

Five Simple Ways to Troubleshoot Using Strace

I keep being surprised how few people are aware of all the things they can use strace for. It's always one of the first debug tools I pull out, because it's usually available on the Linux systems I run, and it can be used to troubleshoot such a wide variety of problems.

What is strace?

Strace is quite simply a tool that traces the execution of system calls. In its simplest form it can trace the execution of a binary from start to end, and output a line of text with the name of the system call, the arguments and the return value for every system call over the lifetime of the process.

But it can do a lot more:

- It can filter based on the specific system call or groups of system calls
- It can profile the use of system calls by tallying up the number of times a specific system call is used, and the time taken, and the number of successes and errors.
- It traces signals sent to the process.
- It can attach to any running process by pid.

If you've used other Unix systems, this is similar to "truss". Another (much more comprehensive) is Sun's Dtrace.

How To Use It

This is just scratching the surface, and in no particular order of importance:

1. Find out which config files a program reads on startup

Ever tried figuring out why some program doesn't read the config file you thought it should? Had to wrestle with custom compiled or distro-specific binaries that read their config from what you consider the "wrong" location?

The naive approach:

```
$ strace php 2>&1 | grep php.ini
open("/usr/local/bin/php.ini", O_RDONLY) = -1 ENOENT (No such file or
directory)
open("/usr/local/lib/php.ini", O_RDONLY) = 4
lstat64("/usr/local/lib/php.ini", {st_mode=S_IFLNK|0777, st_size=27, ...}) = 0
readlink("/usr/local/lib/php.ini", "/usr/local/Zend/etc/php.ini", 4096) = 27
lstat64("/usr/local/Zend/etc/php.ini", {st_mode=S_IFREG|0664,
st_size=40971, ...}) = 0
```

So this version of PHP reads php.ini from /usr/local/lib/php.ini (but it tries /usr/local/bin first).

The more sophisticated approach if I only care about a specific syscall:

```
$ strace -e open php 2>&1 | grep php.ini
open("/usr/local/bin/php.ini", O_RDONLY) = -1 ENOENT (No such file or
directory)
open("/usr/local/lib/php.ini", O_RDONLY) = 4
```

The same approach work for a lot of other things. Have multiple versions of a library installed at different paths and wonder exactly which actually gets loaded? etc.

Five Simple Ways to Troubleshoot Using Strace

2. Why does this program not open my file?

Ever run into a program that silently refuse to read a file it doesn't have read access to, but you only figured out after swearing for ages because you thought it didn't actually find the file? Well, you already know what to do:

```
$ strace -e open,access 2>&1 | grep your-filename
```

Look for an open() or access() syscall that fails

3. What is that process doing RIGHT NOW?

Ever had a process suddenly hog lots of CPU? Or had a process seem to be hanging? Then you find the pid, and do this:

```
root@dev:~# strace -p 15427
Process 15427 attached - interrupt to quit
futex(0x402f4900, FUTEX_WAIT, 2, NULL
Process 15427 detached
```

Ah. So in this case it's hanging in a call to futex(). Incidentally in this case it doesn't tell us all that much - hanging on a futex can be caused by a lot of things (a futex is a locking mechanism in the Linux kernel). The above is from a normally working but idle Apache child process that's just waiting to be handed a request.

But "strace -p" is highly useful because it removes a lot of guesswork, and often removes the need for restarting an app with more extensive logging (or even recompile it).

4. What is taking time?

You can always recompile an app with profiling turned on, and for accurate information, especially about what parts of your own code that is taking time that is what you should do. But often it is tremendously useful to be able to just quickly attach strace to a process to see what it's currently spending time on, especially to diagnose problems. Is that 90% CPU use because it's actually doing real work, or is something spinning out of control.

Here's what you do:

```
root@dev:~# strace -c -p 11084
Process 11084 attached - interrupt to quit
Process 11084 detached
% time      seconds  usecs/call   calls   errors syscall
-----
 94.59     0.001014      48         21         0  select
  2.89     0.000031       1         21         0  getppid
  2.52     0.000027       1         21         0  time
-----
100.00     0.001072                63         0  total
root@dev:~#
```

After you've started strace with -c -p you just wait for as long as you care to, and then exit with ctrl-c. Strace will spit out profiling data as above.

In this case, it's an idle Postgres "postmaster" process that's spending most of it's time quietly waiting in select(). In this case it's calling getppid() and time() in between each select() call, which is a fairly standard event loop.

Five Simple Ways to Troubleshoot Using Strace

You can also run this "start to finish", here with "ls":

```
root@dev:~# strace -c >/dev/null ls
% time      seconds  usecs/call   calls   errors syscall
-----  -
23.62     0.000205      103         2         getdents64
18.78     0.000163       15         11         1 open
15.09     0.000131       19          7         read
12.79     0.000111        7         16         old_mmap
 7.03     0.000061        6         11         close
 4.84     0.000042        11          4         munmap
 4.84     0.000042        11          4         mmap2
 4.03     0.000035         6          6         6 access
 3.80     0.000033         3         11         fstat64
 1.38     0.000012         3          4         brk
 0.92     0.000008         3          3         3 ioctl
 0.69     0.000006         6          1         uname
 0.58     0.000005         5          1         set_thread_area
 0.35     0.000003         3          1         write
 0.35     0.000003         3          1         rt_sigaction
 0.35     0.000003         3          1         fcntl64
 0.23     0.000002         2          1         getrlimit
 0.23     0.000002         2          1         set_tid_address
 0.12     0.000001         1          1         rt_sigprocmask
-----
100.00     0.000868                                87         10 total
```

Pretty much what you'd expect, it spends most of its time in two calls to read the directory entries (only two since it was run on a small directory).

5. Why the **** can't I connect to that server?

Debugging why some process isn't connecting to a remote server can be exceedingly frustrating. DNS can fail, connect can hang, the server might send something unexpected back etc. You can use tcpdump to analyze a lot of that, and that too is a very nice tool, but a lot of the time strace will give you less chatter, simply because it will only ever return data related to the syscalls generated by "your" process. If you're trying to figure out what one of hundreds of running processes connecting to the same database server does for example (where picking out the right connection with tcpdump is a nightmare), strace makes life a lot easier.

This is an example of a trace of "nc" connecting to www.news.com on port 80 without any problems:

```
$ strace -e poll,select,connect,recvfrom,sendto nc www.news.com 80
sendto(3, "\24\0\0\0\26\0\1\3\255\373NH\0\0\0\0\0\0\0\0", 20, 0,
{sa_family=AF_NETLINK, pid=0, groups=00000000}, 12) = 20
connect(3, {sa_family=AF_FILE, path="/var/run/nscd/socket"}, 110) = -1 ENOENT
(No such file or directory)
connect(3, {sa_family=AF_FILE, path="/var/run/nscd/socket"}, 110) = -1 ENOENT
(No such file or directory)
connect(3, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("62.30.112.39")}, 28) = 0
poll([{fd=3, events=POLLOUT, revents=POLLOUT}], 1, 0) = 1
sendto(3, "\213\321\1\0\0\1\0\0\0\0\0\0\0\3www\4news\3com\0\0\34\0\1", 30,
MSG_NOSIGNAL, NULL, 0) = 30
poll([{fd=3, events=POLLIN, revents=POLLIN}], 1, 5000) = 1
recvfrom(3,
"\213\321\201\200\0\1\0\1\0\1\0\0\3www\4news\3com\0\0\34\0\1\300\f"... , 1024, 0,
```

Five Simple Ways to Troubleshoot Using Strace

```
{sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("62.30.112.39")},
[16]) = 153
connect(3, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("62.30.112.39")}, 28) = 0
poll([{fd=3, events=POLLOUT, revents=POLLOUT}], 1, 0) = 1
sendto(3, "k\374\1\0\0\1\0\0\0\0\0\0\0\3www\4news\3com\0\0\1\0\1", 30,
MSG_NOSIGNAL, NULL, 0) = 30
poll([{fd=3, events=POLLIN, revents=POLLIN}], 1, 5000) = 1
recvfrom(3, "k\374\201\200\0\1\0\2\0\0\0\0\0\3www\4news\3com\0\0\1\0\1\300\f"...,
1024, 0, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("62.30.112.39")}, [16]) = 106
connect(3, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("62.30.112.39")}, 28) = 0
poll([{fd=3, events=POLLOUT, revents=POLLOUT}], 1, 0) = 1
sendto(3, "\\2\1\0\0\1\0\0\0\0\0\0\0\3www\4news\3com\0\0\1\0\1", 30,
MSG_NOSIGNAL, NULL, 0) = 30
poll([{fd=3, events=POLLIN, revents=POLLIN}], 1, 5000) = 1
recvfrom(3, "\\2\201\200\0\1\0\2\0\0\0\0\0\3www\4news\3com\0\0\1\0\1\300\f"...,
1024, 0, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("62.30.112.39")}, [16]) = 106
connect(3, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("216.239.122.102")}, 16) = -1 EINPROGRESS (Operation now in
progress)
select(4, NULL, [3], NULL, NULL) = 1 (out [3])
```

So what happens here?

Notice the connection attempts to `/var/run/nscd/socket`? They mean `nc` first tries to connect to NSCD - the Name Service Cache Daemon - which is usually used in setups that rely on NIS, YP, LDAP or similar directory protocols for name lookups. In this case the connects fails.

It then moves on to DNS (DNS is port 53, hence the `"sin_port=htons(53)"` in the following connect. You can see it then does a `"sendto()"` call, sending a DNS packet that contains `www.news.com`. It then reads back a packet. For whatever reason it tries three times, the last with a slightly different request. My best guess why in this case is that `www.news.com` is a CNAME (an "alias"), and the multiple requests may just be an artifact of how `nc` deals with that.

Then in the end, it finally issues a `connect()` to the IP it found. Notice it returns `EINPROGRESS`. That means the connect was non-blocking - `nc` wants to go on processing. It then calls `select()`, which succeeds when the connection was successful.

Try adding "read" and "write" to the list of syscalls given to `strace` and enter a string when connected, and you'll get something like this:

```
read(0, "test\n", 1024) = 5
write(3, "test\n", 5) = 5
poll([{fd=3, events=POLLIN, revents=POLLIN}, {fd=0, events=POLLIN}], 2, -1) = 1
read(3, "
```

This shows it reading "test" + linefeed from standard in, and writing it back out to the network connection, then calling `poll()` to wait for a reply, reading the reply from the network connection and writing it to standard out. Everything seems to be working right.