## 第7课 讲稿（进程间通信-3）

1. **信号**
2. 信号概述

信号是UNIX中所使用的进程通信的一种最古老的方法。它是在**软件层次上对中断机制的一种模拟**，是一种异步通信方式。信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。

**信号可以在任何时候发给某一进程，而无需知道该进程的状态**。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它为止；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

kill命令的“−l”选项可以列出该系统所支持的所有信号的列表，在Linux中，信号值在32 之前的有不同的名称，而信号值在 32 以后的都是用“ SIGRTMIN”或“ SIGRTMAX”开头的，这就是两类典型的信号。前者是从 UNIX 系统中继承下来的信号，为不可靠信号（也称为非实时信号）；后者是为了解决前面“不可靠信号”的问题而进行了更改和扩充的信号，称为“可靠信号”（也称为实时信号）。那么为什么之前的信号不可靠呢？这里首先要介绍一下信号的生命周期。

一个完整的信号生命周期由4 个重要事件来刻画：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数。

一个不可靠信号的处理过程是这样的：如果发现该信号已经在进程中注册，那么就忽略该信号。因此，若前一个信号还未注销又产生了相同的信号就会产生信号丢失。而当可靠信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此信号就不会丢失。

用户进程对信号的响应可以有 3 种方式。

* 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即SIGKILL和SIGSTOP。
* 捕捉信号，定义信号处理函数，当信号发生时，执行相应的自定义处理函数。
* 执行缺省操作， Linux 对每种信号都规定了默认操作。

Linux 中的大多数信号是提供给内核的，表6列出了 Linux 中最为常见信号的含义及其默认操作。

|  |  |  |
| --- | --- | --- |
| 表6 常见信号的含义及其默认操作 | | |
| 信号名 | 含义 | 默认操作 |
| SIGHUP | 该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，通知同一会话内的各个作业与控制终端不再关联 | 终止 |
| SIGINT | 该信号在用户键入INTR字符（通常是Ctrl-C）时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程 | 终止 |
| SIGQUIT | 该信号和SIGINT 类似，但由QUIT字符（通常是Ctrl-\）来控制，类似于一个程序错误信号 | 终止 |
| SIGILL | 该信号在一个进程企图执行一条非法指令时（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）发出 | 终止 |
| SIGFPE | 该信号在发生致命的算术运算错误时发出。这里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 | 终止 |
| SIGKILL | 该信号用来立即结束程序的运行，并且不能被阻塞、处理或忽略 | 终止 |
| SIGALRM | 该信号当一个定时器到时的时候发出 | 终止 |
| SIGSTOP | 该信号用于暂停一个进程，且不能被阻塞、处理或忽略 | 暂停进程 |
| SIGTSTP | 该信号用于交互停止进程，用户键入SUSP 字符时（通常是Ctrl+Z）发出这个信号，可以被阻塞、处理或忽略 | 停止进程 |
| SIGCHLD | 子进程改变状态时，父进程会收到这个信号 | 忽略 |
| SIGCONT | 恢复暂停的进程 | 恢复进程 |
| SIGABORT | 进程异常终止时发出 | 终止 |

1. 信号发送与捕捉

发送和捕捉信号的函数主要有 kill()、 raise()、 alarm()以及 pause()，下面就依次对其进行介绍。

1. kill()和raise()
2. 函数说明。

kill()函数kill系统命令一样，可以发送信号给进程或进程组（kill系统命令只是kill()函数的一个用户接口）。需要注意的是，它不仅可以中止进程（实际上发出SIGKILL信号），也可以向进程发送其他信号。

与kill()函数所不同的是，raise()函数仅允许进程向自身发送信号。

1. 函数格式。

表7列出了kill()函数的语法要点：

|  |  |  |
| --- | --- | --- |
| 表7 kill()函数语法要点 | | |
| 所需头文件 | #include <signal.h>  #include <sys/types.h> | |
| 函数原型 | int kill(pid\_t pid, int sig) | |
|  | pid | 正数：要发送信号的进程号 |
| 0：信号被发送到所有和当前进程在同一个进程组的进程 |
| -1：信号发给所有的进程表中的进程 |
| <-1：信号发送给进程组号为-pid的每一个进程 |
| sig：信号 | |
| 返回值 | 成功：0 | |
| 出错：-1 | |

表8列出了raise()函数的语法要点。

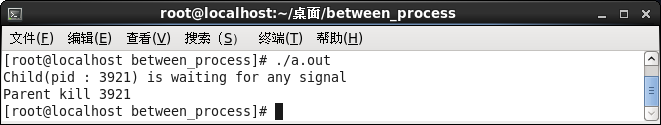
|  |  |
| --- | --- |
| 表8 raise()函数语法要点 | |
| 所需头文件 | #include <signal.h>  #include <sys/types.h> |
| 函数原型 | int raise(int sig) |
| 函数传入值 | sig：信号 |
| 返回值 | 成功：0 |
| 出错：-1 |

1. 函数实例。

下面这个示例首先使用fork()创建了一个子进程，接着为了保证子进程不在父进程调用 kill()之前退出，在子进程中使用raise()函数向自身发送SIGSTOP信号，使子进程暂停。接下来再在父进程中调用kill()向子进程发送信号，在该示例中使用的是SIGKILL，有兴趣的同学可以使用其他信号进行练习。

|  |
| --- |
| /\* kill\_raise.c \*/  #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  int main()  {  pid\_t pid;  int ret;  /\* 创建一子进程 \*/  if ((pid = fork()) < 0)  {  printf("Fork error\n");  exit(1);  }  if (pid == 0)  {  /\* 在子进程中使用 raise() 函数发出 SIGSTOP 信号, 使子进程暂停 \*/  printf("Child(pid : %d) is waiting for any signal\n", getpid());  raise(SIGSTOP);  exit(0);  }  else  {  /\* 在父进程中收集子进程发出的信号，并调用 kill() 函数进行相应的操作 \*/  sleep(1);  if ((waitpid(pid, NULL, WNOHANG) ) == 0)  {  if ((ret = kill(pid, SIGKILL) ) == 0)  {  printf("Parent killed %d\n",pid);  }  }  exit(0);  }  } |

该程序运行结果如下所示：



1. alarm()和pause()
2. 函数说明。

alarm()也称为闹钟函数，它可以在进程中设置一个定时器，当定时器指定的时间到时，它就向进程发送SIGALRM信号。要注意的是，**一个进程只能有一个闹钟时间**，如果在调用 alarm()之前设置过闹钟时间，则任何以前的闹钟时间都会被新值所代替。

pause()函数是用于将调用进程挂起直至捕捉到信号为止。这个函数很常用，通常可以用于判断信号是否已到。

1. 函数格式。

表9列出了 alarm()函数的语法要点。

|  |  |
| --- | --- |
| 表9 alarm()函数语法要点 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | unsigned int alarm(unsigned int seconds) |
| 函数传入值 | seconds：指定秒数，系统经过seconds 秒之后向该进程发送SIGALRM信号 |
| 返回值 | 成功：如果调用此alarm()前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0 |
| 出错：-1 |

表10列出了 pause()函数的语法要点。

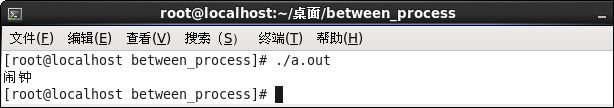
|  |  |
| --- | --- |
| 表10 pause()函数语法要点 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | int pause(void) |
| 函数返回值 | -1，并且把errno值设为EINTR |

1. 函数实例。

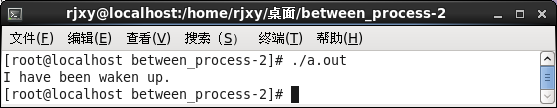
该实例实际上已完成了一个简单的sleep()函数的功能，由于SIGALRM默认的系统动作为终止该进程，因此程序在打印信息之前，就会被结束了。代码如下所示：

|  |
| --- |
| /\* alarm\_pause.c \*/  #include <stdio.h>  #include <stdlib.h>  int main()  {  /\*调用 alarm 定时器函数\*/  int ret = alarm(5) ;  pause();  printf("I have been waken up.\n"); /\* 此语句不会被执行 \*/  } |

执行结果如下：



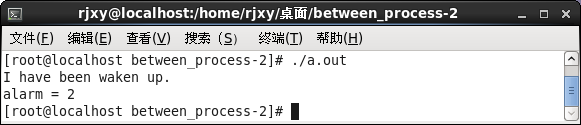
【测试1】去掉“pause()”语句，则程序执行完printf语句后还未达到闹钟设置的时间5s，程序将直接退出。执行结果如下：



【测试2】程序中做如下修改

|  |
| --- |
| /\* alarm\_pause\_mod.c \*/  #include <stdio.h>  #include <stdlib.h>  int main()  {  /\*调用 alarm 定时器函数\*/  int ret = alarm(5) ;  printf("I have been waken up.\n");  sleep(3);  ret = alarm(15);  printf("alarm = %d\n",ret);  } |

因为执行完sleep(3)后，还未到达闹钟设置的时间5s，这时又调用sleep(15)设置新的闹钟，此时alarm的返回值是上次调用alarm的剩余时间，即2s，所以printf输出的alarm值是2。程序执行完printf后，还未达到新设置的闹钟时间15s，程序将直接退出，执行结果如下：



1. 信号的处理

在了解了信号的产生与捕获之后，接下来就要对信号进行具体的操作了。特定的信号是与一定的进程相联系的，也就是说，一个进程可以决定在该进程中需要对哪些信号进行什么样的处理。例如，一个进程可以选择忽略某些信号而只处理其他一些信号，另外，一个进程还可以选择如何处理信号。因此，首先就要建立**进程与其信号之间的对应关系，这就是信号的处理**。

信号处理的主要方法有两种，一种是使用简单的signal()函数或sigaction()函数，另一种是使用信号集函数组。下面分别介绍这两种处理方式。

1. 信号处理函数
2. 函数说明。

使用signal()函数处理时，只需要指出要处理的信号和处理函数即可。它主要是用于前 32 种非实时信号的处理，不支持信号传递信息，但是由于使用简单、易于理解，因此也受到很多程序员的欢迎。

Linux还支持一个更健壮、更新的信号处理函数sigaction()，推荐使用该函数。

1. 函数格式。

signal()函数的语法要点如表11所示。

|  |  |  |
| --- | --- | --- |
| 表11 signal()函数语法要点 | | |
| 所需头文件 | #include <signal.h> | |
| 函数原型 | void (\*signal(int signum, void (\*func)(int)))(int) | |
| 函数传入值 | signum：指定信号代码，signal一旦获取到signum指定的信号，就开始执行func所指定的函数。 | |
| mode： | SIG\_IGN：忽略该信号 |
| SIG\_DFL：采用系统默认方式处理信号 |
| 自定义的信号处理函数指针，此函数必须在signal()被调用前申明，func就是这个函数的名字。当接收到一个类型为sig的信号时，就执行func所指定的函数。这个函数应有如下形式的定义：  void func(int sig); |
| 返回值 | 成功：以前的信号处理配置 | |
| 出错：-1 | |

这个函数原型“void (\*signal(int signum, void (\*func)(int)))(int)”很复杂，这里不占用太多时间说明，简单说明其使用方式：

* 最后一个“int” 是获取的信号，由进程自动获取；
* signal函数的第一个参数signum是一个整数，是指定的信号代码（在signal.h头文件中引用的“bits/signum.h”中定义，如信号SIGINT的代码是2），signal函数把由进程获取到的信号和signum对比，如果一致，再将该信号传递给signal的第二个参数，即函数指针func，这样func函数所使用的参数（倒数第二个“int”）其实就是由进程获取到的信号（最后一个“int”）；
* 以获取到的信号作为参数执行func函数；

表12列举了sigaction()的语法要点。

|  |  |
| --- | --- |
| 表12 sigaction()函数语法要点 | |
| 所需头文件 | #include <signal.h> |
| 函数原型 | int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact) |
| 函数传入值 | signum：信号代码，可以是除SIGKILL及SIGSTOP外的任何一个有效的信号 |
| act：指向结构sigaction的一个实例的指针，**指定对特定信号的处理** |
| oldact：保存原来对相应信号的处理 |
| 返回值 | 成功：0 |
| 出错：-1 |

这里要说明的是sigaction()函数中第2个和第3个参数用到的sigaction结构，这是一个看似非常复杂的结构。

首先给出了sigaction的定义，如下所示：

|  |
| --- |
| struct sigaction  {  void (\*sa\_handler)(int signo);  sigset\_t sa\_mask;  int sa\_flags;  void (\*sa\_restore)(void);  } |

sa\_handler是一个函数指针，指定信号处理函数，这里除可以是用户自定义的处理函数外，还可以为SIG\_DFL（采用缺省的处理方式）或 SIG\_IGN（忽略信号）。它的处理函数只有一个参数，即信号值。

sa\_mask是一个信号集，它可以指定在信号处理程序执行过程中哪些信号应当被屏蔽。在调用信号捕获函数之前，该信号集要加入到信号的信号屏蔽字中。如果没有需要屏蔽的信号，需要用sigemptyset函数将sa\_mask初始化（即清空）。

sa\_flags中包含了许多标志位，是对信号进行处理的各个选择项。它的常见可选值如表13所示。

sa\_restore参数未使用。

|  |  |
| --- | --- |
| 表13 常见标志位的含义及其默认操作 | |
| 信号名 | 含义 |
| SA\_NOCLDSTOP | 进程忽略子进程产生的任何SIGSTOP、SIGTSTP、SIGTTIN和SIGTTOU 信号 |
| SA\_NOCLDWAIT | 当调用此系统调用的进程之子进程终止时，系统不会建立zombie进程 |
| SA\_RESETHAND | 信号处理函数接收到信号后，会先将对信号处理的方式设为预设方式，而且当函数处理该信号时，后来发生的信号将不会被阻塞 |
| SA\_ONSTACK | 如果利用sigaltstack()建立信号专用堆栈，则此标志会把所有信号送往该堆栈 |
| SA\_NODEFER | 在信号处理函数处置信号的时段中，核心程序不会把这个间隙中产生的信号阻塞 |
| SA\_SIGINFO | 指定信号处理函数需要三个参数，所以应使用sa\_sigaction替sa\_handler |

1. 使用实例。

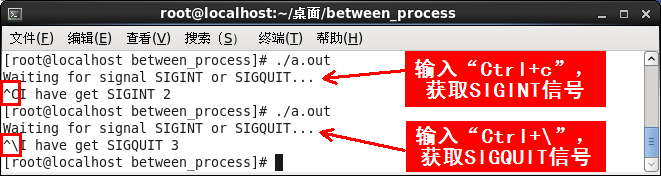
第一个实例表明了如何使用 signal()函数捕捉相应信号，并做出给定的处理。这里，my\_func就是信号处理的函数指针；第二个实例是用sigaction()函数实现同样的功能。

以下是使用 signal()函数的示例：

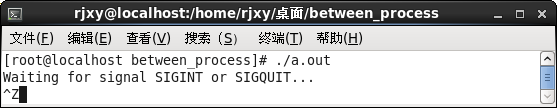
|  |
| --- |
| /\* signal.c \*/  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  /\*自定义信号处理函数\*/  void my\_func(int sign\_no)  {  if (sign\_no == SIGINT)  {  printf("I have get SIGINT %d\n",sign\_no);  }  else if (sign\_no == SIGQUIT)  {  printf("I have get SIGQUIT %d\n",sign\_no);  }  }  int main()  {  printf("Waiting for signal SIGINT or SIGQUIT...\n");  /\* 发出相应的信号，并跳转到信号处理函数处 \*/  signal(SIGINT, my\_func);  signal(SIGQUIT, my\_func);  signal(SIGTSTP,SIG\_IGN); /\* 忽略SIGTSTP（ctrl+z）信号 \*/  pause();  exit(0);  } |

运行结果如下所示：

【测试1】在程序运行的终端里直接输入“Ctrl+c”或“Ctrl+\”：

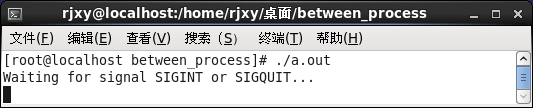


输入“Ctrl+z”后程序将忽略该信号。

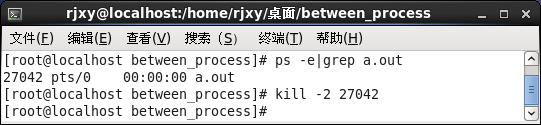


【测试2】在另一个终端中使用kill命令给该进程相应的信号，命令格式“kill -signo PID”，其中signo是要发送的信号值，可通过“kill -l”查看，SIGINT对应的信号值是2，SIGQUIT对应的信号值是3；PID是程序“a.out”所对应的PID。

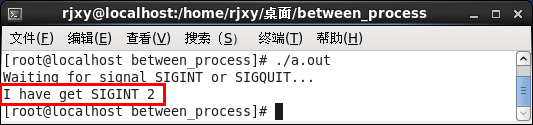
* 在第一个终端中运行程序



* 在第2个终端中查看程序的PID，并通过kill命令向其发送SIGINT信号



* 第1个终端中的程序接收到信号，做出相应的操作。



以下是用sigaction()函数实现同样的功能：

|  |
| --- |
| /\* sigaction.c \*/  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  /\*自定义信号处理函数\*/  void my\_func(int sign\_no)  {  if (sign\_no == SIGINT)  {  printf("I have get SIGINT %d\n",sign\_no);  }  else if (sign\_no == SIGQUIT)  {  printf("I have get SIGQUIT %d\n",sign\_no);  }  }  int main()  {  struct sigaction action;  printf("Waiting for signal SIGINT or SIGQUIT...\n");  /\* sigaction 结构初始化 \*/  action.sa\_handler = my\_func;  sigemptyset(&action.sa\_mask);  action.sa\_flags = 0;  /\* 发出相应的信号，并跳转到信号处理函数处 \*/  sigaction(SIGINT, &action, NULL);  sigaction(SIGQUIT, &action, NULL);  pause();  exit(0);  } |

1. 信号集函数组
2. 函数说明。

使用信号集函数组处理信号时涉及一系列的函数，这些函数按照调用的先后次序可分为以下几大功能模块：创建信号集合、注册信号、处理函数以及检测信号。

其中，创建信号集合主要用于处理用户感兴趣的一些信号，其函数包括以下几个。

* sigemptyset()：将信号集合初始化为空。
* sigfillset()：将所有已定义的信号加入信号集合中。
* sigaddset()：将指定信号加入到信号集合中。
* sigdelset()：将指定信号从信号集合中删除。
* sigismember()：查询指定信号是否在信号集合之中。

**注册信号就是将信号值加入到进程的未决信号集中**。其中“未决”是一种状态，指的是从信号产生到信号被处理前的这段时间。

处理函数主要用于决定进程如何处理信号。这里要注意的是，信号集里的信号并不是真正可以处理的信号，只有当信号的状态处于非阻塞状态时才会真正起作用。因此，首先使用sigprocmask()函数检测并更改信号屏蔽字（信号屏蔽字是用来指定当前被阻塞的一组信号，它们不会被进程接收），然后使用sigaction()函数来定义进程接收到特定信号之后的行为。

检测信号是信号处理的后续步骤，因为被阻塞的信号不会传递给进程，所以这些信号就处于“未处理”状态（也就是进程不清楚它的存在）。sigpending()函数允许进程检测“未处理”信号，并进一步决定对它们作何处理。

1. 函数格式。

首先介绍创建信号集合的函数格式，表14列举了这一组函数的语法要点。

|  |  |
| --- | --- |
| 表14 创建信号集合函数语法要点 | |
| 所需头文件 | #include <signal.h> |
| 函数原型 | int sigemptyset(sigset\_t \*set) |
| int sigfillset(sigset\_t \*set) |
| int sigaddset(sigset\_t \*set, int signum) |
| int sigdelset(sigset\_t \*set, int signum) |
| int sigismember(sigset\_t \*set, int signum) |
| 函数传入值 | set：信号集 |
| signum：指定信号代码 |
| 返回值 | 成功：0（sigismember成功返回1，失败返回0） |
| 出错：-1 |

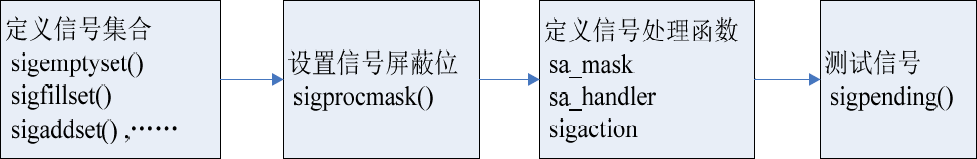
表15列举了sigprocmask的语法要点。

|  |  |  |
| --- | --- | --- |
| 表15 sigprocmask()函数语法要点 | | |
| 所需头文件 | #include <signal.h> | |
| 函数原型 | int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset) | |
| 函数传入值 | how：决定函数的操作方式 | SIG\_BLOCK：将set所指向的信号集中包含的信号加到当前的信号掩码中 |
| SIG\_UNBLOCK：将set所指向的信号集中包含的信号从当前的信号掩码中删除 |
| SIG\_SETMASK：将set的值设定为新的进程信号掩码 |
| set：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，可将其置为NULL。 | |
| oset：也是指向信号集的指针，在此存放原来的信号集。可用来检测信号掩码中存在什么信号。 | |
| 返回值 | 成功：0 | |
| 出错：-1 | |

表16列举了sigpending函数的语法要点。

|  |  |
| --- | --- |
| 表16 sigpending()函数语法要点 | |
| 所需头文件 | #include <signal.h> |
| 函数原型 | int sigpending(sigset\_t \*set) |
| 函数传入值 | set：要检测的信号集 |
| 返回值 | 成功：0 |
| 出错：-1 |

总之，在处理信号时，一般遵循如下图所示的操作流程。



1. 使用实例。

该实例首先把 SIGQUIT、 SIGINT 两个信号加入信号集，然后将该信号集合设为阻塞状态，并进入用户输入状态。只需按任意键，就可以立刻将信号集合设置为非阻塞状态，再对这两个信号分别操作，其中SIGQUIT 执行默认操作，而 SIGINT 执行用户自定义函数的操作。源代码如下所示：

|  |
| --- |
| /\* sigset.c \*/  #include <sys/types.h>  #include <unistd.h>  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  /\*自定义的信号处理函数\*/  void my\_func(int signum)  {  printf("If you want to quit, please try SIGQUIT\n");  }  int main()  {  sigset\_t set;  struct sigaction action1,action2;  /\* 初始化信号集为空 \*/  if (sigemptyset(&set) < 0)  {  perror ("sigemptyset");  exit(1);  }  /\* 将相应的信号加入信号集 \*/  if (sigaddset(&set, SIGQUIT) < 0)  {  perror("sigaddset");  exit(1);  }  if (sigaddset(&set, SIGINT) < 0)  {  perror("sigaddset");  exit(1);  }  if (sigismember(&set, SIGINT))  {  sigemptyset(&action1.sa\_mask);  action1.sa\_handler = my\_func;  action1.sa\_flags = 0;  sigaction(SIGINT, &action1, NULL);  }  if (sigismember(&set, SIGQUIT))  {  sigemptyset(&action2.sa\_mask);  action2.sa\_handler = SIG\_DFL;  action2.sa\_flags = 0;  sigaction(SIGQUIT, &action2,NULL);  }  /\* 设置信号集屏蔽字，此时set中的信号不会被传递给进程，暂时进入待处理状态 \*/  if (sigprocmask(SIG\_BLOCK, &set, NULL) < 0)  {  perror("sigprocmask");  exit(1);  }  else  {  printf("Signal set was blocked, Press any key!");  getchar();  }  /\* 在信号屏蔽字中删除set 中的信号 \*/  if (sigprocmask(SIG\_UNBLOCK, &set, NULL) < 0)  {  perror("sigprocmask");  exit(1);  }  else  {  printf("Signal set is in unblock state\n");  }  while(1);  exit(0);  } |

该程序的运行结果如下所示，可以看见，在信号处于阻塞状态时，所发出的信号对进程不起作用，并且该信号进入待处理状态。读者输入任意键，并且信号脱离了阻塞状态之后，用户发出的信号才能正常运行。

这里 SIGINT 已按照用户自定义的函数运行，请注意阻塞状态下 SIGINT 的处理和非阻塞状态下SIGINT 的处理有何不同。

