[Linux进程间通信——使用信号](http://blog.csdn.net/ljianhui/article/details/10128731)

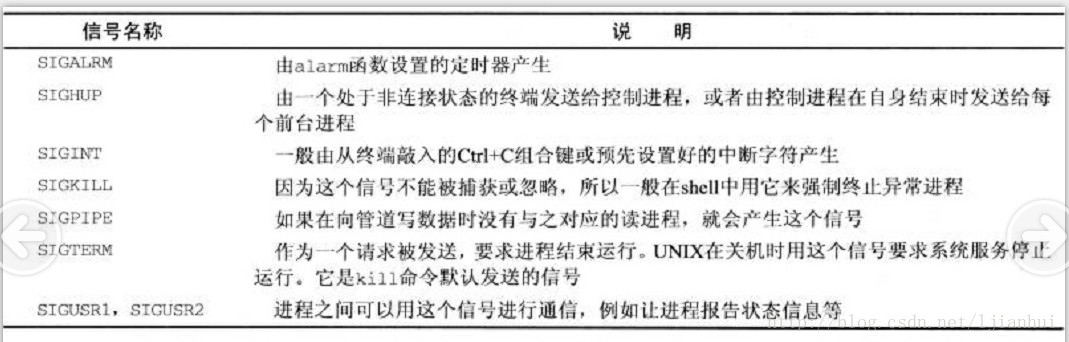
一、什么是信号

用过Windows的我们都知道，当我们无法正常结束一个程序时，可以用任务管理器强制结束这个进程，但这其实是怎么实现的呢？同样的功能在Linux上是通过生成信号和捕获信号来实现的，运行中的进程捕获到这个信号然后作出一定的操作并最终被终止。

信号是UNIX和Linux系统响应某些条件而产生的一个事件，接收到该信号的进程会相应地采取一些行动。通常信号是由一个错误产生的。但它们还可以作为进程间通信或修改行为的一种方式，明确地由一个进程发送给另一个进程。一个信号的产生叫生成，接收到一个信号叫捕获。

二、信号的种类

信号的名称是在头文件signal.h中定义的，信号都以SIG开头，常用的信号并不多，常用的信号如下：



更多的信号类型可查看附录表。

三、信号的处理——signal函数

程序可用使用signal函数来处理指定的信号，主要通过忽略和恢复其默认行为来工作。signal函数的原型如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <signal.h>
2. **void** (\*signal(**int** sig, **void** (\*func)(**int**)))(**int**);

这是一个相当复杂的声明，耐心点看可以知道signal是一个带有sig和func两个参数的函数，func是一个类型为void (\*)(int)的函数指针。该函数返回一个与func相同类型的指针，指向先前指定信号处理函数的函数指针。准备捕获的信号的参数由sig给出，接收到的指定信号后要调用的函数由参数func给出。其实这个函数的使用是相当简单的，通过下面的例子就可以知道。注意信号处理函数的原型必须为void func（int），或者是下面的特殊值：

    SIG\_IGN:忽略信号

    SIG\_DFL:恢复信号的默认行为

说了这么多，还是给出一个例子来说明一下吧，源文件名为signal1.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <signal.h>
2. #include <stdio.h>
3. #include <unistd.h>
5. **void** ouch(**int** sig)
6. {
7. printf("\nOUCH! - I got signal %d\n", sig);
8. //恢复终端中断信号SIGINT的默认行为
9. (**void**) signal(SIGINT, SIG\_DFL);
10. }
12. **int** main()
13. {
14. //改变终端中断信号SIGINT的默认行为，使之执行ouch函数
15. //而不是终止程序的执行
16. (**void**) signal(SIGINT, ouch);
17. **while**(1)
18. {
19. printf("Hello World!\n");
20. sleep(1);
21. }
22. **return** 0;
23. }

运行结果如下：



可以看到，第一次按下终止命令（ctrl+c）时，进程并没有被终止，面是输出OUCH! - I got signal 2，因为SIGINT的默认行为被signal函数改变了，当进程接受到信号SIGINT时，它就去调用函数ouch去处理，注意ouch函数把信号SIGINT的处理方式改变成默认的方式，所以当你再按一次ctrl+c时，进程就像之前那样被终止了。

四、信号处理——sigaction函数

前面我们看到了signal函数对信号的处理，但是一般情况下我们可以使用一个更加健壮的信号接口——sigaction函数。它的原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <signal.h>
2. **int** sigaction(**int** sig, **const** **struct** sigaction \*act, **struct** sigaction \*oact);

该函数与signal函数一样，用于设置与信号sig关联的动作，而oact如果不是空指针的话，就用它来保存原先对该信号的动作的位置，act则用于设置指定信号的动作。

sigaction结构体定义在signal.h中，但是它至少包括以下成员：

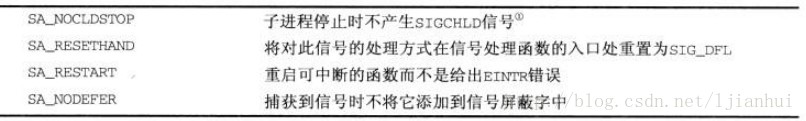
void (\*) (int) sa\_handler;处理函数指针，相当于signal函数的func参数。

sigset\_t sa\_mask; 指定一个。信号集，在调用sa\_handler所指向的信号处理函数之前，该信号集将被加入到进程的信号屏蔽字中。信号屏蔽字是指当前被阻塞的一组信号，它们不能被当前进程接收到

int sa\_flags;信号处理修改器;

sa\_mask的值通常是通过使用信号集函数来设置的，关于信号集函数，我将会在我的下一篇文章——[Linux进程间通信——信号集函数](http://blog.csdn.net/ljianhui/article/details/10130539" \t "_blank)，详细讲述。

sa\_flags，通常可以取以下的值：



此外，现在有一个这样的问题，我们使用signal或sigaction函数来指定处理信号的函数，但是如果这个信号处理函数建立之前就接收到要处理的信号的话，进程会有怎样的反应呢？它就不会像我们想像的那样用我们设定的处理函数来处理了。sa\_mask就可以解决这样的问题，sa\_mask指定了一个信号集，在调用sa\_handler所指向的信号处理函数之前，该信号集将被加入到进程的信号屏蔽字中，设置信号屏蔽字可以防止信号在它的处理函数还未运行结束时就被接收到的情况，即使用sa\_mask字段可以消除这一竞态条件。

承接上面的例子，下面给出用sigaction函数重写的例子代码，源文件为signal2.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <unistd.h>
2. #include <stdio.h>
3. #include <signal.h>
5. **void** ouch(**int** sig)
6. {
7. printf("\nOUCH! - I got signal %d\n", sig);
8. }
10. **int** main()
11. {
12. **struct** sigaction act;
13. act.sa\_handler = ouch;
14. //创建空的信号屏蔽字，即不屏蔽任何信息
15. sigemptyset(&act.sa\_mask);
16. //使sigaction函数重置为默认行为
17. act.sa\_flags = SA\_RESETHAND;
19. sigaction(SIGINT, &act, 0);
21. **while**(1)
22. {
23. printf("Hello World!\n");
24. sleep(1);
25. }
26. **return** 0;
27. }

运行结果与前一个例子中的相同。注意sigaction函数在默认情况下是不被重置的，如果要想它重置，则sa\_flags就要为SA\_RESETHAND。

五、发送信号

上面说到的函数都是一些进程接收到一个信号之后怎么对这个信号作出反应，即信号的处理的问题，有没有什么函数可以向一个进程主动地发出一个信号呢？我们可以通过两个函数kill和alarm来发送一个信号。

1、kill函数

先来看看kill函数，进程可以通过kill函数向包括它本身在内的其他进程发送一个信号，如果程序没有发送这个信号的权限，对kill函数的调用就将失败，而失败的常见原因是目标进程由另一个用户所拥有。想一想也是容易明白的，你总不能控制别人的程序吧，当然超级用户root，这种上帝般的存在就除外了。

kill函数的原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <sys/types.h>
2. #include <signal.h>
3. **int** kill(pid\_t pid, **int** sig);

它的作用把信号sig发送给进程号为pid的进程，成功时返回0。

kill调用失败返回-1，调用失败通常有三大原因：

1、给定的信号无效（errno = EINVAL)

2、发送权限不够( errno = EPERM ）

3、目标进程不存在( errno = ESRCH )

2、alarm函数

这个函数跟它的名字一样，给我们提供了一个闹钟的功能，进程可以调用alarm函数在经过预定时间后向发送一个SIGALRM信号。

alarm函数的型如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <unistd.h>
2. unsigned **int** alarm(unsigned **int** seconds);

alarm函数用来在seconds秒之后安排发送一个SIGALRM信号，如果seconds为0，将取消所有已设置的闹钟请求。alarm函数的返回值是以前设置的闹钟时间的余留秒数，如果返回失败返回-1。

马不停蹄，下面就给合fork、sleep和signal函数，用一个例子来说明kill函数的用法吧，源文件为signal3.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <unistd.h>
2. #include <sys/types.h>
3. #include <stdlib.h>
4. #include <stdio.h>
5. #include <signal.h>
7. **static** **int** alarm\_fired = 0;
9. **void** ouch(**int** sig)
10. {
11. alarm\_fired = 1;
12. }
14. **int** main()
15. {
16. pid\_t pid;
17. pid = fork();
18. **switch**(pid)
19. {
20. **case** -1:
21. perror("fork failed\n");
22. exit(1);
23. **case** 0:
24. //子进程
25. sleep(5);
26. //向父进程发送信号
27. kill(getppid(), SIGALRM);
28. exit(0);
29. **default**:;
30. }
31. //设置处理函数
32. signal(SIGALRM, ouch);
33. **while**(!alarm\_fired)
34. {
35. printf("Hello World!\n");
36. sleep(1);
37. }
38. **if**(alarm\_fired)
39. printf("\nI got a signal %d\n", SIGALRM);
41. exit(0);
42. }

运行结果如下：



在代码中我们使用fork调用复制了一个新进程，在子进程中，5秒后向父进程中发送一个SIGALRM信号，父进程中捕获这个信号，并用ouch函数来处理，变改alarm\_fired的值，然后退出循环。从结果中我们也可以看到输出了5个Hello World！之后，程序就收到一个SIGARLM信号，然后结束了进程。

注：如果父进程在子进程的信号到来之前没有事情可做，我们可以用函数pause（）来挂起父进程，直到父进程接收到信号。当进程接收到一个信号时，预设好的信号处理函数将开始运行，程序也将恢复正常的执行。这样可以节省CPU的资源，因为可以避免使用一个循环来等待。以本例子为例，则可以把while循环改为一句pause();

下面再以一个小小的例子来说明alarm函数和pause函数的用法吧，源文件名为，signal4.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10128731) [copy](http://blog.csdn.net/ljianhui/article/details/10128731)

[print?](http://blog.csdn.net/ljianhui/article/details/10128731)

1. #include <unistd.h>
2. #include <sys/types.h>
3. #include <stdlib.h>
4. #include <stdio.h>
5. #include <signal.h>
7. **static** **int** alarm\_fired = 0;
9. **void** ouch(**int** sig)
10. {
11. alarm\_fired = 1;
12. }
14. **int** main()
15. {
16. //关联信号处理函数
17. signal(SIGALRM, ouch);
18. //调用alarm函数，5秒后发送信号SIGALRM
19. alarm(5);
20. //挂起进程
21. pause();
22. //接收到信号后，恢复正常执行
23. **if**(alarm\_fired == 1)
24. printf("Receive a signal %d\n", SIGALRM);
25. exit(0);
26. }

运行结果如下：



进程在5秒后接收到一个SIGALRM，进程恢复运行，打印信息并退出。

六、信号处理函数的安全问题

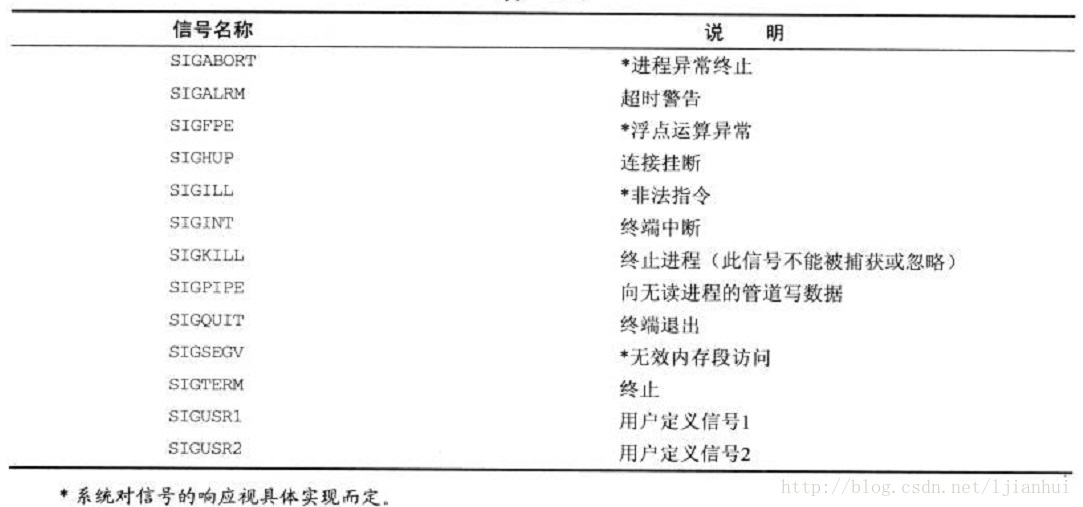
试想一个问题，当进程接收到一个信号时，转到你关联的函数中执行，但是在执行的时候，进程又接收到同一个信号或另一个信号，又要执行相关联的函数时，程序会怎么执行？

也就是说，信号处理函数可以在其执行期间被中断并被再次调用。当返回到第一次调用时，它能否继续正确操作是很关键的。这不仅仅是递归的问题，而是可重入的（即可以完全地进入和再次执行）的问题。而反观Linux，其内核在同一时期负责处理多个设备的中断服务例程就需要可重入的，因为优先级更高的中断可能会在同一段代码的执行期间“插入”进来。

简言之，就是说，我们的信号处理函数要是可重入的，即离开后可再次安全地进入和再次执行，要使信号处理函数是可重入的，则在信息处理函数中不能调用不可重入的函数。下面给出可重入的函数在列表，不在此表中的函数都是不可重入的，可重入函数表如下：



七、附录——信号表



如果进程接收到上面这些信号中的一个，而事先又没有安排捕获它，进程就会终止。

还有其他的一些信号，如下：

