

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

goldcase edited this page 12 days ago · 2 revisions

Big idea: Forget names of files: The 'inode' is the file.

It is common to think of the file name as the 'actual' file. It's not! Instead consider the inode as the file. The inode holds the meta-information (last accessed, ownership, size) and points to the disk blocks used to hold the file contents.

So... How do we implement a directory?

A directory is just a mapping of names to inode numbers. POSIX provides a small set of functions to read the filename and inode number for each entry (see below)

How can I find the inode number of a file?

From a shell, use `ls` with the `-i` option

```
$ ls -i
12983989 dirlist.c      12984068 sandwich.c
```

From C, call one of the stat functions (introduced below).

How do I find out meta-information about a file (or directory)?

Use the stat calls. For example, to find out when my 'notes.txt' file was last accessed -

```
struct stat s;
stat("notes.txt", & s);
printf("Last accessed %s", ctime(s.st_atime));
```

There are actually three versions of `stat` ;

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

For example you can use `fstat` to find out the meta-information about a file if you already have an file descriptor associated with that file

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











```
FILE *file = fopen("notes.txt", "r");
int fd = fileno(file); /* Just for fun - extract the file descriptor from a C
struct stat s;
fstat(fd, & s);
printf("Last accessed %s", ctime(s.st_atime));
```

The third call 'lstat' we will discuss when we introduce symbolic links.

In addition to access,creation, and modified times, the stat structure includes the inode number, length of the file and owner information.

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;     /* user ID of owner */
    gid_t      st_gid;     /* group ID of owner */
    dev_t      st_rdev;     /* device ID (if special file) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t  st_blksize;  /* blocksize for file system I/O */
    blkcnt_t   st_blocks;   /* number of 512B blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last status change */
};
```

How do I list the contents of a directory ?

Let's write our own version of 'ls' to list the contents of a directory.

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    if(argc == 1) {
        printf("Usage: %s [directory]\n", *argv);
        exit(0);
    }
    struct dirent *dp;
    DIR *dirp = opendir(argv[1]);
    while ((dp = readdir(dirp)) != NULL) {
        puts(dp->d_name);
    }

    closedir(dirp);
    return 0;
}
```

How do I read the contents of a directory?

Ans: Use opendir readdir closedir For example, here's a very simple implementation of 'ls' to list the contents of a directory.

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    if(argc ==1) {
        printf("Usage: %s [directory]\n", *argv);
        exit(0);
    }
    struct dirent *dp;
    DIR *dirp = opendir(argv[1]);
    while ((dp = readdir(dirp)) != NULL) {
        printf("%s %lu\n", dp-> d_name, (unsigned long)dp-> d_ino );
    }

    closedir(dirp);
    return 0;
}
```

Note: after a call to `fork()`, either (XOR) the parent or the child can use `readdir()`, `rewinddir()` or `seekdir()`. If both the parent and the child use the above, behavior is undefined.

How do I check to see if a file is in the current directory?

For example, to see if a particular directory includes a file (or filename) 'name', we might write the following code. (Hint: Can you spot the bug?)

```
int exists(char *directory, char *name) {
    struct dirent *dp;
    DIR *dirp = opendir(directory);
    while ((dp = readdir(dirp)) != NULL) {
        puts(dp->d_name);
        if (!strcmp(dp->d_name, name)) {
            return 1; /* Found */
        }
    }
    closedir(dirp);
    return 0; /* Not Found */
}
```

The above code has a subtle bug: It leaks resources! If a matching filename is found then 'closedir' is never called as part of the early return. Any file descriptors opened, and any memory allocated, by `opendir` are never released. This means eventually the process will run out of resources and an `open` or `opendir` call will fail.

The fix is to ensure we free up resources in every possible code-path. In the above code this means calling `closedir` before `return 1` . Forgetting to release resources is a common C programming bug because there is no support in the C lanaguage to ensure resources are always released with all codepaths.

An aside - A System programming pattern to clean up resources - goto considered useful!

Note If C supported exception handling this discussion would be unnecessary. Imagine your function required several temporary resources that need to be freed before returning. How can we write readable code that correctly frees resources under all code paths? Some system programs use `goto` to jump forward into the clean up code, using the following pattern:

```
int f() {
    Acquire resource r1
    if(...) goto clean_up_r1
    Acquire resource r2
    if(...) goto clean_up_r2

    perform work
clean_up_r2:
    clean up r2
clean_up_r1:
    clean up r1
    return result
}
```

Whether this is a good thing or not has led to long rigorous debates that have generally helped system programmers stay warm during the cold winter months. Are there alternatives? Yes! For example using conditional logic, breaking out of do-while loops and writing secondary functions that perform the innermost work. However all choices are problematic and cumbersome as we are attempting to shoe-horn in exception handling in a language that has no inbuilt support for it.

What are the gotcha's of using readdir? For example to recursively search directories?

There are two main gotchas and one consideration: The `readdir` function returns "." (current directory) and ".." (parent directory). If you are looking for sub-directories, you need to explicitly exclude these directories.

For many applications it's reasonable to check the current directory first before recursively searching sub-directories. This can be achieved by storing the results in a linked list, or resetting the directory struct to restart from the beginning.

One final note of caution: `readdir` is not thread-safe! For multi-threaded searches use `readdir_r` which requires the caller to pass in the address of an existing dirent struct.

See the man page of readdir for more details.

How I do determine if a directory entry is a directory?

Ans: Use `S_ISDIR` to check the mode bits stored in the stat structure

And to check if a file is regular file use `S_ISREG` ,

```
struct stat s;
if (0 == stat(name, &s)) {
```

```
printf("%s ", name);  
if (S_ISDIR( s.st_mode)) puts("is a directory");  
if (S_ISREG( s.st_mode)) puts("is a regular file");  
} else {  
    perror("stat failed - are you sure I can read this file's meta data?");  
}
```

Go to File System, Part 3

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.

