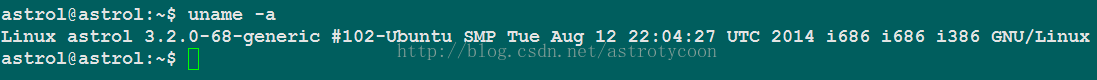
<https://blog.csdn.net/linuxheik/article/details/52825010>

最近在看APUE第10章中关于system函数的POSIX.1的实现。关于POSIX.1要求system函数忽略SIGINT和SIGQUIT，并且阻塞信号SIGCHLD的论述，理解得不是很透彻，本文就通过实际的实例来一探究竟吧。



**一、为什么要阻塞SIGCHLD信号**

**#include <stdlib.h>**

**int system(const char \*command);**

**函数工作大致流程**：system()函数先fork一个子进程，在这个子进程中调用/bin/sh -c来执行command指定的命令。/bin/sh在系统中一般是个软链接，指向dash或者bash等常用的shell，-c选项是告诉shell从字符串command中读取要执行的命令（shell将扩展command中的任何特殊字符）。父进程则调用waitpid()函数来为变成僵尸的子进程收尸，获得其结束状态，然后将这个结束状态返回给system()函数的调用者。

知道了以上基本知识点，也就好理解为什么偏偏是SIGCHLD信号了，而不是其他的信号：因为fork的子进程结束后，内核会向其父进程发送SIGHLD信号，即system()函数的调用者。

那么为什么在调用system()函数，运行command指定的命令时要阻塞SIGCHLD这个信号呢？ 接下来我们就通过两个不同的system版本对比运行的结果，从而找到阻塞SIGCHLD信号的真正原因。

先来具体看看这两个不同的system函数实现版本：

system\_without\_signal.c：

**[cpp]** [view plain](http://blog.csdn.net/astrotycoon/article/details/40626355) [copy](http://blog.csdn.net/astrotycoon/article/details/40626355)

[在CODE上查看代码片](https://code.csdn.net/snippets/502485)

1. #include <stdio.h>
2. #include <errno.h>
3. #include <unistd.h>
4. #include <stdlib.h>
5. #include <string.h>
6. #include "apue.h"
8. /\* version without signal handling \*/
9. **int** system\_without\_signal(**const** **char** \*cmd\_string)
10. {
11. pid\_t pid;
12. **int** status = -1;
14. **if** (cmd\_string == NULL)
15. **return** (1);     /\* always a command processor with UNIX \*/
17. **if** ((pid = fork()) < 0) {
18. status = -1;    /\* probably out of processes \*/
19. } **else** **if** (pid == 0) {  /\* child \*/
20. execl("/bin/sh", "sh", "-c", cmd\_string, (**char** \*)0);
21. \_exit(127); /\* execl error \*/
22. } **else** {                /\* parent \*/
23. //      sleep(1);
24. pid\_t wait\_pid;
25. **while** ((wait\_pid = waitpid(pid, &status, 0)) < 0) {
26. printf("[in system\_without\_signal]: errno = %d(%s)\n",
27. errno, strerror(errno));
28. **if** (errno != EINTR) {
29. status = -1;    /\* error other than EINTR form waitpid() \*/
30. **break**;
31. }
32. }
33. printf("[in system\_without\_signal]: pid = %ld, wait\_pid = %ld\n",
34. (**long**)pid, (**long**)wait\_pid);
35. pr\_exit("[in system\_without\_signal]", status);
36. }
38. **return** (status);
39. }

system\_with\_signal.c

**[cpp]** [view plain](http://blog.csdn.net/astrotycoon/article/details/40626355) [copy](http://blog.csdn.net/astrotycoon/article/details/40626355)

[在CODE上查看代码片](https://code.csdn.net/snippets/502485)

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <errno.h>
4. #include <string.h>
5. #include <unistd.h>
6. #include <signal.h>
7. #include <sys/types.h>
8. #include <sys/wait.h>
10. /\* with appropriate signal handling \*/
11. **int** system\_with\_signal(**const** **char** \*cmd\_string)
12. {
13. pid\_t       pid;
14. **int**         status;
15. **struct**      sigaction ignore, saveintr, savequit;
16. sigset\_t    chld\_mask, save\_mask;
18. **if** (cmd\_string == NULL)
19. **return** (1);     /\* always a command processor with UNIX \*/
21. /\* ignore signal SIGINT and SIGQUIT \*/
22. ignore.sa\_handler = SIG\_IGN;
23. ignore.sa\_flags = 0;
24. sigemptyset(&ignore.sa\_mask);
25. **if** (sigaction(SIGINT, &ignore, &saveintr) < 0)
26. **return** (-1);
27. **if** (sigaction(SIGQUIT, &ignore, &savequit) < 0)
28. **return** (-1);
30. /\* block SIGCHLD and save current signal mask \*/
31. sigemptyset(&chld\_mask);
32. sigaddset(&chld\_mask, SIGCHLD);
33. **if** (sigprocmask(SIG\_BLOCK, &chld\_mask, &save\_mask) < 0)
34. **return** (-1);
36. **if** ((pid = fork()) < 0) {
37. status = -1;    /\* probably out of processes \*/
38. } **else** **if** (pid == 0) {      /\* child \*/
39. /\* restore previous signal actions & reset signal mask \*/
40. sigaction(SIGINT, &saveintr, NULL);
41. sigaction(SIGQUIT, &savequit, NULL);
42. sigprocmask(SIG\_SETMASK, &save\_mask, (sigset\_t \*)NULL);
44. execl("/bin/sh", "sh", "-c", cmd\_string, (**char** \*)0);
45. \_exit(127);
46. } **else** {                    /\* parent \*/
47. **int** wait\_pid;
48. //  sleep(10);  /\* \*/
49. **while** ((wait\_pid = waitpid(pid, &status, 0)) < 0) {
50. printf("[in system\_with\_signal]: errno = %d(%s)\n",
51. errno, strerror(errno));
52. **if** (errno != EINTR) {
53. status = -1;    /\* error other than EINTR from waitpid() \*/
54. **break**;
55. }
56. }
57. printf("[in system\_with\_signal]: pid = %ld, wait\_pid = %ld\n",
58. (**long**)pid, (**long**)wait\_pid);
59. pr\_exit("[in system\_with\_signal]", status);
60. }
62. /\* in parent: restore previous signal action & reset signal mask \*/
63. **if** (sigaction(SIGINT, &saveintr, NULL) < 0)
64. **return** (-1);
65. **if** (sigaction(SIGQUIT, &savequit, NULL) < 0)
66. **return** (-1);
67. **if** (sigprocmask(SIG\_SETMASK, &save\_mask, (sigset\_t \*)NULL) < 0)  /\* \*/
68. **return** (-1);
70. **return** (status);
71. }

好，接下来具体看一个例子：

**[cpp]** [view plain](http://blog.csdn.net/astrotycoon/article/details/40626355) [copy](http://blog.csdn.net/astrotycoon/article/details/40626355)

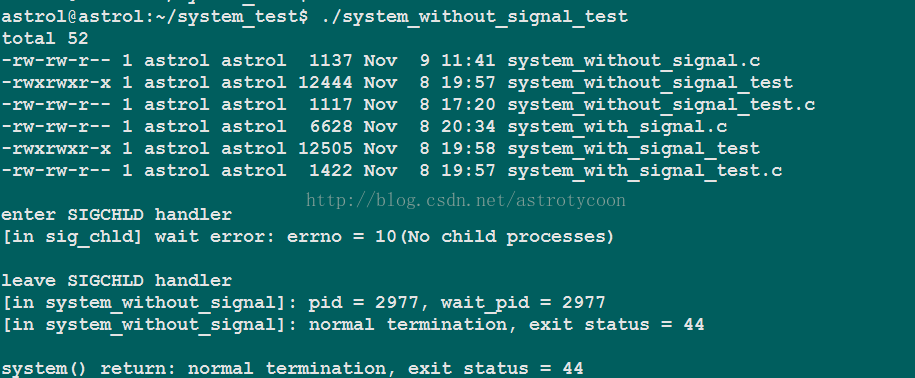
[在CODE上查看代码片](https://code.csdn.net/snippets/502485)

1. #include <stdio.h>
2. #include <errno.h>
3. #include <stdlib.h>
4. #include <signal.h>
5. #include "apue.h"
7. #define SETSIG(sa, sig, fun, flags) \
8. **do** {                                \
9. sa.sa\_handler = fun;            \
10. sa.sa\_flags = flags;            \
11. sigemptyset(&sa.sa\_mask);       \
12. sigaction(sig, &sa, NULL);      \
13. } **while** (0)
15. **extern** **int** system\_without\_signal(**const** **char** \*cmd\_string);
17. **static** **void** sig\_chld(**int** signo)
18. {
19. printf("\nenter SIGCHLD handler\n");
21. pid\_t pid;
22. **int** exit\_status = -1;
23. **int** errno\_saved = errno;
24. pid = wait(&exit\_status);
25. **if** (pid != -1) {
26. printf("[in sig\_chld] reaped %ld child,", (**long**)pid);
27. pr\_exit("wait: ", exit\_status);
28. printf("\n");
29. } **else** {
30. printf("[in sig\_chld] wait error: errno = %d(%s)\n\n",
31. errno, strerror(errno));
32. }
34. errno = errno\_saved;
35. printf("leave SIGCHLD handler\n");
36. }
38. **int** main(**int** argc, **const** **char** \*argv[])
39. {
40. pid\_t pid;
41. **struct** sigaction sigchld\_act;
43. SETSIG(sigchld\_act, SIGCHLD, sig\_chld, 0);
45. **int** status;
46. **if** ((status = system\_without\_signal("/bin/ls -l; exit 44")) < 0) {
47. err\_sys("system() error(status = %d): ", status);
48. }
49. pr\_exit("system() return:", status);
51. exit(EXIT\_SUCCESS);
52. }

在这个例子中，我们调用的是system\_without\_signal，即不处理信号的system实现版本，并且调用者还设置了SIGCHLD的信号处理函数。好，基于这些条件，接下来我们考虑两种情形：

情形1:在子进程正在运行指定程序时，或者说在子进程结束之前，父进程中的waitpid阻塞在那里。

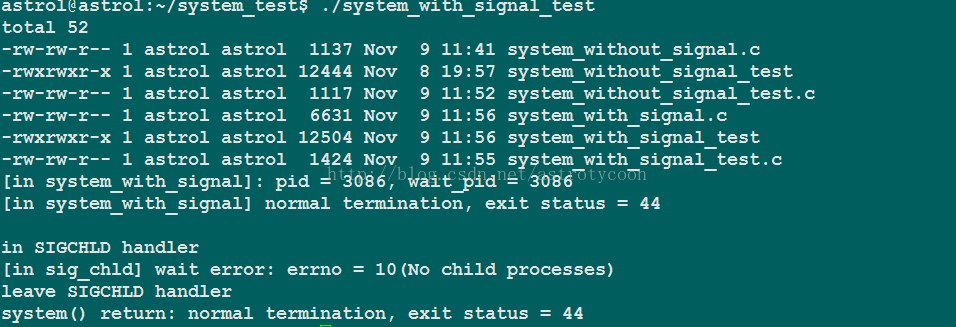
这种情形下，一旦子进程结束，内核会向应用程序递送SIGCHLD信号，运行信号处理函数，在信号处理函数中调用wait系列函数，那么现在问题来了：究竟是信号处理函数中的wait系列函数还是system\_without\_signal中的waitpid为子进程收尸呢？ 答案是未知的。因为信号本身是异步的，我们掌控不了（在我的系统中，waitpid还总能正确的获取子进程退出状态，而在信号处理函数中的wait却返回-1，errno设置为ECHLD，表明没有可收尸的子进程，见下图。但是，在你的系统中，结果也许就是相反的噢）。所以，在这种情形下，我们得出的结论是：尽管system函数完成了其任务（正确执行了我们指定的程序），但却有可能返回-1。很显然，这不是我们希望发生的。



情形2:在一个繁忙的系统中，很可能在调用waitpid之前子进程就已经结束了，此时内核会向父进程递送SIGCHLD信号。

在这种情形下，问题就更明显了。在调用waitpid之前就已经调用了SIGCHLD信号的信号处理函数，信号处理函数中的wait函数为子进程收了尸，那么接下来的waitpid不就获取不了子进程的退出状态了吗？ 事实也的确如此！我们可以在waitpid之前调用加个sleep来模拟系统负荷重的情形，会发现waitpid会出错，返回-1，errno设置为ECHLD，表明没有可收尸的子进程，最终system函数返回-1。所以，在这种情形下，我们得出的结论是：尽管system函数完成了其任务（正确执行了我们指定的程序），但却一直返回-1。很显然，这也不是我们希望发生的。

如果将上面例子中的system\_without\_signal替换成system\_with\_signal，那么system函数在调用fork之前就已经阻塞了SIGCHLD信号的话，那么就不会出现上述两种情况了。因为阻塞了SIGCHLD信号，那么不管system函数创建的子进程什么时候结束，即不管SIGCHLD信号什么时候来，在没有解除阻塞之前，是不会处理该信号的，即SIGCHLD信号是未决的。所以，无论如何，waitpid都会正确获取子进程的退出状态。只有在最后调用sigprocmask时，系统才会解除对SIGCHLD的阻塞。解除阻塞后，这才调用信号处理函数，不过这次信号处理函数中的wait会出错，返回-1，errno设置为ECHLD,表明没有可收尸的子进程。那么system函数就能正确的返回子进程的退出状态了。



看到这里，你可能会说，问题都是SIGCHLD信号处理函数中的wait惹的祸，如果去掉SIGCHLD信号处理函数中的wait函数，不就不会带来上述的两个问题了吗？ 我的答案是：的确可以避免上述两个问题，即system函数可以正确的获取子进程的退出状态。但是这样做还是会有问题的：我们先不管在SIGCHLD信号处理函数中不调用wait系列函数这种不正统的做法，我们在这里考虑这样一种情形：如果信号处理函数需要运行一分钟的时间才返回（实际编程中，信号处理函数要尽量短噢，这里只是一种极端的假设），那么system函数岂不是也要阻塞一分钟才能返回？因为如果不阻塞SIGCHLD信号并且主进程注册了SIGCHLD信号处理函数（未调用wait系列函数）,那么就需要等主进程的信号处理函数返回后waitpid才能接受到子进程的退出状态,也就是信号处理函数需要运行多长时间，那么system也就需要这么多时间才能返回。一个函数的运行受到外界不确定因素的影响，这种情形还是应该避免的。所以在调用system函数的时候阻塞SIGCHLD,这样在执行期间信号被阻塞就不会调用信号处理函数了,system中的waitpid就能"及时"地获取到子进程的状态。-- 但是仔细想想，其实system函数还是避免不了这种情形的，因为在最后调用sigprocmask解除阻塞时（一般在sigprocmask返回之前，就至少递送一个阻塞的信号），还是会调用信号处理函数，system依然会阻塞，唯一的不同是，这种情况下waitpid是在调用信号处理函数之前就获取了子进程的退出状态，避免了多线程的诸多影响。所以，在平时的编程实践当中，信号处理函数要尽量的短，这样才不会对其他函数造成不必要的未知影响。

好，稍微总结一下：

system函数之所以阻塞SIGCHLD，是为了保证system函数能够正确获取子进程的退出状态，并返回给system的调用者。

由此我们也可以引申出以下结论：

如果以后要写一个函数，函数中fork了一个子进程，并且定义的函数要得到子进程的一些信息，例如子进程的ID、子进程的终止状态等，而该函数的调用者所注册的SIGCHLD信号处理函数会影响这个函数获取这些信息，因此为了避免该函数在获取这些信息之前，由于子进程的终止触发SIGCHLD信号而先调用信号处理函数，在fork之前应该将SIGCHLD信号阻塞，在函数正确获取相关信息后，才对SIGCHLD信号解除阻塞。

**二、为什么忽略SIGINT和SIGQUIT**

关于这点，APUE的解释已经很明白了：因为由system执行的命令可能是交互式命令（例如ed程序），以及因为system的调用者在指定的命令执行期间放弃了对程序的控制（waitpid阻塞在那里），等待该执行程序的结束，所以system的调用者就不应该接收SIGINT和SIGQUIT信号，而只由子进程接收，这也是在子进程中一开始恢复SIGINT和SIGQUIT信号的原因。其实说白了，还是因为希望获取子进程的退出状态不受到外界干扰。

**三、system函数的返回值**

很多人不推荐使用system函数，是因为它的返回值很多人没有弄清楚。

（1）当参数command是NULL的时候

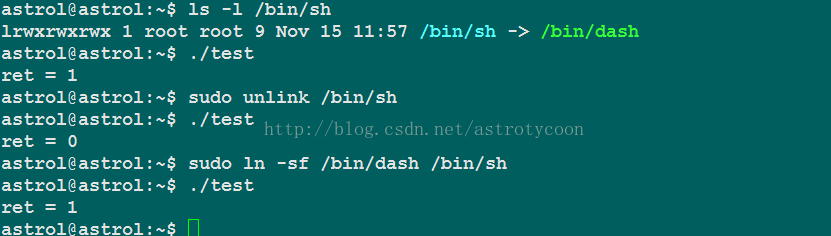
在参数为NULL的情况下，system函数的返回值很简单明了，只有0和1。返回1，表明系统的命令处理程序，即/bin/sh是可用的。相反，如果命令处理程序不可用，则返回0。我们可以通过一个简单的实验来验证下这个结论：

**[cpp]** [view plain](http://blog.csdn.net/astrotycoon/article/details/40626355) [copy](http://blog.csdn.net/astrotycoon/article/details/40626355)

[在CODE上查看代码片](https://code.csdn.net/snippets/502485)

1. <span style="font-family:Microsoft YaHei;">#include <stdio.h>
2. #include <stdlib.h>
4. **int** main(**int** argc, **const** **char** \*argv[])
5. {
6. **int** ret = system(NULL);
7. printf("ret = %d\n", ret);
9. **return** 0;
10. }
11. </span>

 在我的系统上通过ls -l /bin/sh可以看出/bin/sh是个软链接，指向/bin/dash这个SHELL，我们可以通过unlink命令先取消这个软链接，会发现程序返回0，如果再次建立这个软链接，则system返回1.



（2）当参数command不是NULL的时候

当参数不为NULL的时候，情况有些小复杂，根据APUE这里可以分为以下三种情况：

    （2.1）如果fork等系统调用失败，或者waitpid函数发生除EINTR外的错误时，system返回-1

    这种情况下，我们没有办法了，只能检测errno的值来判断是哪个系统调用出错以及出错的原因！

     那么为什么要排除waitpid发生EIINTR呢？ 对于这个问题，我们可以假设system函数的调用者设置了SIGUSR1信号的处理函数，那么当waitpid阻塞在那里时，向程序发送SIGUSR1信号，则waitpid会返回-1，errno被设置为EINTR。所以应该排除EINTR错误值，否则就获取不到/bin/sh的退出状态了。

    （2.2）一切致使execl失败的情况下，system返回127

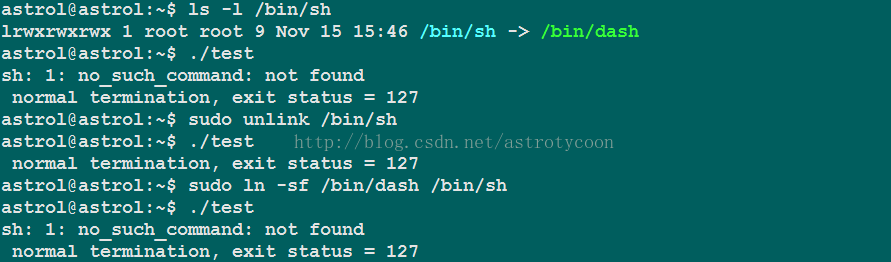
    致使execl失败的原因应该只有两个：/bin/sh不存在，再者就是指定的shell命令是非法的。

**[cpp]** [view plain](http://blog.csdn.net/astrotycoon/article/details/40626355) [copy](http://blog.csdn.net/astrotycoon/article/details/40626355)

[在CODE上查看代码片](https://code.csdn.net/snippets/502485)

1. <span style="font-family:Microsoft YaHei;">#include <stdio.h>
2. #include <stdlib.h>
3. #include "apue.h"
5. **int** main(**int** argc, **const** **char** \*argv[])
6. {
7. **int** ret = system("no\_such\_command");
8. pr\_exit("", ret);
10. **return** 0;
11. }
12. </span>

测试结果：

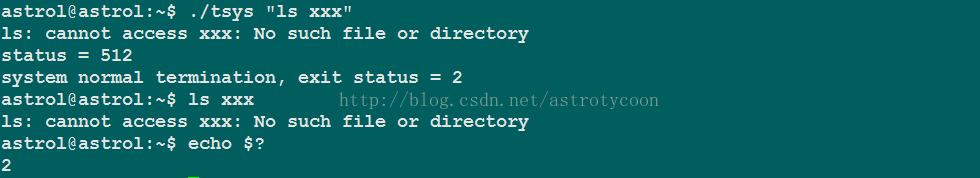


第一次返回127是因为非法的指令，第二次却是/bin/sh不存在导致的。  
那么现在的问题是：如果指定的指令执行成功，且指令的返回值正好也是127，那么如何分辨是什么原因呢（例如上述程序中的是system("exit 127")）？ 貌似没有办法哦，所以我们在程序中尽量避免使用127作为返回值。

    （2.3）除此之外，system返回/bin/sh的终止状态

          到这里，要强调的一点是：system返回的是/bin/sh的结束状态，而不是我们指定的指令的返回状态，尽管大部分时间它们是一样的。因为/bin/sh也有可能异常终止，例如人为的通过kill向其发送SIGKILL，那么/bin/sh退出状态就是9，而这跟指定的指令没有任何关系。

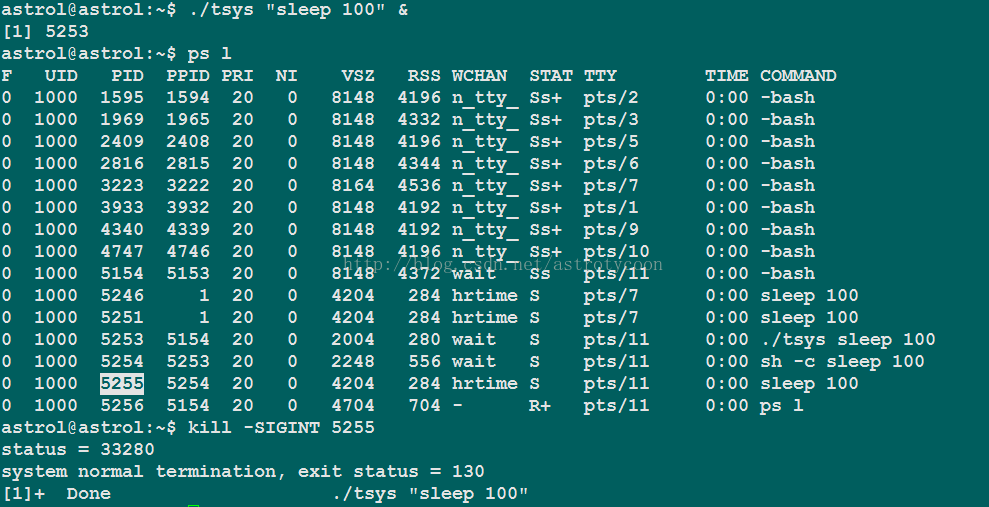
     尽管有时参数command代表的指令执行过程中出了错，但这不会影响/bin/sh的正常退出，看下面实例:



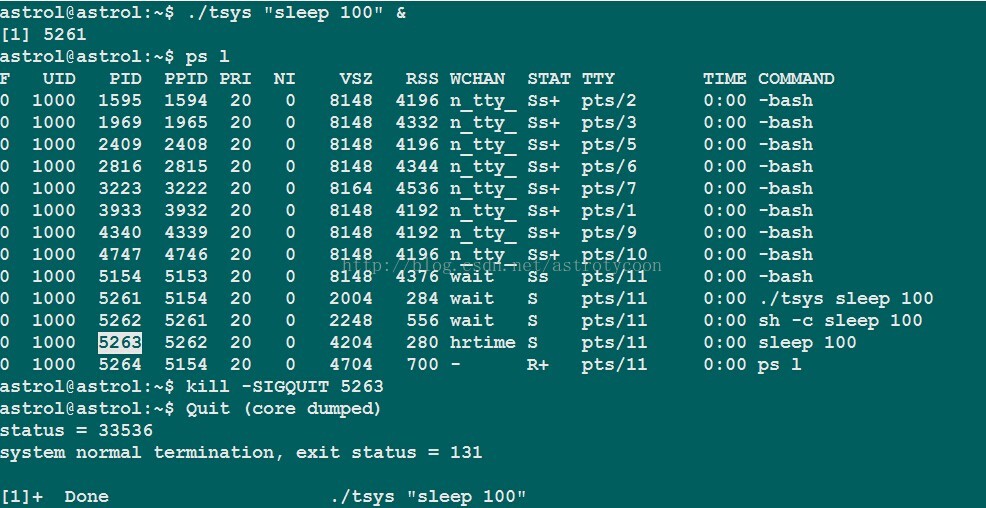
其中的tsys请自行参考APUE。很明显，xxx目录是不存在的，ls执行过程中发生了错误，返回值为2，shell接收到的就是512（为什么是512，具体下篇文章），shell将该值转换成2后，最后由waitpid接收到该终止状态，即512，pr\_exit打印的结果是2，正是ls返回的终止状态。

好了，通过之前的陈述我们知道system函数的返回值即shell的终止状态，这个终止状态是通过waitpid获得的，那么怎么解释这个返回值也就很明朗了 -- 使用检查waitpid返回值的那些宏就可以了，这也正式pr\_exit实现的方式（参考APUE第8章）。

以上说的都是指令正常终止，那么如果是异常终止了？system函数返回值可以正确反映这种状态吗？我们通过实验来验证，先看信号SIGINT：

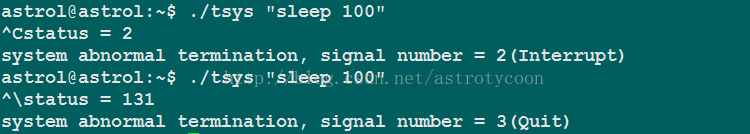


再来看下信号SIGQUIT：



可见通过system函数的返回值是不可能知道程序是异常终止的，上面的返回值之所以分别是130和131，是/bin/sh特殊处理的：当正在执行的指令是被信号终止的话，那么终止状态是128加上这个信号的编码。

这里提醒一下读者，如果你照着APUE的实验操作，即直接在终端键入Ctrl+C和Ctrl+\的话，你的结果可能与作者的是不一样的。我的结果就与作者的不一样：



你的系统上的结果也许和我的也不一样的，原因是不同的shell对信号的处理方式是不一样的，APUE作者使用的shell对SIGINT和SIGQUI的处理应该都是忽略，从我上面的结果可以看出，dash忽略信号SIGQUIT。未完待续！

参考链接：

<http://bbs.chinaunix.net/forum.php?mod=viewthread&tid=2078496>

[http://blog.chinaunix](http://blog.chinaunix.net/uid-24774106-id-3048281.html?page=3)[**.NET**](http://lib.csdn.net/base/dotnet)/uid-24774106-id-3048281.html?page=3