

Using Strace To Debug Application Errors

Recently I inherited ownership of an SVN server which was misbehaving. Trying to determine why it wasn't working correctly involved a few hours of testing, careful thought, and caffeine. Eventually I got it working correctly using the often-overlooked tool strace.

strace is a common tool upon many GNU/Linux systems including Debian. Put simply strace is a "system call tracer" - which is where it gets its name from.

Using strace, as root, you can monitor the system calls made by any process upon your system. This can be enormously beneficial when you have a misbehaving program.

If you don't have it installed you can install it as easily with apt-get (or aptitude):

```
root@itchy:~# apt-get install strace
Reading package lists... Done
Building dependency tree... Done
The following NEW packages will be installed
  strace
0 upgraded, 1 newly installed, 0 to remove and 1 not upgraded.
Need to get 92.7kB of archives.
After unpacking 213kB of additional disk space will be used.
Get: 1 http://http.us.debian.org sid/main strace 4.5.12-1 [92.7kB]
Fetched 92.7kB in 1s (81.8kB/s)
Selecting previously deselected package strace.
(Reading database ... 73032 files and directories currently installed.)
Unpacking strace (from ../strace_4.5.12-1_i386.deb) ...
Setting up strace (4.5.12-1) ...
root@itchy:~#
```

Now that you have it installed you can experiment with using it. The usage is:

```
strace [strace options] program [program arguments]
```

You run strace giving it the name of a binary to run, and any arguments that binary might take. As a simple example take a look at running /usr/bin/uptime via strace:

```
strace /usr/bin/uptime
```

Output will immediately fill your screen, and will probably look something like this:

```
execve("/usr/bin/uptime", ["uptime"], [/* 15 vars */]) = 0
uname({sys="Linux", node="itchy", ...}) = 0
brk(0) = 0x804a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
..
..
snip
..
..
open("/proc/uptime", O_RDONLY) = 3
lseek(3, 0, SEEK_SET) = 0
read(3, "114232.96 111223.25\n", 1023) = 20
access("/var/run/utmpx", F_OK) = -1 ENOENT (No such file or directory)
open("/var/run/utmp", O_RDWR) = 4
fcntl64(4, F_GETFD) = 0
fcntl64(4, F_SETFD, FD_CLOEXEC) = 0
_llseek(4, 0, [0], SEEK_SET) = 0
```

Using Strace To Debug Application Errors

```
..
..
snip
..
..
open("/proc/loadavg", O_RDONLY)          = 4
lseek(4, 0, SEEK_SET)                   = 0
read(4, "0.03 0.07 0.02 1/122 11930\n", 1023) = 27
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fe8000
write(1, " 19:37:34 up 1 day,  7:43,  3 us"... , 69) = 69
munmap(0xb7fe8000, 4096)                 = 0
exit_group(0)                             = ?
```

I've snipped the loading of several library files, but even with that truncated output you can see a lot of actions taking place such as:

- The opening of files.
- The return codes of things which failed.
- The writing of text to the console.

Each of the lines of output we see correspond to the invocation of a particular kernel system call. These system-calls are the primitives which are used by the C runtime, and are the kernels interface to userspace.

For example the "open" syscall is used to open files, etc. We can see several details of how the uname binary operates - because we can see that it opens several local files for reading and writing. Some of these fail and others succeed:

- It tries to access /etc/ld.so.nohwcap - which fails
- It tries to access /etc/ld.so.preload - which also fails.
- It opens /proc/loadavg which succeeds and returns the filehandle "4".
- It writes to filehandle 1 the string "19:37:34 up 1 day, 7:43, 3 us"... which is the output of the program.

Because the output can be quite unwieldy dumped to the console it is common to redirect this text to a file, via the -o option. For example to record the system calls used by the ls command to the file /tmp/ls-trace we can execute:

```
strace -o /tmp/ls-trace /bin/ls
```

Another common option is the "-p", or PID, option. This allows you to connect to a running program and see its output. This is useful in the case of long-running daemons which you cannot restart easily - or which only need to be monitored very rarely.

For more details of the available options please see the manpage by running "man strace".

Returning back to the subversion problem we'll see how useful strace was to me. The basic problem was simple enough:

- An SVN server was setup in the dim and distant past.
- Clients running Microsoft Windows connected to this repository via the tortoise SVN client.

Using Strace To Debug Application Errors

- Randomly SVN clients would "hang" and once they did so nobody could access the SVN server again - not just the hung client all clients in the office would be unable to use SVN.
- Rebooting the server "fixed" it, but not permanently.

Initially I tried looking for logfiles, but there weren't any.

Once I excluded that I tried to see how things were setup. The basic installation seemed simple enough. Clients had SSH access and the SVN client appeared to essentially run:

```
ssh username@svnRepository svnservice
```

This svnservice process was responsible for actually conducting the operations, and for some reason was hanging. Since there were no useful debugging options that I could see for altering svnservice I resorted to a hack.

Rather than asking the clients to adjust their setup to add logging, or verbosity (which might not have shown the problem anyway) I moved the svnservice binary to a safe name:

```
# mv /usr/bin/svnservice /usr/bin/svnservice.real
```

Once that was done I wrote small shell script which would run the real binary under strace so I could try to see why it was hanging, or stalling:

```
#!/bin/sh
#
# temporary wrapper for svnservice, log output to /tmp
#
strace -o/tmp/svnservice.$$ /usr/bin/svnservice.real $*
```

This script, when made executable, meant that clients could still connect and run the svnservice command they expected to use - but that now I'd have a strace logfile generated in /tmp for each invocation.

Note that there is no quoting of the arguments to the child process - something that wasn't an issue here.

After a few clients had connected I could immediately see the problem as each logfile ended with the lines:

```
...
...
open("/dev/random", O_RDONLY) = 5
read(5,
```

Here we see that the client attempted to open the file /dev/random to read some random data - this open succeeded and the client was given the filehandle number 5. Unfortunately the reading of this filehandle failed. (We can see that it failed since the read call didn't return.)

/dev/random stalls if there isn't enough entropy available to generate a random number stream. This explains why rebooting the host fixed the problem temporarily - since rebooting would cause this pool of entropy to fill as the system started back up.

Using Strace To Debug Application Errors

A real solution to this problem would be to investigate why there wasn't enough entropy available, but as a simple "hotfix" to allow things to work it was sufficient to link /dev/random to /dev/urandom - which doesn't stall when entropy isn't available.

In many cases this might give you security weaknesses, especially if you need a good random number stream, but for this particular server I don't believe there to be any significant problems.

Tracking down this problem and creating a fix would have been almost impossible without the ability to use strace or a debugger - and using strace is much simpler.

The idea of using a shell script as a wrapper around commands which you cannot modify is also a neat thing to remember. I first saw it used by a colleague a couple of years ago and thought it was an *inspired* solution at the time, and still do now..

(Just remember to put things back the way they were when you're done!)