strace 命令用法

作者：　　来源：zz　　发表时间：2007-02-01

http://blog.linuxmine.com/i554

strace命令用法

调用:

strace [-dffhiqrtttTvxx] [-acolumn] [-eexpr] ...

[-ofile] [-ppid] ... [-sstrsize] [-uusername] [command [arg ...]]

strace -c [-eexpr] ... [-Ooverhead] [-Ssortby] [command [arg ...]]

功能:

跟踪程式执行时的系统调用和所接收的信号.通常的用法是strace执行一直到commande结束.

并且将所调用的系统调用的名称、参数和返回值输出到标准输出或者输出到-o指定的文件.

strace是一个功能强大的调试,分析诊断工具.你将发现他是一个极好的帮手在你要调试一个无法看到源码或者源码无法在编译的程序.

你将轻松的学习到一个软件是如何通过系统调用来实现他的功能的.而且作为一个程序设计师,你可以了解到在用户态和内核态是如何通过系统调用和信号来实现程序的功能的.

strace的每一行输出包括系统调用名称,然后是参数和返回值.这个例子:

strace cat /dev/null

他的输出会有:

open(\"/dev/null\",O_RDONLY) = 3

有错误产生时,一般会返回-1.所以会有错误标志和描述:

open(\"/foor/bar\",)_RDONLY) = -1 ENOENT (no such file or directory)

信号将输出喂信号标志和信号的描述.跟踪并中断这个命令\"sleep 600\":

sigsuspend({}

--- SIGINT (Interrupt) ---

+++ killed by SIGINT +++

参数的输出有些不一致.如shell命令中的 \">>tmp\",将输出:

open(\"tmp\",O_WRONLY|O_APPEND|A_CREAT,0666) = 3

对于结构指针,将进行适当的显示.如:\"ls -l /dev/null\":

lstat(\"/dev/null\",{st_mode=S_IFCHR|0666},st_rdev=makdev[1,3],...}) = 0

请注意\"struct stat\" 的声明和这里的输出.lstat的第一个参数是输入参数,而第二个参数是向外传值.

当你尝试\"ls -l\" 一个不存在的文件时,会有:

lstat(/foot/ball\",0xb004) = -1 ENOENT (no such file or directory)

char*将作为C的字符串类型输出.没有字符串输出时一般是char* 是一个转义字符,只输出字符串的长度.

当字符串过长是会使用\"...\"省略.如在\"ls -l\"会有一个gepwuid调用读取password文件:

read(3,\"root::0:0:System Administrator:/\"...,1024) = 422

当参数是结构数组时,将按照简单的指针和数组输出如:

getgroups(4,[0,2,4,5]) = 4

关于bit作为参数的情形,也是使用方括号,并且用空格将每一项参数隔开.如:

sigprocmask(SIG_BLOCK,[CHLD TTOU],[]) = 0

这里第二个参数代表两个信号SIGCHLD 和 SIGTTOU.如果bit型参数全部置位,则有如下的输出:

sigprocmask(SIG_UNBLOCK,~[],NULL) = 0

这里第二个参数全部置位.

参数说明:

-c 统计每一系统调用的所执行的时间,次数和出错的次数等.

-d 输出strace关于标准错误的调试信息.

-f 跟踪由fork调用所产生的子进程.

-ff 如果提供-o filename,则所有进程的跟踪结果输出到相应的filename.pid中,pid是各进程的进程号.

-F 尝试跟踪vfork调用.在-f时,vfork不被跟踪.

-h 输出简要的帮助信息.

-i 输出系统调用的入口指针.

-q 禁止输出关于脱离的消息.

-r 打印出相对时间关于,,每一个系统调用.

-t 在输出中的每一行前加上时间信息.

-tt 在输出中的每一行前加上时间信息,微秒级.

-ttt 微秒级输出,以秒了表示时间.

-T 显示每一调用所耗的时间.

-v 输出所有的系统调用.一些调用关于环境变量,状态,输入输出等调用由于使用频繁,默认不输出.

-V 输出strace的版本信息.

-x 以十六进制形式输出非标准字符串

-xx 所有字符串以十六进制形式输出.

-a column

设置返回值的输出位置.默认为 40.

-e expr

指定一个表达式,用来控制如何跟踪.格式如下:

[qualifier=][!]value1[,value2]...

qualifier只能是 trace,abbrev,verbose,raw,signal,read,write其中之一.value是用来限定的符号或数字.默认的qualifier是 trace.感叹号是否定符号.例如:

-eopen等价于 -e trace=open,表示只跟踪open调用.而-etrace!=open表示跟踪除了open以外的其他调用.有两个特殊的符号 all 和 none.

注意有些shell使用!来执行历史记录里的命令,所以要使用\\.

-e trace=set

只跟踪指定的系统调用.例如:-e trace=open,close,rean,write表示只跟踪这四个系统调用.默认的为set=a
ll.

-e trace=file

只跟踪有关文件操作的系统调用.

-e trace=process

只跟踪有关进程控制的系统调用.

-e trace=network

跟踪与网络有关的所有系统调用.

-e strace=signal

跟踪所有与系统信号有关的系统调用

-e trace=ipc

跟踪所有与进程通讯有关的系统调用

-e abbrev=set

设定strace输出的系统调用的结果集.-v 等与 abbrev=none.默认为abbrev=all.

-e raw=set

将指定的系统调用的参数以十六进制显示.

-e signal=set

指定跟踪的系统信号.默认为all.如signal=!SIGIO(或者signal=!io),表示不跟踪SIGIO信号.

-e read=set

输出从指定文件中读出的数据.例如:

-e read=3,5

-e write=set

输出写入到指定文件中的数据.

-o filename

将strace的输出写入文件filename

-p pid

跟踪指定的进程pid.

-s strsize

指定输出的字符串的最大长度.默认为 32.文件名一直全部输出.

-u username

以username的UID和GID执行被跟踪的命令.

用 strace 调试程序

在理想世界里，每当一个程序不能正常执行一个功能时，它就会给出一个有用的错误提示，告诉你在足够的改正错误的线索。但遗憾的是，我们不是生活在理想世界里，起码不总是生活在理想世界里。有时候一个程序出现了问题，你无法找到原因。

这就是调试程序出现的原因。strace 是一个必不可少的调试工具，strace 用来监视系统调用。你不仅可以调试一个新开始的程序，也可以调试一个已经在运行的程序（把 strace 绑定到一个已有的 PID 上面）。

首先让我们看一个真实的例子：

[BOLD]启动 KDE 时出现问题[/BOLD]

前一段时间，我在启动 KDE 的时候出了问题，KDE 的错误信息无法给我任何有帮助的线索。

代码：

```
_KDE_IceTransSocketCreateListener: failed to bind listener
_KDE_IceTransSocketUNIXCreateListener: ...SocketCreateListener() failed
_KDE_IceTransMakeAllCOTSServerListeners: failed to create listener for local

Cannot establish any listening sockets DCOPServer self-test failed.
```

对我来说这个错误信息没有太多意义，只是一个对 KDE 来说至关重要的负责进程间通信的程序无法启动。我还可以知道这个错误和 ICE 协议（Inter Client Exchange）有关，除此之外，我不知道什么是 KDE 启动出错的原因。

我决定采用 strace 看一下在启动 dcopserver 时到底程序做了什么：

代码：

```
strace -f -F -o ~/dcop-strace.txt dcopserver
```

这里 -f -F 选项告诉 strace 同时跟踪 fork 和 vfork 出来的进程，-o 选项把所有 strace 输出写到~/dcop-strace.txt 里面，dcopserver 是要启动和调试的程序。

再次出现错误之后，我检查了错误输出文件 dcop-strace.txt，文件里有很多系统调用的记录。在程序运行出错前的有关记录如下：

代码：

```
```

```
27207 mkdir("/tmp/.ICE-unix", 0777) = -1 EEXIST (File exists)
27207 lstat64("/tmp/.ICE-unix", {st_mode=S_IFDIR|S_ISVTX|0755, st_size=4096, ...}) = 0
27207 unlink("/tmp/.ICE-unix/dcop27207-1066844596") = -1 ENOENT (No such file or directory)
27207 bind(3, {sin_family=AF_UNIX, path="/tmp/.ICE-unix/dcop27207-1066844596"}, 38) = -1 EACCES (Permission
denied)
27207 write(2, "_KDE_IceTrans", 13) = 13
27207 write(2, "SocketCreateListener: failed to "..., 46) = 46
27207 close(3) = 0 27207 write(2, "_KDE_IceTrans", 13) = 13
27207 write(2, "SocketUNIXCreateListener: ...Soc"..., 59) = 59
27207 umask(0) = 0 27207 write(2, "_KDE_IceTrans", 13) = 13
27207 write(2, "MakeAllCOTSServerListeners: fail"..., 64) = 64
27207 write(2, "Cannot establish any listening s"..., 39) = 39
```

其中第一行显示程序试图创建/tmp/.ICE-unix 目录，权限为 0777，这个操作因为目录已经存在而失败了。第二个系统调用（lstat64）检查了目录状态，并显示这个目录的权限是 0755，这里出现了第一个程序运行错误的线索：程序试图创建属性为 0777 的目录，但是已经存在了一个属性为 0755 的目录。第三个系统调用（unlink）试图删除一个文件，但是这个文件并不存在。这并不奇怪，因为这个操作只是试图删掉可能存在的老文件。

但是，第四行确认了错误所在。他试图绑定到/tmp/.ICE-unix/dcop27207-1066844596，但是出现了拒绝访问错误。. ICE_unix 目录的用户和组都是 root，并且只有所有者具有写权限。一个非 root 用户无法在这个目录下面建立文件，如果把目录属性改成 0777，则前面的操作有可能可以执行，而这正是第一步错误出现时进行过的操作。

所以我运行了 chmod 0777 /tmp/.ICE-unix 之后 KDE 就可以正常启动了，问题解决了，用 strace 进行跟踪调试只需要花很短的几分钟时间跟踪程序运行，然后检查并分析输出文件。

说明：运行 chmod 0777 只是一个测试，一般不要把一个目录设置成所有用户可读写，同时不设置粘滞

位(sticky bit)。给目录设置粘滞位可以阻止一个用户随意删除可写目录下面其他人的文件。一般你会发现/tmp 目录因为这个原因设置了粘滞位。KDE 可以正常启动之后，运行 chmod +t /tmp/.ICE-unix 给.ICE_unix 设置粘滞位。

[BOLD]解决库依赖问题[/BOLD]

starce 的另一个用处是解决和动态库相关的问题。当对一个可执行文件运行 ldd 时，它会告诉你程序使用的动态库和找到动态库的位置。但是如果你正在使用一个比较老的 glibc 版本（2.2 或更早），你可能会有一个有 bug 的 ldd 程序，它可能会报告在一个目录下发现一个动态库，但是真正运行程序时动态连接程序（/lib/ld-linux.so.2）却可能到另外一个目录去找动态连接库。这通常因为/etc/ld.so.conf 和 /etc/ld.so.cache 文件不一致，或者/etc/ld.so.cache 被破坏。在 glibc 2.3.2 版本上这个错误不会出现，可能 ld-linux 的这个 bug 已经被解决了。

尽管这样，ldd 并不能把所有程序依赖的动态库列出来，系统调用 dlopen 可以在需要的时候自动调入需要的动态库，而这些库可能不会被 ldd 列出来。作为 glibc 的一部分的 NSS（Name Server Switch）库就是一个典型的例子，NSS 的一个作用就是告诉应用程序到哪里去寻找系统帐号数据库。应用程序不会直接连接到 NSS 库，glibc 则会通过 dlopen 自动调入 NSS 库。如果这样的库偶然丢失，你不会被告知存在库依赖问题，但这样的程序就无法通过用户名解析得到用户 ID 了。让我们看一个例子：

whoami 程序会给出你自己的用户名，这个程序在一些需要知道运行程序的真正用户的脚本程序里面非常有用，whoami 的一个示例输出如下：

代码：

```
# whoami
root
```

假设因为某种原因在升级 glibc 的过程中负责用户名和用户 ID 转换的库 NSS 丢失，我们可以通过把 nss 库改名来模拟这个环境：

代码：

```
# mv /lib/libnss_files.so.2 /lib/libnss_files.so.2.backup
# whoami
whoami: cannot find username for UID 0
```

这里你可以看到，运行 whoami 时出现了错误，ldd 程序的输出不会提供有用的帮助：

代码：

```
# ldd /usr/bin/whoami
libc.so.6 => /lib/libc.so.6 (0x4001f000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

你只会看到 whoami 依赖 Libc.so.6 和 ld-linux.so.2，它没有给出运行 whoami 所必须的其他库。这里时用 strace 跟踪 whoami 时的输出：

代码：

```
strace -o whoami-strace.txt whoami

open("/lib/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/lib/i686/mmx/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/i686/mmx", 0xbffff190) = -1 ENOENT (No such file or directory)
open("/lib/i686/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/i686", 0xbffff190) = -1 ENOENT (No such file or directory)
```

```
open("/lib/mmx/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/mmx", 0xbffff190) = -1 ENOENT (No such file or directory)
open("/lib/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib", {st_mode=S_IFDIR|0755, st_size=2352, ...}) = 0
open("/usr/lib/i686/mmx/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/usr/lib/i686/mmx", 0xbffff190) = -1 ENOENT (No such file or directory)
open("/usr/lib/i686/libnss_files.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
```

你可以发现在不同目录下面查找 libnss.so.2 的尝试，但是都失败了。如果没有 strace 这样的工具，很难发现这个错误是由于缺少动态库造成的。现在只需要找到 libnss.so.2 并把它放回到正确的位置就可以了。

[BOLD]限制 strace 只跟踪特定的系统调用[/BOLD]

如果你已经知道你要找什么，你可以让 strace 只跟踪一些类型的系统调用。例如，你需要看看在 configure 脚本里面执行的程序，你需要监视的系统调用就是 execve。让 strace 只记录 execve 的调用用这个命令：

代码：

```
strace -f -o configure-strace.txt -e execve ./configure
```

部分输出结果为：

代码：

```
2720 execve("/usr/bin/expr", ["expr", "a", ":", "(a)"], [/* 31 vars */]) = 0
2725 execve("/bin/basename", ["basename", "./configure"], [/* 31 vars */]) = 0
2726 execve("/bin/chmod", ["chmod", "+x", "conftest.sh"], [/* 31 vars */]) = 0
```

```
2729 execve("/bin/rm", ["rm", "-f", "conftest.sh"], [/* 31 vars */]) = 0
2731 execve("/usr/bin/expr", ["expr", "99", "+", "1"], [/* 31 vars */]) = 0
2736 execve("/bin/ln", ["ln", "-s", "conf2693.file", "conf2693"], [/* 31 vars */]) = 0
```

你已经看到了，strace 不仅可以被程序员使用，普通系统管理员和用户也可以使用 strace 来调试系统错误。必须承认，strace 的输出不总是容易理解，但是很多输出对大多数人来说是不重要的。你会慢慢学会从大量输出中找到你可能需要的信息，像权限错误，文件未找到之类的，那时 strace 就会成为一个有力的工具了。