shortcut

Addison-Wesley Professional Ruby Series

Troubleshooting Ruby Processes

Leveraging System Tools When the Usual Ruby Tricks Stop Working

Philippe Hanrigou

Addison-Wesley
Pearson Education

www.informIT.com/Ruby

Visit us on the Web: www.awprofessional.com

ISBN 0-321-54468-4
ISBN13 978-0-321-54468-1

Second release, December 2007

Checking That You Are Using a Native Database Driver

If you are using a MySQL database from Ruby and plan to deploy your application in production, make sure that you use the native MySQL driver (`mysql` gem). The pure Ruby MySQL driver bundled with Rails is convenient for getting started, but it is not very stable (http://rubyforge.org/forum/forum.php?thread_id=9974&forum_id=5450) and runs significantly slower than the native one.

One way to check that the MySQL native drivers are installed properly for a Rails application is to launch `script/console` and make sure that `require_library_or_gem 'mysql'` succeeds. Nevertheless, `lsof` provides a quicker and bulletproof alternative. Just make sure that your Ruby process loaded the `mysql` native library by typing the following:

```
lsof -p <pid> -a -d mem | grep mysql
```

Replace `<pid>` in the preceding command with the `PID` of your Ruby process. If you are using the native drivers, you should see a line similar to this:

```
mongrel_r 20384  www mem    REG    8,2    91302 625398 /usr/local/lib/ruby/site_ruby/1.8/
  i686-linux/mysql.so
```

Detecting Connection Leaks

You can turn `lsof` into a powerful monitoring tool by having it periodically refresh its output. Use the `-r` option:

```
lsof -r 2 -p <pid> -a -i TCP
```

The preceding command monitors all TCP connections for a particular process, refreshing its output every 2 seconds. That can be useful for detecting connection leaks.

If you suspect that your Rails application might be leaking Oracle database connections, you could launch.

```
lsof -r 10 -c mongrel -a -i :1521
```

(Oracle typically listens on port `1521`).

You can then do the following.

- Warm up your application with a reasonable load.
- Look at `lsof` output and count the number of connections.
- Put your application under load for a while.
- Check `lsof` output again and verify that the number of connections is the same; if not, it is likely that you do have a connection leak.

You can also use a similar technique to check that your application is not leaking file descriptors and is closing files properly.

Exploring Other Tricks

Although it's possible to come up with tons of examples on how to use `lsof` to solve a full range of problems, I want to keep this shortcut reasonably focused. I encourage you to explore `lsof` usage on your own and come up with your own tricks. How would you find out the current working directory of a process using `lsof`, for instance?[5]

The Web is also a great resource for discovering ideas on uses for `lsof`: from security audit (www.derks.it/tools.html) to file recovery (http://www.linux.com/articles/58142), to understanding why you can't empty your Mac OS trash can (www.macworld.com/weblogs/macosxhints/2006/05/findinuse/index.php).

Exploring Other `lsof` Options

`lsof` has a lot of options that I won't cover in this shortcut. My objective is to get you interested enough that you start exploring `lsof` on your own. (See the "Usage" section in the "lsof" chapter.) Even the options that I did cover are often more flexible than what I presented. For instance, the `-c` option also understands regular expressions, so you can use it like this:

```
sudo lsof -Pni -a -c '/^ruby|^m.ngrel|^memcache.?$/'
```

There are three modes that I want to point out, though, because they are especially useful: repeat mode, field output, and terse output.

Repeat Mode

As briefly covered when discussing how to use `lsof` to track connection leaks and introducing the `-r` option in the previous section, "Detecting Connection Leaks," you can turn `lsof` into a powerful monitoring tool by using its repeat mode. It is worth noting that using `lsof` repeat mode is more efficient than using the `watch` (www.oreillynet.com/linux/cmd/cmd.csp?path=w/watch) command or a custom shell script, because you avoid the `lsof` startup overhead.

When you use the `-r` repeat mode, `lsof` exits only if it is interrupted (`Ctrl+C`) or receives the quit signal. There is a variant, though: the `+r` repeat mode. When you use it, `lsof` exits the first time that no open file matches the selection criteria. This is useful for triggering a specific action in a script and is often used for supervisory purposes. To make this mode even more script-friendly, `lsof` exit code is meaningful (0 if any open files were ever listed, 1 if none were ever listed).

Finally, scripted usage of the `+r` repeat mode is even more useful when coupled with "field output," which we cover next.

Field Output

By default, `lsof` output is formatted to be easily read by a human in a terminal window. This mode is usually called "formatted display."

You can change this behavior by switching to field output mode designed for scripting use. `lsof` output then produces output that other programs can easily parse. Read the "Output for Other Programs" and the "-F" sections of the `lsof` man page (www.netadmintools.com/html/lsof.man.html) for more details. One of the cool things about "field output" mode is that it is relatively homogeneous across UNIX dialects, making it easier to write portable scripts.

Terse Output

Another useful feature for scripting with `lsof` is the terse output mode. When you activate it with the `-t` option, `lsof` does not output the first line header, suppress all warning messages, and only output the `pids` of the processes with open files matching the selection criteria. This mode is especially useful when used in combination with the `kill` command. The command

```
kill -9 'lsof -t /tmp/obscure.lock'
```

stops any process holding up the `/tmp/obscure.lock` file, for instance.

What Is `lsof` Good For?

`lsof` is especially good for troubleshooting problems related to file access, network access, and native libraries. It essentially provides a static view of resource usage. If you need a better understanding of the dynamics of the process or have a history of system resource usage, it is time to switch to `strace`.

strace

What Is It?

`strace` is a tool that traces all system calls and signals.[6] This refers to all data flow (inputs and outputs) across the user/kernel boundary. If this sounds like Greek to you, think of `strace` as a spy that tells you everything your process asks your operating system to do and what the operating system triggers in your process. As soon as you start playing with `strace`, you realize that your process is interacting with the OS *much* more than you think.

I Cannot Find `strace` on My System!

If you are running OS X, Windows, or BSD, chances are that you will not find versions of `strace` installed on your system. I will soon cover some alternatives in the sibling section on page 35. Keep reading, though, because most of the content here is not `strace`-specific.

The most basic way of using `strace` is to prefix the launching command of the process you want to troubleshoot with `strace`. For instance:

```
strace ruby ./script/server -e test
```

Don't try this yet, though. By default, `strace` writes all its output to standard error and is *extremely* verbose. Therefore, when running `strace`, redirect its output to a file (`trace.log` for instance):

```
strace -o strace.log ruby ./script/server -e test
```

In practice, most of the time you use `strace` to diagnose a live process that has already been launched. Typical scenarios include these:

- You have to diagnose an already-running process that starts to misbehave.
- You launch a process in your desired state and *only then* start tracing system calls.

In these cases, launch `strace` with the following:

```
strace -o strace.log -p <pid>
```

where you replace `<pid>` with the PID[7] of the live process you want to diagnose.

What's the Difference Between Signals and System Calls?

If you feel like you need more background on UNIX signals and system calls, you can start with this introduction (http://ph7spot.com/publications/troubleshooting_ruby_processes/introduction_to_unix_signals_and_system_calls).

Interpreting `strace` Output

If you do not have some knowledge of common POSIX system calls, `strace` output can be somewhat intimidating. The good news is that many problems can be diagnosed simply with a basic