新的消息队列，IPC_CREAT 特殊位必须与其他的权限位进行或操作。如果消息队列已经存在，IPC_CREAT 标记只是简单地被忽略。

IPC_EXCL：和 IPC_CREAT 一起使用（用“|”连接），如果消息对象不存在，则创建之，否则产生一个错误并返回。

如果函数执行成功，则返回一个正数作为消息队列标识符，执行错误返回-1，错误原因存于 error 中。一些常见的错误码如下。

- EACCES：指定的消息队列已存在，但调用进程没有权限访问它。
- EEXIST： key 指定的消息队列已存在，而 msgflg 中同时指定 IPC_CREAT 和 IPC_EXCL 标志。
- ENOENT：key 指定的消息队列不存在，同时 msgflg 中没有指定 IPC_CREAT 标志。
- ENOMEM：需要建立消息队列，但内存不足。
- ENOSPC：需要建立消息队列，但已达到系统的限制。

大家可以想一下，为什么需要键值 key？这是因为是进程间的通信，所以必须有一个公共的标识来确保使用同一个通信通道（比如这个通信通道就是消息队列），再把这个标识与某个消息队列进行绑定，任何一个进程如果使用同一个标识，内核就可以通过该标识找到对应的那个队列，这个标识就是键值。如果没有键值，进程 A 打开或者创建一个队列并返回这个队列的描述符，但其他进程不知道这个队列的描述符是什么，因此就不能通信了。

### 6.3.2 获取和设置消息队列的属性函数 msgctl

函数 msgctl 用于获取和设置消息队列的属性。该函数声明如下：

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

```
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

其中，参数 msqid 是消息队列标识符；cmd 表示要对消息队列进行的操作，它的取值可以是：

- IPC_STAT：读取消息队列的 msqid_ds 数据，并将其存储在 buf 指定的地址中。IPC_SET：设置消息队列的属性，要设置的属性需先存储在 buf 中，可设置的属性包括：msg_perm.uid、msg_perm.gid、msg_perm.mode 以及 msg_qbytes。
- IPC_EMID：将队列从系统内核中删除。

参数 buf 指向消息队列管理结构体 msqid_ds，该结构体定义如下：

```
/* Obsolete, used only for backwards compatibility and libc5 compiles */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;     /* first message on queue,unused  */
    struct msg *msg_last;      /* last message in queue,unused */
    __kernel_time_t msg_stime; /* last msgsnd time */
    __kernel_time_t msg_rtime; /* last msgrcv time */
    __kernel_time_t msg_ctime; /* last change time */
    unsigned long  msg_lcbytes; /* Reuse junk fields for 32 bit */
    unsigned long  msg_lqbytes; /* ditto */
    unsigned short msg_cbytes; /* current number of bytes on queue */
    unsigned short msg_qnum;   /* number of messages in queue */
    unsigned short msg_qbytes; /* max number of bytes on queue */
    __kernel_ipc_pid_t msg_lspid;  /* pid of last msgsnd */
    __kernel_ipc_pid_t msg_lrpid;  /* last receive pid */
};
```

如果函数执行成功就返回 0，失败返回-1，错误原因可以通过错误码 errno 获得，常见错误码如下。

- EACCESS：参数 cmd 为 IPC_STAT，无权限读取该消息队列。
- EFAULT：参数 buf 指向无效的内存地址。
- EIDRM：标识符为 msqid 的消息队列已被删除。
- EINVAL：无效的参数 cmd 或 msqid。
- EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。

### 6.3.3 将消息送入消息队列的函数 msgsnd

msgsnd 函数用来将消息送入消息队列。该函数声明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

其中，参数 msqid 是消息队列对象的标识符（由 msgget 函数得到），第二个参数 msgp 指向消息缓冲区的指针，该缓冲区用来暂时存储要发送的消息，通常可用一个通用结构来表示消息：

```
struct msgbuf {
   long mtype;     /* 消息类型，必须大于0*/
   char mtext[1];  /*消息数据*/
};
```

第三个参数 msgsz 是要发送信息的长度（字节数），可以用以下公式计算：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个参数 msgflg 是控制函数行为的标志，可以取以下的值。

- 0：表示阻塞方式，线程将被阻塞直到消息可以被写入。
- IPC_NOWAIT：表示非阻塞方式，如果消息队列已满或其他情况无法送入消息，函数立即返回。

如果函数执行成功就返回 0，失败返回-1，errno 被设为以下某个值。

- EACCES：调用进程在消息队列上没有写权限，同时没有 CAP_IPC_OWNER 权限。
- EAGAIN：由于消息队列的 msg_qbytes 限制和 msgflg 中指定 IPC_NOWAIT 标志，因此消息不能被发送。
- EFAULT：msgp 指针指向的内存空间不可访问。
- EIDRM：消息队列已被删除。
- EINTR：等待消息队列空间可用时被信号中断。
- EINVAL：参数无效。
- ENOMEM：系统内存不足，无法将 msgp 指向的消息复制进来。

### 6.3.4 从消息队列中读取一条新消息的函数 msgrcv

函数 msgrcv 用于从消息队列中读出一条新消息。该函数声明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);
```

其中，参数 msqid 表示消息队列的标识符，msgp 指向要读出消息的缓冲区。通常消息缓冲区结构为：

```
struct msgbuf {
   long mtype;     /* 消息类型，必须大于0*/
   char mtext[1];  /* 消息数据 */
};
```

msgsz 表示消息数据的长度，msgtyp 表示从消息队列内读取的消息形态。如果值为零，则表

示消息队列中的所有消息都会被读取。参数 msgflg 是控制函数行为的标志，可以取以下的值。

（1）0：表示阻塞方式，当消息队列为空时，一直等待。

（2）IPC_NOWAIT：表示非阻塞方式，消息队列为空时，不等待，马上返回-1，并设定错误码为 ENOMSG。如果函数执行成功，msgrcv 返回复制到 mtext 数组的实际字节数，若失败则返回-1，errno 被设为以下的某个值。

- E2BIG：消息文本长度大于 msgsz，并且 msgflg 中没有指定。
- G_NOERROREACCES：调用进程没有读权能，同时没有 CAP_IPC_OWNER 权能。
- EAGAIN：消息队列为空，并且 msgflg 中没有指定 IPC_NOWAIT。
- EFAULT：msgp 指向的空间不可访问。
- EIDRM：当进程睡眠等待接收消息时，消息已被删除。
- EINTR：当进程睡眠等待接收消息时，被信号中断。
- EINVAL：参数无效。
- ENOMSG：msgflg 中指定了 IPC_NOWAIT，同时所请求类型的消息不存在。

### 6.3.5 生成键值函数 ftok

系统建立 IPC 通信（如消息队列、共享内存时）必须指定一个键值。通常情况下，该键值通过 ftok 函数得到。该函数声明如下：

```
key_t ftok( char * fname, int id );
```

其中，参数 fname 是指定的文件名，这个文件必须是存在的而且可以访问的。id 是子序号，它是一个 8bit 的整数，即范围是 0~255，可以根据自己的约定随意设置，没有什么限制条件。若函数执行成功，则会返回 key_t 键值，否则返回-1。在一般的 UNIX 中，通常是将文件的索引节点取出，然后在前面加上子序号就得到 key_t 的值。

ftok 根据路径名提取文件信息，再根据这些文件信息及参数 id 合成 key，该路径可以随便设置，该路径是必须存在的，ftok 只是根据文件 inode 在系统内的唯一性来取一个数值，和文件的权限无关。

在使用 ftok()函数时，里面有两个参数，即 fname 和 id，fname 为指定的文件名，而 id 为子序列号，这个函数的返回值就是 key，它与指定的文件的索引节点号和子序列号 id 有关，这样就会给我们一个误解，即只要文件的路径、名称和子序列号不变，那么得到的 key 值永远就不会变。

事实上，这种认识是错误的，想一下，假如存在这样一种情况：在访问同一共享内存的多个进程先后调用 ftok()时间段中，如果 fname 指向的文件或者目录被删除而且又重新创建，那么文件系统会赋予这个同名文件新的 i 节点信息，于是这些进程调用的 ftok()都能正常返回，但键值 key 却不一定相同了。由此可能造成的后果是，原本这些进程意图访问一个相同的共享内存对象，然而由于它们各自得到的键值不同，实际上进程指向的共享内存不再一致，如果这些共享内存都得到创建，则在整个应用运行的过程中表面上不会报出任何错误，然而通过一个共享内存对象进行数据传输的目的将无法实现。这是一个很重要的坑，笔者当年因为此问题而苦不堪言，希望大家谨记。

所以要确保 key 值不变，要么确保 ftok()的文件不被删除，要么不用 ftok()，指定一个固定的 key 值。

【例 6.8】生成一个键值

（1）打开 UE，输入代码如下：

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
int main()
{
    key_t semkey;
    if((semkey = ftok("./test", 123))<0)
    {
        printf("ftok failed\n");
        exit(EXIT_FAILURE);
    }
    printf("ftok ok ,semkey = %d\n", semkey);
    return 0;
}
```

代码很简单。用当前路径下的 test 程序文件和 123 一起生成一个键值，最后打印出来。

（2）保存代码为 test.cpp，然后上传到 Linux，编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
ftok ok ,semkey = 2063635014
```

【例 6.9】解开 ftok 产生键值的内幕

（1）打开 UE，输入代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/sem.h>
int main()
{
    char   filename[50];
    struct stat    buf;
    int    ret;
    strcpy( filename, "./test" );
    ret = stat( filename, &buf );
    if( ret )
    {
        printf( "stat error\n" );
        return -1;
    }

    printf( "the file info: ftok( filename, 0x27 ) = %x, st_ino = %x, st_dev=
%x\n", ftok( filename, 0x27 ), buf.st_ino, buf.st_dev );
```

```
    return 0;
}
```

代码中，我们首先生成一个键值，然后用获取文件信息的函数 stat 来获取 test 程序的文件信息，最后打印出键值和文件 tet 的相关属性。

（2）保存代码为 test.cpp，然后上传到 Linux，编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
the file info: ftok( filename, 0x27 ) = 27009246, st_ino = d359246, st_dev=
fd00
```

通过执行结果可以看出，ftok 获取的键值是由 ftok 函数的第二个参数的后 8 个位、st_dev 的后 8 位、st_ino 的后 16 位构成的。注意，它们都是 16 进制的。

其中，st_dev 是文件的设备编号，st_ino 是文件的节点信息，它们都定义在结构体 stat 中：

```
struct stat {
    unsigned long  st_dev;//文件的设备编号
    unsigned long  st_ino;//节点
    unsigned short st_mode; //文件的类型和存取的权限
    unsigned short st_nlink;//连到该文件的硬链接数目，刚建立的文件值为1
    unsigned short st_uid; //用户ID
    unsigned short st_gid; //组ID
    unsigned long st_rdev;
    unsigned long  st_size;
    unsigned long st_blksize;
    unsigned long  st_blocks;
    unsigned long  st_atime;
    unsigned long st_atime_nsec;
    unsigned long  st_mtime;
    unsigned long st_mtime_nsec;
    unsigned long  st_ctime;
    unsigned long st_ctime_nsec;
    unsigned long  __unused4;
    unsigned long  __unused5;
};
```

这个结构体 stat 和函数 stat 都在第 4 章讲述过了，这里不再赘述。

前面我们已经了解了消息队列的定义，下面看一下是如何实际工作的。我们将会编写两个程序：test 用来接收，send 用来发送。我们会允许任意一个程序创建消息队列，但是接收者在接收到最后一条消息后删除消息队列。

### 【例 6.10】消息队列的发送和接收

（1）首先创建接收程序，打开 UE，输入代码如下：

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

```
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msg_st
{
    long int my msg type;
    char some text[BUFSIZ];
};
int main()
{
    int running = 1;
    int msgid;
    struct my_msg_st some_data;
    long int msg_to_receive = 0;//读取消息队列中的全部消息

         //创建消息队列
    msgid = msgget((key_t)1234,0666|IPC_CREAT);
    if(msgid == -1)
    {
        fprintf(stderr,"msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }
   //接收消息队列中的消息直到遇到一个end消息。最后，消息队列被删除
    while(running)
    {   //阻塞方式等待接收消息
        if(msgrcv(msgid, (void *)&some_data, BUFSIZ, msg_to_receive, 0) == -1)
        {
            fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
            exit(EXIT_FAILURE);
        }


        printf("You wrote: %s", some_data.some_text);
        if(strncmp(some_data.some_text,"end",3)==0)//如果收到的是end，就退出循环
        {
            running = 0;
        }
    }

    if(msgctl(msgid, IPC_RMID, 0)==-1)//删除消息队列
    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
```

```
}
```

代码很简单，接收者使用 msgget 来获得消息队列标识符，并且等待接收消息，直到接收到特殊消息 end。然后它会使用 msgctl 删除消息队列进行一些清理工作。

（2）保存代码为 test.cpp，然后上传到 Linux，编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test
```

此时接收程序就处于等待接收消息的状态了。下面我们创建消息发送程序。打开 UE，输入代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#includ <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_TEXT 512
struct my_msg_st
{
    long int my_msg_type;
    char some_text[MAX_TEXT];
};

int main()
{
    int running = 1;
    struct my_msg_st some_data;
    int msgid;
    char buffer[BUFSIZ];

    msgid = msgget((key_t)1234, 0666|IPC_CREAT);//用一个整数作为键值
    if(msgid==-1)
    {
        fprintf(stderr,"msgget failed with errno: %d\n", errno);
        exit(EXIT_FAILURE);
    }
     while(running)
    {
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        some_data.my_msg_type = 1;
        strcpy(some_data.some_text, buffer);
        if(msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0)==-1)
        {
```

```
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }
        if(strncmp(buffer, "end", 3) == 0)
        {
            running = 0;
        }
    }
     exit(EXIT SUCCESS);
}
```

发送者程序使用 msgget 创建一个消息队列，然后使用 msgsnd 函数向队列中添加消息。

与管道的程序不同，消息队列的程序并没有必要提供自己的同步机制。这是消息队列比起管道的一个巨大优点。

（3）保存代码为 send.cpp，上传到 Linux，另开一个终端窗口，在命令行下编译并运行：

```
[root@localhost test]# ./send
Enter some text: abc
Enter some text: zww book
Enter some text: end
```

我们发了 3 条消息，最后一条是 end。此时接收端变为：

```
[root@localhost test]# ./test
You wrote: abc
You wrote: zww book
You wrote: end
[root@localhost test]#
```

3 条消息都接收到了，并且收到最后一条程序就退出了，符合预期。

### 【例 6.11】获取消息队列的属性

（1）打开 UE，输入代码如下：

```
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>
#include <ctime>
#include "stdio.h"
#include "errno.h"
void msg_stat(int,struct msqid_ds );
main()
{
    int gflags,sflags,rflags;
    key_t key;
    int msgid;
    int reval;
    struct msgsbuf{
```

```
            int mtype;
            char mtext[1];
        }msg_sbuf;
    struct msgmbuf
        {
        int mtype;
        char mtext[10];
        }msg_rbuf;
    struct msqid ds msg ginfo,msg sinfo;
    char  msgpath[]="./test";

    key=ftok(msgpath,'b');
    gflags=IPC CREAT|IPC EXCL;
    msgid=msgget(key,gflags|00666);
    if(msgid==-1)
    {
        printf("msg create error\n");
    }
    //创建一个消息队列后，输出消息队列默认属性
    msg_stat(msgid,msg_ginfo);
    sflags=IPC_NOWAIT;
    msg_sbuf.mtype=10;
    msg_sbuf.mtext[0]='a';
    reval=msgsnd(msgid,&msg_sbuf,sizeof(msg_sbuf.mtext),sflags);
    if(reval==-1)
    {
        printf("message send error\n");
    }
    //发送一个消息后，输出消息队列属性
    msg_stat(msgid,msg_ginfo);


    reval=msgctl(msgid,IPC_RMID,NULL);//删除消息队列
    if(reval==-1)
    {
        printf("unlink msg queue error\n");
    }
}
void msg_stat(int msgid,struct msqid_ds msg_info)
{
    int reval;
    sleep(1);//只是为了后面输出时间的方便
    reval=msgctl(msgid,IPC_STAT,&msg_info);
    if(reval==-1)
    {
        printf("get msg info error\n");
    }
    printf("\n");
    printf("current number of bytes on queue is %d\n",msg_info.msg_cbytes);
```

```
        printf("number of messages in queue is %d\n",msg_info.msg_qnum);
        printf("max number of bytes on queue is %d\n",msg_info.msg_qbytes);
        //每个消息队列的容量（字节数）都有限制MSGMNB，值的大小因系统而异。在创建新的消息队列
        //时，msg_qbytes的默认值就是MSGMNB
        printf("pid of last msgsnd is %d\n",msg_info.msg_lspid);
        printf("pid of last msgrcv is %d\n",msg_info.msg_lrpid);
        printf("last msgsnd time is %s", ctime(&(msg_info.msg_stime)));
        printf("last msgrcv time is %s", ctime(&(msg_info.msg_rtime)));
        printf("last change time is %s", ctime(&(msg info.msg ctime)));
        printf("msg uid is %d\n",msg info.msg perm.uid);
        printf("msg gid is %d\n",msg info.msg perm.gid);
    }
```

其中，函数 msg_stat 是一个自定义函数，用来打印消息队列的一些属性信息。我们先创建了一个消息队列，然后打印了其属性信息，接着发送了一个消息，又打印了其属性信息。

（2）保存代码为 test.cpp，然后上传到 Linux，编译并运行：

```
[root@localhost test]# g++ test.cpp -o test
[root@localhost test]# ./test

current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384
pid of last msgsnd is 0
pid of last msgrcv is 0
last msgsnd time is Thu Jan  1 08:00:00 1970
last msgrcv time is Thu Jan  1 08:00:00 1970
last change time is Fri Mar 30 14:55:17 2018
msg uid is 0
msg gid is 0

current number of bytes on queue is 1
number of messages in queue is 1
max number of bytes on queue is 16384
pid of last msgsnd is 90972
pid of last msgrcv is 0
last msgsnd time is Fri Mar 30 14:55:18 2018
last msgrcv time is Thu Jan  1 08:00:00 1970
last change time is Fri Mar 30 14:55:17 2018
msg uid is 0
msg gid is 0
```

可以看到，刚开始的时候，消息队列里的消息是 0，发送一个后，队列里面的消息个数就变成 1 了。因为我们发送的消息是字符“a”，长度是 1，所以消息队列的长度就是一个字节。