19 Advanced File Handling

In This Chapter

- Review of File Types and Attributes
- File System Traversal
- File Mapping
- File Event Notification
- Buffering and Syncing

Introduction

This chapter continues where Chapter 11, "File Handling in GNU/Linux," left off and explores some of the more advanced features of file handling. This includes a discussion of file types and their testing, special files, directory traversal, file system mapping, and many other operations.

Testing File Types

You can start with the simple task of determining a file's type. The type is the classification of the file (for example, whether it's a regular file, symbolic link, or one of the many special files). The GNU C library provides a number of functions that provide this classification, but first you need to retrieve status information of the file for the test using the `stat` command.

The `stat` command returns a number of different data elements about a file. This section focuses on the file type, and then some of the other elements are covered in the next section.

With the `stat` information, you use the `st_mode` field in particular to determine the file type. This field is a coding of the file type, but the GNU C library provides a set of macros that can be used to test for a given file type (returns 1 if the file is of the defined type, otherwise 0). These are shown in Table 19.1.

TABLE 19.1 File Test Macros for `stat`

Macro	Description (Returns 1 If the File Is of That Type)
`S_ISREG`	Regular file
`S_ISDIR`	Directory
`S_ISCHR`	Special character file
`S_ISBLK`	Special block file
`S_ISLNK`	Symbolic link
`S_ISFIFO`	Special FIFO file
`S_ISSOCK`	Socket

To use `stat`, you call it with a buffer that represents a structure of the information to be captured. Listing 19.1 provides a simple application that uses the filename provided as the argument and then uses `stat` and the file test macros to determine the file type.

LISTING 19.1 Using the File Test Macros to Determine the File Type (on the CD-ROM at `./source/ch19/ftype.c`)

```
 1:  int main( int argc, char *argv[] )
 2:  {
 3:    struct stat statbuf;
 4:
 5:    /* Check to see if a filename was provided */
 6:    if (argc < 2) {
 7:      printf("specify a file to test\n");
 8:      return 1;
 9:    }
10:
11:    /* Obtain stat information on the file */
12:    if ( stat(argv[1], &statbuf) == -1 ) {
13:      printf("Can't stat file\n");
14:      return 1;
```

```
15:     }
16:
17:     printf("%s ", argv[1]);
18:
19:     /* Determine the file type from the st_mode */
20:     if        (S_ISDIR(statbuf.st_mode)) {
21:       printf("is a directory\n");
22:     } else if (S_ISCHR(statbuf.st_mode)) {
23:       printf("is a character special file\n");
24:     } else if (S_ISBLK(statbuf.st_mode)) {
25:       printf("is a block special file\n");
26:     } else if (S_ISREG(statbuf.st_mode)) {
27:       printf("is a regular file\n");
28:     } else if (S_ISFIFO(statbuf.st_mode)) {
29:       printf("is a FIFO special file\n");
30:     } else if (S_ISSOCK(statbuf.st_mode)) {
31:       printf("is a socket\n");
32:     } else {
33:       printf("is unknown\n");
34:     }
35:
36:     return 0;
37:   }
```

Using this application is demonstrated in the following with a variety of file types:

```
$ ./ftype ..
.. is a directory
$ ./ftype /dev/mem
/dev/mem is a character special file
$ ./ftype /dev/fd0
a block special file
$ ./ftype /etc/resolv.conf
/etc/resolv.conf is a regular file
$
```

The `stat` command provides much more than just the file type through `st_mode`. The next section takes a look at some of the other attributes that are returned through `stat`.

Other `stat` information

The `stat` command provides a wealth of information about a file, including its size, owner, group, last modified time, and more. Listing 19.2 provides a simple program that implements the emission of the previously mentioned data elements, making use of other helper functions to convert the raw data into human-readable strings. In particular, the `strftime` function takes a time structure (standard time format) and converts it into a string. This function offers a large number of options and output formats.

LISTING 19.2 Using the `stat` Command to Capture File Data (on the CD-ROM at `./software/ch19/ftype2.c`)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <langinfo.h>

int main( int argc, char *argv[] )
{
  struct stat    statbuf;
  struct passwd *pwd;
  struct group  *grp;
  struct tm     *tm;
  char           tmstr[257];

  /* Check to see if a filename was provided */
  if (argc < 2) {
    printf("specify a file to test\n");
    return 1;
  }

  /* Obtain stat information on the file */
  if ( stat(argv[1], &statbuf) == -1 ) {
    printf("Can't stat file\n");
    return 1;
  }

  printf("File Size : %-d\n", statbuf.st_size);

  pwd = getpwuid(statbuf.st_uid);
```

```
32:     if (pwd) printf("File Owner: %s\n", pwd->pw_name);
33:
34:     grp = getgrgid(statbuf.st_gid);
35:     if (grp) printf("File Group: %s\n", grp->gr_name);
36:
37:     tm = localtime(&statbuf.st_mtime);
38:     strftime(tmstr, sizeof(tmstr), nl_langinfo(D_T_FMT), tm);
39:     printf("File Date : %s\n", tmstr);
40:
41:     return 0;
42:   }
```

Executing the new `stat` application (called `ftype2`) produces the following for the `passwd` file:

```
$ ./ftype /etc/passwd
File Size : 1337
File Owner: root
File Group: root
File Date : Sun Jan  6 19:44:50 2008
$
```

The same information can be retrieved with the `fstat` command, although this command operates on a file descriptor instead of a filename.

Determining the Current Working Directory

You can learn the name of the current working directory, including the full path, with the `getcwd` command. Another function called `pathconf` enables you to determine the size of buffer necessary to pass to `getcwd`. This process is demonstrated in Listing 19.3. After the maximum path length is known, a buffer is allocated with `malloc` and then passed to `getcwd` to return the current working directory. After this is emitted to `stdout`, the buffer is freed.

LISTING 19.3 Getting the Current Working Directory with `getcwd` (on the CD-ROM at `./software/ch19/gcwd.c`)

```
1:  #include <stdio.h>
2:  #include <unistd.h>
3:  #include <stdlib.h>
4:
5:  int main()
6:  {
```

```
 7:    char *pathname;
 8:    long  maxbufsize;
 9:
10:    maxbufsize = pathconf( ".", _PC_PATH_MAX );
11:    if (maxbufsize == -1) return 1;
12:
13:    pathname = (char *)malloc(maxbufsize);
14:
15:    (void)getcwd( pathname, (size_t)maxbufsize );
16:
17:    printf("%s\n", pathname);
18:
19:    free(pathname);
20:
21:    return 0;
22:  }
```

Enumerating Directories

GNU/Linux provides variety of directory manipulation functions that are useful for enumerating a directory or an entire filesystem. Similar to operating on a file, you must first open the directory and then read its contents. In this case, the contents of the directory are the files themselves (which can be other directories).

To open a directory, you use the `opendir` API function. This returns a directory stream (a reference of type `DIR`). After the stream is opened, it can be read with the `readdir` function. Each call to `readdir` returns a directory entry or `NULL` if the enumeration is complete. When the directory stream is no longer needed, it should be closed with `closedir`.

The following simple application (Listing 19.4) illustrates using the API functions discussed so far (`opendir`, `readdir`, `closedir`). In this example, you enumerate a directory (specified on the command line) and emit the `stdout` only those directory entries that refer to regular files.

LISTING 19.4 Enumerating a Directory with `readdir` (on the CD-ROM at `./software/ch19/dirlist.c`)

```
1:  #include <stdio.h>
2:  #include <sys/types.h>
3:  #include <sys/stat.h>
4:  #include <dirent.h>
5:
6:  int main( int argc, char *argv[] )
```

```
 7:  {
 8:    DIR *dirp;
 9:    struct stat statbuf;
10:    struct dirent *dp;
11:
12:    if (argc < 2) return 1;
13:
14:    if (stat(argv[1], &statbuf) == -1) return 1;
15:
16:    if (!S_ISDIR(statbuf.st_mode)) return 1;
17:
18:    dirp = opendir( argv[1] );
19:
20:    while ((dp = readdir(dirp)) != NULL) {
21:
22:      if (stat(dp->d_name, &statbuf) == -1) continue;
23:      if (!S_ISREG(statbuf.st_mode)) continue;
24:
25:      printf("%s\n", dp->d_name);
26:
27:    }
28:
29:    closedir(dirp);
30:
31:    return 0;
32:  }
```

During enumeration, you can remember a position in the directory stream using `telldir`, and then return to this later with a call to `seekdir`. To return to the beginning of the stream, you use `rewinddir`.

Another operation for walking a directory is called `ftw` (which is an acronym for "file tree walk"). With this API function, you provide a directory to start the walk and then a function that is called for each file or directory found during the walk.

For each file that's encountered during the walk, a callback function is called and passed the filename, the `stat` structure, and a flag word that indicates the type of file (`FTW_D` for a directory, `FTW_F` for a file, `FTW_SL` for a symbolic link, `FTW_NS` for a file that couldn't be `stat`'ed, and `FTW_DNR` for a directory that could not be read).

An example of the use of `ftw` is shown in Listing 19.5. The `fncheck` function is the callback function called for each file encountered. The `main` function takes the argument passed as the starting directory to walk. This is provided to the `ftw` function along with the callback function and the `ndirs` argument. This argument

defines the number of directory streams that can be simultaneously open (better performance results with a larger number streams).

LISTING 19.5 Walking a Directory with `ftw` (on the CD-ROM at `./software/ch19/dwalk.c`)

```
 1:  #include <stdio.h>
 2:  #include <ftw.h>
 3:
 4:  int fncheck( const char *file, const struct stat *sb, int flag )
 5:  {
 6:    if (flag == FTW_F)
 7:      printf("File %s (%d)\n", file, (int)sb->st_size);
 8:    else if (flag == FTW_D)
 9:      printf("Directory %s\n", file);
10:
11:    return 0;
12:  }
13:
14:  int main( int argc, char *argv[] )
15:  {
16:    if (argc < 2) return 1;
17:
18:    ftw( argv[1], fncheck, 1 );
19:
20:    return 0;
21:  }
```

The final enumeration technique that this section explores is called globbing. The `glob` API function allows you to specify a regular expression representing the files to be returned. The results are stored in a glob buffer, which can be enumerated when the function completes. An example of the `glob` API function is provided in Listing 19.6. The first argument is the pattern (in this case files that end `.c` or `.h`). The second argument is one or more flags (for example `GLOB_APPEND` can be specified to append a new glob to a previous invocation). The third argument is an error function that is called when a directory is attempted to be opened but fails. Finally, the results are stored in the glob buffer (which entails internal allocation of memory). For this reason, you have to free this buffer with `globfree` when it's no longer needed.

This simple program (Listing 19.6) results in a printout of files that end in `.c` or `.h` in the current directory.

LISTING 19.6 Demonstrating the `glob` API Function (on the CD-ROM at `./software/ch19/globtest.c`)

```
 1:  #include <stdio.h>
 2:  #include <glob.h>
 3:
 4:  int main()
 5:  {
 6:    glob_t globbuf;
 7:    int    err, i;
 8:
 9:    err = glob( "*.[ch]", 0, NULL, &globbuf );
10:
11:    if (err == 0) {
12:
13:      for (i = 0 ; i < globbuf.gl_pathc ; i++) {
14:
15:        printf("%s\n", globbuf.gl_pathv[i]);
16:
17:      }
18:
19:      globfree( &globbuf );
20:
21:    }
22:
23:    return 0;
24:  }
```

GNU/Linux (the GNU C Library) provides a number of options for enumerating a subdirectory, whether you want to view all files, files based upon a regular expression, or through a callback means for each file.

File Event Notification with `inotify`

One of the new features in the 2.6 kernel provides support for notification of filesystem events. This new mechanism is called `inotify`, which refers to the fact that it provides for notification of inode events. This capability of the kernel has been used in a variety of applications, the most visible being the Beagle search utility. Beagle uses `inotify` to automatically notify of changes to the filesystem to perform indexing. The alternative is to periodically scan the filesystem for changes, which is very inefficient.

What's interesting about `inotify` is that it uses the file-based mechanisms to provide for notification about filesystem events. For example, when monitoring is

requested, events are provided to the user-space application through a file descriptor. The `inotify` API provides system calls that wrap these mechanisms. It's time to take a look at this capability in more detail.

Notification Process

The basic flow of `inotify` is first to provide a monitoring point. This can be a file or a directory. If a directory is provided (which could be an entire filesystem), then all elements of that directory are candidates for event monitoring. After a watchpoint has been defined, you can read from a file descriptor to receive events.

`inotify` Events

An event (whose structure is shown in Listing 19.7) provides an event type (`mask`), the name of the file affected (if applicable), and other elements.

LISTING 19.7 The `inotify_event` Structure

```
struct inotify_event
{
  int wd;               /* Watch descriptor.  */
  uint32_t mask;        /* Watch mask.  */
  uint32_t cookie;      /* Cookie to sync two events.  */
  uint32_t len;         /* Length (with NULs) of name.  */
  char name __flexarr;  /* Name.  */
};
```

The `mask` field shown in Listing 19.7 is the event tag that indicates the type of operation that was performed (and is being notified). The current list of events is provided in Table 19.2.

TABLE 19.2 Events Supported in the Current Version of `inotify`

Event Name	Value	Description
IN_ACCESS	0x00000001	File was accessed.
IN_MODIFY	0x00000002	File was modified.
IN_ATTRIB	0x00000004	Attributes were changed.
IN_CLOSE_WRITE	0x00000008	File was closed (writeable).
IN_CLOSE_NOWRITE	0x00000010	File was closed (non-writeable).
IN_CLOSE	0x00000018	File was closed (both).

→

Event Name	Value	Description
IN_OPEN	0x00000020	File was opened.
IN_MOVED_FROM	0x00000040	File was moved (from "name").
IN_MOVE_TO	0x00000080	File was moved (to "name").
IN_MOVE	0x000000C0	File was moved (either).
IN_CREATE	0x00000100	File was created.
IN_DELETE	0x00000200	File was deleted.
IN_DELETE_SELF	0x00000400	The file watchpoint was deleted.
IN_MOVE_SELF	0x00000800	The file watchpoint was moved.
IN_UNMOUNT	0x00002000	The filesystem containing file was unmounted.
IN_Q_OVERFLOW	0x00004000	Event queue overflow.
IN_ISDIR	0x40000000	Event file refers to a directory.
IN_ALL_EVENTS	0x00000FFF	All events.

Simple `inotify-Based` Application

Now it's time to take a look at a simple application that registers for events on a user-defined file and emits those events in a human-readable fashion. To begin you initialize `inotify` (which creates an instance of `inotify` for your application). This call to `inotify_init` returns a file descriptor that serves as your means to receive events from the kernel. You also register a watchpoint and then monitor the file (covered later). The `main` function is provided in Listing 19.8.

LISTING 19.8 The `main` Function for Your `inotify` Application (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
 1:  #include <stdio.h>
 2:  #include <sys/inotify.h>
 3:  #include <errno.h>
 4:
 5:  int main( int argc, char *argv[] )
 6:  {
 7:    int ifd, err;
 8:
 9:    int register_watchpoint( int fd, char *dir );
10:    int watch( int fd );
11:
```

```
12:     if (argc < 2) return -1;
13:
14:     ifd = inotify_init();
15:
16:     if (ifd < 0) {
17:       printf("can't init inotify (%d)\n", errno);
18:       return -1;
19:     }
20:
21:     err = register_watchpoint( ifd, argv[1] );
22:     if (err < 0) {
23:       printf("can't add watch (%d)\n", errno);
24:       return -1;
25:     }
26:
27:     watch(ifd);
28:
29:     close(ifd);
30:
31:     return 0;
32:   }
```

Note that because events are posted from the kernel through a file descriptor, you can use the traditional mechanisms on that descriptor such as select to identify when an event is available (rather than sitting on the file).

Next, you have a look at the registration function (`register_watchpoint`). This function, shown in Listing 19.9, uses the `inotify_add_watch` API function to register for one or more events. As shown here, you specify the file descriptor (returned from `inotify_init`), the file to monitor (which was grabbed as the command-line argument), and then the event mask. Here you specify all events with the `IN_ALL_EVENTS` symbolic constant.

LISTING 19.9 Registering for an `inotify` Event (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
1:  int register_watchpoint( int fd, char *dir )
2:  {
3:    int err;
4:
5:    err = inotify_add_watch( fd, dir, IN_ALL_EVENTS );
6:
7:    return err;
8:  }
```

Now that you've registered for events, they queue to your file descriptor. The `watch` function is used as your monitoring loop (as called from `main` and shown in Listing 19.10). In this function you declare a buffer for your incoming events (which can be multiple per invocation) and then enter your monitoring loop. You can call the `read` system call to wait for events (as this call blocks until an event is received). When this returns, you do some quick sanity checking and then walk through each event that's contained in the buffer. This event is emitted with a call to `emit_event`. Because events can vary in size (considering a variable length filename), you increment the index by the size of the event and the size of the accompanying file name (if present). The filename is indicated by a nonzero `len` field within the event structure. When all events have been emitted, you start again by sitting on the `read` function.

LISTING 19.10 The Event Monitoring Loop (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
#define MAX_EVENTS      256

#define BUFFER_SIZE     (MAX_EVENTS * sizeof(struct inotify_event))

int watch( int fd )
{
  char ev_buffer[ BUFFER_SIZE ];
  struct inotify_event *ievent;
  int  len, i;

  void emit_event( struct inotify_event *ievent );

  while (1) {

    len = read( fd, ev_buffer, BUFFER_SIZE );

    if (len < 0) {
      if (errno == EINTR) continue;
    }

    i = 0;
    while (i < len) {

      ievent = (struct inotify_event *)&ev_buffer[i];

      emit_event( ievent );

```

```
28:         i += sizeof(struct inotify_event) + ievent->len;
29:
30:       }
31:
32:     }
33:
34:     return 0;
35:   }
```

The final function, `emit_event` (shown in Listing 19.11), parses the event structure and emits the information to `stdout`. First, you check to see if a string is provided, and if so, you emit this first. This name refers to the filename that was affected (directory or regular file). You then test each of the event tags to see which is present and then emit this to `stdout`.

LISTING 19.11 Parsing and Emitting the `event` Structure (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
 1:  void emit_event( struct inotify_event *ievent )
 2:  {
 3:    if (ievent->len) printf("[%s] ", ievent->name);
 4:
 5:    if ( ievent->mask & IN_ACCESS ) printf("Accessed\n");
 6:    if ( ievent->mask & IN_MODIFY ) printf("Modified\n");
 7:    if ( ievent->mask & IN_ATTRIB ) printf("Attributes Changed\n");
 8:    if ( ievent->mask & IN_CLOSE_WRITE ) printf("Closed
       (writeable)\n");
 9:    if ( ievent->mask & IN_CLOSE_NOWRITE ) printf("Closed
       (unwriteable)\n");
10:    if ( ievent->mask & IN_OPEN ) printf("Opened\n");
11:    if ( ievent->mask & IN_MOVED_FROM ) printf("File moved from\n");
12:    if ( ievent->mask & IN_MOVED_TO ) printf("File moved to\n");
13:    if ( ievent->mask & IN_CREATE ) printf("Subfile created\n");
14:    if ( ievent->mask & IN_DELETE ) printf("Subfile deleted\n");
15:    if ( ievent->mask & IN_DELETE_SELF ) printf("Self deleted\n");
16:    if ( ievent->mask & IN_MOVE_SELF ) printf("Self moved\n");
17:
18:    return;
19:  }
```

Now take a look at an example use of this application on a simple directory (see Listing 19.12). Because this is performed in two different shells, the following example shows one shell as flush and the other shell as indented. On the left is the shell activity and on the right (indented) is the output of your `inotify` application.

You begin by creating the test directory (called `itest`) and then in another shell starting your `inotify` application (called `dirwatch`), specifying the directory to be monitored. You then perform a number of file operations and emit the resulting output by `dirwatch`. In the end, the directory itself is removed, which is detected and reported. At this point, no further monitoring can take place (so ideally the application should exit).

LISTING 19.12 Sample Output from the `inotify` Application

```
$ mkdir itest

    $ ./dirwatch /home/mtj/itest

$ touch itest/newfile

    [newfile] Subfile created
    [newfile] Opened
    [newfile] Attributes Changed
    [newfile] Closed (writeable)

$ ls itest
newfile

    Opened
    Closed (unwriteable)

$ mv itest/newfile itest/newerfile

    [newfile] File moved from
    [newerfile] File moved to

$ rm itest/newerfile

    [newerfile] Subfile deleted

$ rmdir itest

    Self deleted
```

With an API that is clean and simple to understand, the `inotify` event system is a great way to monitor filesystem events and to keep track of what's going on in the filesystem.

Removing Files from the Filesystem

To remove a file from a filesystem, you have a number of API functions available, but one is recommended because of its ability to work on either files or directories. The `remove` function removes a file, whether it is a regular file or directory. If the file to be removed is a regular file, then `remove` acts like `unlink`, but if the file is a directory, then `remove` acts like `rmdir`.

In Listing 19.13 you create a file in the filesystem using the `mkstemp` function. This returns a unique filename given a template provided by the caller (as shown, the `X` characters are returned with a unique signature). After this file is created, it is promptly closed and then deleted with the `remove` function.

LISTING 19.13 Creating and Removing a File with `mkstemp` and `remove` (on the CD-ROM at `./software/ch19/rmtest.c`)

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main()
 5:  {
 6:    FILE *fp;
 7:    char filename[L_tmpnam+1] = "fileXXXXXX";
 8:    int err;
 9:
10:    err = mkstemp( filename );
11:
 2:    fp = fopen( filename, "w" );
13:    if (fp != NULL) {
14:
15:      fclose(fp);
16:
17:      err = remove( filename );
18:
19:      if (!err) printf("%s removed\n", filename);
20:
21:    }
22:
23:    return 0;
24:  }
```

Synchronizing Data

Older UNIX system users no doubt recall the `sync` command for synchronizing data to the disk. The `sync` command forces changed blocks that are buffered in the

kernel out to the disk. Normally, changed blocks are cached in the buffer cache within the kernel that makes them faster to retrieve if they're needed again. Periodically, these blocks are written out to the disk. This process also works for reads, where read data is cached within the buffer cache (making it faster for retrieval). A kernel process controls when read data is removed from the cache or when write data is pushed to the disk.

The GNU C Library provides three API functions for synchronizing data, depending upon your particular need: `sync`, `fsync`, and `fdatasync`. In most systems, `fsync` and `fdatasync` are equivalent.

The `sync` function (which can also be invoked at the command line) causes all pending writes to be scheduled for write on their intended mediums. This means that the write to the device might not be completed when the call returns, but is scheduled for write. Its prototype is as follows:

```
void sync( void );
```

If you're really just interested in a particular file descriptor, you can use the `fsync` API function. This function has the added bonus that it does not return until the data is on its intended medium. So if you're more interested in making sure that the data gets to the external device than you are in just scheduling it for write, `fsync` is your call. Its prototype is as follows:

```
int fsync( int filedes );
```

Note that `fsync` might return an error if an issue occurred (such as EIO, if a failure occurred while reading or writing to the device).

Summary

In addition to providing a rich and expressive file system interface, GNU/Linux provides a number of advanced features for file testing, directory traversal, and even event notification of filesystem changes. This chapter provided a quick summary of some of the more interesting filesystem features.

Advanced File Handling APIs

```
#include <sys/stat.h>

int stat( const char *restrict path,
struct stat *restrict buf );
```

```
int fstat( int filedes, struct stat *buf );


#include <pwd.h>

struct passwd *getpwuid( uid_t uid );


#include <grp.h>

struct group *getgrgid( gid_t gid );


#include <time.h>
#include <langinfo.h>

size_t strftime( char *restric s, size_t maxsize,
                   const char *restrict format,
                   const struct tm *restrict timeptr );

char *nl_langinfo( nl_item item );


#include <unistd.h>

long pathconf( const char *path, int name );

char *getcwd( char *buf, size_t size );


#include <dirent.h>

DIR *opendir( const char *dirname );

struct dirent *readdir( DIR *dirp );

int closedir( DIR *dirp );

long telldir( DIR *dirp );

void seekdir( DIR *dirp, long loc );

void rewinddir( DIR *dirp );


#include <ftw.h>
```

```
int ftw( const char *path, int (*fn)(const char *,
        const struct stat *ptr, int flag),
             int ndirs );


#include <glob.h>

int glob( const char *restrict pattern, int flags,
           int (*errfunc)(const char *epath, int errno),
           glob_t *restrict pglob );

void globfree( glob_t *pglob );


#include <sys/inotify.h>

int inotify_init( void );

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

int inotify_rm_watch( int fd, uint32_t wd );


#include <stdio.h>

int remove( const char *path );

int mkstemp( char *template );


#include <unistd.h>

int unlink( const char *path );

int rmdir( const char *path );


#include <unistd.h>

void sync( void );

int fsync( int filedes );

int fdatasync( int filedes );
```