acmqueue

File-system Litter

Cleaning up your storage space quickly and efficiently

Dear KV,

We recently ran out of storage space on a very large file server—one with many terabytes of space—and upon closer inspection we found that it was just one employee who had used it all up. The space was taken up almost exclusively by small files that were the result of running some data-analysis scripts. These files were completely unnecessary after they had been read once. The code that generated the files had no good way of cleaning them up once they had been created; it just went on believing that storage was infinite. Now we've had to put quotas on our file servers and, of course, deal with weekly cries for more disk space. Surely there is a better way of dealing with this problem than clamping down on everyone for fear that one of them will do the wrong thing.

Caught Between a Block and a Lack of Space

Dear Caught,

Yes, there are better ways of handling this problem, but don't call me Shirley. You have now discovered one of the drawbacks of cheap storage (and yes, that old adage is true): files will always expand to fill the available storage space, just as programs expand to fill all available memory and spawn more threads until all of your CPU is utilized as well.

Shared storage, such as you're dealing with, presents the thorniest problem because it is *shared*, and, it would seem—as I'm sure regular readers of this column are aware—people simply cannot be trusted to police themselves. In reality most people can, but it takes just one, as you found out, to “ruin it for everybody,” as our teachers used to say.

The point you make about the scripts not having a way of cleaning up after themselves is a good one. When you build programs out of many small source files your tools also generate intermediate files—the objects that then get linked into a final executable. All build systems worthy of the name, however, have some form of “clean” target. Although this target was originally created so that you could start a new build from scratch, it is also a handy way of shrinking the size of your work area when a project is either complete or on hold. Having a program that would do the same work with intermediate data files is a good start, but there are other things that can be done to improve the situation.

The act of littering the file system with files that have to be deleted later results in a performance problem. If you need to find all the files via recursive descent of the file system before you can delete them, then you're going to be hammering your file system. In the case of NFS (network file system)-mounted systems, you will also be hammering your network while trying to clean up after yourself. Although it might appear that the best course of action would be to delete the files immediately after use, this would prevent you from debugging problems in your data analysis. Also, if you have to rerun some part of the analysis, then the derived objects you created could come in handy in speeding up the second, or third, or—well, you know—the *n*th run before you finally get it right. Probably the best compromise position is to place all of the derived objects into their own directory

or set of directories, which can be easily located and purged when it's time to free up some space on the file system.

Keeping all the files in one place means you don't have to descend the file system recursively to find all the files that can be safely deleted. That's going to make the process easier, faster, and therefore more likely to be used by the people on your system. If cleaning up after yourself takes 30 *seconds*, you're pretty likely to do it; if it requires 30 *minutes*, you're going to put it off as long as you can, usually long enough for the file system to fill up again.

KV

Dear KV,

You've written in previous columns about not using `printf` to debug programs, and you recommended using a debugger, but you must admit that there are times when a print statement is just an easier way of debugging a program and that using a debugger is overkill.

Still Pounding on Printf

Dear Pounding,

True, I have written in previous columns about the reasons for not using print statements for debugging, and I have recommended that people use finer tools such as debuggers to find problems in their programs. There are two instances in which I agree that a print statement is a better solution.

The first instance where print beats a debugger is when either you have no debugger or the debugger itself is incredibly painful to use. I find that this happens often with interpreted languages, probably because adding a print statement and rerunning your program is just so easy that no one ever bothers to write a decent debugger for the language. Compiled languages, on the other hand, usually have debuggers because the time needed to add a print statement and rebuild a large program is longer than it takes to fire up the debugger. An example of this problem is present in my scripting language of choice, Python. I love writing in Python, but I do not love the Python debugger. It has improved over the past few years, likely because bigger and bigger systems are being built in Python, so having a debugger makes finding the bugs easier. As debuggers go, however, the ones for Python are nothing compared with those available for compiled languages.

The second instance where print beats a debugger is one that perhaps most of my readers have not had to experience: bringing up a new piece of hardware. In the not-too-distant past it was uncommon for anyone except a device-driver writer to worry about bringing up new hardware. With more and more people using open source operating systems, however, it has become more common to have to do some level of work with new hardware. I recently experienced this when I bought a new laptop. Of all the things that didn't work when I installed my OS of choice, it happened to be the built-in keyboard that didn't work with the operating system's keyboard driver. It turned out that I could plug in a USB keyboard and boot with the internal keyboard disabled, but that was not quite how I envisioned using my new, light, slick, laptop—with a USB keyboard attached.

I normally don't work on keyboard drivers, but I know the people who did, and I know that there is nothing more frustrating than having a whiny user send you email saying, "The keyboard doesn't work." The driver itself wasn't long, and I knew about where the hang would happen in the code, so I just backtracked from where I thought the hang point was and used an Emacs macro I'd written for just such an occasion:

```
(defun dbgprintf ()
  "Insert a debug printf for kernel debugging."
  (interactive ""*)
  (indent-for-tab-command)
  (insert "printf(\"reached function %s file %s line %d\\n\",\\n")
  (indent-for-tab-command)
  (insert "__func__, __FILE__, __LINE__);\\n"))
```

Attaching this code to a key sequence, I could insert a print statement anywhere in my code, and when it was reached, it would print out the function, filename, and line that had been reached. Using this primitive method, I was able to track down what was causing the system to hang and thus could avoid it, as well as send a much more detailed bug report to the driver maintainer.

Certainly more could be done with this macro. For example:

```
(defun debug-block ()
  "Insert a debug printf inside a C ifdef debug block."
  (interactive ""*)
  (insert "#if defined(DEBUG)\\n")
  (indent-for-tab-command)
  (dbgprintf)
  (insert "#endif /* DEBUG */\\n")
  (indent-for-tab-command)
)
```

This code builds on the previous code to enclose the print statement in a debug block that can be turned on and off from the makefile or command line.

Yes, there are times when you need or want printf, or print statements, but I still say that those times are, hopefully, few and far between.

KV

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.

© 2011 ACM 1542-7730/11/0700 \$10.00