iVoyeur
inotify

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project. dave-usenix@skeptech.org

The last time I changed jobs, the magnitude of the change didn't really sink in until the morning of my first day, when I took a different combination of freeways to work. The difference was accentuated by the fact that the new commute began the same as the old one, but on this morning, at a particular interchange, I would zig where before I zagged.

It was an unexpectedly emotional and profound metaphor for the change. My old place was off to the side, and down below, while my future was straight ahead, and literally under construction.

The fact that it was under construction was poetic but not surprising. Most of the roads I travel in the Dallas/Fort Worth area are under construction and have been for as long as anyone can remember. And I don't mean a lane closed here or there. Our roads drift and wander like leaves in the water—here today and tomorrow over there. The exits and entrances, neither a part of this road or that, seem unable to anticipate the movements of their brethren, and are constantly forced to react. They stretch and grasp, and struggle to keep pace, and sometimes they even manage to connect things, albeit usually not the things they claim to connect. The GPS units, having given up years ago, refuse to commit themselves, offering only vague, directionless suggestions that begin "continue to merge."

On that particular morning—and really every morning that one takes a rarely traveled road in DFW—that new zig was literally and figuratively the beginning of an adventure into a wholly unexplored country, despite my having traveled the route several times in the past. Often, because of the unfortunate, disembodied exits, these adventures involve the accidental continued merging onto another highway in an utterly unexpected direction (usually north). So it was that I arrived emotionally depleted and 15 minutes late on the first morning of my new job. They didn't seem to notice.

I can imagine neither the ultimate goal of the master plan under which our Department of Transportation labors nor whether its intent is whimsical or malevolent, but it certainly has provided ample food for reflection over the years. Many of my professional undertakings remind me of this or that aspect of our highways.

There are, for example, problems that seem to recur every so often that, like the exits in Grapevine TX, take me in a new and surprising direction every time I visit. File system notification seems to be one such problem. Every time I've had a need to come up with a foolproof way to monitor changes to a set of files or directories, my options seem to have changed radically.

With three in-kernel solutions—dnotify, inotify, and fanotify—kernel instrumentation like systemtap, several external libraries like libevent, and myriad security-focused tools like snoopy logger and samhain, there are now more ways to monitor changes to files than there are types of file in UNIX.

On this visit, after an afternoon of reading, I decided to try out the inotify API, which is built-in to Linux kernels >= 2.6.13. Created to address several shortcomings in dnotify, namely, a vastly simplified (signals free) interface to more precise events about more specific file-system objects (dnotify only works on directories), inotify makes dnotify obsolete. The consensus seems to be that unless you very specifically need an in-kernel solution for monitoring directories that will never be unmounted on Linux kernels < 2.6.13, dnotify should be ignored. I should also mention that there are a few wrapper libraries that provide a more portable and abstract interface to inotify and inotify-like functionality on systems other than Linux. Among these are inotify_tools, FAM, and Gamin.

Further, the inotify_tools package provides two programs that are suitable for use in shell scripts: inotifywatch, which collects and reports inotify events in a "tail -f" fashion, and inotifywait, which blocks waiting for user-specified inotify events.

Inotify's interface is pretty simple. After creating an inotify instance with inotify_init(), the application informs the kernel which files it's interested in watching with one or more calls to inotify_add_watch(). Each call to inotify_add_watch() is accompanied by a path name and a bitmask specifying the event types to watch for. The add_watch function returns a file descriptor, which can be poll()'d, select()'d, or simply read() by your application. When successfully read, the file descriptor returns one or more `inotify_event` structures, each of which contains the details of a single file system event. When the application has finished its monitoring duties, it closes the watch file descriptor. I've provided a small example program in Listing 1.

Adding and Removing Watches

The add_watch function is not recursive, and therefore must be called on each subdirectory in a given directory that you want to monitor. If you call it on a directory, it will monitor all files in that directory (but not files in subdirectories). The function is formally defined as follows:

```
int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

Add_watch returns 0 on success or -1 on failure. The first argument is the file descriptor returned from inotify_init(), and is used as a means of referring to our inotify instance. The second argument is the path to the file or directory you want to monitor. The application needs to have read permission on this object for the call to succeed. The third argument is a collection of event bits OR'd together. There are 23 possible events, 12 of which represent file system actions, like `IN_CREATE` (a file or directory was created).

A few event constants are shorthand for combinations of other events. For example, the event `IN_MOVED_FROM` is set when a file is moved out of a monitored directory, while `IN_MOVED_TO` is set when a file is moved in to a monitored directory. The shorthand event `IN_MOVE` can be used in lieu of both MOVED events. The shorthand event `IN_ALL_EVENTS` can be used to subscribe to all event types.

When a successful read returns an `inotify_event` struct, the same event constants are used in the struct to communicate the type of event that has transpired. Several of the event types only occur as output from inotify in these structs. Examples include `IN_ISDIR`, which is set whenever an event describes a directory, or `IN_UNMOUNT`, which is set when a directory is unmounted.

Finally, a few event types can be set in the bitmask to specify options to the inotify subsystem, like `IN_DONT_FOLLOW`, which turns off dereferencing of symbolic links. The complete list of event types is available in the inotify(7) man page.

Calling inotify_add_watch() on a file or directory that is already being watched replaces the event mask for that file or directory, but specifying `IN_MASK_ADD` in the replacement mask modifies this behavior such that the new mask is OR'd with the old one.

A call to inotify_rm_watch() explicitly removes watches on named files or directories. Whenever you explicitly remove a watch, or a file is moved outside a watched directory structure, inotify generates an event with the `IN_IGNORED` bit set.

Reading Events

The FD returned by add_watch follows the universal I/O convention, and may be treated like any other file descriptor. If no events have occurred when your application attempts to read() the descriptor, it blocks until an event is available. Applications may use poll() or select() for nonblocking behavior. Successful reads yield a stream of bytes, which is composed of one or more serialized `inotify_event` structs. The struct is defined as follows:

```
struct inotify_event {
int        wd;       //the FD on which the event occurred
uint32_t  mask;    //bitmask describing the event
uint32_t  cookie;  //cookie to detect related events
uint32_t   len;     //size of the name field in bytes
char     name[]; //null terminated filename (optional)
};
```

The `wd` descriptor is used by applications that are monitoring multiple files or directories via the same inotify file descriptor. To use it, your application needs to keep an internal map that relates the file descriptors returned by add_watch, to the files you passed into add_watch.

The mask is a bitmask that describes the event using the constants I've discussed above.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <error.h>
#include <sys/inotify.h>
#include <limits.h>

#define BUF_SIZE (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

void
printEvent(struct inotify_event *event)
{
    printf("wd::%d, ", event->wd);

    printf("mask :: ");
    if (event->mask & IN_ACCESS)        printf("IN_ACCESS ");
    if (event->mask & IN_ATTRIB)        printf("IN_ATTRIB ");
    if (event->mask & IN_CLOSE_NOWRITE) printf("IN_CLOSE_NOWRITE ");
    if (event->mask & IN_CLOSE_WRITE)   printf("IN_CLOSE_WRITE ");
    if (event->mask & IN_CREATE)        printf("IN_CREATE ");
    if (event->mask & IN_DELETE)        printf("IN_DELETE ");
    if (event->mask & IN_DELETE_SELF)   printf("IN_DELETE_SELF ");
    if (event->mask & IN_IGNORED)       printf("IN_IGNORED ");
    if (event->mask & IN_ISDIR)         printf("IN_ISDIR ");
    if (event->mask & IN_MODIFY)        printf("IN_MODIFY ");
    if (event->mask & IN_MOVE_SELF)     printf("IN_MOVE_SELF ");
    if (event->mask & IN_MOVED_FROM)    printf("IN_MOVED_FROM ");
    if (event->mask & IN_MOVED_TO)      printf("IN_MOVED_TO ");
    if (event->mask & IN_OPEN)          printf("IN_OPEN ");
    if (event->mask & IN_Q_OVERFLOW)    printf("IN_Q_OVERFLOW ");
    if (event->mask & IN_UNMOUNT)       printf("IN_UNMOUNT ");
    printf("\n");

    if (event->len > 0)
        printf("name :: %s\n", event->name);
}
int main (int argc, char *argv[]){
    int fd, wd, rr,i; //fd and wd are file descriptors
    char buf[BUF_SIZE]; //returned events come here
    char *p;
    struct inotify_event *event;

    fd = inotify_init(); //dear kernel, inotify plsthnx

    if (fd < 0)
        perror("inotify_init");

    for(i=1;i<=argc;i++){
        // setup a watch on all events for every file passed as an arg
        wd=inotify_add_watch(fd,argv[i],IN_ALL_EVENTS);
        if (fd < 0)
            perror("inotify_init");

        printf("watching %s using wd %i\n", argv[i], wd);
    }

    for(;;){ //event loop
        rr=read(fd,buf,BUF_SIZE); //any events?
        if(rr == 0){
            printf("uh-oh, read from inotify returned 0!\n");
            return 42;
      }
        if(rr<0) //something went wrong with the read syscall
            perror("read");

        printf("read %ld bytes from inotify fd\n",(long) rr);

        for(p=buf;p<buf+rr;){ //as long as there are more bytes in buf
            event = (struct inotify_event *) p; //cast the current addr as a struct
            printEvent(event); //decode it
            p += sizeof(struct inotify_event) + event->len; //move to the start of the next struct
        }
    }
}
```

Listing 1

Cookies are currently only used to associate MOVE events that are the result of renaming files. When a file is renamed, two events will be generated: an IN_MOVED_FROM event for the old file name and an IN_MOVED_TO event for the new file name. When this occurs, the value of the cookie field will be the same in both events, so that the application can associate the two.

If an event occurs to a file inside a monitored directory, the name field will be set to the name of the file, and len will indicate the number of bytes allocated for the name field. If, however, the event occurs to the directory itself, name will be NULL and len will be set to 0.

Because name is a dynamically allocated field, predicting the necessary size of the read buffer for the next event struct is impossible until you've read the struct and dereferenced the len field.; however, we can safely assume the size of the next struct will be smaller than:

```
(sizeof(struct inotify_event) + NAME_MAX + 1)
```

where NAME_MAX is the local OS constant that specifies the maximum size of a file name (usually set in limits.h). In Listing 1, I'm passing a buffer 10 times this size to read(). This will allow the application to retrieve at least 10 events with a single read() efficiently, and use pointer arithmetic to split them out. A read from an inotify file descriptor will yield the number of available events that will fit in the supplied buffer. In the event that you pass a buffer that is too small to hold the next single event struct, read() will fail with EINVAL.

Because add_watch is not recursive, for inotify applications to dynamically detect and add_watch newly created subdirectories in a currently watched directory is pretty common. To keep things simple, I didn't include an example of that in my sample code, but I hope given this article's Listing 1, that it's an obvious enough exercise for the reader.

As always I hope you've enjoyed the ride. Until next time, continue merging.