Introduction

In a multi-user, multi-process operating system, files are continually being created, modified and deleted, often by apparently unrelated processes. This means that any software that needs to keep aware of what is happening in a filesystem needs to employ a file monitoring technique.

Monitoring, in this sense, means keeping a watch over a set of files, waiting for any of them to change. Applications that benefit from monitoring techniques include those that need to keep their displayed filesystem information up to date, long running server tools that should automatically reread their configuration files when those files change, applications that watch for and respond to unusual file modifications, system message file monitors, and so on.

Most modern operating systems provide file monitoring facilities to give applications realtime information about changes to the filesystem. A variety of notification methods are used to tell the application when a change happens, ranging from an asynchronous signal being sent from the kernel, through to a user space tool printing the name of the changed file on its standard output. We'll take a look at some of the file monitoring facilities available to the Linux developer, starting with the lowest level mechanism and working up to the highest.

dnotify

All files are ultimately created, updated and deleted by the Linux kernel, so it makes sense that the most direct way for an application to learn that a file has changed is for the kernel to tell it. This is the idea behind the kernel's dnotify feature. dnotify stands for directory notification, and although the feature has been in the kernel for some time, it can only be relied upon in kernels 2.4.19 and later, which was when a tiny but very important bug was fixed.

dnotify works by sending a signal to a process that has asked to be informed when the content of a directory is changed. Being a low level feature, dnotify is normally used from C code. The developer needs to open the appropriate directory file for reading, then issue a `fcntl(2)` system call specifying the F_NOTIFY subcommand. Flags passed to this subcommand can specify which events the process wants to be notified about: file creation, deletion, access, modification, renaming, or access rights being set with chmod. By default, whenever any of the monitored operations happen to any file in the directory, the monitoring process will receive the SIGIO signal from the kernel. The signal used is configurable using the `fcntl(2)` F_SETSIG subcommand, and in order to ensure no signals are missed it is best to change the signal generated to one of the POSIX4 real-time ones, which can be queued.

Note that we are talking about *directory* notification here, not files. If only a single file needs to be watched its directory should be dnotified and the file examined each time the monitoring process receives the signal. Listing 1 shows an example of some C code that watches a particular file waiting for it to be edited.

```
#define _GNU_SOURCE
#include <fcntl.h>

#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
```

```
int requiredEvent = 0;

void
eventHandler( int sig )
{
  requiredEvent = 1;
}


int
main( int argc, char *argv[] )
{
  struct stat      statBuf;
  struct sigaction action;
  int              fd;
  char            *dir;
  char            *slashPtr;
  char             splitFilename[128];

  if( argc != 2 ) {
    printf( "Usage: %s filename", argv[0] );
    exit( -1 );
  }

  if( lstat( argv[1], &statBuf ) < 0 ) {
    printf( "Cannot lstat %s : %s\n", argv[1], strerror(errno) );
    exit( -1 );
  }

  if( ! S_ISREG( statBuf.st_mode ) ) {
    printf( "%s is not a regular file\n", argv[1] );
    exit( -1 );
  }

  /* Find directory the file is in, or assume the current directory */
  strncpy( splitFilename, argv[1], 127 );
  if( (slashPtr = rindex( splitFilename, '/' )) == NULL ) {
    dir       = ".";
  } else {
    *slashPtr = 0;
    dir       = splitFilename;
  }

  /* Set up the signal handler */
  action.sa_handler = eventHandler;
  sigemptyset( &action.sa_mask );
  action.sa_flags = SA_RESTART;
  sigaction( SIGRTMIN, &action, NULL );

  /* Open the directory the file is in */
  if( (fd = open( dir, O_RDONLY )) < 0 ) {
    printf( "Cannot open %s for reading : %s\n", dir, strerror(errno) );
    exit( -1 );
  }

  /* Choose the signal I want to receive when the directory content changes */
  if( fcntl( fd, F_SETSIG, SIGRTMIN) < 0 ) {
    printf( "Cannot set signal : %s\n", strerror(errno) );
    exit( -1 );
  }

  /* Ask for notification when a modification is made in the directory */
  if( fcntl( fd, F_NOTIFY, DN_MODIFY|DN_MULTISHOT ) < 0 ) {
    printf( "Cannot fcntl %s : %s\n", dir, strerror(errno) );
    exit( -1 );
  }

  /* Infinite loop - a real program would be doing stuff */
  while( 1 ) {
    time_t modifiedTime;

    /* This demo has nothing to do, so just wait for a signal */
    pause();

    /* Check the flag which indicates the right signal has been received */
    if( requiredEvent ) {
```

```
      /* Something has been modified in the directory - stat our file */
      modifiedTime = statBuf.st_mtime;

      if( lstat( argv[1], &statBuf ) < 0 ) {
        printf( "Cannot lstat %s : %s\n", argv[1], strerror(errno) );
        exit( -1 );
      }

      /* If the modification time has changed, the file has been altered */
      if( modifiedTime != statBuf.st_mtime ) {
        printf( "File %s has been modified\n", argv[1] );
      }

      requiredEvent = 0;
    }
  }

}
```

This program finds the directory the given file is in, then starts monitoring that directory for file modifications. When a notification arrives, the program checks the monitored file with the `lstat(2)` system call to see if the modification time has been changed.

Compile the example code with `cc -o checkchanged checkchanged.c` and try it:

```
> mkdir testdir
> touch testdir/datafile
> ./checkchanged testdir/datafile
```

Now, in another shell do something like `echo data >> testdir/datafile`. The checkchanged program immediately prints `File testdir/datafile has been modified`.

This example uses the multishot feature of dnotify. Normally, once a notification is issued, dnotify stops monitoring since it is, by default, a single event system. If the DN_MULTISHOT flag is passed to the `fcntl(2)` F_NOTIFY subcommand, the monitoring stays in place and a signal is sent each and every time a change to the directory is made.

FAM

The main problem with dnotify is that it is Linux specific. It is also not particularly simple to work with, and if a lot of files need monitoring in lots of different directories, it can get rather complicated.

Silicon Graphics Inc.'s File Alteration Monitor, or FAM, was designed to simplify file monitoring, and to work on systems without a dnotify type feature. It is now one of SGI's contributions to the free software community.

FAM ideally requires the operating system to have SGI's inode monitoring pseudo device driver available. This is called imon and is available for Linux as a kernel patch. There is also a patch available to FAM which allows it to use Linux's dnotify feature instead of imon, and this patch is applied to the FAM package by virtually all Linux distributors. Where neither imon nor dnotify are available, FAM falls back to polling for file changes. Although this might appear to defeat the object somewhat, it does have the advantage that a single process ends up doing all the polling for all monitoring applications.

FAM comes in two parts: a daemon and a user level library. The system administrator will configure the daemon to either be started at boot time, or from xinetd as required. It is this daemon process that sits watching for file alterations. Applications, known as FAM clients, link to the user level library which provides routines to connect to the daemon and ask it to monitor a given set of files and/or directories. When a change

happens, the daemon sends the relevant information down a socket to the client application.

An important difference between dnotify and FAM is that FAM doesn't work in an asynchronous way. The client program is expected to either block waiting for some information to arrive on its socket, or to occasionally poll the socket to see if anything has happened. If asynchronous behaviour is required by a FAM client, that client should fork off a dedicated process which can block on the socket, and which can send a signal to its parent when something happens.

Working with FAM is a lot more straightforward than dnotify. The procedure is basically to open a connection to the daemon, use that connection to tell the daemon what you want to monitor, then either do a blocking read on the connection, or occasionally poll it, waiting for the daemon to tell you your file has changed. When you are finished monitoring, tell the daemon to stop, then close the connection. All of these steps are performed using calls to functions in the FAM library. See the documentation for details. Listing 2 shows a simple FAM client, written in C:

```
#include <fam.h>

#include <stdio.h>
#include <errno.h>
#include <signal.h>

void
ctrlCCatcher( int sig ) { /* No action, just EINTR the system call */ }

int
main( int argc, char *argv[] )
{
  FAMConnection    famConn;
  FAMRequest       famRequest;
  FAMEvent         famEvent;
  struct sigaction action;

  if( argc != 2 ) {
    printf( "Usage: %s filename\n", argv[0] );
    exit( -1 );
  }

  action.sa_handler = ctrlCCatcher;
  sigemptyset( &action.sa_mask );
  sigaction( SIGINT, &action, NULL );

  if( FAMOpen( &famConn ) < 0 ) {
    printf( "Cannot connect to FAM : %s\n", strerror(errno) );
    exit( -1 );
  }

  if( FAMMonitorFile( &famConn, argv[1], &famRequest, NULL ) < 0 ) {
    printf( "Cannot monitor file %s : %s\n", argv[1], strerror(errno) );
    FAMClose( &famConn );
    exit( -1 );
  }

  while( 1 ) {
    if( FAMNextEvent( &famConn, &famEvent ) < 0 ) {
      if( errno == EINTR ) {
        printf( "Interrupted\n" );
        break;
      }

      printf( "Error retrieving next FAM event : %s\n", strerror(errno) );
      FAMClose( &famConn );
      exit( -1 );
    }

    if( famEvent.code == FAMChanged ) {
```

```
      printf( "FAM indicates file %s has changed\n", famEvent.filename );
    }
    if( famEvent.code == FAMDeleted ) {
      printf( "FAM indicates file %s has been deleted\n", famEvent.filename );
    }
  }

  if( FAMClose( &famConn ) < 0 ) {
    printf( "Cannot close FAM connection : %s\n", strerror(errno) );
    exit( -1 );
  }

  exit( 0 );
}
```

This should be complied with a command like `cc -o famchanged -lfam famchanged.c`. The file to be monitored should be passed to the program as a command line argument, and *must* be an absolute pathname, starting with a '/'. For demonstration purposes this code makes the effort to trap a Ctrl-C and close the connection to the FAM daemon neatly. In normal practise it's safe to just exit the program and have the FAM connection close automatically.

FAM can also be used from Perl, using the SGI::FAM module from CPAN. With this module you can write code like Listing 3:

```
#!/usr/bin/perl -w
use strict;

use SGI::FAM;

my $conn = new SGI::FAM;
$conn->monitor( "/var/log/messages" );

while (1) {
  my $event = $conn->next_event;
  print $event->filename, " ", $event->type, "\n";
}
```

There's a Python FAM module too:

```
#!/usr/bin/python

import _fam

conn    = _fam.open()
request = conn.monitorFile( "/var/log/messages", None )

while True:
    event = conn.nextEvent()
    print event.filename, event.code2str()
```

Being a user space tool, FAM can do a lot more than is possible with a kernel based facility like dnotify. For example, if a file to be monitored is in an NFS directory, FAM will attempt to contact the FAM daemon running on the NFS server. The server's FAM daemon will then do the actual monitoring, and when the file changes the information will be relayed via the local daemon to the client. If the server isn't running a FAM daemon then the local daemon will resort to polling the file. See the FAM documentation for details on how to configure this and other more specialised FAM features.

FilesChanged

If you only have a simple task which requires file monitoring it is often possible to do it from shell script. The `fileschanged` utility is a FAM client program designed to be used from a script. Although not as flexible as a hand coded FAM client, it is often a quick and simple solution to a problem.

`fileschanged` has recently been updated to version 0.6.0. Previous versions contained a bug which prevented the tool working with bash under some circumstances. Ensure you have the latest version before experimenting with it.

`fileschanged` takes the names of the files to monitor on its command line and can report when any of the files are created, deleted, changed or executed. When one of these actions happens, the name of the file is printed to standard output.

This is simple, but effective. If you're working on a shared source tree, and you want to keep a log of which source files are being updated by your colleagues, you might take advantage of `fileschanged`'s recursive directories feature, and run something like this:

```
fileschanged –r /shared/source/tree | while read fd; do
        stat –format=”%n modified by %U at %y” $fc
done
```

`fileschanged` is often used to monitor log files or incoming mailbox files. It's also possible to use it for more complex tasks such as watching and backing up rarely changed configuration files.

Conclusion

File monitoring is one of those issues of software development that is often classed as the icing on the cake. A user can always press F5 to refresh a file manager window, or send a SIGHUP to a program when they've changed its configuration file, so automatically monitoring for changes isn't very often essential. It is, however, something which can make a program more intuitive, or easier to use. The system facilities that provide file monitoring are now widespread and mature, so users will always benefit if developers look for ways to utilise them.