对于linux下system()函数的深度理解(整理)

(2013-02-07 08:58:54)

[[http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif](javascript:;)转载▼](javascript:;)

|  |  |
| --- | --- |
| 标签：  [system()](http://search.sina.com.cn/?c=blog&q=system%28%29&by=tag)  [popen()](http://search.sina.com.cn/?c=blog&q=popen%28%29&by=tag)  [system()不稳定](http://search.sina.com.cn/?c=blog&q=system%28%29%B2%BB%CE%C8%B6%A8&by=tag)  [linux调用系统命令](http://search.sina.com.cn/?c=blog&q=linux%B5%F7%D3%C3%CF%B5%CD%B3%C3%FC%C1%EE&by=tag)  [system()详解](http://search.sina.com.cn/?c=blog&q=system%28%29%CF%EA%BD%E2&by=tag) | 分类： [linuxC](http://blog.sina.com.cn/s/articlelist_2151896182_7_1.html) |

这几天调程序(嵌入式linux),发现程序有时就莫名其妙的死掉,每次都定位在程序中不同的system()函数,直接在shell下输入system()函数中调用的命令也都一切正常.就没理这个bug,以为是其他的代码影响到这个,或是内核驱动文件系统什么的异常导致,昨天有出现了这个问题,就随手百了一下度,问题出现了,很多人都说system()函数要慎用要少用要能不用则不用,system()函数不稳定?

下面对system函数做一个简单的介绍:

头文件

＃i nclude

定义函数

int system(const char \* string);

函数说明

system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，此命>令执行完后随即返回原调用的进程。在调用system()期间SIGCHLD 信号会被暂时搁置，SIGINT和SIGQUIT 信号则会被忽略。 返回值 =-1:出现错误 =0:调用成功但是没有出现子进程 >0:成功退出的子进程的id 如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值>。如果system()调用成功则最后会返回执行shell命令后的返回值，但是此返回值也有可能为 system()调用/bin/sh失败所返回的127，因此最好能再检查errno 来确认执行成功。

附加说明

在编写具有SUID/SGID权限的程序时请勿使用system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。 system函数已经被收录在标准c库中，可以直接调用,使用system()函数调用系统命令的基本使用方法如下:

#include

int main()

{

system("mkdir $HOME/.SmartPlatform/");

system("mkdir $HOME/.SmartPlatform/Files/");

system("cp mainnew.cpp $HOME/.SmartPlatform/Files/");

return 0;

}

下面我们来看看system函数的源码:

#include

#include

#include

#include

int system(const char \* cmdstring)

{

pid\_t pid;

int status;

if(cmdstring == NULL)

{

return (1);

}

if((pid = fork())<0)

{

status = -1;

}

else if(pid = 0)

{

execl("/bin/sh", "sh", "-c", cmdstring, (char \*)0); -exit(127); //子进程正常执行则不会执行此语句 }

else

{

while(waitpid(pid, &status, 0) < 0)

{

if(errno != EINTER){ status = -1; break;

}

}

}

return status;

}

花了两天时间仔细研究了一下,在网上发现了一篇精品博客,介绍的很详细了,谢谢博主,直接转, 原文如下:

http://my.oschina.net/renhc/blog/53580

**【C/C++】Linux下使用system()函数一定要谨慎**

曾经的曾经，被system()函数折磨过，之所以这样，是因为对system()函数了解不够深入。只是简单的知道用这个函数执行一个系统命令，这远远不够，它的返回值、它所执行命令的返回值以及命令执行失败原因如何定位，这才是重点。当初因为这个函数风险较多，故抛弃不用，改用其他的方法。这里先不说我用了什么方法，这里必须要搞懂system()函数，因为还是有很多人用了system()函数，有时你不得不面对它。

先来看一下system()函数的简单介绍：

#include

int system(const char \*command);

system() executes a command specified in command by calling /bin/sh -c command, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored. system()函数调用/bin/sh来执行参数指定的命令，/bin/sh 一般是一个软连接，指向某个具体的shell，比如bash，-c选项是告诉shell从字符串command中读取命令；在该command执行期间，SIGCHLD是被阻塞的，好比在说：hi，内核，这会不要给我送SIGCHLD信号，等我忙完再说；在该command执行期间，SIGINT和SIGQUIT是被忽略的，意思是进程收到这两个信号后没有任何动作。

再来看一下system()函数返回值：

The value returned is -1 on error (e.g. fork(2) failed), and the return status of the command otherwise. This latter return status is in the format specified in wait(2). Thus, the exit code of the command will be WEXITSTATUS(status). In case /bin/sh could not be executed, the exit status will be that of a command that does exit(127). If the value of command is NULL, system() returns nonzero if the shell is available, and zero if not.

为了更好的理解system()函数返回值，需要了解其执行过程，实际上system()函数执行了三步操作：

1.fork一个子进程；

2.在子进程中调用exec函数去执行command；

3.在父进程中调用wait去等待子进程结束。 对于fork失败，system()函数返回-1。如果exec执行成功，也即command顺利执行完毕，则返回command通过exit或return返回的值。（注意，command顺利执行不代表执行成功，比如command："rm debuglog.txt"，不管文件存不存在该command都顺利执行了）如果exec执行失败，也即command没有顺利执行，比如被信号中断，或者command命令根本不存在，system()函数返回127. 如果command为NULL，则system()函数返回非0值，一般为1.

看一下system()函数的源码

看完这些，

我想肯定有人对system()函数返回值还是不清楚，看源码最清楚，下面给出一个system()函数的实现：

int system(const char \* cmdstring)

{

pid\_t pid;

int status;

if(cmdstring == NULL)

{

return (1); //如果cmdstring为空，返回非零值，一般为1

}

if((pid = fork())<0)

{

status = -1; //fork失败，返回-1

}

else if(pid == 0)

{

execl("/bin/sh", "sh", "-c", cmdstring, (char \*)0);

\_exit(127); // exec执行失败返回127，注意exec只在失败时才返回现在的进程，成功的话现在的进程就不存在啦~~

}

else //父进程

{

while(waitpid(pid, &status, 0) < 0)

{

if(errno != EINTR)

{

status = -1; //如果waitpid被信号中断，则返回-1

break;

}

}

}

return status; //如果waitpid成功，则返回子进程的返回状态

}

仔细看完这个system()函数的简单实现，那么该函数的返回值就清晰了吧，那么什么时候system()函数返回0呢？只在command命令返回0时。看一下该怎么监控system()函数执行状态 这里给我出的做法：

int status;

if(NULL == cmdstring) //如果cmdstring为空趁早闪退吧，尽管system()函数也能处理空指针

{

return XXX;

}

status = system(cmdstring);

if(status < 0)

{

printf("cmd: %s\t error: %s", cmdstring, strerror(errno)); // 这里务必要把errno信息输出或记入Log

return XXX;

}

if(WIFEXITED(status))

{

printf("normal termination, exit status = %d\n", WEXITSTATUS(status)); //取得cmdstring执行结果 }

else if(WIFSIGNALED(status))

{

printf("abnormal termination,signal number =%d\n", WTERMSIG(status)); //如果cmdstring被信号中断，取得信号值

}

else if(WIFSTOPPED(status))

{

printf("process stopped, signal number =%d\n", WSTOPSIG(status)); //如果cmdstring被信号暂停执行，取得信号值

}

至于取得子进程返回值的相关介绍可以参考另一篇文章：http://my.oschina.net/renhc/blog/35116 system()函数用起来很容易出错，返回值太多，而且返回值很容易跟command的返回值混淆。

这里推荐使用popen()函数替代，关于popen()函数的简单使用也可以通过上面的链接查看。

popen()函数较于system()函数的优势在于使用简单，popen()函数只返回两个值：成功返回子进程的status，使用WIFEXITED相关宏就可以取得command的返回结果；失败返回-1，我们可以使用perro()函数或strerror()函数得到有用的错误信息。这篇文章只涉及了system()函数的简单使用，还没有谈及SIGCHLD、SIGINT和SIGQUIT对system()函数的影响，事实上，之所以今天写这篇文章，是因为项目中因有人使用了system()函数而造成了很严重的事故。现像是system()函数执行时会产生一个错误：“No child processes”。关于这个错误的分析，感兴趣的朋友可以看一下：http://my.oschina.net/renhc/blog/54582 2012-04-14

qdurenhongcai@163.com 转载请注明出处。

下面是第二篇,对于system()函数的错误详细分析,再次感谢博主

**【C/C++】Linux下system()函数引发的错误**

今天，一个运行了近一年的程序突然挂掉了，问题定位到是system()函数出的问题，关于该函数的简单使用在我上篇文章做过介绍： http://my.oschina.net/renhc/blog/53580

先看一下问题简单封装了一下system()函数：

int pox\_system(const char \*cmd\_line)

{

return system(cmd\_line);

}

函数调用：

int ret = 0;

ret = pox\_system("gzip -c /var/opt/I00005.xml > /var/opt/I00005.z");

if(0 != ret)

{

Log("zip file failed\n");

}

问题现象：每次执行到此处，都会zip failed。而单独把该命令拿出来在shell里执行却总是对的，事实上该段代码已运行了很长时间，从没出过问题。

糟糕的日志

分析log时，我们只能看到“zip file failed”这个我们自定义的信息，至于为什么fail，毫无线索。那好，我们先试着找出更多的线索：

int ret = 0;

ret = pox\_system("gzip -c /var/opt/I00005.xml > /var/opt/I00005.z");

if(0 != ret)

{

Log("zip file failed: %s\n", strerror(errno)); //尝试打印出系统错误信息

}

我们增加了log，通过system()函数设置的errno，我们得到一个非常有用的线索：system()函数失败是由于“ No child processes”。继续找Root Cause。

谁动了errno

我们通过上面的线索，知道system()函数设置了errno为ECHILD，然而从system()函数的man手册里我们找不到任何有关EHILD的信息。我们知道system()函数执行过程为：fork()->exec()->waitpid(). 很显然waitpid()有重大嫌疑，我们去查一下man手册，看该函数有没有可能设置

ECHILD： ECHILD (for waitpid() or waitid()) The process specified by pid (waitpid()) or idtype and id (waitid()) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for SIGCHLD is set to SIG\_IGN. See also the Linux Notes section about threads.)

果然有料，如果SIGCHLD信号行为被设置为SIG\_IGN时，waitpid()函数有可能因为找不到子进程而报ECHILD错误。似乎我们找到了问题的解决方案：在调用system()函数前重新设置SIGCHLD信号为缺省值，即signal(SIGCHLD, SIG\_DFL)。

我们很兴奋，暂时顾不上看Linux Notes部分，直接加上代码测试！乖乖，问题解决了！

如此处理问题是你的风格吗 正当我们急于check in 代码时，一个疑问出现了：“这个错误为什么以前没发生”？是啊，运行良好的程序怎么突然就挂了呢？首先我们代码没有改动，那么肯定是外部因素了。一想到外部因素，我们开始抱怨：“肯定是其他组的程序影响我们了！”但抱怨这是没用的，如果你这么认为，那么请拿出证据！但静下来分析一下不难发现，这不可能是其他程序的影响，其他进程不可能影响我们进程对信号的处理方式。 system()函数之前没出错，是因为systeme()函数依赖了系统的一个特性，那就是内核初始化进程时对SIGCHLD信号的处理方式为SIG\_DFL，这是什么什么意思呢？即内核发现进程的子进程终止后给进程发送一个SIGCHLD信号，进程收到该信号后采用SIG\_DFL方式处理，那么SIG\_DFL又是什么方式呢？SIG\_DFL是一个宏，定义了一个信号处理函数指针，事实上该信号处理函数什么也没做。这个特性正是system()函数需要的，system()函数首先fork()一个子进程执行command命令，执行完后system()函数会使用waitpid()函数对子进程进行收尸。通过上面的分析，我们可以清醒的得知，system()执行前，SIGCHLD信号的处理方式肯定变了，不再是SIG\_DFL了，至于变成什么暂时不知道，事实上，我们也不需要知道，我们只需要记得使用system()函数前把SIGCHLD信号处理方式显式修改为SIG\_DFL方式，同时记录原来的处理方式，使用完system()后再设为原来的处理方式。

这样我们可以屏蔽因系统升级或信号处理方式改变带来的影响。验证猜想我们公司采用的是持续集成+敏捷开发模式，每天都会由专门的team负责自动化case的测试，每次称为一个build，我们分析了本次build与上次build使用的系统版本，发现版本确实升级了。于是我们找到了相关team进行验证，我们把问题详细的描述了一下，很快对方给了反馈，下面是邮件回复原文： LIBGEN 里新增加了SIGCHLD的处理。将其ignore。为了避免僵尸进程的产生。看来我们的猜想没错！问题分析到这里，解决方法也清晰了，于是我们修改了我们的pox\_system()函数：

typedef void (\*sighandler\_t)(int);

int pox\_system(const char \*cmd\_line)

{

int ret = 0;

sighandler\_t old\_handler;

old\_handler = signal(SIGCHLD, SIG\_DFL);

ret = system(cmd\_line);

signal(SIGCHLD, old\_handler);

return ret;

}

我想这是调用system()比较完美的解决方案了，同时使用pox\_system()函数封装带来了非常棒的易维护性，我们只需要修改此处一个函数，其他调用处都不需要改。后来，查看了对方修改的代码，果然从代码上找到了答案：

if (signal(SIGCHLD, SIG\_IGN) == SIG\_ERR)

{

return -1;

}

else {

return 0;

} 其他思考我们公司的代码使用SVN进程管理的，到目前为止有很多branch，逐渐的，几乎每个branch都出现了上面的问题，于是我逐个在各个branchc上fix这个问题，几乎忙了一天，因为有的branch已被锁定，再想merge代码必须找相关负责人说明问题的严重性，还要在不同的环境上测试，我边做这些边想，系统这样升级合适吗？首先，由于系统的升级导致我们的代码在测试时发现问题，这时再急忙去fix，造成了我们的被动，我想这是他们的一个失误。你做的升级必须要考虑到对其他team的影响吧？何况你做的是系统升级。升级前需要做个风险评估，对可能造成的影响通知大家，这样才职业嘛。再者，据他们的说法，修改信号处理方式是为了避免僵尸进程，当然初衷是好的，但这样的升级影响了一些函数的使用方式，比如system()函数、wait()函数、waipid()、fork()函数，这些函数都与子进程有关，如果你希望使用wait()或waitpid()对子进程收尸，那么你必须使用上面介绍的方式：在调用前（事实上是fork()前）将SIGCHLD信号置为SIG\_DFL处理方式，调用后（事实上wait()/waitpid()后）再将信号处理方式设置为从前的值。你的系统升级，强制大家完善代码，确实提高了代码质量，但是对于这种升级我不是很认同，试想一下，你见过多少fork()->waitpid()前后都设置SIGCHLD信号的代码？

使用system()函数的建议上在给出了调用system()函数的比较安全的用法，但使用system()函数还是容易出错，错在哪？

那就是system()函数的返回值，关于其返回值的介绍请见上篇文章。system()函数有时很方便，但不可滥用！

1、建议system()函数只用来执行shell命令，因为一般来讲，system()返回值不是0就说明出错了；

2、建议监控一下system()函数的执行完毕后的errno值，争取出错时给出更多有用信息；

3、建议考虑一下system()函数的替代函数popen()；其用法在我的另一篇文章有介绍。

qdurenhongcai@163.com 转载请注明出处。

继续转该牛X博主的博客,对于上文提到的system()函数的替换函数popen()的详细介绍...万分感谢博主:

**【IPC通信】基于管道的popen和pclose函数**

标准I/O函数库提供了popen函数，它启动另外一个进程去执行一个shell命令行。这里我们称调用popen的进程为父进程，由popen启动的进程称为子进程。 popen函数还创建一个管道用于父子进程间通信。父进程要么从管道读信息，要么向管道写信息，至于是读还是写取决于父进程调用popen时传递的参数。下在给出popen、pclose的定义：

#include

FILE \* popen( const char \* command,const char \* type);

int pclose(FILE \* stream);

下面通过例子看下popen的使用：假如我们想取得当前目录下的文件个数，在shell下我们可以使用：

ls | wc -l 我们可以在程序中这样写：

#include

#include

#include

#include

#define MAXLINE 1024

int main()

{

char result\_buf[MAXLINE], command[MAXLINE];

int rc = 0; // 用于接收命令返回值

FILE \*fp;

snprintf(command, sizeof(command), "ls ./ | wc -l");

fp = popen(command, "r");

if(NULL == fp)

{

perror("popen执行失败！");

exit(1);

}

while(fgets(result\_buf, sizeof(result\_buf), fp) != NULL)

{

if('\n' == result\_buf[strlen(result\_buf)-1])

{

result\_buf[strlen(result\_buf)-1] = '\0';

}

printf("命令【%s】 输出【%s】\r\n", command, result\_buf);

}

rc = pclose(fp);

if(-1 == rc)

{

perror("关闭文件指针失败");

exit(1);

}

else

{

printf("命令【%s】子进程结束状态【%d】命令返回值【%d】\r\n", command, rc, WEXITSTATUS(rc));

}

return 0;

}

编译并执行：

$ gcc popen.c

$ ./a.out 命令【ls ./ | wc -l】

输出【2】 命令【ls ./ | wc -l】子进程结束状态【0】命令返回值【0】

上面popen只捕获了command的标准输出，如果command执行失败，子进程会把错误信息打印到标准错误输出，父进程就无法获取。比如，command命令为“ls nofile.txt” ，事实上我们根本没有nofile.txt这个文件，这时shell会输出“ls: nofile.txt: No such file or directory”。这个输出是在标准错误输出上的。通过上面的程序并无法获取。注：如果你把上面程序中的command设成“ls nofile.txt”,编译执行程序你会看到如下结果：

$ gcc popen.c

$ ./a.out ls: nofile.txt: No such file or directory

命令【ls nofile.txt】子进程结束状态【256】命令返回值【1】 需要注意的是第一行输出并不是父进程的输出，而是子进程的标准错误输出。有时子进程的错误信息是很有用的，那么父进程怎么才能获取子进程的错误信息呢？这里我们可以重定向子进程的错误输出，让错误输出重定向到标准输出（2>&1），这样父进程就可以捕获子进程的错误信息了。

例如command为“ls nofile.txt 2>&1”,输出如下：

命令【ls nofile.txt 2>&1】 输出【ls: nofile.txt: No such file or directory】

命令【ls nofile.txt 2>&1】子进程结束状态【256】命令返回值【1】

附：子进程的终止状态判断涉及到的宏，设进程终止状态为status. WIFEXITED(status)如果子进程正常结束则为非0值。 WEXITSTATUS(status)取得子进程exit()返回的结束代码，一般会先用WIFEXITED 来判断是否正常结束才能使用此宏。 WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真。 WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED 来判断后才使用此宏。 WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED 时才会有此情况。 WSTOPSIG(status)取得引发子进程暂停的信号代码，一般会先用WIFSTOPPED 来判断后才使用此宏。 2011-11-12 任洪彩

qdurenhongcai@163.com 转载请注明出处。

但是根据上面那位博主说的使用system()函数前把SIGCHLD信号处理方式显式修改为SIG\_DFL方式，同时记录原来的处理方式，使用完system()后再设为原来的处理方式后,程序还是会死掉.而且看不到system的返回值是多少(因为system在执行系统命令的时候,程序已经挂掉了),故暂时使用博主提到的第二种解决方式使用popen()函数替代system()函数.修改后的函数如下

int my\_system(const char \* cmd)

{

FILE \* fp;

int res; char buf[1024];

if (cmd == NULL)

{

printf("my\_system cmd is NULL!\n");

return -1;

}

if ((fp = popen(cmd, "r") ) == NULL)

{

perror("popen");

printf("popen error: %s/n", strerror(errno)); return -1;

}

else

{

while(fgets(buf, sizeof(buf), fp))

{

printf("%s", buf);

}

if ( (res = pclose(fp)) == -1)

{

printf("close popen file pointer fp error!\n"); return res;

}

else if (res == 0)

{

return res;

}

else

{

printf("popen res is :%d\n", res); return res;

}

}

}

此时调用my\_system()来执行system函数的功能(my\_system函数中是使用popen()函数来实现的), 测试了一天,没有再次出现程序突然死掉的问题(修改前连续循环调用system()函数测试,每10次就会至少导致程序挂掉一次.连续不停顿的调用). 以上是我对这个问题的总结,先做个记录,待修复bug后再回来仔细研究.