# 实验六 进程通信——信号

课程名称 操作系统原理实验

实验项目名称 实验六 进程通信——信号

学生姓名 司晨旭 专业班级 计算2002 学号 32001019

**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。

1. **例程**
2. 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。

|  |
| --- |
| 为节省空间，课本代码省略 |

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。

结果一

|  |
| --- |
|  |

分析如下

|  |
| --- |
| (void) signal(SIGINT, ouch);  当进程接受到中断信号时，运行ouch函数，输出OUCH – I got signal 2 |

结果二

|  |
| --- |
|  |

分析如下

|  |
| --- |
| 可以发现，其他信号无法输出OUCH – I got signal 2 |

1. 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。

删除语句1之后的运行结果

|  |
| --- |
|  |

|  |
| --- |
| 删除语句1之后，没有了信号处理的函数，因此ctrl c之后没有发送信号 |

删除语句2之后的运行结果

|  |
| --- |
|  |

|  |
| --- |
| 由于删去了语句2，，导致Ctrl C无法正常无法正常退出进程  这是因为该语句的功能为程序会在中断函数ouch结束后继续执行，但信号处理动作恢复为默认动作。  因此删去后，信号处理动作无法回复为默认动作 |

1. 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。

|  |
| --- |
| 更改了一下ouch函数，运行结果如下 |

1. 闹钟，通过系统调用alarm( )定时发送信号

|  |
| --- |
| 为节省空间，课本代码省略 |

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。

|  |
| --- |
| 运行结果如下 |

|  |
| --- |
| alarm（3）可以理解为每过3s发送一个信号 |

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。

|  |
| --- |
| 为节省空间，课本代码省略 |

1)了解 sts/time.h中关于 struct itimerval 的定义

|  |
| --- |
| struct itimerval{  struct timeval it\_interval;*//下一次的取值*  struct timeval it\_value;*//本次的设定值*  };  *//该结构中timeval 结构定义如下*  struct timeval{  long tv\_sec;*//秒*  long tv\_usec*//微秒*  } |

2)了解定时器ITIMER\_REAL和ITIMER\_VIRTUAL的差异，理解实际时间和进程执行时间的概念

|  |
| --- |
| TIMER\_REAL: 按实际时间计时，计时到达将给进程发送SIGALARM信号  ITIMER\_VITTUAL:仅当进程执行时才进行计时。计时到达将发送SIGVTALAR信号给进程 |

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。

|  |
| --- |
| 首先给出无信号丢失的情况 |

|  |
| --- |
| 下面给出有信号丢失的情况    连着输出了三个SIGALARM，而没有SIGCVTALRM的输出。这是由于SIGVTALARM信号的丢失造成的 |

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。

|  |
| --- |
| 教材代码省去 |

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。

|  |
| --- |
| 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。 |

|  |
| --- |
| 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。  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。  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。  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。  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。 |

|  |
| --- |
| 正常运行结果如下 |

1. 掌握父子进程获取对方进程号的方法。

|  |
| --- |
| 父进程得到子进程id号:getpid()  子进程得到父进程id号:getppid() |

1. 取消语句①②，观察输出的变化，解释原因。

|  |
| --- |
| 因为删掉了pause，而子进程又要运行至少五秒，因此父进程先提前结束掉，从而没有任何的输出结果 |

1. 取消语句①，保留语句②，观察输出的变化，解释原因。

|  |
| --- |
|  |

|  |
| --- |
| 可以看到运行结果和先前的运行结果一样，pause的作用为挂起进程，直到有一个信号出现。Wait的作用为，等待子进程运行结束，因此二者都可以使得父进程晚于子进程结束，从而保证了运行结果的正常输出（即父进程收到了来自子进程的信号）。 |

1. 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!

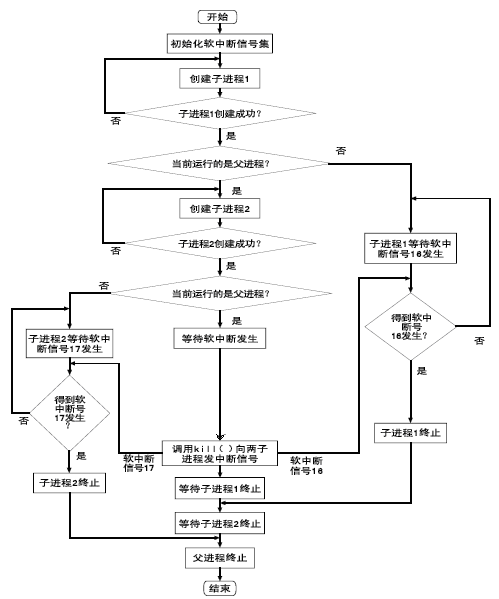
Child process2 is killed by parent!

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。

|  |
| --- |
| 课本代码略 |



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。

|  |
| --- |
| 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。  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。  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。  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。  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. 父进程的处理中有两个wait（0）；他们有什么作用？

|  |
| --- |
| 使得两个子进程结束父进程再结束 |

1. 每个进程退出时都使用了exit（0）；他们有什么作用？

|  |
| --- |
| 结束进程 |

1. 使用sigaction注册信号

|  |
| --- |
| 书中代码省略  下面给出两个终端的运行结构和输入内容 |

1）了解sigaction结构的定义

|  |
| --- |
| struct sigaction  {      union      {          sighandler\_t \_sa\_handler;          void (\*\_sa\_sigaction)(int, struct siginfo \*, void \*);      } \_u          sigset\_t sa\_mask;      unsigned long sa\_flags;  } |

2）了解使用sigaction()函数注册信号

|  |
| --- |
| 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。 |

3）理解信号注册，信号处理，信号发送的三个环节

|  |
| --- |
| 信号注册  struct sigaction act;      int sig;      sig=atoi(*argv*[1]);      sigemptyset(&act.sa\_mask);      act.sa\_flags=SA\_SIGINFO;      act.sa\_sigaction=new\_op;  信号处理  void new\_op(int *signum*,siginfo\_t \**info*,void \**myact*)  {      printf("receive signal %d\n", *signum*);      sleep(1);  }  信号发送 |

7．使用sigqueue发送信号并附加传送信息

|  |
| --- |
| 代码省略 |

1. 了解union sigval的定义

|  |
| --- |
| 1. typedef union sigval { int sival\_int; 3. void \*sival\_ptr; 4. }sigval\_t; |

2）了解使用sigqueue（）函数发送信号

|  |
| --- |
| mysigval.sival\_ptr=data;  读入值（要发送的附带信息）  sigqueue(pid,sig,mysigval);*//向本进程发送信号，并传递附加信息*  发送信号 |

3）观察进程给自身发送信号

|  |
| --- |
|  |

8.修改前面的例程，把信号发送和接受放在两个程序中，并且在发送过程中传递整型参数

|  |
| --- |
| 省略课本代码 |

1. 理解发送进程和接受进程

|  |
| --- |
| 一号进程接受来自二号进程发送的信号   pid=getpid();       sig=atoi(*argv*[1]);  pid存放进程号，sig存放接受的信号的值  act.sa\_sigaction=new\_op;  存放处理函数  二号进程发送信号   signum=atoi(*argv*[1]);       pid=(pid\_t)atoi(*argv*[2]);  signum存放信号的值  pid存放输入的进程号，也就是进程一的进程号 |

1. 按程序要求在运行时附加合适的参数，并运行检验

|  |
| --- |
| 第一个进程运行结果如下    第二个进程运行结果如下 |

1. **编写程序**
2. 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!

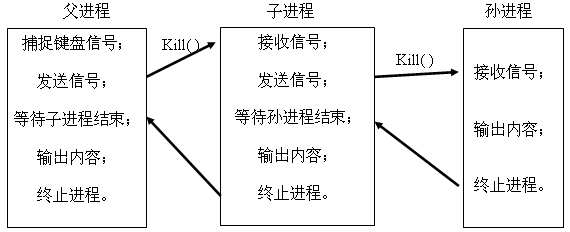
子进程等待孙子进程终止后，输出如下的信息后终止：

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

|  |
| --- |
| #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\*/  {  wait\_mark = 1;  /\*等待接收ctrl+c信号\*/  waiting();  /\*向p1发出信号16\*/  kill(p1, 16);    /\*同步\*/  wait(0);  wait(0);  printf("parents is killed \n");  exit(0);  }  else /\*p1进程的处理\*/  {  while ((p2 = fork()) == -1)  ;  if (p2 > 0) // p1进程处理  {  wait\_mark = 1;  signal(16, stop);  waiting(); /\*等待信号16\*/  /\*向p2发出信号17\*/  kill(p2, 17);  /\*用上锁的方法实现互斥\*/  wait(0);  lockf(stdout, 1, 0);  printf("Child process is killed by parent! \n");  lockf(stdout, 0, 0);  /\*模拟P1被kill时进程的工作\*/  exit(0);  }  else /\*p2进程的处理\*/  {  wait\_mark = 1;  signal(17, stop);  waiting(); /\*等待信号17\*/    /\*用上锁的方法实现互斥\*/  lockf(stdout, 1, 0);  printf("Grandson process1 is killed by son! \n");  lockf(stdout, 0, 0);  /\*模拟P2被kill时进程的工作\*/  exit(0);  }  //  }  }  void waiting()  {  while (wait\_mark != 0)  ;  }  void stop()  {  wait\_mark = 0;  } |

|  |
| --- |
| 代码运行结果如下    注意使用wait（0）就好 |

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

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  int i = 1;  void hander1()  {  printf("child caught signal \n");  }  void hander2()  {  printf("parent caught signal \n");  }  void main()  {  int p1, p2;  while ((p2 = fork()) == -1)  ;  if (p2 > 0) //主进程  {  signal(17, hander1);  while (1)  {  pause();  kill(p2, 16);  }  }  else  {  signal(16, hander2);  while (1)  {  p1 = getppid();  kill(p1, 17);  pause();  }  }  } |

|  |
| --- |
| 运行结果如下 |

将编程题1用sigaction（）和sigqueue（）再实现一次，并在信号传递时附带上信息。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <sys/types.h>  #include <signal.h>  int wait\_mark;  void waiting(), stop();  void op(int signum, siginfo\_t \*info, void \*myact)  {  printf("%s\n", (\*info).si\_ptr);  }  int main(int argc, char \*\*argv)  {  struct sigaction act;  union sigval mysigval\_father, mysigval\_son, mysigval\_grandson;  int sig = atoi(argv[1]);  int p1, p2;  //数据输入  char data\_father[40] = "father process is killed";  mysigval\_father.sival\_ptr = data\_father;  char data\_son[40] = "child process is killed by father";  mysigval\_son.sival\_ptr = data\_son;  char data\_grandson[50] = "Grandson process is killed by son";  mysigval\_grandson.sival\_ptr = data\_grandson;  sigemptyset(&act.sa\_mask);  act.sa\_flags = SA\_SIGINFO;  act.sa\_sigaction = op;  if (sigaction(sig, &act, NULL) < 0)  {  printf("install sigal error\n");  }  while ((p1 = fork()) == -1)  ;  if (p1 > 0) // father  {  wait(0);  sigqueue(getpid(), sig, mysigval\_father);  exit(0);  }  else // son  {  while ((p2 = fork()) == -1)  ;  if (p2 > 0) // son  {  wait(0);  sigqueue(getpid(), sig, mysigval\_son);  exit(0);  }  else // grandson  {  sigqueue(getpid(), sig, mysigval\_grandson);  }  }  } |

|  |
| --- |
| 运行结果如下 |

**实验心得**

|  |
| --- |
| 有关sigaction和sigqueue里的一些参数，指针仍然有些问题，其他部分还好 |