Swamped by Automation

Whenever someone asks you to trust them, don't.

Dear KV,

As part of a recent push to automate everything from test builds to documentation updates, my group—at the request of one of our development groups—deployed a job-scheduling system. The idea behind the deployment is that anyone should be able to set up a periodic job to run in order to do some work that takes a long time, but that isn't absolutely critical to the day-to-day work of the company. It's a way of avoiding having people run cron jobs on their desktops and of providing a centralized set of background processing services.

There are a couple of problems with the system, though. The first is that it's very resource-intensive, particularly in terms of memory use. The second is that no one in our group knows how it works, or how it's really used, but only how to deploy it on our servers—every week or so someone uses the system in a new and unexpected way, which then breaks the system for all the previous users. The people who use the system are too busy to explain how they use it, which actually defeats the main reason we deployed it in the first place—to save them time. The documentation isn't very good either. No one in the group that supports the system has the time to read and understand the source code, but we have to do something to learn how the system works and how it scales in order to save ourselves from this code. Can you shed some light on how to proceed without becoming mired in the code?

Swamped

Dear Swamped,

So your group fell for the “just install this software and things will be great” ploy. It's an old trick that continues to snag sysadmins and others who have supporting roles around developers. Whenever someone asks you to trust them, don't. Cynical as that might be, it's better than being suckered.

But now that you've been suckered, how do you un-sucker yourselves? While wading through thousands of lines of unknown code of dubious provenance is the normal approach to such a problem—a sort of “suck it up” effort—there are some other ways of trying to understand the system without starting from `main()` and reading every function.

The first is to build a second system, just for yourselves, and create a set of typical test jobs for your environment. The second is to use the system already in place to test how far you can push it. In both cases, you will want to instrument the machine so that you can measure the effect that adding work has on the system.

Once you have the set of test jobs or you're running on the production machine, you instrument your machine(s) to measure the effect each job has on the system. In your original question, you say that memory is one of the things the job-control system uses in large amounts, so that's the first thing to look at. How much real memory, not virtual, does the system use when you add a job.

If you add two jobs, does it take twice as much? What about three? How does the memory usage scale? Once you can graph how the memory usage scales, you can get an idea of how much work the system can take before you start to have memory problems. You should continue to add work until the system begins to swap, at which point you'll know the memory limit of the system.

Do not make the mistake of trying only one or two jobs—go all the way to the limit of the system, because there are effects that you will not find with only a small amount of work. If the system had failed with one or two jobs, you wouldn't have deployed it at all, right? Please tell me that's right.

Another thing to measure is what happens when a job ends. Does the memory get freed? On most modern systems you will not see memory freed until another program needs memory, so you'll have to test by running jobs until the system swaps, then remove all the jobs, and then add the same number of jobs again. Does the system swap with fewer jobs after the warm-up run? The system may have a memory leak. If you can't fix the leak, then guess what, you will get to reboot the system periodically, since you're unlikely to have time to find the leak yourself.

When you're trying to understand how a system scales, it's also good to look at how it uses resources other than memory. All systems have simple tools to look at CPU utilization, and you should, of course, make sure that the job-control system is the one taking all the CPU time, as that adds to the total system overhead.

The files and network resources a system uses can be understood using programs such as `netstat` and `procstat`, as well as `lsof`. Does the system open lots of files and just leave them? That's a waste of resources you need to know about, because most operating systems limit the number of open files a process can have. Is the system disk-intensive, or does it use lock files for a lot of work? A system that uses lots of lock files needs to have space on a local, non-networked disk for the lock files, as network file systems are particularly bad at file locking.

A rather drastic measure, and one that I favor, is the use of `ktrace`, `strace`, and particularly `DTrace` to figure out just what a program is doing to a system. The first two will definitely slow down the system they are measuring, but they can quickly show you what a program is doing, including the system calls it makes when waiting for I/O to complete, plus what files it's using, etc. On systems that support `DTrace`, the overhead of tracing is reduced, and on a system that is not latency-sensitive, it's acceptable to do a great deal more tracing with `DTrace` than with either `ktrace` or `strace`. There is even a script, `dtruss`, provided with `DTrace`, that works like `ktrace` or `strace`, but that has the lower overhead associated with `DTrace`. If you want to know what a program is doing without tiptoeing through the source code, I strongly recommend using some form of tracing.

In the end it's always better to understand the goals of a system, but with engineers and programmers being who they are, this might be like pulling teeth. Not that pulling teeth isn't fun—trust me, I've done it—but it's more work than it looks like and sometimes the tooth fairy doesn't give you that extra buck for all your hard work.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2013 ACM 1542-7730/13/0200 \$10.00