inotify
a powerful yet simple file change notification system

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

(i) User Interface

Inotify is controlled by a set of three system calls and normal file I/O on a returned file descriptor.

First step in using inotify is to initialise an inotify instance:

```
        int fd = inotify_init ();
```

Each instance is associated with a unique, ordered queue.

Change events are managed by "watches". A watch is an (object,mask) pair where the object is a file or directory and the mask is a bit mask of one or more inotify events that the application wishes to receive. See <linux/inotify.h> for valid events. A watch is referenced by a watch descriptor, or wd.

Watches are added via a path to the file.

Watches on a directory will return events on any files inside of the directory.

Adding a watch is simple:

```
        int wd = inotify_add_watch (fd, path, mask);
```

Where "fd" is the return value from inotify_init(), path is the path to the object to watch, and mask is the watch mask (see <linux/inotify.h>).

You can update an existing watch in the same manner, by passing in a new mask.

An existing watch is removed via

```
        int ret = inotify_rm_watch (fd, wd);
```

Events are provided in the form of an inotify_event structure that is read(2) from a given inotify instance. The filename is of dynamic length and follows the struct. It is of size len. The filename is padded with null bytes to ensure proper alignment. This padding is reflected in len.

You can slurp multiple events by passing a large buffer, for example

```
        size_t len = read (fd, buf, BUF_LEN);
```

Where "buf" is a pointer to an array of "inotify_event" structures at least BUF_LEN bytes in size. The above example will return as many events as are available and fit in BUF_LEN.

Each inotify instance fd is also select()- and poll()-able.

You can find the size of the current event queue via the standard FIONREAD ioctl on the fd returned by inotify_init().

All watches are destroyed and cleaned up on close.

(ii)

Prototypes:

```
        int inotify_init (void);
        int inotify_add_watch (int fd, const char *path, __u32 mask);
        int inotify_rm_watch (int fd, __u32 mask);
```

(iii) Kernel Interface

Inotify's kernel API consists a set of functions for managing watches and an event callback.

To use the kernel API, you must first initialize an inotify instance with a set of inotify_operations. You are given an opaque inotify_handle, which you use for any further calls to inotify.

```
    struct inotify_handle *ih = inotify_init(my_event_handler);
```

You must provide a function for processing events and a function for destroying the inotify watch.

```
    void handle_event(struct inotify_watch *watch, u32 wd, u32 mask,
                      u32 cookie, const char *name, struct inode *inode)

        watch - the pointer to the inotify_watch that triggered this call
        wd - the watch descriptor
        mask - describes the event that occurred
        cookie - an identifier for synchronizing events
        name - the dentry name for affected files in a directory-based event
        inode - the affected inode in a directory-based event

    void destroy_watch(struct inotify_watch *watch)
```

You may add watches by providing a pre-allocated and initialized inotify_watch structure and specifying the inode to watch along with an inotify event mask. You must pin the inode during the call. You will likely wish to embed the inotify_watch structure in a structure of your own which contains other information about the watch. Once you add an inotify watch, it is immediately subject to removal depending on filesystem events. You must grab a reference if you depend on the watch hanging around after the call.

```
    inotify_init_watch(&my_watch->iwatch);
    inotify_get_watch(&my_watch->iwatch);       // optional
    s32 wd = inotify_add_watch(ih, &my_watch->iwatch, inode, mask);
    inotify_put_watch(&my_watch->iwatch);       // optional
```

You may use the watch descriptor (wd) or the address of the inotify_watch for other inotify operations. You must not directly read or manipulate data in the inotify_watch. Additionally, you must not call inotify_add_watch() more than once for a given inotify_watch structure, unless you have first called either inotify_rm_watch() or inotify_rm_wd().

To determine if you have already registered a watch for a given inode, you may call inotify_find_watch(), which gives you both the wd and the watch pointer for the inotify_watch, or an error if the watch does not exist.

```
    wd = inotify_find_watch(ih, inode, &watchp);
```

You may use container_of() on the watch pointer to access your own data associated with a given watch. When an existing watch is found, inotify_find_watch() bumps the refcount before releasing its locks. You must put that reference with:

```
    put_inotify_watch(watchp);
```

Call inotify_find_update_watch() to update the event mask for an existing watch. inotify_find_update_watch() returns the wd of the updated watch, or an error if the watch does not exist.

```
    wd = inotify_find_update_watch(ih, inode, mask);
```

An existing watch may be removed by calling either inotify_rm_watch() or inotify_rm_wd().

```
    int ret = inotify_rm_watch(ih, &my_watch->iwatch);
    int ret = inotify_rm_wd(ih, wd);
```

A watch may be removed while executing your event handler with the following:

```
    inotify_remove_watch_locked(ih, iwatch);
```

Call inotify_destroy() to remove all watches from your inotify instance and release it. If there are no outstanding references, inotify_destroy() will call your destroy_watch op for each watch.

```
    inotify_destroy(ih);
```

When inotify removes a watch, it sends an IN_IGNORED event to your callback. You may use this event as an indication to free the watch memory. Note that inotify may remove a watch due to filesystem events, as well as by your request. If you use IN_ONESHOT, inotify will remove the watch after the first event, at which point you may call the final inotify_put_watch.

(iv) Kernel Interface Prototypes

```
        struct inotify_handle *inotify_init(struct inotify_operations *ops);

        inotify_init_watch(struct inotify_watch *watch);

        s32 inotify_add_watch(struct inotify_handle *ih,
                              struct inotify_watch *watch,
                              struct inode *inode, u32 mask);

        s32 inotify_find_watch(struct inotify_handle *ih, struct inode *inode,
```

```
                                 struct inotify_watch **watchp);

        s32 inotify_find_update_watch(struct inotify_handle *ih,
                                      struct inode *inode, u32 mask);

        int inotify_rm_wd(struct inotify_handle *ih, u32 wd);

        int inotify_rm_watch(struct inotify_handle *ih,
                             struct inotify_watch *watch);

        void inotify_remove_watch_locked(struct inotify_handle *ih,
                                         struct inotify_watch *watch);

        void inotify_destroy(struct inotify_handle *ih);

        void get_inotify_watch(struct inotify_watch *watch);
        void put_inotify_watch(struct inotify_watch *watch);
```

(v) Internal Kernel Implementation

Each inotify instance is represented by an inotify_handle structure. Inotify's userspace consumers also have an inotify_device which is associated with the inotify_handle, and on which events are queued.

Each watch is associated with an inotify_watch structure. Watches are chained off of each associated inotify_handle and each associated inode.

See fs/inotify.c and fs/inotify_user.c for the locking and lifetime rules.

(vi) 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.