CSci 4061
Introduction to Operating Systems

File Systems: Basics
Chapter 5

File as Abstraction

- Container for related information
- Named
- Associated attributes

Persistence

Process abstraction

Container for a program

I/O abstraction

Source/ sink for data (using fd's FILE*)

Naming a File

```
creat/open ("path/name", …);
```

Links: files with multiple names

Each name is an alias

```
#include <unistd.h>
int link (const char *original_path,
          const char *new_path)
```

cannot exist as a file already

```
link ("foo", "bar");  // "bar" refers to file "foo"
unlink ("bar");       // remove name "bar"
// if file is open by someone, will not actually get deleted until
// all fd's to it are closed
```

What if unlink 'foo' here?
A; sort of like removing last pointer to object (indirect deletion)

File Attributes: Access to metadata

LIB

MUST open file ☆

```
#include <sys/stat.h>
int fstat (int filedes, struct stat *buf)

int stat (const char *pathname,
            struct stat *buf)
```

← don't need to open file ☺

Structure contains file/directory info:

```
off_t st_size;      // file size
nlink_t st_nlink;   // links
mode_t st_mode;     // type + permission
time_t st_mtime;    // last modification time
```

`fcntl` can also be used to set or get lower-level attrs

Exercise: Metadata

- Write a program that monitors a given file every minute and if the file size has changed, it outputs the new size to stdout

sleep(60)

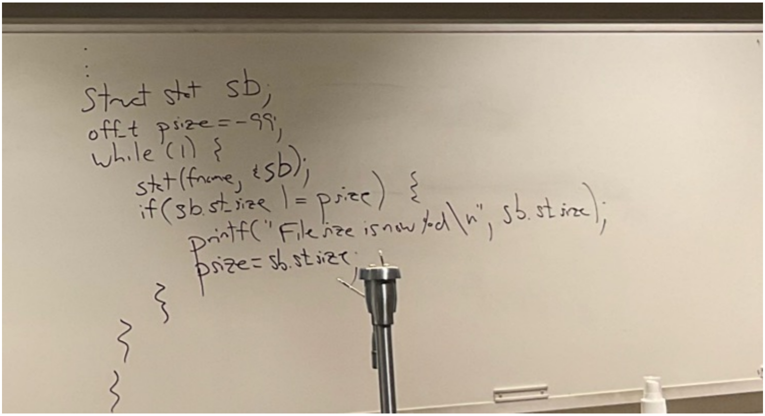
```
#include <sys/stat.h>
int stat (const char *pathname,
            struct stat *buf)
```

Structure contains file/directory level info:

```
off_t st_size;      // file size
nlink_t st_nlink;   // links
…
```

SLIDE INSERT

(bookkeeping code here)

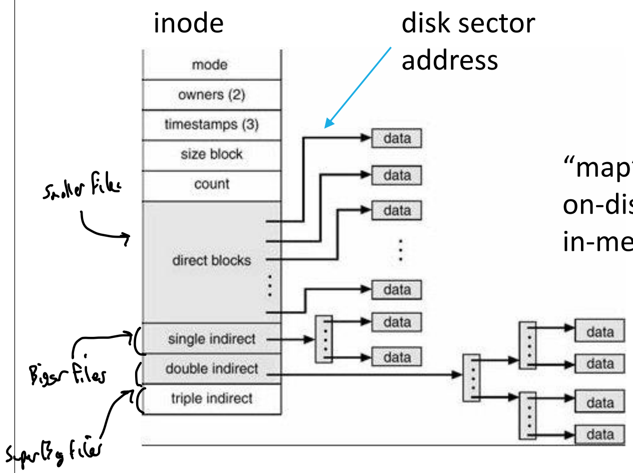


```
:
struct stat sb;
off_t psize = -99;
while (1) {
    stat(fname, &sb);
    if(sb.st_size != psize) {
        printf("File size is now %d\n", sb.st_size);
        psize = sb.st_size;
    }
  }
}
```

Storing File Meta-data: Unix inode

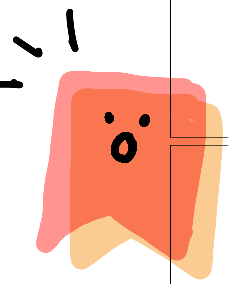
Allows fast for common case which is smaller file sizes

While also flexible for the less common case which is larger file sizes



"map" for a file
on-disk
in-memory

`ls -i <file>` shows inode #



Filesystem

- Directory is a file as well
  - it has an $\boxed{\text{inode}}$
  - what are file contents?
    - list of file_name, inode pairs
- Filesystem
  - Files
  - Directories
  - Free disk sectors (free list)
  - Root dir

Filesystem (cont’d)

- On-disk organization
  - inode for root dir of filesystem “/” stored in well-known sector on the disk
  - inode for disk sector *free-list* also stored in a well-known sector on the disk

Inode table or file (inode #, sector)

  - These are stored in the *superblock*

Unix file types/modes

- Indicated by the first character in ls -l
  - - regular file
  - d directory
  - c **character special file**
  - b **block special file**
  - p pipe
  - s socket
  - l symbolic link

File types

- Within `stat` structure:

```
struct_t stat st;
stat ("foo", &st);
```

Macros:

```
    int S_ISDIR (st.st_mode);
    int S_ISREG (st.st_mode);
    int S_ISSOCK (st.st_mode);
    …
```

P. 158

Another look at ls -l

Example:

all from inode

```
drwx-xr-x   3   jon   fac 4066 Nov 2 09:14 st
```

file type

hard links

allocation size

Filesystem semantics: Unix

- Two processes open the same file
- Reader sees most recent write
- One reader and one writer – run together
  - File "foo" contains "aaaaaaaaaaaaaaaaaaaaaaa"

Filesystem semantics (cont'd)

```
// reader.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void main () {
   int fd, n;

   char c, buf[100];
   read (0, &c, 1);
   fd = open ("foo", O_RDONLY);
   n = read (fd, buf, 10);
   buf[n] = '\0';
   printf ("buf=%s\n", buf);
   …
}
```

```
// writer.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void main () {
   int fd, n;

   char c, buf[100] = "bbbbbbbbbb…";
   read (0, &c, 1);
   fd = open ("foo", O_WRONLY);
   write (fd, buf, 10);
   read (0, &c, 1);
   close (fd) }
```

Missing

File permissions

Operations (r, w, x): read, write, execute

Subjects (u: user/owner, g: group, o: others)

Users may belong to any number of groups (type `groups` at the shell)

↑ terminal

File permissions (cont'd)

```
shell> ls -l
```

```
drwxr-xr-w     3     jon   fac   46
 u   g   o           owner group
```

user
group
other

When a file is created it is given a restricted permission and a default group

You can broaden or further restrict permissions

File permissions (cont'd)

0077
each octal digit is rwx (or 1) for ugo
000, 111, 111

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (char *path, mode_t mode);
```

prefer symbolic flags: `man fstat, e.g. S_IRGRP : GRouP has Read`)

Also at the command-line (absolute)

```
    chmod 0077 st.txt
```

Also at the command-line (relative)

```
    chmod go-xr st.txt
    chmod u+xrw st.txt
```

Power of IDs

- Real user-id: user that actually initiated a process
  - Not executable owner!

  -r-x … 1 jon fac 203 Feb 10 10:47 test

  Bill> `/usr/jon/fac`
- Effective user-id: user that system associates with the process for purposes of protection
  - Usually the same as the real user-id: this would be?
  - Sometimes want effective user-id to that of the file owner and not the user … why?

Power of IDs (cont'd)

- How do to it?

  shell> chmod u+s my_file

  -r-s… 1 jon fac 203 Feb 10 10:47 test

Bill> `/usr/jon/test`

has priviledge of 'jon'

shell> `chmod g+s`

Has priviledge of group 'fac'

Masks

```
creat ("my_file", 0777);
```

- Expectation:
  - shell> `ls -l`
  - -rwxrwxrwx jon … my_file
- Instead:
  - -rw-------- …. … my_file
- What happened?
  - To prevent against accidental exposure, Unix sets a default **mask** with your process (type `umask`)
  - Typically: 077 (1 means mask out)

  `creat ("name", PERM & (~mask));` AND
  regular files also mask out execute

Masks (cont’d)

```
creat (“name”, PERM & (~mask));
```

`umask` is 022: what is this one?

To change the mask:

```
mode_t umask (mode_t newmask);
```

CSci 4061
Introduction to Operating Systems

File Systems: Directories
Chapter 5

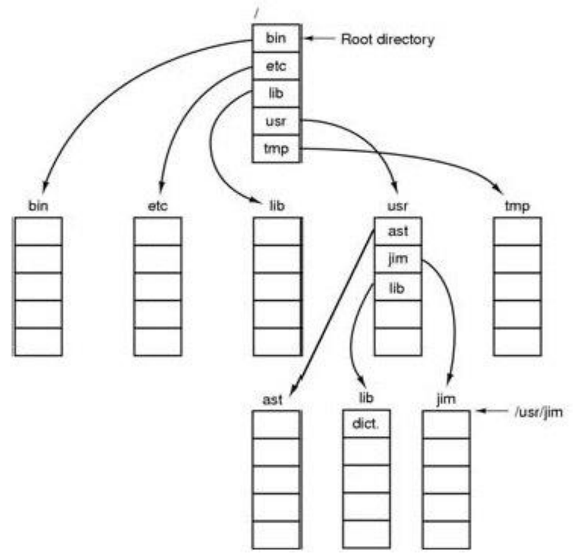
Directory

- What is it?

Directory

- Abstraction
  - Container for related files (and other directories)
  - name
  - location
  - contents
  - attributes

Unix Path Names



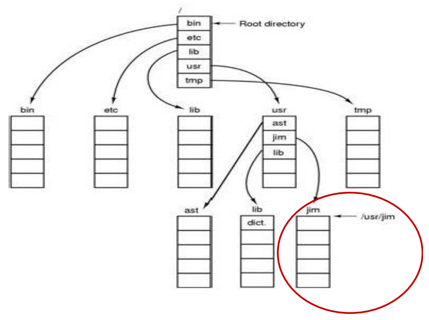
A Unix directory tree
MAX_PATH: 1024 chars

Path Names (cont’d)

- Home directory: dir you are logged into (~)
- Current working directory (`cwd`): `cd /usr/jim`
  - shell> `pwd`
  - shell> `/usr/jim`
- Relative file names(w/r to `cwd`)
  - shell> `ls foo`

  … advantage?
- Absolute file names (rooted from /)
  - Shell> `ls /usr/jim/foo`

  … advantage?



Path Names (cont’d)

- Value of absolute path names:

`foo.c/foo`

```
    …
    f = fopen (“bar”, “r”);
    …
```

Bill> `/usr/jon/foo` will fail unless “bar” is in `cwd`
“bar” must be in the `cwd` of whomever runs it, instead:

```
f = fopen (“/user/jon/bar”, “r”);
```

Bill> `/usr/jon/foo` works now

On the other hand, if we were distributing `foo` …

Path Names (cont'd)

```
int chdir (const char *path);
```

shell> `cd foo`

Ex:

```
     fd1 = open ("/usr/ben/abs", O_RDONLY);
     chdir ("/usr/ben");
     fd1 = open ("abs", O_RDONLY);
```

```
char *getcwd (char *name,
                  size_t size);
```

shell> `pwd`

Links: files with multiple names

Each name is an alias or a "hard link"

```
#include <unistd.h>
int link (const char *original_path,
           const char *new_path)
```

cannot exist as a file already

```
link ("foo", "bar");
```
// "bar" refers to file "foo"

```
unlink ("bar");
```

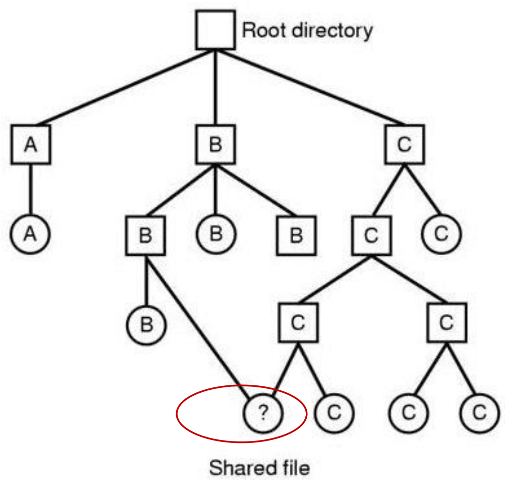
Number of links is the link count

Last `unlink` will delete the file (when no fd's to it)

Cannot `unlink` a directory

Does link affect the fd table?

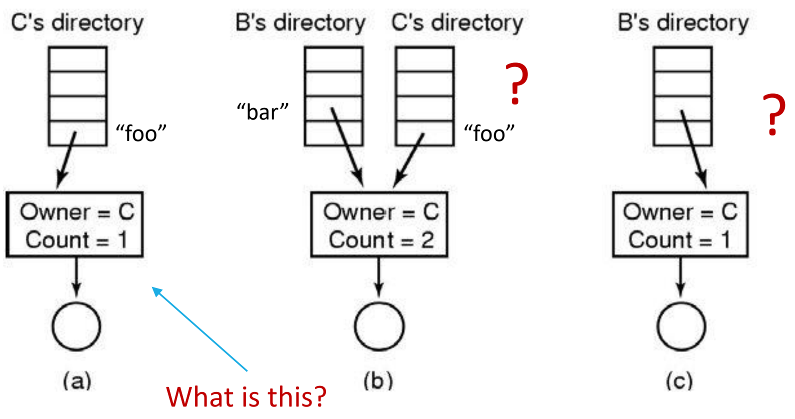
Directories and Hard Links



File system containing a shared file

Directories and Hard Links (cont'd)

```
link ("…/C_dir/foo", "…/B_dir/bar")
```



```
unlink ("…/C_dir/foo")
```

a) Situation prior is linking
b) After the link is created
c) After the original owner unlinks/removes the file

Symbolic Links

- Hard links cannot be made to directories or to files in other file systems
- Symbolic link: allows a file/dir name to "point to" another file/dir name

```
int symlink (const char *realname,
             const char *symname);
symlink ("/usr/jon/tmp1",
         "/usr/bill/tmp2");
```

New inode created for symname `/usr/bill/tmp2`

Symbolic Links (cont'd)

```
symlink ("/user/jon/tmp1/f1", "f2");
```

lrwxrwxrwx f2 -> /usr/jon/tmp1/f1

Remove `f2`, symbolic link goes away, file does not

Remove `/user/jon/tmp1/f1`, symbolic link remains!

<board>

Default Hard Links

- Two links: . and .. (ls -id <dir>)
  - `.` Refers to `cwd`
  - `..` Refers to one level up from `cwd`
  - shell> `./cat`

  - shell> `cd /usr/jim/tmp`
  - shell> `ls ./foo`
    - same as `foo` or `/usr/jim/tmp/foo`
  - shell> `ls ../bar`
    - same as `/usr/jim/bar`

Directory Permissions

- Directories are themselves represented by files
  - Have a name
  - Contents are file names
  - Same protection bits are used for directories (rwx)

Directory Permissions (cont’d)

- Read means class of users can list ‘`ls`’ contents of directory
- Write means class of users can create or remove files in the directory
- Execute means class of users can ‘`cd`’ into directory also allows open and execute for files in the directory

Directory Operations

- create/remove
- opendir/closedir
- readdir

Create

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

int mkdir (const char *pathname,
              mode_t mode);

mkdir (“tmp/dir1”, 0777);
```

Also places two links (. and .. in directory)

Remove

```
int rmdir (const char *pathname);
```

Removes the directory: directory must be empty!

Can be executed in the shell as well

```
shell> rmdir foo
```

Open/Close Directory

- Open a directory to look at its contents

```
#include <dirent.h>

DIR *opendir (const char *dirname);
struct dirent *readdir (DIR *dirptr);
int closedir (DIR *dirptr);

DIR *dp;
dp = opendir (“/tmp/dir1”);

struct dirent {
      ino_t d_ino;
      char d_name[NAMESIZE];
}
```

readdir

```
#include <dirent.h>
struct dirent *readdir
                 (DIR *dirptr);
```

Returns each directory entry, NULL at the end

readdir: example

Example: (very simple) `my_ls`

```
int my_ls (const char *name) {
    struct dirtent *d;
    DIR *dp;
    if ((dp = opendir (name)) == NULL)
        return -1;
    while (d = readdir (dp))
        printf (“%s\n”, d->d_name);
    closedir (dp);
    return 1;
}
```