The C10K problem

It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now.

And computers are big, too. You can buy a 1000MHz machine with 2 gigabytes of RAM and an 1000Mbit/sec Ethernet card for $1200 or so. Let's see - at 20000 clients, that's 50KHz, 100Kbytes, and 50Kbits/sec per client. It shouldn't take any more horsepower than that to take four kilobytes from the disk and send them to the network once a second for each of twenty thousand clients. (That works out to $0.08 per client, by the way. Those $100/client licensing fees some operating systems charge are starting to look a little heavy!) So hardware is no longer the bottleneck.

In 1999 one of the busiest ftp sites, cdrom.com, actually handled 10000 clients simultaneously through a Gigabit Ethernet pipe. As of 2001, that same speed is now being offered by several ISPs, who expect it to become increasingly popular with large business customers.

And the thin client model of computing appears to be coming back in style -- this time with the server out on the Internet, serving thousands of clients.

With that in mind, here are a few notes on how to configure operating systems and write code to support thousands of clients. The discussion centers around Unix-like operating systems, as that's my personal area of interest, but Windows is also covered a bit.

Contents

Related Sites

See Nick Black's execellent Fast UNIX Servers page for a circa-2009 look at the situation.

In October 2003, Felix von Leitner put together an excellent web page and presentation about network scalability, complete with benchmarks comparing various networking system calls and operating systems. One of his observations is that the 2.6 Linux kernel really does beat the 2.4 kernel, but there are many, many good graphs that will give the OS developers food for thought for some time. (See also the Slashdot comments; it'll be interesting to see whether anyone does followup benchmarks improving on Felix's results.)

Book to Read First

If you haven't read it already, go out and get a copy of Unix Network Programming : Networking Apis: Sockets and Xti (Volume 1) by the late W. Richard Stevens. It describes many of the I/O strategies and pitfalls related to writing high-performance servers. It even talks about the 'thundering herd' problem. And while you're at it, go read Jeff Darcy's notes on high-performance server design.

(Another book which might be more helpful for those who are *using* rather than *writing* a web server is Building Scalable Web Sites by Cal Henderson.)

I/O frameworks

Prepackaged libraries are available that abstract some of the techniques presented below, insulating your code from the operating system and making it more portable.

- ACE, a heavyweight C++ I/O framework, contains object-oriented implementations of some of these I/O strategies and many other useful things. In particular, his Reactor is an OO way of doing nonblocking I/O, and Proactor is an OO way of doing asynchronous I/O.
- ASIO is an C++ I/O framework which is becoming part of the Boost library. It's like ACE updated for the STL era.

- libevent is a lightweight C I/O framework by Niels Provos. It supports kqueue and select, and soon will support poll and epoll. It's level-triggered only, I think, which has both good and bad sides. Niels has a nice graph of time to handle one event as a function of the number of connections. It shows kqueue and sys_epoll as clear winners.
- My own attempts at lightweight frameworks (sadly, not kept up to date):
    - Poller is a lightweight C++ I/O framework that implements a level-triggered readiness API using whatever underlying readiness API you want (poll, select, /dev/poll, kqueue, or sigio). It's useful for benchmarks that compare the performance of the various APIs. This document links to Poller subclasses below to illustrate how each of the readiness APIs can be used.
    - rn is a lightweight C I/O framework that was my second try after Poller. It's lgpl (so it's easier to use in commercial apps) and C (so it's easier to use in non-C++ apps). It was used in some commercial products.
- Matt Welsh wrote a paper in April 2000 about how to balance the use of worker thread and event-driven techniques when building scalable servers. The paper describes part of his Sandstorm I/O framework.
- Cory Nelson's Scale! library - an async socket, file, and pipe I/O library for Windows

I/O Strategies

Designers of networking software have many options. Here are a few:

- Whether and how to issue multiple I/O calls from a single thread
    - Don't; use blocking/synchronous calls throughout, and possibly use multiple threads or processes to achieve concurrency
    - Use nonblocking calls (e.g. write() on a socket set to O_NONBLOCK) to start I/O, and readiness notification (e.g. poll() or /dev/poll) to know when it's OK to start the next I/O on that channel. Generally only usable with network I/O, not disk I/O.
    - Use asynchronous calls (e.g. aio_write()) to start I/O, and completion notification (e.g. signals or completion ports) to know when the I/O finishes. Good for both network and disk I/O.
- How to control the code servicing each client
    - one process for each client (classic Unix approach, used since 1980 or so)
    - one OS-level thread handles many clients; each client is controlled by:
        - a user-level thread (e.g. GNU state threads, classic Java with green threads)
        - a state machine (a bit esoteric, but popular in some circles; my favorite)
        - a continuation (a bit esoteric, but popular in some circles)
    - one OS-level thread for each client (e.g. classic Java with native threads)
    - one OS-level thread for each active client (e.g. Tomcat with apache front end; NT completion ports; thread pools)

- Whether to use standard O/S services, or put some code into the kernel (e.g. in a custom driver, kernel module, or VxD)

The following five combinations seem to be popular:

1. Serve many clients with each thread, and use nonblocking I/O and level-triggered readiness notification
2. Serve many clients with each thread, and use nonblocking I/O and readiness change notification
3. Serve many clients with each server thread, and use asynchronous I/O
4. serve one client with each server thread, and use blocking I/O
5. Build the server code into the kernel

1. Serve many clients with each thread, and use nonblocking I/O and level-triggered readiness notification

... set nonblocking mode on all network handles, and use select() or poll() to tell which network handle has data waiting. This is the traditional favorite. With this scheme, the kernel tells you whether a file descriptor is ready, whether or not you've done anything with that file descriptor since the last time the kernel told you about it. (The name 'level triggered' comes from computer hardware design; it's the opposite of 'edge triggered'. Jonathon Lemon introduced the terms in his BSDCON 2000 paper on kqueue().)

Note: it's particularly important to remember that readiness notification from the kernel is only a hint; the file descriptor might not be ready anymore when you try to read from it. That's why it's important to use nonblocking mode when using readiness notification.

An important bottleneck in this method is that read() or sendfile() from disk blocks if the page is not in core at the moment; setting nonblocking mode on a disk file handle has no effect. Same thing goes for memory-mapped disk files. The first time a server needs disk I/O, its process blocks, all clients must wait, and that raw nonthreaded performance goes to waste.
This is what asynchronous I/O is for, but on systems that lack AIO, worker threads or processes that do the disk I/O can also get around this bottleneck. One approach is to use memory-mapped files, and if mincore() indicates I/O is needed, ask a worker to do the I/O, and continue handling network traffic. Jef Poskanzer mentions that Pai, Druschel, and Zwaenepoel's 1999 Flash web server uses this trick; they gave a talk at Usenix '99 on it. It looks like mincore() is available in BSD-derived Unixes like FreeBSD and Solaris, but is not part of the Single Unix Specification. It's available as part of Linux as of kernel 2.3.51, thanks to Chuck Lever.

But in November 2003 on the freebsd-hackers list, Vivek Pei et al reported very good results using system-wide profiling of their Flash web server to attack bottlenecks. One bottleneck they found was mincore (guess that wasn't such a good idea after all) Another was the fact that sendfile blocks on disk access; they improved performance by introducing a modified sendfile() that return something like EWOULDBLOCK when the disk page it's fetching is not yet in core. (Not sure how you tell the user the page is now resident... seems to me what's really needed here is aio_sendfile().) The end result of their optimizations is a SpecWeb99 score of about 800 on a 1GHZ/1GB FreeBSD box, which is better than anything on file at spec.org.

There are several ways for a single thread to tell which of a set of nonblocking sockets are ready for I/O:

- The traditional select()
  Unfortunately, select() is limited to FD_SETSIZE handles. This limit is compiled in to the standard library and user programs. (Some versions of the C library let you raise this limit at user app compile time.)

  See Poller_select (cc, h) for an example of how to use select() interchangeably with other readiness notification schemes.

- The traditional poll()
  There is no hardcoded limit to the number of file descriptors poll() can handle, but it does get slow about a few thousand, since most of the file descriptors are idle at any one time, and scanning through thousands of file descriptors takes time.

  Some OS's (e.g. Solaris 8) speed up poll() et al by use of techniques like poll hinting, which was implemented and benchmarked by Niels Provos for Linux in 1999.

  See Poller_poll (cc, h, benchmarks) for an example of how to use poll() interchangeably with other readiness notification schemes.

- /dev/poll
  This is the recommended poll replacement for Solaris.

  The idea behind /dev/poll is to take advantage of the fact that often poll() is called many times with the same arguments. With /dev/poll, you get an open handle to /dev/poll, and tell the OS just once what files you're interested in by writing to that handle; from then on, you just read the set of currently ready file descriptors from that handle.

  It appeared quietly in Solaris 7 (see patchid 106541) but its first public appearance was in Solaris 8; according to Sun, at 750 clients, this has 10% of the overhead of poll().

  Various implementations of /dev/poll were tried on Linux, but none of them perform as well as epoll, and were never really completed. /dev/poll use on Linux is not recommended.

See Poller devpoll (cc, h benchmarks) for an example of how to use /dev/poll interchangeably with many other readiness notification schemes. (Caution - the example is for Linux /dev/poll, might not work right on Solaris.)

- kqueue()
  This is the recommended poll replacement for FreeBSD (and, soon, NetBSD).

  See below. kqueue() can specify either edge triggering or level triggering.

2. Serve many clients with each thread, and use nonblocking I/O and readiness change notification

Readiness change notification (or edge-triggered readiness notification) means you give the kernel a file descriptor, and later, when that descriptor transitions from not ready to ready, the kernel notifies you somehow. It then assumes you know the file descriptor is ready, and will not send any more readiness notifications of that type for that file descriptor until you do something that causes the file descriptor to no longer be ready (e.g. until you receive the EWOULDBLOCK error on a send, recv, or accept call, or a send or recv transfers less than the requested number of bytes).

When you use readiness change notification, you must be prepared for spurious events, since one common implementation is to signal readiness whenever any packets are received, regardless of whether the file descriptor was already ready.

This is the opposite of "level-triggered" readiness notification. It's a bit less forgiving of programming mistakes, since if you miss just one event, the connection that event was for gets stuck forever. Nevertheless, I have found that edge-triggered readiness notification made programming nonblocking clients with OpenSSL easier, so it's worth trying.

[Banga, Mogul, Drusha '99] described this kind of scheme in 1999.

There are several APIs which let the application retrieve 'file descriptor became ready' notifications:

- kqueue() This is the recommended edge-triggered poll replacement for FreeBSD (and, soon, NetBSD).

  FreeBSD 4.3 and later, and NetBSD-current as of Oct 2002, support a generalized alternative to poll() called kqueue()/kevent(); it supports both edge-triggering and level-triggering. (See also Jonathan Lemon's page and his BSDCon 2000 paper on kqueue().)

  Like /dev/poll, you allocate a listening object, but rather than opening the file /dev/poll, you call kqueue() to allocate one. To change the events you are listening for, or to get the list of current events, you call kevent() on the descriptor returned by kqueue(). It can listen not just for socket readiness, but also for plain file readiness, signals, and even for I/O completion.

Note: as of October 2000, the threading library on FreeBSD does not interact well with kqueue(); evidently, when kqueue() blocks, the entire process blocks, not just the calling thread.

See Poller_kqueue (cc, h, benchmarks) for an example of how to use kqueue() interchangeably with many other readiness notification schemes.

Examples and libraries using kqueue():

- PyKQueue -- a Python binding for kqueue()
- Ronald F. Guilmette's example echo server; see also his 28 Sept 2000 post on freebsd.questions.

- epoll
This is the recommended edge-triggered poll replacement for the 2.6 Linux kernel.

  On 11 July 2001, Davide Libenzi proposed an alternative to realtime signals; his patch provides what he now calls /dev/epoll www.xmailserver.org/linux-patches/nio-improve.html. This is just like the realtime signal readiness notification, but it coalesces redundant events, and has a more efficient scheme for bulk event retrieval.

  Epoll was merged into the 2.5 kernel tree as of 2.5.46 after its interface was changed from a special file in /dev to a system call, sys_epoll. A patch for the older version of epoll is available for the 2.4 kernel.

  There was a lengthy debate about unifying epoll, aio, and other event sources on the linux-kernel mailing list around Halloween 2002. It may yet happen, but Davide is concentrating on firming up epoll in general first.

- Polyakov's kevent (Linux 2.6+) News flash: On 9 Feb 2006, and again on 9 July 2006, Evgeniy Polyakov posted patches which seem to unify epoll and aio; his goal is to support network AIO. See:
  - the LWN article about kevent
  - his July announcement
  - his kevent page
  - his naio page
  - some recent discussion

- Drepper's New Network Interface (proposal for Linux 2.6+)
At OLS 2006, Ulrich Drepper proposed a new high-speed asynchronous networking API. See:

    - his paper, "The Need for Asynchronous, Zero-Copy Network I/O"
    - his slides
    - LWN article from July 22

- Realtime Signals
  This is the recommended edge-triggered poll replacement for the 2.4 Linux kernel.

  The 2.4 linux kernel can deliver socket readiness events via a particular realtime signal. Here's how to turn this behavior on:

  ```
  /* Mask off SIGIO and the signal you want to use. */
  sigemptyset(&sigset);
  sigaddset(&sigset, signum);
  sigaddset(&sigset, SIGIO);
  sigprocmask(SIG_BLOCK, &m_sigset, NULL);
  /* For each file descriptor, invoke F_SETOWN, F_SETSIG, and set O_ASYNC. */
  fcntl(fd, F_SETOWN, (int) getpid());
  fcntl(fd, F_SETSIG, signum);
  flags = fcntl(fd, F_GETFL);
  flags |= O_NONBLOCK|O_ASYNC;
  fcntl(fd, F_SETFL, flags);
  ```

  This sends that signal when a normal I/O function like read() or write() completes. To use this, write a normal poll() outer loop, and inside it, after you've handled all the fd's noticed by poll(), you loop calling sigwaitinfo().
  If sigwaitinfo or sigtimedwait returns your realtime signal, siginfo.si_fd and siginfo.si_band give almost the same information as pollfd.fd and pollfd.revents would after a call to poll(), so you handle the i/o, and continue calling sigwaitinfo().
  If sigwaitinfo returns a traditional SIGIO, the signal queue overflowed, so you flush the signal queue by temporarily changing the signal handler to SIG_DFL, and break back to the outer poll() loop.

  See Poller_sigio (cc, h) for an example of how to use rtsignals interchangeably with many other readiness notification schemes.

  See Zach Brown's phhttpd for example code that uses this feature directly. (Or don't; phhttpd is a bit hard to figure out...)

  [Provos, Lever, and Tweedie 2000] describes a recent benchmark of phhttpd using a variant of sigtimedwait(), sigtimedwait4(), that lets you retrieve multiple signals with one call. Interestingly, the chief benefit of sigtimedwait4() for them seemed to be it allowed the app to gauge system overload (so it could behave appropriately). (Note that poll() provides the same measure of system overload.)

- Signal-per-fd
  Chandra and Mosberger proposed a modification to the realtime signal approach called "signal-per-fd" which reduces or eliminates realtime signal queue overflow by coalescing redundant events. It doesn't outperform epoll, though. Their paper (www.hpl.hp.com/techreports/2000/HPL-2000-174.html) compares performance of this scheme with select() and /dev/poll.

  Vitaly Luban announced a patch implementing this scheme on 18 May 2001; his patch lives at www.luban.org/GPL/gpl.html. (Note: as of Sept 2001, there may still be stability problems with this patch under heavy load. dkftpbench at about 4500 users may be able to trigger an oops.)

  See Poller_sigfd (cc, h) for an example of how to use signal-per-fd interchangeably with many other readiness notification schemes.

3. Serve many clients with each server thread, and use asynchronous I/O

This has not yet become popular in Unix, probably because few operating systems support asynchronous I/O, also possibly because it (like nonblocking I/O) requires rethinking your application. Under standard Unix, asynchronous I/O is provided by the aio_ interface (scroll down from that link to "Asynchronous input and output"), which associates a signal and value with each I/O operation. Signals and their values are queued and delivered efficiently to the user process. This is from the POSIX 1003.1b realtime extensions, and is also in the Single Unix Specification, version 2.

AIO is normally used with edge-triggered completion notification, i.e. a signal is queued when the operation is complete. (It can also be used with level triggered completion notification by calling aio_suspend(), though I suspect few people do this.)

glibc 2.1 and later provide a generic implementation written for standards compliance rather than performance.

Ben LaHaise's implementation for Linux AIO was merged into the main Linux kernel as of 2.5.32. It doesn't use kernel threads, and has a very efficient underlying api, but (as of 2.6.0-test2) doesn't yet support sockets. (There is also an AIO patch for the 2.4 kernels, but the 2.5/2.6 implementation is somewhat different.) More info:

- The page "Kernel Asynchronous I/O (AIO) Support for Linux" which tries to tie together all info about the 2.6 kernel's implementation of AIO (posted 16 Sept 2003)
- Round 3: aio vs /dev/epoll by Benjamin C.R. LaHaise (presented at 2002 OLS)
- Asynchronous I/O Suport in Linux 2.5, by Bhattacharya, Pratt, Pulaverty, and Morgan, IBM; presented at OLS '2003

- Design Notes on Asynchronous I/O (aio) for Linux by Suparna Bhattacharya -- compares Ben's AIO with SGI's KAIO and a few other AIO projects
- Linux AIO home page - Ben's preliminary patches, mailing list, etc.
- linux-aio mailing list archives
- libaio-oracle - library implementing standard Posix AIO on top of libaio. First mentioned by Joel Becker on 18 Apr 2003.

Suparna also suggests having a look at the the DAFS API's approach to AIO.

Red Hat AS and Suse SLES both provide a high-performance implementation on the 2.4 kernel; it is related to, but not completely identical to, the 2.6 kernel implementation.

In February 2006, a new attempt is being made to provide network AIO; see the note above about Evgeniy Polyakov's kevent-based AIO.

In 1999, SGI implemented high-speed AIO for Linux. As of version 1.1, it's said to work well with both disk I/O and sockets. It seems to use kernel threads. It is still useful for people who can't wait for Ben's AIO to support sockets.

The O'Reilly book POSIX.4: Programming for the Real World is said to include a good introduction to aio.

A tutorial for the earlier, nonstandard, aio implementation on Solaris is online at Sunsite. It's probably worth a look, but keep in mind you'll need to mentally convert "aioread" to "aio_read", etc.

Note that AIO doesn't provide a way to open files without blocking for disk I/O; if you care about the sleep caused by opening a disk file, Linus suggests you should simply do the open() in a different thread rather than wishing for an aio_open() system call.

Under Windows, asynchronous I/O is associated with the terms "Overlapped I/O" and IOCP or "I/O Completion Port". Microsoft's IOCP combines techniques from the prior art like asynchronous I/O (like aio_write) and queued completion notification (like when using the aio_sigevent field with aio_write) with a new idea of holding back some requests to try to keep the number of running threads associated with a single IOCP constant. For more information, see Inside I/O Completion Ports by Mark Russinovich at sysinternals.com, Jeffrey Richter's book "Programming Server-Side Applications for Microsoft Windows 2000" (Amazon, MSPress), U.S. patent #06223207, or MSDN.

4. Serve one client with each server thread

... and let read() and write() block. Has the disadvantage of using a whole stack frame for each client, which costs memory. Many OS's also have trouble handling more than a few hundred threads. If each thread gets a 2MB stack (not an uncommon default value), you run out of *virtual memory* at (2^30 / 2^21) = 512 threads on a 32 bit machine with 1GB user-accessible VM (like, say, Linux as normally shipped on x86). You can work around this by giving each thread a smaller stack, but since most thread libraries don't allow growing thread stacks once created, doing this means designing your program to minimize stack use. You can also work around this by moving to a 64 bit processor.

The thread support in Linux, FreeBSD, and Solaris is improving, and 64 bit processors are just around the corner even for mainstream users. Perhaps in the not-too-distant future, those who prefer using one thread per client will be able to use that paradigm even for 10000 clients. Nevertheless, at the current time, if you actually want to support that many clients, you're probably better off using some other paradigm.

For an unabashedly pro-thread viewpoint, see Why Events Are A Bad Idea (for High-concurrency Servers) by von Behren, Condit, and Brewer, UCB, presented at HotOS IX. Anyone from the anti-thread camp care to point out a paper that rebuts this one? :-)

LinuxThreads

LinuxTheads is the name for the standard Linux thread library. It is integrated into glibc since glibc2.0, and is mostly Posix-compliant, but with less than stellar performance and signal support.

NGPT: Next Generation Posix Threads for Linux

NGPT is a project started by IBM to bring good Posix-compliant thread support to Linux. It's at stable version 2.2 now, and works well... but the NGPT team has announced that they are putting the NGPT codebase into support-only mode because they feel it's "the best way to support the community for the long term". The NGPT team will continue working to improve Linux thread support, but now focused on improving NPTL. (Kudos to the NGPT team for their good work and the graceful way they conceded to NPTL.)

NPTL: Native Posix Thread Library for Linux

NPTL is a project by Ulrich Drepper (the benevolent dict^H^H^H^Hmaintainer of glibc) and Ingo Molnar to bring world-class Posix threading support to Linux.

As of 5 October 2003, NPTL is now merged into the glibc cvs tree as an add-on directory (just like linuxthreads), so it will almost certainly be released along with the next release of glibc.

The first major distribution to include an early snapshot of NPTL was Red Hat 9. (This was a bit inconvenient for some users, but somebody had to break the ice...)

NPTL links:

- Mailing list for NPTL discussion
- NPTL source code
- Initial announcement for NPTL
- Original whitepaper describing the goals for NPTL
- Revised whitepaper describing the final design of NPTL
- Ingo Molnar's first benchmark showing it could handle 10^6 threads
- Ulrich's benchmark comparing performance of LinuxThreads, NPTL, and IBM's NGPT. It seems to show NPTL is much faster than NGPT.

Here's my try at describing the history of NPTL (see also Jerry Cooperstein's article):

In March 2002, Bill Abt of the NGPT team, the glibc maintainer Ulrich Drepper, and others met to figure out what to do about LinuxThreads. One idea that came out of the meeting was to improve mutex performance; Rusty Russell et al subsequently implemented fast userspace mutexes (futexes)), which are now used by both NGPT and NPTL. Most of the attendees figured NGPT should be merged into glibc.

Ulrich Drepper, though, didn't like NGPT, and figured he could do better. (For those who have ever tried to contribute a patch to glibc, this may not come as a big surprise :-) Over the next few months, Ulrich Drepper, Ingo Molnar, and others contributed glibc and kernel changes that make up something called the Native Posix Threads Library (NPTL). NPTL uses all the kernel enhancements designed for NGPT, and takes advantage of a few new ones. Ingo Molnar described the kernel enhancements as follows:

> While NPTL uses the three kernel features introduced by NGPT: getpid() returns PID, CLONE_THREAD and futexes; NPTL also uses (and relies on) a much wider set of new kernel features, developed as part of this project.
>
> Some of the items NGPT introduced into the kernel around 2.5.8 got modified, cleaned up and extended, such as thread group handling (CLONE_THREAD). [the CLONE_THREAD changes which impacted NGPT's compatibility got synced with the NGPT folks, to make sure NGPT does not break in any unacceptable way.]
>
> The kernel features developed for and used by NPTL are described in the design whitepaper, http://people.redhat.com/drepper/nptl-design.pdf ...
>
> A short list: TLS support, various clone extensions (CLONE_SETTLS, CLONE_SETTID, CLONE_CLEARTID), POSIX

> thread-signal handling, sys_exit() extension (release TID futex upon VM-release), the sys_exit_group() system-call, sys_execve() enhancements and support for detached threads.
>
> There was also work put into extending the PID space - eg. procfs crashed due to 64K PID assumptions, max_pid, and pid allocation scalability work. Plus a number of performance-only improvements were done as well.
>
> In essence the new features are a no-compromises approach to 1:1 threading - the kernel now helps in everything where it can improve threading, and we precisely do the minimally necessary set of context switches and kernel calls for every basic threading primitive.

One big difference between the two is that NPTL is a 1:1 threading model, whereas NGPT is an M:N threading model (see below). In spite of this, Ulrich's initial benchmarks seem to show that NPTL is indeed much faster than NGPT. (The NGPT team is looking forward to seeing Ulrich's benchmark code to verify the result.)

FreeBSD threading support

FreeBSD supports both LinuxThreads and a userspace threading library. Also, a M:N implementation called KSE was introduced in FreeBSD 5.0. For one overview, see www.unobvious.com/bsd/freebsd-threads.html.

On 25 Mar 2003, Jeff Roberson posted on freebsd-arch:

> ... Thanks to the foundation provided by Julian, David Xu, Mini, Dan Eischen, and everyone else who has participated with KSE and libpthread development Mini and I have developed a 1:1 threading implementation. This code works in parallel with KSE and does not break it in any way. It actually helps bring M:N threading closer by testing out shared bits. ...

And in July 2006, Robert Watson proposed that the 1:1 threading implementation become the default in FreeBsd 7.x:

> I know this has been discussed in the past, but I figured with 7.x trundling forward, it was time to think about it again. In benchmarks for many common applications and scenarios, libthr demonstrates significantly better performance over libpthread... libthr is also implemented across a larger number of our platforms, and is already libpthread on several. The first recommendation we make to MySQL and other heavy thread users is "Switch to libthr", which is suggestive, also! ... So the strawman proposal is: make libthr the default threading library on 7.x.

NetBSD threading support

According to a note from Noriyuki Soda:

> Kernel supported M:N thread library based on the Scheduler Activations model is merged into NetBSD-current on Jan 18 2003.

For details, see An Implementation of Scheduler Activations on the NetBSD Operating System by Nathan J. Williams, Wasabi Systems, Inc., presented at FREENIX '02.

Solaris threading support

The thread support in Solaris is evolving... from Solaris 2 to Solaris 8, the default threading library used an M:N model, but Solaris 9 defaults to 1:1 model thread support. See Sun's multithreaded programming guide and Sun's note about Java and Solaris threading.

Java threading support in JDK 1.3.x and earlier

As is well known, Java up to JDK1.3.x did not support any method of handling network connections other than one thread per client. Volanomark is a good microbenchmark which measures throughput in messsages per second at various numbers of simultaneous connections. As of May 2003, JDK 1.3 implementations from various vendors are in fact able to handle ten thousand simultaneous connections -- albeit with significant performance degradation. See Table 4 for an idea of which JVMs can handle 10000 connections, and how performance suffers as the number of connections increases.

Note: 1:1 threading vs. M:N threading

There is a choice when implementing a threading library: you can either put all the threading support in the kernel (this is called the 1:1 threading model), or you can move a fair bit of it into userspace (this is called the M:N threading model). At one point, M:N was thought to be higher performance, but it's so complex that it's hard to get right, and most people are moving away from it.

- Why Ingo Molnar prefers 1:1 over M:N
- Sun is moving to 1:1 threads
- NGPT is an M:N threading library for Linux.
- Although Ulrich Drepper planned to use M:N threads in the new glibc threading library, he has since switched to the 1:1 threading model.
- MacOSX appears to use 1:1 threading.
- FreeBSD and NetBSD appear to still believe in M:N threading... The lone holdouts? Looks like freebsd 7.0 might

> switch to 1:1 threading (see above), so perhaps M:N threading's believers have finally been proven wrong everywhere.

5. Build the server code into the kernel

Novell and Microsoft are both said to have done this at various times, at least one NFS implementation does this, khttpd does this for Linux and static web pages, and "TUX" (Threaded linUX webserver) is a blindingly fast and flexible kernel-space HTTP server by Ingo Molnar for Linux. Ingo's September 1, 2000 announcement says an alpha version of TUX can be downloaded from ftp://ftp.redhat.com/pub/redhat/tux, and explains how to join a mailing list for more info.
The linux-kernel list has been discussing the pros and cons of this approach, and the consensus seems to be instead of moving web servers into the kernel, the kernel should have the smallest possible hooks added to improve web server performance. That way, other kinds of servers can benefit. See e.g. Zach Brown's remarks about userland vs. kernel http servers. It appears that the 2.4 linux kernel provides sufficient power to user programs, as the X15 server runs about as fast as Tux, but doesn't use any kernel modifications.

Bring the TCP stack into userspace

See for instance the netmap packet I/O framework, and the Sandstorm proof-of-concept web server based on it.

Comments

Richard Gooch has written a paper discussing I/O options.

In 2001, Tim Brecht and MMichal Ostrowski measured various strategies for simple select-based servers. Their data is worth a look.

In 2003, Tim Brecht posted source code for userver, a small web server put together from several servers written by Abhishek Chandra, David Mosberger, David Pariag, and Michal Ostrowski. It can use select(), poll(), epoll(), or sigio.

Back in March 1999, Dean Gaudet posted:

> I keep getting asked "why don't you guys use a select/event based model like Zeus? It's clearly the fastest."
> ...

His reasons boiled down to "it's really hard, and the payoff isn't clear". Within a few months, though, it became clear that people were willing to work on it.

Mark Russinovich wrote an editorial and an article discussing I/O strategy issues in the 2.2 Linux kernel. Worth reading, even he seems misinformed on some points. In particular, he seems to think that Linux 2.2's asynchronous I/O (see F_SETSIG above) doesn't notify the user process when data is ready, only when new connections arrive. This seems like a bizarre misunderstanding. See also comments on an earlier draft, Ingo Molnar's rebuttal of 30 April 1999, Russinovich's comments of 2 May 1999, a rebuttal from Alan Cox, and various posts to linux-kernel. I suspect he was trying to say that Linux doesn't support asynchronous disk I/O, which used to be true, but now that SGI has implemented KAIO, it's not so true anymore.

See these pages at sysinternals.com and MSDN for information on "completion ports", which he said were unique to NT; in a nutshell, win32's "overlapped I/O" turned out to be too low level to be convenient, and a "completion port" is a wrapper that provides a queue of completion events, plus scheduling magic that tries to keep the number of running threads constant by allowing more threads to pick up completion events if other threads that had picked up completion events from this port are sleeping (perhaps doing blocking I/O).

See also OS/400's support for I/O completion ports.

There was an interesting discussion on linux-kernel in September 1999 titled "> 15,000 Simultaneous Connections" (and the second week of the thread). Highlights:

- Ed Hall posted a few notes on his experiences; he's achieved >1000 connects/second on a UP P2/333 running Solaris. His code used a small pool of threads (1 or 2 per CPU) each managing a large number of clients using "an event-based model".
- Mike Jagdis posted an analysis of poll/select overhead, and said "The current select/poll implementation can be improved significantly, especially in the blocking case, but the overhead will still increase with the number of descriptors because select/poll does not, and cannot, remember what descriptors are interesting. This would be easy to fix with a new API. Suggestions are welcome..."
- Mike posted about his work on improving select() and poll().
- Mike posted a bit about a possible API to replace poll()/select(): "How about a 'device like' API where you write 'pollfd like' structs, the 'device' listens for events and delivers 'pollfd like' structs representing them when you read it? ... "
- Rogier Wolff suggested using "the API that the digital guys suggested", http://www.cs.rice.edu/~gaurav/papers/usenix99.ps
- Joerg Pommnitz pointed out that any new API along these lines should be able to wait for not just file descriptor events, but also signals and maybe SYSV-IPC. Our synchronization primitives should certainly be able to do what Win32's WaitForMultipleObjects can, at least.
- Stephen Tweedie asserted that the combination of F_SETSIG, queued realtime signals, and sigwaitinfo() was a superset of the API proposed in http://www.cs.rice.edu/~gaurav/papers/usenix99.ps. He also mentions that you keep

the signal blocked at all times if you're interested in performance; instead of the signal being delivered asynchronously, the process grabs the next one from the queue with sigwaitinfo().
- Jayson Nordwick compared completion ports with the F_SETSIG synchronous event model, and concluded they're pretty similar.
- Alan Cox noted that an older rev of SCT's SIGIO patch is included in 2.3.18ac.
- Jordan Mendelson posted some example code showing how to use F_SETSIG.
- Stephen C. Tweedie continued the comparison of completion ports and F_SETSIG, and noted: "With a signal dequeuing mechanism, your application is going to get signals destined for various library components if libraries are using the same mechanism," but the library can set up its own signal handler, so this shouldn't affect the program (much).
- Doug Royer noted that he'd gotten 100,000 connections on Solaris 2.6 while he was working on the Sun calendar server. Others chimed in with estimates of how much RAM that would require on Linux, and what bottlenecks would be hit.

Interesting reading!

Limits on open filehandles

- Any Unix: the limits set by ulimit or setrlimit.
- Solaris: see the Solaris FAQ, question 3.46 (or thereabouts; they renumber the questions periodically).
- FreeBSD:

  Edit /boot/loader.conf, add the line

  ```
  set kern.maxfiles=XXXX
  ```

  where XXXX is the desired system limit on file descriptors, and reboot. Thanks to an anonymous reader, who wrote in to say he'd achieved far more than 10000 connections on FreeBSD 4.3, and says

  > "FWIW: You can't actually tune the maximum number of connections in FreeBSD trivially, via sysctl.... You have to do it in the /boot/loader.conf file.
  > The reason for this is that the zalloci() calls for initializing the sockets and tcpcb structures zones occurs very early in system startup, in order that the zone be both type stable and that it be swappable.
  > You will also need to set the number of mbufs much higher, since you will (on an unmodified kernel) chew up one mbuf per connection for tcptempl structures, which are used to implement keepalive."

  Another reader says

  > "As of FreeBSD 4.4, the tcptempl structure is no longer allocated; you no longer have to worry about one mbuf being chewed up per connection."

  See also:

  - the FreeBSD handbook
  - SYSCTL TUNING, LOADER TUNABLES, and KERNEL CONFIG TUNING in 'man tuning'
  - The Effects of Tuning a FreeBSD 4.3 Box for High Performance, Daemon News, Aug 2001
  - postfix.org tuning notes, covering FreeBSD 4.2 and 4.4
  - the Measurement Factory's notes, circa FreeBSD 4.3

- OpenBSD: A reader says

  > "In OpenBSD, an additional tweak is required to increase the number of open filehandles available per process: the openfiles-cur parameter in /etc/login.conf needs to be increased. You can change kern.maxfiles either with sysctl -w or in sysctl.conf but it has no effect. This matters because as shipped, the login.conf limits are a quite low 64 for nonprivileged processes, 128 for privileged."

- Linux: See Bodo Bauer's /proc documentation. On 2.4 kernels:

  ```
  echo 32768 > /proc/sys/fs/file-max
  ```

  increases the system limit on open files, and

  ```
  ulimit -n 32768
  ```

  increases the current process' limit.

  On 2.2.x kernels,

  ```
  echo 32768 > /proc/sys/fs/file-max
  echo 65536 > /proc/sys/fs/inode-max
  ```

  increases the system limit on open files, and

  ```
  ulimit -n 32768
  ```

  increases the current process' limit.

  I verified that a process on Red Hat 6.0 (2.2.5 or so plus patches) can open at least 31000 file descriptors this way. Another fellow has verified that a process on 2.2.12 can open at least 90000 file descriptors this way (with appropriate limits). The upper bound seems to be available memory.

Stephen C. Tweedie posted about how to set ulimit limits globally or per-user at boot time using initscript and pam_limit.
In older 2.2 kernels, though, the number of open files per process is still limited to 1024, even with the above changes.
See also Oskar's 1998 post, which talks about the per-process and system-wide limits on file descriptors in the 2.0.36 kernel.

Limits on threads

On any architecture, you may need to reduce the amount of stack space allocated for each thread to avoid running out of virtual memory. You can set this at runtime with pthread_attr_init() if you're using pthreads.

- Solaris: it supports as many threads as will fit in memory, I hear.
- Linux 2.6 kernels with NPTL: /proc/sys/vm/max_map_count may need to be increased to go above 32000 or so threads. (You'll need to use very small stack threads to get anywhere near that number of threads, though, unless you're on a 64 bit processor.) See the NPTL mailing list, e.g. the thread with subject "Cannot create more than 32K threads?", for more info.
- Linux 2.4: /proc/sys/kernel/threads-max is the max number of threads; it defaults to 2047 on my Red Hat 8 system. You can set increase this as usual by echoing new values into that file, e.g. "echo 4000 > /proc/sys/kernel/threads-max"
- Linux 2.2: Even the 2.2.13 kernel limits the number of threads, at least on Intel. I don't know what the limits are on other architectures. Mingo posted a patch for 2.1.131 on Intel that removed this limit. It appears to be integrated into 2.3.20.

  See also Volano's detailed instructions for raising file, thread, and FD_SET limits in the 2.2 kernel. Wow. This document steps you through a lot of stuff that would be hard to figure out yourself, but is somewhat dated.

- Java: See Volano's detailed benchmark info, plus their info on how to tune various systems to handle lots of threads.

Java issues

Up through JDK 1.3, Java's standard networking libraries mostly offered the one-thread-per-client model. There was a way to do nonblocking reads, but no way to do nonblocking writes.

In May 2001, JDK 1.4 introduced the package java.nio to provide full support for nonblocking I/O (and some other goodies). See the release notes for some caveats. Try it out and give Sun feedback!

HP's java also includes a Thread Polling API.

In 2000, Matt Welsh implemented nonblocking sockets for Java; his performance benchmarks show that they have advantages over blocking sockets in servers handling many (up to 10000) connections. His class library is called java-nbio; it's part of the Sandstorm project. Benchmarks showing performance with 10000 connections are available.

See also Dean Gaudet's essay on the subject of Java, network I/O, and threads, and the paper by Matt Welsh on events vs. worker threads.

Before NIO, there were several proposals for improving Java's networking APIs:

- Matt Welsh's Jaguar system proposes preserialized objects, new Java bytecodes, and memory management changes to allow the use of asynchronous I/O with Java.
- Interfacing Java to the Virtual Interface Architecture, by C-C. Chang and T. von Eicken, proposes memory management changes to allow the use of asynchronous I/O with Java.
- JSR-51 was the Sun project that came up with the java.nio package. Matt Welsh participated (who says Sun doesn't listen?).

Other tips

- Zero-Copy
  Normally, data gets copied many times on its way from here to there. Any scheme that eliminates these copies to the bare physical minimum is called "zero-copy".
  - Thomas Ogrisegg's zero-copy send patch for mmaped files under Linux 2.4.17-2.4.20. Claims it's faster than sendfile().
  - IO-Lite is a proposal for a set of I/O primitives that gets rid of the need for many copies.
  - Alan Cox noted that zero-copy is sometimes not worth the trouble back in 1999. (He did like sendfile(), though.)
  - Ingo implemented a form of zero-copy TCP in the 2.4 kernel for TUX 1.0 in July 2000, and says he'll make it available to userspace soon.
  - Drew Gallatin and Robert Picco have added some zero-copy features to FreeBSD; the idea seems to be that if you call write() or read() on a socket, the pointer is page-aligned, and the amount of data transferred is at least a page, *and* you don't immediately reuse the buffer, memory management tricks will be used to avoid copies. But see followups to this message on linux-kernel for people's misgivings about the speed of those memory management tricks.

    According to a note from Noriyuki Soda:

    > Sending side zero-copy is supported since NetBSD-1.6 release by specifying "SOSEND_LOAN" kernel option. This option is now default on NetBSD-current (you can disable this feature by specifying "SOSEND_NO_LOAN" in the kernel option on NetBSD_current). With this feature, zero-copy is automatically enabled, if data more than 4096 bytes are specified as data to be sent.

    - The sendfile() system call can implement zero-copy networking.
      The sendfile() function in Linux and FreeBSD lets you tell the kernel to send part or all of a file. This lets the OS do it as efficiently as possible. It can be used equally well in servers using threads or servers using nonblocking I/O. (In Linux, it's poorly documented at the moment; use _syscall4 to call it. Andi Kleen is writing new man pages that cover this. See also Exploring The sendfile System Call by Jeff Tranter in Linux Gazette issue 91.) Rumor has it, ftp.cdrom.com benefitted noticeably from sendfile().

      A zero-copy implementation of sendfile() is on its way for the 2.4 kernel. See LWN Jan 25 2001.

      One developer using sendfile() with Freebsd reports that using POLLWRBAND instead of POLLOUT makes a big difference.

      Solaris 8 (as of the July 2001 update) has a new system call 'sendfilev'. A copy of the man page is here.. The Solaris 8 7/01 release notes also mention it. I suspect that this will be most useful when sending to a socket in blocking mode; it'd be a bit of a pain to use with a nonblocking socket.

- Avoid small frames by using writev (or TCP_CORK)
  A new socket option under Linux, TCP_CORK, tells the kernel to avoid sending partial frames, which helps a bit e.g. when there are lots of little write() calls you can't bundle together for some reason. Unsetting the option flushes the buffer. Better to use writev(), though...

  See LWN Jan 25 2001 for a summary of some very interesting discussions on linux-kernel about TCP_CORK and a possible alternative MSG_MORE.

- Behave sensibly on overload.
  [Provos, Lever, and Tweedie 2000] notes that dropping incoming connections when the server is overloaded improved the shape of the performance curve, and reduced the overall error rate. They used a smoothed version of "number of clients with I/O ready" as a measure of overload. This technique should be easily applicable to servers written with select, poll, or any system call that returns a count of readiness events per call (e.g. /dev/poll or sigtimedwait4()).
- Some programs can benefit from using non-Posix threads.

Not all threads are created equal. The clone() function in Linux (and its friends in other operating systems) lets you create a thread that has its own current working directory, for instance, which can be very helpful when implementing an ftp server. See Hoser FTPd for an example of the use of native threads rather than pthreads.
- Caching your own data can sometimes be a win.
"Re: fix for hybrid server problems" by Vivek Sadananda Pai (vivek@cs.rice.edu) on new-httpd, May 9th, states:

> "I've compared the raw performance of a select-based server with a multiple-process server on both FreeBSD and Solaris/x86. On microbenchmarks, there's only a marginal difference in performance stemming from the software architecture. The big performance win for select-based servers stems from doing application-level caching. While multiple-process servers can do it at a higher cost, it's harder to get the same benefits on real workloads (vs microbenchmarks). I'll be presenting those measurements as part of a paper that'll appear at the next Usenix conference. If you've got postscript, the paper is available at http://www.cs.rice.edu/~vivek/flash99/"

Other limits

- Old system libraries might use 16 bit variables to hold file handles, which causes trouble above 32767 handles. glibc2.1 should be ok.
- Many systems use 16 bit variables to hold process or thread id's. It would be interesting to port the Volano scalability benchmark to C, and see what the upper limit on number of threads is for the various operating systems.
- Too much thread-local memory is preallocated by some operating systems; if each thread gets 1MB, and total VM space is 2GB, that creates an upper limit of 2000 threads.
- Look at the performance comparison graph at the bottom of http://www.acme.com/software/thttpd/benchmarks.html. Notice how various servers have trouble above 128 connections, even on Solaris 2.6? Anyone who figures out why, let me know.
Note: if the TCP stack has a bug that causes a short (200ms) delay at SYN or FIN time, as Linux 2.2.0-2.2.6 had, and the OS or http daemon has a hard limit on the number of connections open, you would expect exactly this behavior. There may be other causes.

Kernel Issues

For Linux, it looks like kernel bottlenecks are being fixed constantly. See Linux Weekly News, Kernel Traffic, the Linux-Kernel mailing list, and my Mindcraft Redux page.

In March 1999, Microsoft sponsored a benchmark comparing NT to Linux at serving large numbers of http and smb clients, in which they failed to see good results from Linux. See also my article on Mindcraft's April 1999 Benchmarks for more info.

See also The Linux Scalability Project. They're doing interesting work, including Niels Provos' hinting poll patch, and some work on the thundering herd problem.

See also Mike Jagdis' work on improving select() and poll(); here's Mike's post about it.

Mohit Aron (aron@cs.rice.edu) writes that rate-based clocking in TCP can improve HTTP response time over 'slow' connections by 80%.

Measuring Server Performance

Two tests in particular are simple, interesting, and hard:

1. raw connections per second (how many 512 byte files per second can you serve?)
2. total transfer rate on large files with many slow clients (how many 28.8k modem clients can simultaneously download from your server before performance goes to pot?)

Jef Poskanzer has published benchmarks comparing many web servers. See http://www.acme.com/software/thttpd/benchmarks.html for his results.

I also have a few old notes about comparing thttpd to Apache that may be of interest to beginners.

Chuck Lever keeps reminding us about Banga and Druschel's paper on web server benchmarking. It's worth a read.

IBM has an excellent paper titled Java server benchmarks [Baylor et al, 2000]. It's worth a read.

Examples

Nginx is a web server that uses whatever high-efficiency network event mechanism is available on the target OS. It's getting popular; there are even two books about it.

Interesting select()-based servers

- thttpd Very simple. Uses a single process. It has good performance, but doesn't scale with the number of CPU's. Can also use kqueue.

- mathopd. Similar to thttpd.
- fhttpd
- boa
- Roxen
- Zeus, a commercial server that tries to be the absolute fastest. See their tuning guide.
- The other non-Java servers listed at http://www.acme.com/software/thttpd/benchmarks.html
- BetaFTPd
- Flash-Lite - web server using IO-Lite.
- Flash: An efficient and portable Web server -- uses select(), mmap(), mincore()
- The Flash web server as of 2003 -- uses select(), modified sendfile(), async open()
- xitami - uses select() to implement its own thread abstraction for portability to systems without threads.
- Medusa - a server-writing toolkit in Python that tries to deliver very high performance.
- userver - a small http server that can use select, poll, epoll, or sigio

Interesting /dev/poll-based servers

- N. Provos, C. Lever, "Scalable Network I/O in Linux," May, 2000. [FREENIX track, Proc. USENIX 2000, San Diego, California (June, 2000).] Describes a version of thttpd modified to support /dev/poll. Performance is compared with phhttpd.

Interesting epoll-based servers

- ribs2
- cmogstored - uses epoll/kqueue for most networking, threads for disk and accept4

Interesting kqueue()-based servers

- thttpd (as of version 2.21?)
- Adrian Chadd says "I'm doing a lot of work to make squid actually LIKE a kqueue IO system"; it's an official Squid subproject; see http://squid.sourceforge.net/projects.html#commloops. (This is apparently newer than Benno's patch.)

Interesting realtime signal-based servers

- Chromium's X15. This uses the 2.4 kernel's SIGIO feature together with sendfile() and TCP_CORK, and reportedly achieves higher speed than even TUX. The source is available under a community source (not open source) license. See the original announcement by Fabio Riccardi.
- Zach Brown's phhttpd - "a quick web server that was written to showcase the sigio/siginfo event model. consider this code highly experimental and yourself highly mental if you try and use it in a production environment." Uses the siginfo features of 2.3.21 or later, and includes the needed patches for earlier kernels. Rumored to be even faster than khttpd. See his post of 31 May 1999 for some notes.

Interesting thread-based servers

- Hoser FTPD. See their benchmark page.
- Peter Eriksson's phttpd and
- pftpd
- The Java-based servers listed at http://www.acme.com/software/thttpd/benchmarks.html
- Sun's Java Web Server (which has been reported to handle 500 simultaneous clients)

Interesting in-kernel servers

- khttpd
- "TUX" (Threaded linUX webserver) by Ingo Molnar et al. For 2.4 kernel.

Other interesting links

- Jeff Darcy's notes on high-performance server design
- Ericsson's ARIES project -- benchmark results for Apache 1 vs. Apache 2 vs. Tomcat on 1 to 12 processors
- Prof. Peter Ladkin's Web Server Performance page.
- Novell's FastCache -- claims 10000 hits per second. Quite the pretty performance graph.
- Rik van Riel's Linux Performance Tuning site

Translations

Belorussian translation provided by Patric Conrad at Ucallweconn

Changelog

2011/07/21
Added nginx.org

```
$Log: c10k.html,v $
Revision 1.212  2006/09/02 14:52:13  dank
added asio

Revision 1.211  2006/07/27 10:28:58  dank
Link to Cal Henderson's book.

Revision 1.210  2006/07/27 10:18:58  dank
Listify polyakov links, add Drepper's new proposal, note that FreeBSD 7 might move to 1:1

Revision 1.209  2006/07/13 15:07:03  dank
link to Scale! library, updated Polyakov links

Revision 1.208  2006/07/13 14:50:29  dank
Link to Polyakov's patches

Revision 1.207  2003/11/03 08:09:39  dank
Link to Linus's message deprecating the idea of aio_open

Revision 1.206  2003/11/03 07:44:34  dank
link to userver

Revision 1.205  2003/11/03 06:55:26  dank
Link to Vivek Pei's new Flash paper, mention great specweb99 score
```

---

dank@kegel.com
Last updated: 5 February 2014
[Return to www.kegel.com]