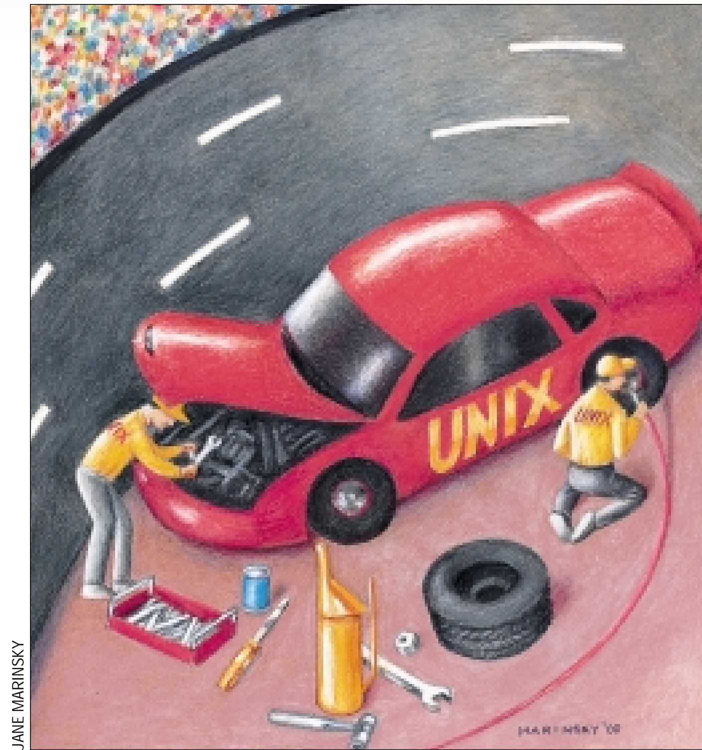
by Peter Collinson, Hillside Systems



JANE MARINSKY

Performance

UNIX started life on what would be considered a very slow machine by today's standards. In fact, I often remark, somewhat wryly, that there is 90% more memory in my printer than there was in the first machine I used to run UNIX. The desire to make a system that would perform well with limited resources generated a software ethos where small is beautiful. "Make each program do one thing well," was an early aphorism from the team at Bell Labs.

The philosophy generated a vocabulary of tools that could be used together to get the job done. A recent visitor to my crowded office grew up in the Windows world (what I often like to refer to as the "One Microsoft Way") and remarked with some surprise about the "tools approach" that I was using to show him around the systems.

The tools are still there, but we ask our systems to do much more for us now. We now typically view the system using a color bitmapped screen that is supported by X11. Designed to be flexible and portable, X11 allows the user to have maximum choice over the look and feel of their own interface to the system.

Most, if not all, UNIX systems now contain TCP/IP networking code, which lead to the development of networked file systems, which in turn lead to the development of "the file system switch," where a UNIX kernel is now able to access a file system formatted with some external standard as if the files were sitting on a native UNIX disk.

Kernels have needed to provide flexibility, and in general, have become more modular, so that it's possible to "mix and match" in the kernel more easily. A module will offer a set of actions that can be used by modules "above it," and will also have a clearly defined way of using the more primitive operations provided by modules that are "below." The UNIX system call interface provides a model of how a file system should operate, and the user process uses this model to handle files. Inside the kernel, successive modules will translate the desired user action into an action on the hardware. Each module provides a general-purpose model that can be simply switched to operate on different hardware.

For example, the user process may ask to open a file, and the request will be passed into code that performs the necessary translation from a file name into the inode that points to the file's information on the disk. These days, the code will consist of several modules that will present the necessary information to the user process, but may end up calling routines that access a local disk, or read something from a CD-ROM, or send a message out to the network to request the file to be opened on another machine. All these actions will happen transparently, as far as the user process is concerned, and are usually structured internally so that adding a new handler for a different type of source file system on possibly different media can be done easily.

Modules are used also to help the programmer. As the kernels became

more complex, some way was needed to create order from the chaos. Modules with clearly defined inputs and outputs mean that the programmer has a better chance of understanding the problem and creating a correct solution. One way of returning to the notion of "doing one thing well" has been the development of the message passing kernel. All actions that the kernel is requested to make are turned into discrete messages, which are sent to a specific server process whose job is to service the request. The job of the server process is intrinsically simple, it sits waiting for messages to arrive, takes the next one, does its job and sends a message back to the caller with the result.

Of course, all these changes sound reasonable. They are all developmental, taking us along some forward path and allowing us to do new things with our systems. There are also a bunch of changes that are there because of politics or history. For example, Sun Microsystems Inc.'s operating system kernels have always attempted to provide maximum software flexibility to user processes. They often support more than one way of doing something because there are programs out there coded with different UNIX standards. Sun aims to support all the programs without source modification, and will also generally support old binaries. As a result, kernels (and more often the user libraries that interface directly with the kernels) will have layers of conversion code.

Finally, there is a whole bunch of what can be termed "creeping featurism." Since the beginning, people have delighted in adding the odd new bit of functionality. Worse, people have often changed the old functionality so that it's "no longer broken." When UNIX System III came out in the early '80s, so many changes had been made to the original UNIX system, that to me, System III didn't "feel" like UNIX.

This has happened again to a certain extent with GNU and Linux. As programmers replicated "standard UNIX ways of doing things," they found bugs and inadequacies that they couldn't help themselves from fixing. Mostly this doesn't matter, as long as there's some way of obtaining the old "broken" way of doing things. GNU software is often good at providing such bug-for-bug compatibility. Other people have simply changed things, and this is a pain, because extant UNIX software no longer works the same way on Linux as it does on other UNIX systems.

A worse situation occurs when a command has been completely reimplemented, and yet retains the same name. Maybe it's performing the same function, which is why it hangs onto the name, however, the options the user can give the command have changed, and the output from the command is altered. It's actually a completely new command that happens to fit into the niche of the old. An example of this misbehavior is the `ps` command, which behaved differently in UNIX System III from the original UNIX release. We have the legacy today on Solaris, where `/usr/bin/ps` is different from `/usr/ucb/ps` (I choose to use the latter).

All of these changes in both the kernel and user processes have not come cheap. They've generally cost us code to implement. User processes have grown in size to cope with new requirements. Kernels are now pretty massive. It would be interesting to compare the number of machine instructions that are executed by, say, a "write" system call in a modern UNIX system with the number that were used in, say, UNIX Version 7. I'll bet that the increase will be staggering. The code bloat costs us memory and processor cycles.

Does it Matter?

You may choose to argue that none of this concern about performance and efficiency matters in the slightest. After all, computer systems are now cheap. I'll guess that I can now buy perhaps 15 or 20 complete PC systems for the price of the original PDP-11 that I used for UNIX. Memory is cheap and getting cheaper. CPU performance increases all the time. The code bloat doesn't remove the benefits of these improvements completely from users, they are able to see that the processors are faster and watch their programs run more quickly on the new hardware. Programming efficiency and size is no longer a primary goal, what's important is features and usability.

I'm in sympathy with this line of argument, but I tend to doubt the actuality. People don't often go out and buy new computers because their old ones are creaking at the seams. I confess that I did do just that when I upgraded to an Ultra 10 in November. One reason for doing the upgrade was a feeling that the old SPARCstation II would not cope with Solaris 7.

Windows, and especially Windows NT, is noted for its large size. My main NT machine has 64 MB of memory, which is barely enough, but upgrading the machine means throwing all the old memory away (or trading it in), and this seems painful. The layers in the code slows it down, too. I didn't understand how fast my Windows 95 laptop was until I bought a new disk and installed a version of UNIX on it. I couldn't throw away the Window system, my son wanted to play games on it.

I have a SunPCi card in my Ultra 10 and choose to run Windows 95 because it's markedly more lively than Windows NT for the small number of PC applications that I use. This may be a feature of the different ways that the two Windows systems drive the screen, but I doubt it. With NT, you pay for the "New Technology" operating system. It's rare to get the opportunity to run different operating systems on the same PC hardware, so people are probably often unable to make the direct comparison.

So although I am broadly in sympathy with the argument that says we should be worrying about functionality and not resource limits, I still don't buy into the idea completely. Other people seem to feel the same, at least one of my email correspondents in the past few months did care about optimizing the performance of his Suns (sorry I've lost your email and name). He is in the large-scale modelling business and needs to squeeze as much as he can from his systems.

So, what about UNIX?

How can we make our UNIX system run faster? Well, it may not be possible to do that. It may be possible to configure it so that it will not be slowed down. You can certainly work on tuning it for your load, because when "out of the box," UNIX is usually configured for some general-purpose load that may not match your environment. (I give some tuning references at the end of the article.)

Ensuring that there is sufficient memory in the machine to support the tasks that you want it to undertake is perhaps the most important aspect of tuning the system. Setting memory sizes is a little bit of a black art, largely because the way it works is not well-understood. The extant papers require some basic knowledge of how the memory is managed on UNIX, and it seems a good idea to give a starting point for that here.

UNIX was designed as a multiprocessing system. This means that several processes can be running in the machine at any one time. Of course, if the machine has only one CPU, then the reality is only one process will be really running. The remainder will be waiting, perhaps to run, or more usually, for a system call to terminate. Eventually, as new processes start, the kernel will run out of available memory and will have to take some action to make some memory available for the next deserving process.

When the kernel needed some memory in the early UNIX systems, processes that were waiting were “swapped out.” Their entire address space was moved completely from memory to a special area of disk. At some point later, when memory became available, they could be swapped into memory again. A characteristic of these early systems was that the process was split into three fairly large chunks: the memory holding the instructions for the program; the memory holding the data for the program; and one segment for the stack area for the program. All three chunks had to be resident in memory for the process to be able to run.

UNIX was ported to the VAX architecture by the Computer Systems Research Group at the University of California at Berkeley in the early 1980s and this work formed what we now call the BSD releases of UNIX. The BSD systems formed the basis of SunOS, which in turn was merged back into UNIX System V, creating Solaris.

The VAX architecture supports “virtual memory.” The fundamental idea is that all the memory for each process is split into fixed-sized sections called “pages.” The system hardware supports an address mapping system, and so the three large chunks of the earlier processes could now be split into smaller sections, each section being one page in size. The address mapping allows the parts of the process to actually be spread in separate pages all over the “real” memory. So while the process thinks it has a contiguous address space, it doesn’t, the mapping is used to provide this illusion.

The process memory is virtual because the pages can be written out to disk at any time, and the hardware has the hooks to alert the kernel when a process wants to access a page that’s actually out on the disk. In this case, the process is paused, the required page is brought into memory, the address map is adjusted and the process continues.

We now have a system where a process can be split into small page-sized chunks, and at any one time, it will only need some portion of its address space resident in the computer. This portion is known as the “resident set.” Another benefit is “shared libraries.” Because the address space of a process is accessed via an address map, there is no reason why two processes cannot point entries in each of their maps at the same page, “sharing” it.

So, virtual memory gives us the ability to create processes that are larger than the physical memory of the machine, because some portion of the address space of the process can be resident on disk. It also allows us to move information between the disks and memory only when it is needed.

In the lightly loaded state, information should be trickling between the memory and the disks, the system will be mostly reading information from the disks into memory. When things become more hectic, processes will begin to have parts of their address space paged out and will be reduced to their resident set. As we proceed toward heavy loading, processes will be swapped out completely to make space for other processes that need the memory. As the load increases the kernel can get more desperate for memory, and the system can start thrashing, swapping processes in and out in a vain attempt to get the work done.

Thrashing should be avoided on any system, because this means it’s spending more time moving process images about than doing work. I suspect that the EIDE disks on my Ultra 10 will be spectacularly bad at handling this type of load. Actually, I had the case this morning where I had inadvertently left my SunPCi machine on overnight, and worse, had left a Web page displaying a scrolling marquee. The CDE disk performance indicator showed that the machine was engaged in loads of disk activity. Poking about with `ps` showed that nothing obvious was running. I began to wonder if I was being hacked and someone had installed something nasty on the machine to prevent the `ps` command from showing the full activity. I then realized that the activity I was seeing was probably swapping activity. I closed the SunPCi GUI down and the unwanted activity stopped.

Sizing

The explanation of virtual memory above is, of course, much simplified. The kernel adds a bunch of optimizations that are intended to make the system run more quickly. First, the kernel tries to keep as many pages as it can hanging around in memory for as long as it can. The attitude is that once something has been read in from disk, there is no point in getting rid of it unless memory is needed. After all, accessing something in memory is 100,000 times faster than getting it from disk.

Second, recent Solaris kernels will fill up spare kernel memory with file system buffer cache. UNIX kernels have always retained information about open files in memory to provide processes with faster access to the data. The cache used to be a dedicated chunk of kernel memory, but Solaris now stores the

buffer cache information in free pages in the general memory page pool. I tend to notice the effect of this cache in the mornings after a large nighttime system dump when it takes some time to page in the various programs that I want to use, like the X server. Incidentally, you can use the `vmstat` command to look at the performance of the virtual memory system, and typical output shows that the memory is always full. This is the effect of storing the buffer cache in memory, and is not a problem with memory leaks in some program or other.

OK, that's all well and good. How do I decide how much memory to put on my Solaris system, and how much swap space do I need? Well, I am going to punt on the sizing issue and point you at "The Solaris Memory System" available on the Web (see below for URL information), this is a 100-page document and you need to read it somewhat carefully. However, when I bought my Ultra 10 system, the jump in price between 256 and 512 MB seemed a huge one, so I plumped for 256, which has proved to be OK.

Actually, the resident set size of all the background daemons that seem to be considered part of the "system" these days is not small. Typing `ps -ef` or `/usr/ucb/ps ax` on my system shows a huge number of processes that are sitting around waiting for something or other to happen. The X server, `Xsun`, can grow to prodigious proportions.

What about swap and paging space? We tend to call this just swap space, for historical reasons. In the past, the general wisdom from Sun was that you should have as much swap space as you have RAM. In general, this is to allow you to take core dumps of the system should the need arise. Solaris 8 gives you 512 MB as part of the standard Webstart install process. I tend to try and have loads of swap space (I've got about 1 GB on my recently installed Solaris 8) because I like to run `tmpfs`, and have `/tmp` resident in the swap area. Because I have two disks on my system, I have them set up with the same partitions, and swap into the same-sized partition on both. The swapping system interleaves swapping and paging requests between the disks, so each disk should see the same load from paging or swapping that the system needs.

I also tend to overallocate swap space when I install a system. Sometimes the swap space can be a useful resource when you need an extra 20 or 30 MB to install the next system, and systems always seem to grow with every release.

Further Reading

I had intended to get into some discussion on disk and NFS tuning, but as often seems to be the case, space has defeated me. There are several useful references both in print and on the Web if you want to follow up on this topic. First, you can get hold of *Sun Performance and Tuning* by Adrian Cockcroft and Richard Pettit (published by Prentice-Hall Inc., 2nd Edition, 1998, ISBN 0-13-095249-4). I have the first edition, originally printed in 1995, which has dated quite a bit in those five years. However, it still contains much wisdom and information.

Searching for "Performance" on `http://www.sun.com` lead me to `http://www.sun.com/sun-on-net`, which is a useful starting point for investigating more recent work by Cockcroft and others at Sun on performance management. Cockcroft and others also have generated the Sunworld SE3.0 Tuning Toolkit program suite that can be used to look at your system and tell you what parts of it are in need of attention.

Finally, the Answerbook collection contains a section called *NFS Server Performance and Tuning Guide for Sun Hardware*, which contains a plethora of useful tips about how to get the best from workstations whose main files are stored on NFS.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever… He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.