Multithreading with Solaris:

Automated Teller Machine (ATM) Demo

For the past two years, I have been working as a technical support engineer for SunSoft advising our third party developers on the benefits of the Solaris operating environment. I've had many discussions with developers on how to best utilize the features of Solaris 2, which adds multiprocessing and multithreading features that many developers want to learn more about.

With multithreading (MT), applications have access to user-level threads which allow them to concurrently execute independent tasks in the application simultaneously on a multiprocessor machine. On a single processor machine, the application does not have to block I/O since a single thread can be dedicated to do the I/O while the rest of the application continues processing. This gives the application writer higher throughput and better CPU utilization.

As an experienced software engineer, I felt compelled to

show myself just how important multithreading is. In other words, I wanted to put MT to the test by writing a concept demo to demonstrate thread technology. I decided to test a single CPU scenario by writing a simple Automated Teller Machine (ATM) server to perform database I/O and networking I/O.

The demonstration was written using conventional UNIX facilities and then rewritten using user-level threads. The demo consisted of two ATMs communicating to the server, which could reside either on the same machine or on a networked machine. The ATMs could do a read request (BALANCE) or write request (DEPOSIT/WITHDRAW) with the server employing a reader/writer algorithm to fulfill requests. To emulate many teller machines on the network, the rate of request from the tellers was increased to 50 times the normal teller rate to allow for creation of contention in the server. And, to add some realism (and some fun!) to the demo, animated people were added walking up to the ATMs. As the teller requests were made, the ATMs communicated with the display server to show how many transactions had been processed (see Figure 1). This demo was able to achieve 100% CPU utilization of a SPARCstation™ 10 running Solaris 2.2.

Figure 1. Single-threaded version of ATM demo on SPARCstation 10 with 32 MB RAM achieves 238 transactions per minute.

In developing the ATM demo, my first task was to use a conventional Unix API. In the server and the display server, one process was dedicated to listen to the network for ATM requests. Once a request came in, a new process was forked to take care of the request. Since different processes needed to access the database, there were two choices: Either read the database into memory once and share it between the processes, or have each process read the database into its own memory. I chose the latter because it was much easier to implement than shared memory and synchronization.

Next, I developed a locking mechanism to implement the reader/writer locks on the database using the fcntl() system call. This is not a very efficient way to do high speed locking. However, when you have an NFS-based filesystem, creating a high speed lock without corrupting the database can be difficult. I also used signals and pipes to communicate between the processes.

Using Threads

Once a working server was developed, the demo was re-implemented using threads to see how it compared to the original approach. It turned out that user-level threads were very simple to use. The API was made up of a small set of function calls, of which I used only a few. The calls used in the demo are:

- Create a thread
- Use mutex lock
- Use reader/writer lock
- Provide hint to the threads library on desired concurrency.

Since all the threads share the same process address space, the code size was reduced by 20%. The code was also much simpler and cleaner with fewer file descriptors hanging around. To do asynchronous I/O, a single dedicated thread was allowed to block in the read system call. This was much simpler than using aioread(). Threads were far more efficient in the use of system resources than processes. They used less resources and were cheaper for the quick create/destroy scenario used in the server. Since all the threads shared the same address space, there was no need for using shared memory. With this approach, both mutex and reader/writer locks were extremely efficient and comparable to the best locking schemes on the market (see Figure 2).

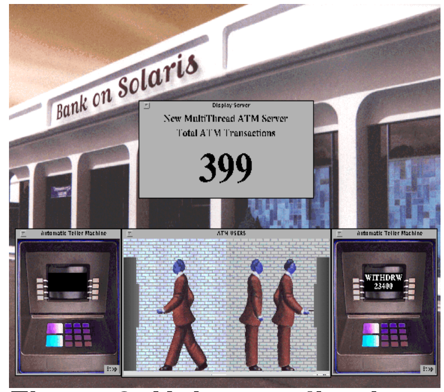

Figure 2. Using application-level threads, the ATM demo provides 399 transactions per minute, or 160 extra transactions. Therefore, on uniprocessor machines, threads achieve better I/O, resulting in higher throughput.

While the demo does not illustrate all the benefits of threads, I was convinced that going with the user-level threads gave significant performance gains. I also found that when a local filesystem is used, a threaded server is 1.6 times faster, whereas when an NFS-based filesystem is used, the threaded server is 10 times faster. The ATM demo achieved its goal in demonstrating enhanced system performance by utilizing threads, which ultimately resulted in lower cost per transaction.