Inotify - A Powerful yet Simple File Change Notification System

Document started 15 Mar 2005 by Robert Love <rml@novell.com>

Document updated 4 Jan 2015 by Zhang Zhen <zhenzhang.zhang@huawei.com>

- Deleted obsoleted interface, just refer to manpages for user interface.

i. Rationale

Q:

What is the design decision behind not tying the watch to the open fd of the watched object?

A:

Watches are associated with an open inotify device, not an open file. This solves the primary problem with dnotify: keeping the file open pins the file and thus, worse, pins the mount. Dnotify is therefore infeasible for use on a desktop system with removable media as the media cannot be unmounted. Watching a file should not require that it be open.

Q:

What is the design decision behind using an-fd-per-instance as opposed to an fd-per-watch?

A:

An fd-per-watch quickly consumes more file descriptors than are allowed, more fd's than are feasible to manage, and more fd's than are optimally select()-able. Yes, root can bump the per-process fd limit and yes, users can use epoll, but requiring both is a silly and extraneous requirement. A watch consumes less memory than an open file, separating the number spaces is thus sensible. The current design is what user-space developers want: Users initialize inotify, once, and add n watches, requiring but one fd and no twiddling with fd limits. Initializing an inotify instance two thousand times is silly. If we can implement user-space's preferences cleanly--and we can, the idr layer makes stuff like this trivial--then we should.

There are other good arguments. With a single fd, there is a single item to block on, which is mapped to a single queue of events. The single fd returns all watch events and also any potential out-of-band data. If every fd was a separate watch,

- There would be no way to get event ordering. Events on file foo and file bar would pop poll() on both fd's, but there would be no way to tell which happened first. A single queue trivially gives you ordering. Such ordering is crucial to existing applications such as Beagle. Imagine "mv a b ; mv b a" events without ordering.
- We'd have to maintain n fd's and n internal queues with state, versus just one. It is a lot messier in the kernel. A single, linear queue is the data structure that makes sense.
- User-space developers prefer the current API. The Beagle guys, for example, love it. Trust me, I asked. It is not a surprise: Who'd want to manage and block on 1000 fd's via select?
- No way to get out of band data.
- 1024 is still too low. ;-)

When you talk about designing a file change notification system that scales to 1000s of directories, juggling 1000s of fd's just does not seem the right interface. It is too heavy.

Additionally, it _is_ possible to more than one instance and juggle more than one queue and thus more than one associated fd. There need not be a one-fd-per-process mapping; it is one-fd-per-queue and a process can easily want more than one queue.

Q:

Why the system call approach?

A:

The poor user-space interface is the second biggest problem with dnotify. Signals are a terrible, terrible interface for file notification. Or for anything, for that matter. The ideal solution, from all perspectives, is a file descriptor-based one that allows basic file I/O and poll/select. Obtaining the fd and managing the watches could have been done either via a device file or a family of new system calls. We decided to implement a family of system calls because that is the preferred approach for new kernel interfaces. The only real difference was whether we wanted to use open(2) and ioctl(2) or a couple of new system calls. System calls beat ioctls.