FILES AND DIRECTORIES

- Overview
- Filesystem Structure
- Filesystem Structure Continued
- Hard Links
- Hard Links Continued
- Directory Links
- Symbolic Links
- Reading Directories
- Accessing the `stat` Structure
- Details of the `stat` Structure
- File Types
- User and Group IDs
- Process IDs
- The `passwd` Command
- File Permissions
- Directory Permissions
- `ls -F` Program Log
- `ls -F` Program
- `ls -F` Main Program
- `ls -F` Main Program Continued
- `setuid` and `setgid` Bits
- Sticky Bit
- File Times
- `utime()` Function
- Special Device Files
- Creating New Files
- Renaming a File
- Changing File Permissions
- Changing File Ownership
- Standard I/O and Unix I/O
- Directories
- `link()` Function
- `unlink()` and `remove()` Functions
- Symbolic Link Functions
- Making and Removing Directories
- File Tree Walk Main Program
- FTW Program: `myftw()` Function
- FTW Program: `doPath`() Function
- FTW Program: `doPath`() Function Continued
- FTW Program: `myfunc()` Declarations
- FTW Program: `myfunc()`

- References

Overview

- Filesystem structure.
- Files, directories and inodes.
- Hard and soft links.
- The `stat` structure.
- Authorization. File permissions.
- File timestamps.

Filesystem Structure

Typically, a filesystem is divided as follows:

Boot Area
: Contains boot program if filesystem is used for booting.

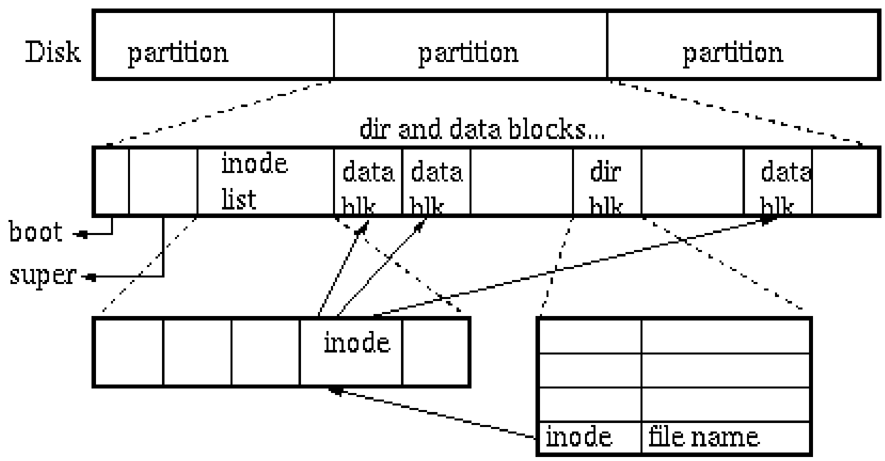
Super Block Area
: Describes layout of filesystem.

I-node List
: A sequence of i-nodes.

Data and Directory Blocks
: Blocks which contain file data or the contents of directories.

A directory block is simply a sequence of `i-node number` and `filename` entries. It is the i-node which contains all the information about the file like its size, permissions, and where its data-blocks occur. There is a 1:1 correspondence between a i-node and a file.

Filesystem Structure Continued



Hard Links

- A filename is contained only within the directory entry; the i-node does not contain any filenames.
- It is possible to have multiple directory entries with different filenames pointing to the same i-node: i.e. a file can be references under multiple names.
- Each name referring to a file is said to be a **hard link** to the file.
- Because a i-node number is local to a filesystem, hard links **cannot** cross filesystems.

Hard Links Continued

- Each i-node contains the number of links to the file. `rm`'ing a file, only removes the directory entry. The file contents are not deleted until after the link count reaches 0.
- Additional links can be added to an existing file using `ln(1)`.
- `ls -l` prints out the link count for a file or directory.

Directory Links

- Hard links to directories can only be added by `root`, to preserve file system sanity.
- Each directory always contains two directory entries: one for `.` (itself), and one for `..` (its parent directory).
- The link count for a directory is at least 2: one link **to** itself and one link **from** its parent.

Symbolic Links

- A symbolic link is a file of a certain type which contains the pathname of the file being linked to.
- Symbolic links can cross filesystems.
- Can be produced using `ln(1)` with the `-s` option.

Reading Directories

`DIR *opendir(const char *dirName)`

Opens a directory stream.

`struct dirent *readdir(DIR *dp)`

Gets next directory entry.

- `struct dirent` always contains a field `d_name`, giving a `NUL`-terminated filename.
- `struct dirent` usually contains a field `d_ino`, giving the i-node number.
- May not see changes which occur during scan.

`void rewinddir(DIR *dirp)`

Rewinds so `readdir()` gets first file.

`int closedir(DIR *dp)`

Closes previously opened directory stream.

Accessing the `stat` Structure

Gets i-node information.

```
int stat(const char *pathname, struct stat *statBuf);

int fstat(int fd, struct stat *statBuf);

int lstat(const char *pathname, struct stat *statBuf);
```

Details of the `stat` Structure

```
struct stat {
    dev_t         st_dev;    /* device */
    ino_t         st_ino;    /* inode */
    umode_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 type
                              * (if inode device) */
    off_t         st_size;   /* total size, in bytes */
    unsigned long st_blksize;/* blocksize for
                              * filesystem I/O */
    unsigned long st_blocks; /* number of 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 change */
};
```

File Types

The `st_mode` field specifies the file type (access via macros). Can be tested using following macros:

Regular file
`S_ISREG`()

Directory file
`S_ISDIR`()

Character special file
`S_ISCHR`()

Block special file
`S_ISBLK`()

Pipe or FIFO
`S_ISFIFO`()

Symbolic link
`S_ISLNK`()

Socket
`S_ISSOCK`()

User and Group IDs

- Each *user name* has a corresponding number or `UID`.
- User name to `UID` mapping maintained in `/etc/passwd` file or equivalent.
- A user belongs to a *primary group* and possibly to multiple *supplementary groups*.
- Each *group name* has a corresponding number or `GID`.
- Group name to `GID` mapping maintained in `/etc/groups` file or equivalent.

Process IDs

Each process has 6 or more user/group IDs associated with it.

Real `UID` and `GID`
: Who we really are.

Effective `UID` and `GID`, Supplementary `GID`s
: Used for resource access permission checks.

Saved-set `UID` and `GID`
: Effective `UID` and `GID` which is saved after a `exec()`.

Usually, the saved-set `UID` and effective `UID` are merely the real `UID`. Similarly, for the `GID`s. However, they can be different, providing needed flexibility (and security holes!).

The `passwd` Command

- Illustrates the need for a process having multiple `UID`s.
- The `passwd` command can be used by any user to change his or her password.
- The `passwd` command operates by changing `/etc/passwd` or equivalent.
- `/etc/passwd` is owned by `root`. Normally, regular users are not allowed to change it.
- The `passwd` command operates by changing its *effective* `UID` to root before attempting to change the `/etc/passwd` file.

File Permissions

- Associated with each file's i-node, are 9 bits giving *read*, *write* or *execute* permissions for *user* (owner), *group* or *other*.
- Permissions can be set using the `chmod(1)` command, or by using the `chmod()` function.
- If effective `UID` of the process matches the owner of the file, then only the *user* permissions of the file are used to grant access.
- If the effective `UID` of the process does not match the owner of the file, but the effective `GID` or one of the supplementary `GID`s matches the group ID of the file, then the *group* permissions are used to grant access.
- If the effective `UID` of the process does not match the owner of the file, and neither the effective `GID` or one of the supplementary `GID`s matches the group ID of the file, then the *other* permissions are used to grant access.

Directory Permissions

Interpretation of 9 permission bits is slightly different for directories.

- *Read* permission allows the contents of the directory to be listed.
- *Write* permission allows the contents of a directory to be changed. Hence to remove or create a new file, it is not necessary to have write permissions on the file, but it is necessary to have write permissions on the directory.
- Execute permission allows the directory to be searched for a particular file.

`ls -F` Program Log

```
$ ./lsF lsF /dev/tty0 /dev/hda1 ~/upload/FIFOS
lsF*
/dev/tty0’
/dev/hda1#
/home/umrigar/upload/FIFOS/./
/home/umrigar/upload/FIFOS/../
/home/umrigar/upload/FIFOS/REQUESTS|
/home/umrigar/upload/FIFOS/symLink@
$
```

`ls -F` Program

Implemented by following program:

```
const char *fileType(const struct stat *statBufP)
{
  umode_t m = statBufP->st_mode;
  int isExec = S_ISREG(m) &&
    (m & ( S_IXUSR | S_IXGRP | S_IXOTH));
  return   (isExec)      ? "*"
         : (S_ISDIR(m))  ? "/"
         : (S_ISLNK(m))  ? "@"
         : (S_ISCHR(m))  ? "'"
         : (S_ISBLK(m))  ? "#"
         : (S_ISFIFO(m)) ? "|"
         : (S_ISSOCK(m)) ? "="
         :                 "";
}
```

`ls -F` Main Program

`main()` driver:

```
int main(int argc, char *argv[])
{
  int i;
  char *cwd = getcwd(NULL, 0);
  if (!cwd) err_sys("no cwd");
  for (i = 1; i < argc; i++) {
    struct stat statBuf;
    if (stat(argv[i], &statBuf) < 0) err_sys("stat");
    if (S_ISDIR(statBuf.st_mode)) {
      DIR *dirP = opendir(argv[i]);
      struct dirent *direntP;
      if (!dirP) err_sys("dir open failed");
      if (chdir(argv[i]) != 0) err_sys("chdir");
      while ((direntP = readdir(dirP))) {
        struct stat statBuf1;
        const char *name = direntP->d_name;
        if (lstat(name, &statBuf1) < 0)
          err_sys("2nd stat");
        printf("%s/%s%s\n", argv[i], name,
               fileType(&statBuf1));
      }
```

`ls -F` Main Program Continued

```
      if (chdir(cwd) != 0) err_sys("could not cd");
      if (closedir(dirP) < 0) err_sys("closedir");
    }
    else {
      printf("%s%s\n", argv[i], fileType(&statBuf));
    }
  }
  free(cwd);
  return 0;
}
```

`setuid` and `setgid` Bits

These are two additional bits associated with the `st_mode` `stat` field:

`setuid`
: (`S_ISUID`) If this bit is set for a executable file, then when the file is `exec()`'d, the effective `UID` is set to the owner of the file.

`setgid`
: (`S_ISGID`) For a regular file, if group execute, then when the file is `exec()`'d, the effective `GID` is set to the group of the file. If not a group-executable, then turn on mandatory record locking for file.

  For a directory, set `GID` of new files created in the directory to `GID` of directory.

Sticky Bit

- S_ISVTX is used on a executable file to force its image to remain within the swap area after its execution completes. This enables quicker subsequent startup. Typically used for popular programs like editors and compiler phases.
- Can only be set by root.
- Today's faster filesystems need this technique less.
- If the S_ISVTX sticky bit is set on a directory, then a file can be removed from a directory if the user has write permission on the directory and either owns the file or directory, or is root.

  This is typically used for /tmp, which is usually writable by all.

File Times

`st_atime`
: Last access time of data. Listed by `ls -u`. Very useful to check if a program is trying to access a file.

`st_mtime`
: Last modification time of data. Listed by default by `ls`.

`st_ctime`
: Last modification time of inode. Listed by `ls -c`.

`utime()` Function

```
int utime(const char *pathname,
          const struct utimbuf *times);

struct utimbuf {
  time_t actime;    /* access time */
  time_t modtime;   /* data modification time */
};
```

`utime()` sets access and modification times set to specified time (current time if `times` is NULL).
`time_t` represents the number of elapsed seconds since the Epoch (UTC Midnight, Jan 1, 1970).

Special Device Files

- Unix I/O devices are divided into *character special devices* and *block special devices*.
- Each device number typically has a *major* and *minor* component, accessed using the macros `major` and `minor` respectively.
- `st_dev` value for every filename is the device number of the filesystem containing that filename and its corresponding i-node.
- Only character special files and block special files have an `st_rdev` value which contains the device number of the actual device.

Creating New Files

- Owner UID of a new file is set to the effective UID of the process creating it.
- GID of a new file is either the effective GID of the process creating it, or the GID of the directory in which it is created.
- The process's `umask` affects permissions on the created file. Specifically, bits in the `umask` which are on, specify the permissions which should be denied. A common `umask` value is `022`, which denies write permissions to group and other.
- To ensure that all files are created with specified permissions, a program may initially set its `umask` to 0 using `umask()`.

Renaming a File

```
int rename(const char *oldName,
           const char *newName);
```

- Can be used to rename directories or files.
- If `oldName` specifies a file, then `newName` cannot specify a directory.
- If `oldName` specifies a directory, then if `newName` exists, it must specify a empty directory.
- Process must have write permissions in both source and destination directories.
- Does not follow symbolic links.
- Largely atomic. There may be a window when both `oldName` and `newName` refer to the same file.
- Does nothing if `oldName` and `newName` are the same.
- Cannot cross file systems.

Changing File Permissions

```
int chmod(const char *pathName,
          mode_t mode);
int fchmod(int fileDesc, mode_t mode);
```

- `mode` is bitwise-or of `S_ISUID`, `S_ISGID`, `S_ISVTX`, `S_I[RWX]USR`, `S_I[RWX]GRP`, `S_I[RWX]OTH`, `S_IRWXU`, `S_IRWXG`, `S_IRWXO`.
- Mode is often specified as a octal number like `0755`.
- To just change a particular permission, it is necessary to call `stat()` first to get current `mode`.
- `S_SISVTX` sticky bit can only be set by `root`.
- `S_ISGID` bit turned off if GID of file does not equal effective GID of process or supplementary GID of process.

Changing File Ownership

```
int chown(const char *pathName,
          uid_t owner, gid_t group);
int fchown(int fileDesc,
           uid_t owner, gid_t group);
int lchown(const char *pathName,
           uid_t owner, gid_t group);
```

- `chown()` follows symbolic links; `lchown()` does not.
- BSD only allows `root` to `chown()`; Sys V allows any user. Posix allows either, depending on `_POSIX_CHOWN_RESTRICTED` filesystem configuration parameter.
- If called by other than `root`, then `S_ISUID` and `S_ISGID` bits are cleared.

Standard I/O and Unix I/O

`int fileno(FILE *stream)`

Returns Unix file descriptor for a open standard I/O stream.

`FILE *fdopen(int fd, const char *mode)`

Return FILE stream for open file descriptor `fd`. `mode` is as for `fopen()` and must be consistent with how `fd` was opened. Underlying descriptor will be closed when returned stream is closed.

Directories

- Each process always has a *root* directory (controlled by privileged `chroot()` call) and *current* directory (controlled by `chdir()`, `fchdir()` calls, retrieved by `getcwd()`).
- It is not possible to portably `read()` or `write` directory files directly. Instead, use suitable directory API.
- Directory consists of filename to inode-number mapping.
- Unix file systems support multiple filenames mapping to same inode-number (*hard links*). Not supported by all file systems (like Microsoft VFAT).

`link()` Function

```
int link(const char *oldPath,
         const char *newPath);
```

- Can be used to link directories only by `root`.
- Creation of directory entry and link count increment is done atomically.
- Different behaviors for `link()` when `pathname` is a symlink: Linux and Solaris do not dereference; standard requires dereference.

`unlink()` and `remove()` Functions

```
int unlink(const char *pathName);
int remove(const char *pathName);
```

- Contents of `pathName` actually removed only when link count goes to zero **and** no process has the file open.
- A common idiom for temporary files, is to open the temporary file and immediately `unlink()` it: this ensures cleanup if the program crashes.
- For files, `remove()` is same as `unlink()`; for directories, it is like `rmdir()`.
- `unlink`'ing a sym-link remove the sym-link, not the file referred to by the sym-link.

Symbolic Link Functions

```
int symlink(const char *oldPath,
            const char *symPath);
```

- `oldPath` need not exist.
- Must have write access to `symPath` directory.

  ```
  int readlink(const char *pathName,
               char *buf, int bufSize);
  ```

- Reads non-NUL terminated link-name referred to by `pathName` into `buf` of size `bufSize`. Returns number of bytes put into `buf`.
- Combines `open()`, `read()`, `close()`.

Making and Removing Directories

```
int mkdir(const char *pathName,
          mode_t mode);
int rmdir(const char *pathName);
```

- `mkdir()` creates a *empty* directory.
- `rmdir()` requires directory to be empty. Directory is actually removed only when its link count reaches 0 and all processes close the directory.
- Typically, `mode` for `mkdir()` must specify one-or-more *execute* permissions, if files within the directory are to be accessed.
- Sticky bit in `mode` respected.
- `S_ISGID` bit in mode ignored; inherited from parent.

File Tree Walk Main Program

Following program (from APUE, non-reentrant):

```
int
main(int argc, char *argv[])
{
  int ret;
  long ntot = 0;
  int i;

  if (argc != 2) {
    fprintf(stderr, "usage:  %s  <starting-pathname>\n", argv[0]); exit(1);
  }

  ret = myftw(argv[1], myfunc);         /* does it all */

  for (i = 0; i < NTYPES_S; i++) ntot += counts[i];
  if (ntot == 0) ntot = 1; /* avoid divide by 0 */

  for (i = 0; i < NTYPES_S; i++) {
    printf("%20s = %7ld, %5.2f %%\n",
           types[i], counts[i], counts[i]*100.0/ntot);
  }

  exit(ret);
}
```

FTW Program: `myftw()` Function

```
/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller’s func() is called for every file.
 */

#define FTW_F   1 /* file other than directory */
#define FTW_D   2 /* directory */
#define FTW_DNR 3 /* directory that can’t be read */
#define FTW_NS  4 /* file that we can’t stat */

/* Type of function called for each file name */
typedef int Myfunc(const char *, const struct stat *, int);

/* contains full pathname for every file */
static char *fullpath;

static int  /* return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
  fullpath = path_alloc(NULL);  /* malloc’s for PATH_MAX+1 bytes */
                                /* ({Prog pathalloc}) */
  strcpy(fullpath, pathname);   /* initialize fullpath */

  return(dopath(func));
}
```

FTW Program: doPath() Function

```
static int /* return whatever func() returns */
dopath(Myfunc* func)
{
  struct stat statbuf;
  struct dirent *dirp;
  DIR *dp;
  int ret;
  char *ptr;

  if (lstat(fullpath, &statbuf) < 0) {
    return(func(fullpath, &statbuf, FTW_NS));   /* stat error */
  }
  if (S_ISDIR(statbuf.st_mode) == 0) {     /* not a directory */
    return(func(fullpath, &statbuf, FTW_F));
  }
  /* It’s a directory.  First call func() for the directory,
   * then process each filename in the directory.
   */

  if ( (ret = func(fullpath, &statbuf, FTW_D)) != 0) {
    return(ret);
  }
  ptr = fullpath + strlen(fullpath); /* point to fullpath end */
  *ptr++ = ’/’;
  *ptr = 0;
```

FTW Program: `doPath()` Function Continued

```
  if ( (dp = opendir(fullpath)) == NULL) { /* can’t read directory */
    return(func(fullpath, &statbuf, FTW_DNR));
  }
  while ( (dirp = readdir(dp)) != NULL) {
    if (strcmp(dirp->d_name, ".") == 0  ||
        strcmp(dirp->d_name, "..") == 0) {
      continue; /* ignore dot and dot-dot */
    }
    strcpy(ptr, dirp->d_name);  /* append name after slash */

    if ( (ret = dopath(func)) != 0) { /* recursive */
        break;  /* time to leave */
    }
  } /* while */
  ptr[-1] = 0;  /* erase everything from slash onwards */

  if (closedir(dp) < 0) {
    fprintf(stderr, "can’t close directory %s\n", fullpath); return -1;
  }
  return(ret);
}
```

FTW Program: `myfunc()` Declarations

```
enum { FILE_S, DIR_S, BLK_S, CHR_S, FIFO_S, SYMLINK_S, SOCK_S, NTYPES_S };
static const char *types[] = {
  "regular files",
  "directories",
  "block specials",
  "char specials",
  "fifos",
  "symbolic links",
  "sockets",
};

static long counts[NTYPES_S];
```

FTW Program: `myfunc()`

```
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
  switch (type) {
  case FTW_F:
    switch (statptr->st_mode & S_IFMT) {
      case S_IFREG: counts[FILE_S]++; break;
      case S_IFBLK: counts[BLK_S]++; break;
      case S_IFCHR: counts[CHR_S]++; break;
      case S_IFIFO: counts[FIFO_S]++; break;
      case S_IFLNK: counts[SYMLINK_S]++; break;
      case S_IFSOCK: counts[SOCK_S]++; break;
      case S_IFDIR:
        /* directories should have type = FTW_D */
        fprintf(stderr, "for S_IFDIR for %s", pathname); abort();
    }
    break;
  case FTW_D: counts[DIR_S]++; break;
  case FTW_DNR:
    fprintf(stderr, "can’t read directory %s\n", pathname);
    break;
  case FTW_NS:
    fprintf(stderr, "stat error for %s\n", pathname);
    break;
  default:
    fprintf(stderr, "unknown type %d for pathname %s\n", type, pathname);
    abort();
  }
  return(0);
}
```

References

Text, Chapter 14, 15, 18.

APUE, Ch 4.