# 进程通信——信号

**实验目的：**

1. 理解IPC通信中的信号通信原理和基本技术；
2. 掌握在Linux环境中构造信号通信机制的方法和步骤。

**实验内容：**

1. 实现进程间通过信号进行通信。
2. 使用和信号相关的系统调用：signal，sigaction，kill，sigqueue等。
3. 使用和时间相关的系统调用：alarm，setitimer等。

**实验步骤：**

1. **信号通信机制基本概念**

信号机制是一种进程之间进行通信的简单通信机制。信号是一种异步通信的处理机制，进程或内核通过发送一个指定信号来通知进程某个异常事件发生，接收进程根据所接收到的信号的值以及相关参数将会做出相应动作。进程并不知道信号何时到来，进程运行时会不时地检查有无软中断信号到达，如果有，则中断原来正在执行的程序，转向该信号的处理程序对该事件进行处理，处理结束后便返回原程序的断点执行。一般地可以分成OS标准信号和用户进程自定义信号。这种机制类似硬件中断，不分优先级，简单有效。

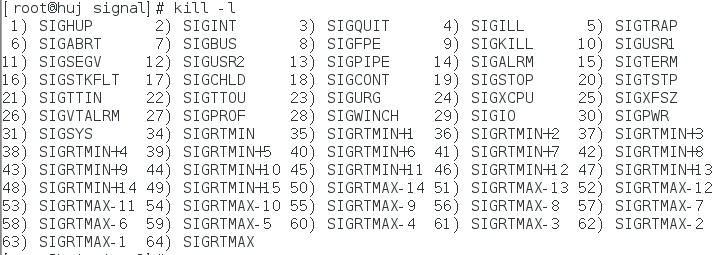
从某种角度来说，信号机制是对中断机制在软件层上的模拟，所以信号也称软中断。信号与中断的相似之处为：两者采用了相同的异步通信方式，当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序，而在处理完毕后返回到原来的断点。对信号或中断都可进行屏蔽。信号与中断的差别在于：中断有优先级，而信号没有优先级，所有的信号都是平等的。信号处理程序是在用户态下运行的，而中断处理程序是在内核态下运行。中断响应是及时的，而信号响应通常都有较大的时间延迟。

举例来说，当系统正在运行一个耗时的前台程序，若已发现有错误，并断定该程序要失败，为了节省时间，用户可以按软中断键停止程序的执行，这一过程中就用到了信号。系统具体的操作为：响应键盘输入的中断处理程序向发来中断信号的终端进程发一个信号，进程收到信号后，完成相关处理，然后执行终止。

信号机制具有以下三方面的功能：发送信号、预置对信号的处理方式和收受信号的进程按事先规定完成对相应事件的处理。把从开始发送信号直到信号处理结束的过程称为信号生命周期，它可以由四个事件来刻画：信号注册、信号发送、信号处理和信号注销。

信号不仅能从内核发给一个进程，也能由一个进程发给同组的另一个进程或多个进程。一个信号的发送是指把它送到指定进程的PCB中的软中断域的的某一位。由于每个信号都被看作一个单位，给定类型的信号不能排队，只是在进程被唤醒继续运行时，或者在进程准备从系统调用请求返回时，才处理信号。进程可以通过执行默认操作、执行一个处理函数或忽略该信号来对信号做出响应。

在头文件<signal.h>中定义了64种信号，这些信号的名字都以SIG开头，且都被定义为正整数，称为信号编号。每个数字有特定含义，用来通知进程某一异步事件的发生。可以用“kill -l”查看信号的具体名称。其中编号为1~31的信号为早期Linux所支持的信号，是不可靠信号（非实时的），编号为34~64的信号时后来扩充的，称为可靠信号（实时信号）。



Linux支持的信号列表如下。很多信号是与机器的体系结构相关的。

|  |  |  |
| --- | --- | --- |
| 信号 | 值 | 发出信号的原因 |
| SIGHUP | 1 | 终端挂起或者控制进程终止 |
| SIGINT | 2 | 键盘中断（如break键被按下） |
| SIGQUIT | 3 | 键盘的退出键被按下 |
| SIGILL | 4 | 非法指令 |
| SIGABRT | 6 | 由abort(3)发出的退出指令 |
| SIGFPE | 8 | 浮点异常 |
| SIGKILL | 9 | Kill信号 |
| SIGSEGV | 11 | 无效的内存引用 |
| SIGPIPE | 13 | 管道破裂：写一个没有读端口的管道 |
| SIGALRM | 14 | 由alarm(2)发出的信号 |
| SIGTERM | 15 | 终止信号 |
| SIGUSR1 | 30,10,16 | 用户自定义信号1 |
| SIGUSR2 | 31,12,17 | 用户自定义信号2 |
| SIGCHLD | 20,17,18 | 子进程结束信号 |
| SIGCONT | 19,18,25 | 进程继续（曾被停止的进程） |
| SIGSTOP | 17,19,23 | 终止进程 |
| SIGTSTP | 18,20,24 | 控制终端（tty）上按下停止键 |
| SIGTTIN | 21,21,26 | 后台进程企图从控制终端读 |
| SIGTTOU | 22,22,27 | 后台进程企图从控制终端写 |

如果进程接受到表中的某个信号，但事先没有安排捕捉信号，进程就会立刻终止。否则，进程可以使用系统默认的动作或自己编写的信号处理函数对信号做出响应。因为信号只是一些用以表示系统发生异常的预设的代码，所以不涉及信息的泄漏。

1. **本实验涉及的系统调用和函数**
2. 设置信号处理方式一：void (\*signal(int signum,void(\* handler) (int)))(int);

函数说明 ：

signal( )会依参数signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler指定的函数执行。

如果参数handler不是函数指针，则必须是下列两个常数之一：

* SIG\_IGN 忽略参数signum指定的信号。
* SIG\_DFL 将参数signum 指定的信号重设为核心预设的信号处理方式。

1. 设置信号处理方式二：int sigaction(int signum,conststruct sigaction \*act,struct sigaction \*oldact));

函数说明 ：

sigaction函数用于改变进程接收到特定信号后的行为。

参数说明：

* signum为信号的值，可以为除SIGKILL及SIGSTOP外的任何一个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）。
* act是指向结构sigaction的一个实例的指针，在结构sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理；
* oldact指向的对象用来保存返回的原来对相应信号的处理，可指定oldact为NULL。

1. 向任何进程或进程组发送任何信号：kill(pid\_t pid, int sig);

参数说明 ：

* pid为信号发送对象。
  1. 如果参数pid是正数，那么该调用将信号sig发送到进程号为pid的进程。
  2. 如果pid等于0，那么信号sig将发送给当前进程所属进程组里的所有进程。
  3. 如果参数pid等于-1，信号sig将发送给除了进程1和自身以外的所有进程。
  4. 如果参数pid小于-1，信号sig将发送给属于进程组-pid的所有进程。
* sig为所发送的信号值。如果sig为0，将不发送信号。

1. 新的发送信号系统调用：int sigqueue(pid\_t pid, int sig,const union sigval val)

函数说明 ：

sigqueue()是比较新的发送信号系统调用，主要是针对实时信号提出的（当然也支持前32种），支持信号带有参数，与函数sigaction()配合使用。

参数说明：

* pid是指定接收信号的进程ID；
* sig是确定即将发送的信号；
* val是一个联合数据结构union sigval，指定了信号传递的参数。

sigqueue()比kill()传递了更多的附加信息，但sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。

1. 定时发送SIGALRM的信号：unsigned int alarm(unsigned int seconds)
2. 获取定时器状态：int getitimer(int which, struct itimerval \*value);
3. 设置定时器：int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue);

参数说明：

* which：setitimer（）给进程提供了三个定时器，由which参数指定定时器类型。这三个定时器各自有其独有的计时域，当其中任何一个到达，就发送一个相应的信号给进程，并使得计时器重新开始。
  1. ITIMER\_REAL：以系统真实的时间来计算，计时到达将给进程发送SIGALRM信号。
  2. ITIMER\_VIRTUAL：以该进程在用户态下花费的时间来计算，计时到达将发送SIGVTALRM信号给进程。
  3. ITIMER\_PROF：以该进程在用户态下和内核态下所费的时间来计算，计时到达将发送SIGPROF信号给进程。
* value：第二个参数是结构itimerval的一个实例，
* 第三个参数可不做处理。

1. 让进程暂停直到信号出现：int pause(void);
2. 向自身所在进程发送一个信号：int raise(int signo);
3. **例程**
4. 通过键盘发送信号：编写程序，使它对Ctrl-C组合键的响应是输出一条消息而不是结束运行。程序将在用户第2次按下Ctrl-C组合键后结束。

|  |
| --- |
| #include <signal.h>  #include <stdio.h>  #include <unistd.h>  void ouch(int sig)  {  printf("OUCH! - I got signal %d\n", sig);  (void) signal(SIGINT, SIG\_DFL); ①//恢复SIGINT信号的处理动作  }  int main()  {  //设置SIGINT信号的处理动作为响应ouch函数  (void) signal(SIGINT, ouch); ②  //主函数每隔1秒输出字符串  while(1) {  printf("Hello World!\n");  sleep(1);  }  } |

程序安排如下：当用户通过按下Ctrl-C组合键给出SIGINT信号时，函数ouch将被调用。程序会在中断函数ouch结束后继续执行，但信号处理动作恢复为默认动作。当程序接受到第二个SIGINT信号时，程序采取默认行动，结束运行。

思考：运行该例子，理解信号机制具有以下三方面的功能：

1. 发送信号：通过键盘Ctrl-C发送终止进程的信号。试通过键盘发送其他信号，观察程序是否有反应。
2. 预置对信号的处理方式：通过signal设置信号的处理方式。删除语句①或②，观察程序的变化。
3. 收受信号的进程按事先规定完成对相应事件的处理：函数ouch(int sig)安排设置接收到信号后实施的动作。
4. 闹钟，通过系统调用alarm( )定时发送信号

|  |
| --- |
| #include<unistd.h>  #include<signal.h>  void handler() {  printf("hello\n");  }  main()  {  int i;  signal(SIGALRM,handler);  alarm(3);  for(i=1;i<7;i++){  printf("sleep %d ...\n",i);  sleep(1);  }  } |

思考：运行例程，观察输出结果。理解程序通过alarm( )定时发送信号。

1. 由于alarm（）的局限性，现在多使用setitimer（）来设置定时器，用getitimer（）来获得定时器的状态。

|  |
| --- |
| #include <signal.h>  #include <unistd.h>  #include <stdio.h>  #include <sys/time.h>  int sec;  void sigroutine(int signo) {  switch (signo) {  case SIGALRM:printf("Catch a signal -- SIGALRM \n");break;  case SIGVTALRM:printf("Catch a signal -- SIGVTALRM \n");break;  }  return;  }  int main()  {  struct itimerval value,ovalue,value2;  sec = 5;  printf("process id is %d \n",getpid());  signal(SIGALRM, sigroutine);  signal(SIGVTALRM, sigroutine);  value.it\_value.tv\_sec = 1;  value.it\_value.tv\_usec = 0;  value.it\_interval.tv\_sec = 1;  value.it\_interval.tv\_usec = 0;  setitimer(ITIMER\_REAL, &value, &ovalue);  value2.it\_value.tv\_sec = 0;  value2.it\_value.tv\_usec = 250000;  value2.it\_interval.tv\_sec = 0;  value2.it\_interval.tv\_usec =250000;  setitimer(ITIMER\_VIRTUAL, &value2, &ovalue);  for (;;) ;  } |

思考：运行例程，观察两个信号量触发的交替输出结果。理解不可靠信号存在的问题，即信号丢失的问题。

1. 父子进程间通过kill发送信号

用fork( )创建两个子进程，子进程在等待5秒后用系统调用kill（）向父进程发送SIGALARM信号，父进程用系统调用signal( )捕捉SIGLARM信号。参考程序如下所示。

|  |
| --- |
| #include <signal.h>  #include <stdio.h>  #include <unistd.h>  static int alarm\_fired = 0; //闹钟未设置  //模拟闹钟  void ding(int sig)  {  alarm\_fired = 1; //设置闹钟  }  int main()  {  int pid;  printf("alarm application starting\n");  if((pid = fork( )) == 0)  { //子进程5秒后发送信号SIGALRM给父进程  sleep(5);  kill(getppid(), SIGALRM);  exit(0);  }  //父进程安排好捕捉到SIGALRM信号后执行ding函数  printf("waiting for alarm to go off\n");  (void) signal(SIGALRM, ding);  pause( ); ① //挂起父进程，直到有一个信号出现  //wait(0); ②  if (alarm\_fired)  printf("Ding!\n");  printf("done\n");  } |

编译连接通过后，多次运行程序，查看输出结果。并思考下述问题：

1. 理解系统调用kill（）和signal( )的功能和使用方法。
2. 掌握父子进程获取对方进程号的方法。
3. 取消语句①②，观察输出的变化，解释原因。
4. 取消语句①，保留语句②，观察输出的变化，解释原因。
5. 进程使用信号通信

用fork( )创建两个子进程，再用系统调用signal( )让父进程捕捉键盘上来的中断信号（即按^c键）；捕捉到中断信号后，父进程用系统调用kill( )向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child process1 is killed by parent!

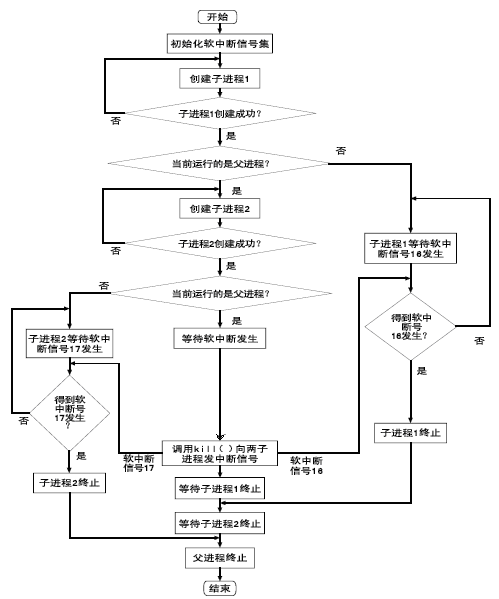
Child process2 is killed by parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent process is killed!

参考程序和流程图分别如图 13、图 14所示。

|  |
| --- |
| 略 |



进程信号通信程序的流程图

思考：

1. 编译连接通过后，多次运行程序，查看输出结果。

|  |
| --- |
|  |

|  |
| --- |
| 可以看见进程并发的进行，输出结果并不确定  课本中要求两个子进程的输出不交替，接下来给出使得不交替的代码和运行结果    代码如下  #include <stdio.h>  #include <signal.h>  #include <unistd.h>  int wait\_mark;  void waiting(), stop();  void main()  {      int p1, p2;      signal(SIGINT, stop);      while ((p1 = fork()) == -1)          ;      if (p1 > 0) */\*主进程的处理\*/*      {          while ((p2 = fork()) == -1)              ;  */\*主进程的处理\*/*          if (p2 > 0)          {              wait\_mark = 1;  */\*等待接收ctrl+c信号\*/*              waiting();  */\*向p1发出信号16\*/*              kill(p1, 16);  */\*向p2发出信号17\*/*              kill(p2, 17);  */\*同步\*/*              wait(0);              wait(0);              printf("parents is killed \n");              exit(0);          }          else */\*p2进程的处理\*/*          {              wait\_mark = 1;              signal(17, stop);              waiting(); */\*等待信号17\*/*              sleep(1);  */\*用上锁的方法实现互斥\*/*              lockf(stdout, 1, 0);              printf("P2 is killed by parent \n");              lockf(stdout, 0, 0);  */\*模拟P2被kill时进程的工作\*/*              exit(0);          }      }      else */\*p1进程的处理\*/*      {          wait\_mark = 1;          signal(16, stop);          waiting(); */\*等待信号16\*/*          sleep(1);  */\*用上锁的方法实现互斥\*/*          lockf(stdout, 1, 0);          printf("P1 is killed by parent \n");          lockf(stdout, 0, 0);  */\*模拟P1被kill时进程的工作\*/*          exit(0);      }  }  void waiting()  {      while (wait\_mark != 0)          ;  }  void stop()  {      wait\_mark = 0;  } |

1. 请说明系统调用kill（）的功能和使用方法。

|  |
| --- |
|  |

1. 父进程的处理中有两个wait（0）；他们有什么作用？
2. 每个进程退出时都使用了exit（0）；他们有什么作用？
3. **编写程序**
4. 用fork( )创建一个子进程，在子进程中再次调用fork( )创建孙子进程。调用系统调用signal( )，让父进程捕捉键盘上来的中断信号（即按^c键）；捕捉到中断信号后，父进程用系统调用kill( )向子进程发出信号，子进程捕捉到信号后，调用kill( )向孙子进程发出信号，孙子进程捕捉到信号后，输出下列信息后终止：

Grandson process1 is killed by son!

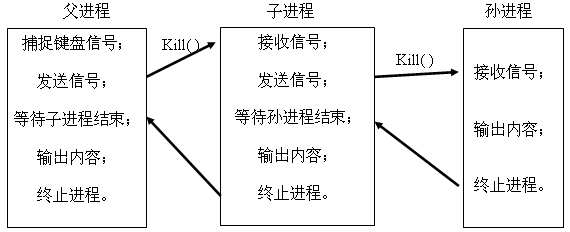
子进程等待孙子进程终止后，输出如下的信息后终止：

Child process is killed by parent!

父进程等待子进程终止后，输出如下的信息后终止：

Parent process is killed!

提示：过程如下图所示。



三代进程间通信流程图

1. 父子进程间互相发送信号：父进程无限循环向子进程发送信号，子进程收到信号后向父进程发送相同信号，父子进程之间各自记录全局变量的变化，结果如：

child caught signal #1

parent caught signal #1

child caught signal #2

parent caught signal #2

child caught signal #3

parent caught signal #3

……