Monitor file system activity with inotify

Write your own applications, or use a suite of open source tools

Martin Streicher (martin.streicher@gmail.com)
Web developer
自由职业者

16 September 2008

Inotify is a Linux® feature that monitors file system operations, such as read, write, and create. Inotify is reactive, surprisingly simple to use, and far more efficient than, say, busy polling from a cron job. Learn how to integrate inotify into your own applications, and discover a set of command-line tools you can use to further automate system administration.

Systems administration is a lot like life. Like brushing your teeth and eating your veggies, a little daily maintenance keeps your machine humming. You must regularly empty cruft, such as temporary files or digested log files, among any number of interruptions such as filling out forms, returning calls, downloading updates, and monitoring processes. Thankfully, automation through shell scripts, monitoring using tools such as Nagios, and job scheduling via the ubiquitous cron can ease the burden.

Oddly, though, none of these tools are *reactive.* Certainly, you can schedule a cron job to run frequently to monitor a condition, but such busy polling—resource-intensive and speculative work—does not scale particularly well. For instance, if you must monitor several File Transfer Protocol (FTP) dropboxes for incoming data, you might scan each target directory with a `find` command to enumerate what's new. However, though the operation seems harmless, each invocation spawns a new shell along with the `find` command itself, which requires scads of system calls to open the directory, scan it, and so on. Very frequent or numerous busy polling jobs can quickly add up. (Worse, busy polling is not always appropriate. Imagine the expense and complexity if a file system browser, such as Mac OS X's Finder, polled for updates.)

So, what's an administrator to do? Happily, you can once again turn to your trusty computer for assistance.

Get to know inotify

Inotify is a Linux kernel feature that monitors file systems and immediately alerts an attentive application to relevant events, such as a delete, read, write, and even an unmount operation. You can also track the origin and destination of a move, among other niceties.

Using inotify is simple: Create a file descriptor, attach one or more watches (a *watch* is a path and set of events), and use the `read()` method to receive event information from the descriptor. Rather than burn scarce cycles, `read()` blocks until events occur.

Better yet, because inotify works through a traditional file descriptor, you can leverage the traditional `select()` system call to passively monitor your watches and a multitude of other input sources at the same time. Both approaches—blocking on a file descriptor and multiplexing with `select()`—avoid busy polling.

Now, let's peep into inotify, write a little bit of C code, and then look at a set of command-line tools you can build and use to attach commands and scripts to file system events. Inotify won't let your cat out in the middle of the night, but it can run `cat` and `wget` and do so precisely when it needs to.

To use inotify, you must have a Linux machine with kernel 2.6.13 or later. (Prior versions of the Linux kernel use a far less capable file monitor called *dnotify*). If you don't know the version of your kernel, go to the shell, and type `uname -a`:

```
% uname -a
Linux ubuntu-desktop 2.6.24-19-generic #1 SMP ... i686 GNU/Linux
```

If the kernel version listed is at least 2.6.13, your system should support inotify. You can also check your machine for the file /usr/include/sys/inotify.h. If it exists, chances are your kernel supports inotify.

Note: FreeBSD and thus Mac OS X provide an analog of inotify called *kqueue.* Type `man 2 kqueue` on a FreeBSD machine for more information.

This article is based on Ubuntu Desktop version 8.04.1 (also known as *Hardy*) running under Parallels Desktop version 3.0 on Mac OS X version 10.5 Leopard.

The inotify C API

Inotify provides three system calls to build file system monitors of all kinds:

- `inotify_init()` creates an instance of the inotify subsystem in the kernel and returns a file descriptor on success and -1 on failure. Like other system calls, if `inotify_init()` fails, check `errno` for diagnostics.
- `inotify_add_watch()`, as its name implies, adds a watch. Each watch must provide a pathname and a list of pertinent events, where each event is specified by a constant, such as IN_MODIFY. To monitor more than one event, simply use the logical *or*—the pipe (`|`) operator in C—between each event. If `inotify_add_watch()` succeeds, the call returns a unique identifier for the registered watch; otherwise, it returns -1. Use the identifier to alter or remove the associated watch.
- `inotify_rm_watch()` removes a watch.

The `read()` and `close()` system calls are also needed. Given the descriptor yielded by `inotify_init()`, call `read()` to wait for alerts. Assuming a typical file descriptor, the application

blocks pending the receipt of events, which are expressed as data in the stream. The common `close()` on the file descriptor yielded from `inotify_init()` deletes and frees all active watches as well as all memory associated with the inotify instance. (The typical reference count caveat applies here, too. All file descriptors associated with an instance must be closed before the memory consumed by the watches and by inotify is freed.)

And that's it—powerful stuff given just three application program interface (API) calls and the simple, familiar "everything is a file" paradigm. Now, you're ready to move on to an example application.

Example application: Event monitoring

Listing 1 is a short C program for monitoring a directory for two events: file creation and file deletion.

Listing 1. A simple inotify application to monitor a directory for create, delete, and modify events

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/inotify.h>

#define EVENT_SIZE  ( sizeof (struct inotify_event) )
#define BUF_LEN     ( 1024 * ( EVENT_SIZE + 16 ) )

int main( int argc, char **argv )
{
  int length, i = 0;
  int fd;
  int wd;
  char buffer[BUF_LEN];

  fd = inotify_init();

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

  wd = inotify_add_watch( fd, "/home/strike",
                          IN_MODIFY | IN_CREATE | IN_DELETE );
  length = read( fd, buffer, BUF_LEN );

  if ( length < 0 ) {
    perror( "read" );
  }

  while ( i < length ) {
    struct inotify_event *event = ( struct inotify_event * ) &buffer[ i ];
    if ( event->len ) {
      if ( event->mask & IN_CREATE ) {
        if ( event->mask & IN_ISDIR ) {
          printf( "The directory %s was created.\n", event->name );
        }
        else {
          printf( "The file %s was created.\n", event->name );
        }
      }
      else if ( event->mask & IN_DELETE ) {
        if ( event->mask & IN_ISDIR ) {
```

```
          printf( "The directory %s was deleted.\n", event->name );
        }
        else {
          printf( "The file %s was deleted.\n", event->name );
        }
      }
      else if ( event->mask & IN_MODIFY ) {
        if ( event->mask & IN_ISDIR ) {
          printf( "The directory %s was modified.\n", event->name );
        }
        else {
          printf( "The file %s was modified.\n", event->name );
        }
      }
    }
    i += EVENT_SIZE + event->len;
  }

  ( void ) inotify_rm_watch( fd, wd );
  ( void ) close( fd );

  exit( 0 );
}
```

The application creates an inotify instance with `fd = inotify_init();` and adds one watch to monitor modifications, new files, and destroyed files in /home/strike, as specified by `wd = inotify_add_watch(...)`. The `read()` method blocks until one or more alerts arrive. The specifics of the alert(s)—each file, each event—are sent as a stream of bytes; hence, the loop in the application casts the stream of bytes into a series of event structures.

You can find the definition of the event structure, a C struct, in the file /usr/include/sys/inotify.h., as shown in Listing 2.

Listing 2. Definition of an event structure

```
struct inotify_event
{
  int wd;   /* The watch descriptor */
  uint32_t mask;  /* Watch mask */
  uint32_t cookie; /* A cookie to tie two events together */
  uint32_t len;  /* The length of the filename found in the name field */
  char name __flexarr; /* The name of the file, padding to the end with NULs */
}
```

The `wd` field refers to the watch associated with the event. If you have more than one watch per inotify instance, you can use this field to determine how to proceed with further processing. The `mask` field is a set of bits that specifies what happened. Test each bit separately.

You use `cookie` to tie two events together, as when a file is moved from one directory to another. If and only if you are watching the source and destination directory, inotify generates two move events—one for the source and one for the destination—and ties the two together by setting `cookie`. To watch for a move, specify `IN_MOVED_FROM` or `IN_MOVED_TO`, or use the shorthand `IN_MOVE`, which watches for either. Use `IN_MOVED_FROM` and `IN_MOVED_TO` to test for event type.

Finally, `name` and `len` contain the file name (but not the path) and the length of the name of the file affected.

Build the sample application code

To build the code, change the directory /home/strike to your home directory, say, save the code to a file, and invoke the C compiler—typically, gcc on most Linux systems. Then, run the executable file, as shown in Listing 3.

Listing 3. Run the executable file

```
% cc -o watcher watcher.c
% ./watcher
```

With watcher running, open a second terminal window and use `touch`, `cat`, and `rm` to alter the contents of your home directory, as shown in Listing 4. After each experiment, restart your new application.

Listing 4. Use touch, cat, and rm

```
% cd $HOME
% touch a b c
The file a was created.
The file b was created.
The file c was created.

% ./watcher  &
% rm a b c
The file a was deleted.
The file b was deleted.
The file c was deleted.

% ./watcher  &
% touch a b c
The file a was created.
The file b was created.
The file c was created.

% ./watcher  &
% cat /etc/passwd >> a
The file a was modified.

% ./watcher  &
% mkdir d
The directory d was created.
```

Experiment with the other available watch flags. To catch changes to permissions, add `IN_ATTRIB` to the mask.

Tips for using inotify

You can also experiment with `select()`, `pselect()`, `poll()`, and `epoll()` to avoid blocking, which is useful if you want to monitor watches as part of a graphical application's main event processing loop or as part of a daemon that watches for other kinds of incoming connections. Simply add the inotify descriptor to the set of descriptors to monitor concurrently. Listing 5 shows a canonical form for `select()`.

Listing 5. Canonical form for select()

```
int return_value;
```

```
fd_set descriptors;
struct timeval time_to_wait;

FD_ZERO ( &descriptors );
FD_SET( ..., &descriptors );
FD_SET ( fd, &descriptors );

...

time_to_wait.tv_sec = 3;
time.to_waittv_usec = 0;

return_value = select ( fd + 1, &descriptors, NULL, NULL, &time_to_wait);

if ( return_value < 0 ) {
 /* Error */
}

else if ( ! return_value ) {
 /* Timeout */
}

else if ( FD_ISSET ( fd, &descriptors ) ) {
 /* Process the inotify events */
 ...
}

else if ...
```

The `select()` method pauses the program for `time_to_wait` seconds. However, if any activity occurs on any of the file descriptors in the set descriptors during that delay, execution resumes immediately. Otherwise, the call times out, allowing the application to do other processing, such as respond to mouse or keyboard events in a graphical user interface (GUI) tool.

Here are some other tips for inotify:

- If a file or directory under observation is deleted, its watches are removed automatically (after a delete event is delivered, if appropriate).
- If you're monitoring a file or directory on a file system that is unmounted, your watch receives an unmount event before all affected watches are deleted.
- Add the `IN_ONESHOT` flag to the watch mask to set a one-time alert. After the alert is sent once, it is deleted.
- To modify an event, provide the same pathname but a different mask. The new watch replaces the old one.
- For all practical purposes, you're unlikely to run out of watches in any given inotify instance. However, you can run out of space in your event queue, depending on how often you process events. A queue overflow causes the `IN_Q_OVERFLOW` event.
- The `close()` method destroys the inotify instance and all associated watches and empties all pending events in the queue.

Installing the inotify-tools suite

The inotify programming interface is simple to use, but if you'd prefer not to write your own tool, open source provides a nice, flexible alternative. The Inotify-tools library (see Resources below for a link) provides a pair of command-line utilities to monitor file system activity:

- `inotifywait` simply blocks to wait for inotify events. You can monitor any set of files and directories and monitor an entire directory tree (a directory, its subdirectories, its sub-subdirectories, and so on). Use `inotifywait` in shell scripts.
- `inotifywatch` collects statistics about the watched file system, including how many times each inotify event occurred.

As of this writing, the latest version of the inotify-tools library is version 3.13, released on 1 Jan 2008. There are two ways to install inotify-tools: You can download and build the software yourself, or you can install a collection of binaries using your Linux distribution's package manager, if a known repository contains inotify-tools. To do the latter on a Debian-based distribution, run `apt-cache search inotify`, and look for matching tools, as shown in Listing 6. On the example system I used to write this article, Ubuntu Desktop version 8.04, the tools are readily available.

Listing 6. Search for inotify-tools

```
% apt-cache search inotify
incron - cron-like daemon which handles filesystem events
inotail - tail replacement using inotify
inoticoming - trigger actions when files hit an incoming directory
inotify-tools - command-line programs providing a simple interface to inotify
iwatch - realtime filesystem monitoring program using inotify
libinotify-ruby - Ruby interface to Linux's inotify system
libinotify-ruby1.8 - Ruby interface to Linux's inotify system
libinotify-ruby1.9 - Ruby interface to Linux's inotify system
libinotifytools0 - utility wrapper around inotify
libinotifytools0-dev - Development library and header files for libinotifytools0
liblinux-inotify2-perl - scalable directory/file change notification
muine-plugin-inotify - INotify Plugin for the Muine music player
python-kaa-base - Base Kaa Framework for all Kaa Modules
python-pyinotify - Simple Linux inotify Python bindings
python-pyinotify-doc - Simple Linux inotify Python bindings
% sudo apt-get install inotify-tools
...
Setting up inotify-tools.
```

Building the code is easy, though. Download the source; extract it; then configure, compile, and install it as shown in Listing 7. The entire process might take three minutes.

Listing 7. Building the code

```
% wget \
    http://internap.dl.sourceforge.net/sourceforge/inotify-tools/inotify-tools-3.13.tar.gz
% tar zxvf inotify-tools-3.13.tar.gz
inotify-tools-3.13/
inotify-tools-3.13/missing
inotify-tools-3.13/src/
inotify-tools-3.13/src/Makefile.in
...
inotify-tools-3.13/ltmain.sh

% cd inotify-tools.3.13
% ./configure
% make
% make install
```

You're now ready to use the tools. For example, if you want to monitor your entire home directory for changes, run `inotifywait` . The simplest invocation is `inotifywait -r -m`, which recursively monitors its arguments (`-r`) and leaves the utility running after each event (`-m`):

```
% inotifywait -r  -m $HOME
Watches established.
```

Run another terminal window, and tinker with your home directory. Interestingly, even a simple directory listing with `ls` generates an event:

```
/home/strike OPEN,ISDIR
```

Read the `inotifywait` man page for options to restrict events to a specific list (use the `-e` *`event_name`* option repeatedly to create the list), and exclude matching files (`--exclude` *`pattern`*) from recursive watches.

Stay "inotified"

As `apt-cache` revealed above, there are other inotify-based utilities to consider adding to your bag of tricks. The `incron` utility is a corollary to cron but reacts to inotify events instead of a schedule. The `inoticoming` utility is specifically designed to monitor dropboxes. And if you're a Perl, Ruby, or Python developer, you can find modules and libraries to call inotify from the comfort of your favorite scripting language.

For instance, Perl coders can use `Linux::Inotify2` (see Resources for details) to embed inotify features in any Perl application. This code, taken from the `Linux::Inotify2` README file, demonstrates a callback interface to monitor events, as shown in Listing 8.

Listing 8. A callback interface monitors events

```
use Linux::Inotify2;

my $inotify = new Linux::Inotify2
 or die "Unable to create new inotify object: $!";

 # for Event:
 Event->io (fd =>$inotify->fileno, poll => 'r', cb => sub { $inotify->poll });

 # for Glib:
 add_watch Glib::IO $inotify->fileno, in => sub { $inotify->poll };

 # manually:
 1 while $inotify->poll;

 # add watchers
 $inotify->watch ("/etc/passwd", IN_ACCESS, sub {
    my $e = shift;
    my $name = $e->fullname;
    print "$name was accessed\n" if $e->IN_ACCESS;
    print "$name is no longer mounted\n" if $e->IN_UNMOUNT;
    print "$name is gone\n" if $e->IN_IGNORED;
    print "events for $name have been lost\n" if $e->IN_Q_OVERFLOW;

    # cancel this watcher: remove no further events
    $e->w->cancel;
 });
```

Because everything in Linux is a file, you'll no doubt find countless uses for inotify watches.

So, the real question is, "Who watches the watches?"

Resources

Learn

- Learn more about the history of inotify.
- Still bent on using cron? "Linux tip: Job scheduling with cron and at" (developerWorks, July 2007) and "Linux tip: Controlling the duration of scheduled jobs" (developerWorks, July 2007) will help you get more out of cron.
- Read "Systems Administration Toolkit: Log file basics" (developerWorks, February 2008) for more on managing log files.
- Read more about incron, a corollary to cron that runs jobs in reaction to file system events.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Learn more about `Linux::Inotify2`.
- Download the source code for inotify-tools, a set of command-line utilities to monitor file system events.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

Martin Streicher

Martin Streicher is a freelance Web developer and the former Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has programmed UNIX-like systems since 1986. He collects art and toys.