

# File System, Part 4: Working with directories

goldcase edited this page 12 days ago · 13 revisions

## How do I find out if file (an inode) is a regular file or directory?

Use the `S_ISDIR` macro to check the mode bits in the stat struct:

```
struct stat s;
stat("/tmp", &s);
if (S_ISDIR(s.st_mode)) { ...
```

Note, later we will write robust code to verify that the stat call succeeds (returns 0); if the `stat` call fails, we should assume the stat struct content is arbitrary.

## How do I recurse into subdirectories?

First a puzzle - how many bugs can you find in the following code?

```
void dirlist(char *path) {

    struct dirent *dp;
    DIR *dirp = opendir(path);
    while ((dp = readdir(dirp)) != NULL) {
        char newpath[strlen(path) + strlen(dp->d_name) + 1];
        sprintf(newpath,"%s/%s", newpath, dp->d_name);
        printf("%s\n", dp->d_name);
        dirlist(newpath);
    }
}

int main(int argc, char **argv) { dirlist(argv[1]); return 0; }
```

Did you find all 5 bugs?

```
// Check opendir result (perhaps user gave us a path that can not be opened as a
if (!dirp) { perror("Could not open directory"); return; }
// +2 as we need space for the / and the terminating 0
char newpath[strlen(path) + strlen(dp->d_name) + 2];
// Correct parameter
sprintf(newpath,"%s/%s", path, dp->d_name);
// Perform stat test (and verify) before recursing
if (0 == stat(newpath,&s) && S_ISDIR(s.st_mode)) dirlist(new path)
// Resource leak: the directory file handle is not closed after the while loop
closedir(dirp);
```

Edit

New Page

▼ Pages 51

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)


File System, Part 3: Permissions


File System, Part 4: Working with directories


File System, Part 5: Virtual file systems


Show 36 more pages...


Clone this wiki locally





 Clone in Desktop











# What are symbolic links? How do they work?

## How do I make one?

```
symlink(const char *target, const char *symlink);
```

To create a symbolic link in the shell use `ln -s`

To read the contents of the link as just a file use `readlink`

```
$ readlink myfile.txt
../../dir1/notes.txt
```

To read the meta-(stat) information of a symbolic link use `lsstat` not `stat`

```
struct stat s;
stat("myfile.txt", &s1); // stat info about the notes.txt file
lsstat("myfile.txt", &s2); // stat info about the symbolic link
```

## Advantages of symbolic links

- Can refer to a files that don't exist yet
- Unlike hard links, can refer to directories as well as regular files
- Can refer to files (and directories) that exist outside of the current file system

Main disadvantage: Slower than regular files and directories. When the links contents are read, they must be interpreted as a new path to the target file.

## What is `/dev/null` and when is it used?

The file `/dev/null` is a great place to store bits that you never need to read! Bytes sent to `/dev/null/` are never stored - they are simply discarded. A common use of `/dev/null` is to discard standard output. For example,

```
$ ls . >/dev/null
```

## Why would I want to set a directory's sticky bit?

When a directory's sticky bit is set only the file's owner, the directory's owner, and the root user can rename (or delete) the file. This is useful when multiple users have write access to a common directory

A common use of the sticky bit is for the shared and writable `/tmp` directory.

## Why do shell and script programs start with `#!/usr/bin/env python` ?

Ans: For portability! While it is possible to write the fully qualified path to a python or perl interpreter, this approach is not portable because you may have installed python in a different directory me.

To overcome this use the `env` utility is used to find and execute the program on the user's path. The env utility itself has historically been stored in `/usr/bin` - and it must be specified with an absolute path.

## How do I make 'hidden' files i.e. not listed by "ls"? How do I list them?

Easy! Create files (or directories) that start with a "." - then (by default) they are not displayed by standard tools and utilities.

This is often used to hide configuration files inside the user's home directory. For example `ssh` stores its preferences inside a directory called `.ssh`

To list all files including the normally hidden entries use `ls` with `-a` option

```
$ ls -a
.          a.c          myls
..         a.out          other.txt
.secret
```

## What happens if I turn off the execute bit on directories?

The execute bit for a directory is used to control whether the directory contents is listable.

```
$ chmod ugo-x dir1
$ ls -l
drw-r--r--  3 angrave  staff   102 Nov 10 11:22 dir1
```

However when attempting to list the contents of the directory,

```
$ ls dir1
ls: dir1: Permission denied
```

In other words, the directory itself is discoverable but its contents cannot be listed.

## What is file globbing (and who does it)?

Before executing the program the shell expands parameters into matching filenames. For example, if the current directory has three filenames that start with my ( `my1.txt` `mytext.txt` `myomy`), then

```
$ echo my*
```

Expands to

```
$ echo my1.txt mytext.txt myomy
```

This is known as file globbing and is processed before the command is executed. ie the command's parameters are identical to manually typing every matching filename.

## Creating secure directories

Suppose you created your own directory in /tmp and then set the permissions so that only you can use the directory (see below). Is this secure?

```
$ mkdir /tmp/mystuff
$ chmod 700 /tmp/mystuff
```

There is a window of opportunity between when the directory is created and when it's permissions are changed. This leads to several vulnerabilities that are based on a race condition (where an attacker modifies the directory in some way before the privileges are removed). Some examples include:

Another user replaces `mystuff` with a hardlink to an existing file or directory owned by the second user, then they would be able to read and control the contents of the `mystuff` directory. Oh no - our secrets are no longer secret!

However in this specific example the `/tmp` directory has the sticky bit set, so other users may not delete the `mystuff` directory, and the simple attack scenario described above is impossible. This does not mean that creating the directory and then later making the directory private is secure! A better version is to atomically create the directory with the correct permissions from its inception -

```
$ mkdir -m 700 /tmp/mystuff
```

## How do I automatically create parent directories?

```
$ mkdir -p d1/d2/d3
```

Will automatically create d1 and d2 if they don't exist.

## My default umask 022; what does this mean?

The umask *subtracts* (reduces) permission bits from 777 and is used when new files and new directories are created by open,mkdir etc. Thus `022` (octal) means that group and other privileges will not include the writable bit . Each process (including the shell) has a current umask value. When forking, the child inherits the parent's umask value.

For example, by setting the umask to 077 in the shell, ensures that future file and directory creation will only be accessible to the current user,

```
$ umask 077
$ mkdir secret_dir
```

As a code example, suppose a new file is created with `open()` and mode bits `666` (write and read bits for user,group and other):

```
open("myfile", O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

If umask is octal 022, then the permissions of the created file will be 0666 & ~022 ie.

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

# How can I copy bytes from one file to another?

Use the versatile `dd` command. For example, the following command copies 1mb of data from the file `/dev/urandom` to the file `/dev/null` . The data is copied as 1024 blocks of blocksize 1024 bytes.

```
$ dd if=/dev/urandom of=/dev/null bs=1k count=1024
```

Both the input and output files in the example above are virtual - they don't exist on a disk. This means the speed of the transfer is unaffected by hardware power. Instead they are part of the `dev` filesystem, which is virtual filesystem provided by the kernel. The virtual file `/dev/urandom` provides an infinite stream of random bytes, while the virtual file `/dev/null` ignores all bytes written to it. A common use of `/dev/null` is to discard the output of a command,

```
$ myverboseexecutable > /dev/null
```

Another commonly used `/dev` virtual file is `/dev/zero` which provides an infinite stream of zero bytes. For example, we can benchmark the operating system performance of reading stream zero bytes in the kernel into a process memory and writing the bytes back to the kernel without any disk I/O. Note the throughput (~20GB/s) is strongly dependent on blocksize. For small block sizes the overhead of additional `read` and `write` system calls will dominate.

```
$ dd if=/dev/zero of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 0.0539153 s, 19.9 GB/s
```

# What happens when I touch a file?

The `touch` executable creates file if it does not exist and also updates the file's last modified time to be the current time. For example, we can make a new private file with the current time:

```
$ umask 077      # all future new files will maskout all r,w,x bits for group an
$ touch file123  # create a file if it does not exist, and update its modified t
$ stat file123
  File: `file123'
  Size: 0          Blocks: 0          IO Block: 65536   regular empty file
Device: 21h/33d Inode: 226148        Links: 1
Access: (0600/-rw-----)  Uid: (395606/  angrave)   Gid: (61019/    ews)
Access: 2014-11-12 13:42:06.000000000 -0600
Modify: 2014-11-12 13:42:06.001787000 -0600
Change: 2014-11-12 13:42:06.001787000 -0600
```

An example use of touch is to force make to recompile a file that is unchanged after modifying the compiler options inside the makefile. Remeber that make is 'lazy' - it will compare the modified time of the source file with the corresponding output file to see if the file needs to be recompiled

```
$ touch myprogram.c  # force my source file to be recompiled
$ make
```

Go to File System: Part 5

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

