Graham King

Solvitas perambulum

Linux: What can you epoll?

October 20, 2022 software linux epoll

At the start of this year I argued that epoll is the API that powers the modern Internet, but what can you actually do with `epoll`?

Once we structure our application around an epoll event loop or use an async engine such as Go or Rust's tokio, it becomes really useful to integrate everything else into that event loop.

> What types of file descriptors can you add to `epoll_ctl`?
>
> What can you turn into a file descriptor?

I was not able to find a comprehensive list of epoll-compatible APIs on the Internet, so I made one.

Network sockets

Network sockets are the reason we want and use `epoll`. These are nearly always IP sockets (TCP and UDP), but it also works with less common families such as UNIX, PACKET, NETLINK and many others. Indeed it should work with any of the protocol families that socket accepts, although I have not tested all of them.

Timers

timerfd_create allows us to create a one-shot or repeating timer and get a file descriptor for it, which becomes ready when the timer fires. Ideal for async `sleep` or timeout.

Signals

signalfd gives us a file descriptor that becomes ready when an operating system signal fires. Example uses include asking a program to reload it's configuration on SIGHUP, or logging debug info on SIGUSR1.

Filesystem events

The inotify API allows us to monitor the file system for events such as file creation and modification. The file descriptor returned by `inotify_init` can be polled with epoll. Example uses include reloading a config file when it changes, and waking up when new data appears in a directory.

Child processes

We can get a file descriptor for a child process either by setting the CLONE_PIDFD flag on clone or by calling pidfd_open. It becomes ready when the process exits. Useful for monitoring sub-processes and preventing them becoming zombies. If we need the exit code we would call waitid once epoll says it's exited.

Terminals

epoll accepts fd's from terminals and pseudo-terminals. It accepts fd 0 (stdin), 1 (stdout) and 2 (stderr). This is useful for programs designed to be used in a cmd line pipe, or as a sub-process where the parent writes to it's stdin and reads from it's stdout.

It also allows giving our servers an interactive operator / debug mode and wiring that into the main event loop. It's also quite fun to experiment with.

Epoll inception

`epoll_create` returns a file descriptor which can itself be monitored via `epoll_ctl`. This allows building a multi-level hierarchy of epoll listeners.

Notifications: The everything-else plan

If our async engine cannot ask the kernel for asynchronous (epoll-able) notification when a task completes, we have to run that task on a different thread. All we need then is a way for that thread to notify the event loop upon completion. There are four epoll-able ways for processes or threads to communicate:

POSIX message queues: See man 7 mq_overview.

UNIX socket pairs: man socketpair.

Pipes (or FIFOs aka named pipe): man 7 pipe.

eventfd: Designed for exactly our use case, eventfd is a modern alternative to pipes for epoll-able signalling. Has optional semaphore semantics. From man eventfd:

> *Applications can use an eventfd file descriptor instead of a pipe in all cases where a pipe is used simply to signal events. The kernel overhead of an eventfd file descriptor is much lower than that of a pipe, and only one file descriptor is required (versus the two required for a pipe).*

And what you can't epoll: regular files

Calling `epoll_ctl` with a regular (disk) file will return error `EPERM`:

> *EPERM The target file fd does not support epoll. This error can occur if fd refers to, for example, a regular file or a directory.*

On Linux `write` to a regular file never blocks. Writing to a file copies data from our user space buffer to the kernel buffer and returns immediately. At some later point in time the kernel will send it to the disk. A regular file is hence always ready for writing and epoll wouldn't add anything.

A file `read` is a copy from kernel buffer to user buffer. The kernel does try to anticipate

our reads by prefetching disk data, but if the data is not available our thread will very much block, going into uninterruptible sleep. There are edge cases where this could block indefinitely, such as some NTFS failures and FUSE setups (e.g. sshfs).

Aside By 'blocking' above I mean blocking I/O, particularly I/O that could block indefinitely such as reading from a socket. Copying data from user space to kernel space does indeed block our thread, but that doesn't count here. It is both very fast compared to I/O, and most importantly not indefinite.

File reads don't fit epoll particularly well because knowing that a file is ready to read (as indicated by `epoll_wait`) isn't useful unless we know the offset at which we can read. We may only want the last 100 bytes of an enormouse file. Still, it would be really nice to integrate disk reads with our event loop.

Linux does have two facilities for true async disk I/O: libaio and io_uring. Neither of them work directly with epoll. They could be integrated thanks to `eventfd`, or io_uring could become the event loop and monitor the epoll fd.

In pratice

I had a quick look at tokio and Go to see how much of this they are using. They both always use epoll and start multiple threads (num CPU * 2 seems standard). Go uses a pipe (`pipe2` syscall). Tokio uses eventfd via mio. Neither appears to use timerfd or signalfd. Tokio handles events (epoll) via Mio, and there are crates to extend Mio for all of the above APIs.

Discuss on Hacker News

Underrust: Multiple Return Values →
October 18, 2022