Graham King

Solvitas perambulum

epoll: The API that powers the modern internet

January 3, 2022 software epoll history

You used epoll to fetch this blog post. For almost anything you do on the Internet the server will be running Linux and it will use `epoll` to receive and answer your request in a timely and affordable manner.

> epoll is what makes Go such a great language for writing server software. Here is epoll in Go’s netpoll.
>
> epoll is what makes nginx the most popular web server in the world (this blog runs nginx). Here is nginx’s use of epoll.
>
> and it is often what we mean when we say ‘async’ in most programming languages. For example, of Rust’s two main async frameworks, async-std uses polling and tokio uses mio.

Aside: All of the above work on many operating systems and support API’s other than `epoll`, which is Linux specific. The Internet is mostly made of Linux, so epoll is the API that matters.

The problem

The core problem of running a network service, the problem epoll fixes, is that your network is very fast and your clients network is very slow. A server handling a request typically looks like this:

```
read the user's request (e.g. a browser HTTP GET)
do what they asked (e.g. load some information from the database)
write a response (e.g. HTML that the browser will display)
```

During the “read” and “write” parts above the server is idle, waiting for data or acknowledgments of that data to move across the network.

Before epoll

Before epoll the standard way to overcome this was to run a pool of processes each handling a different user request, typically with Apache mod_prefork. While one process waits on the user to acknowledge a packet of data, a different process can use the CPU. An emerging alternative was to use a thread pool which is lighter than a process pool and could handle low-hundreds of concurrent users. Multi-threading was risky as many libraries were not thread safe. Steven's 2004 reference UNIX Network Programming has a chapter discussing preforked vs prethreaded designs, because those were your options back then.

Then along came everybody, and even hundreds of concurrent users turned out not to be enough. An influential article, The C10K problem started this discussion in 1999. It was not uncommon for web requests to timeout. People would mirror popular sites in an effort to spread the traffic.

The solution

In 2000 Jonathan Lemon solved this problem for FreeBSD 4.3 by designing and building kqueue/kevent, making BSD the early choice for high performance networking.

Independently, in July 2001 Davide Libenzi solved the problem for Linux, with the first draft of epoll, which evolved, was merged into Linux kernel 2.5.44 (a development release) in October 2002 and became widely available in December 2003 with the release of stable kernel 2.6.

Jim Blandy has a fantastic comparison of threads vs epoll-based async here.

How it works

epoll allows a single thread or process to register interest in a long list of network sockets (it supports things other than network sockets such as pipes and terminals, but you rarely have thousands of those). An `epoll_wait` call will then block until one of those is ready for reading or writing. A single thread using epoll can handle tens of thousands of concurrent (and mostly idle) requests.

The downside of epoll is that it changes the architecture of your application. Instead of a handling each connection with a straightforward `{read request, handle, write response}`, you now have a main loop more akin to a game engine. The code becomes:

```
loop
  epoll_wait on all the connections
  for each of the ready connections:
    continue from where you left off
```

You might be part way through reading a request on one of the ready sockets, and part way through writing a response on another socket. You have to remember your state, do only as much I/O as the socket can take without blocking, and then epoll_wait again. A large part of the popularity of Go, and of the ‘async/await’ model in languages like C#, Javascript and Rust, is that they hide that event loop, allowing you to write straight-line code as if you were still doing thread-per-connection.

Conclusion

Without epoll either the economics of today’s Internet would look quite different (fewer requests per machine, so more machines, costing more money), or we’d be running our servers on a BSD. And without BSD’s kqueue (which preceded epoll by two years), we’d really be in trouble because the only alternatives were proprietary (/dev/poll in Solaris 8 and I/O Completion Ports in Windows NT 3.5).

epoll has been improved since it’s initial release, particularly with EPOLLONESHOT and EPOLLEXCLUSIVE flags, but the core API has stayed the same. epoll solved the C10K problem on Linux, which powers the Internet, allowing us to build fast and cheap Internet services.

Thanks Davide!

Addendum: predecessors

Linux had `poll` and `select` before `epoll`. They were designed to handle a handful of file descriptors and they scale O(n) on that count. epoll scales O(1). Kerrisk has

performance numbers showing poll and select becoming unusable beyond the hundreds of file descriptors, while epoll remains fast into the tens of thousands.

Linux also had signal-driven I/O before `epoll`. To quote UNIX Network Programming:

> *Unfortunately, signal-driven I/O is next to useless with a TCP socket*

and

> *The only real-world use of signal-driven I/O with sockets that the authors were able to find is the NTP server, which uses UDP.*

Davide Libenzi kindly read this post before publication

← Rust is also C
January 17, 2022

Scripting Minecraft server with Python →
December 11, 2021