File systems: User-level view

File system API (a sample)

- **The basics:** `open(), close(), read(), write()`
- **Permissions:** `chmod(), chown()`
- **Metadata:** `stat(), fstat(), lstat()`
- **Directories:** `mkdir(), rmdir(), opendir(), closedir()`
- **Links:** `link(), unlink(), symlink()`

open – read – close

```
int fd;
fd = open("/tmp/1.txt", options);
read(fd,buffer,sizeof(buffer));
close(fd);
```

Note: any of these can result in an error!

What can go wrong with open?

What can go wrong with open?

- EACCES The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of pathname, or the file did not exist yet and write access to the parent directory is not allowed. (See also path_resolution(2).)
- EEXIST pathname already exists and O_CREAT and O_EXCL were used.
- EFAULT pathname points outside your accessible address space.
- EISDIR pathname refers to a directory and the access requested involved writing (that is, O_WRONLY or O_RDWR is set).
- ELOOP Too many symbolic links were encountered in resolving pathname, or O_NOFOLLOW was specified but pathname was a symbolic link.
- EMFILE The process already has the maximum number of files open.
- ENAMETOOLONG pathname was too long.
- ENFILE The system limit on the total number of open files has been reached.
- ENODEV pathname refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation ENXIO must be returned.)
- ENOENT O_CREAT is not set and the named file does not exist. Or, a directory component in pathname does not exist or is a dangling symbolic link.
- ENOMEM Insufficient kernel memory was available.
- ENOSPC pathname was to be created but the device containing pathname has no room for the new file.
- ENOTDIR A component used as a directory in pathname is not, in fact, a directory, or O_DIRECTORY was specified and pathname was not a directory.
- ENXIO O_NONBLOCK | O_WRONLY is set, the named file is a FIFO and no process has the file open for reading. Or, the file is a device special file and no corresponding device exists.
- EOVERFLOW pathname refers to a regular file, too large to be opened; see O_LARGEFILE above.

Open returns a file descriptor

- A structure created inside kernel for each file opened by each process
  - Contains pointer to process's current location inside file
- ID passed back to process for future `read`, `write`, and `close` calls
- E.g.: Unique file descriptors created for
  - Process p opening two different files f and g
  - Process p opening f, and process q opening g
  - Processes p and q both opening f
  - Each of the above cases result in 2 FDs being created
- File desc. structure is destroyed by `close()` call
- Note: Upon `fork()`, child inherits a copy of the parent's file descriptors

File system API (a sample)

- **The basics:** `open()`, `close()`, `read()`, `write()`
- **Permissions:** `chmod()`, `chown()`
- **Metadata:** `stat()`, `fstat()`, `lstat()`
- **Directories:** `mkdir()`, `rmdir()`, `opendir()`, `closedir()`
- **Links:** `link()`, `unlink()`, `symlink()`

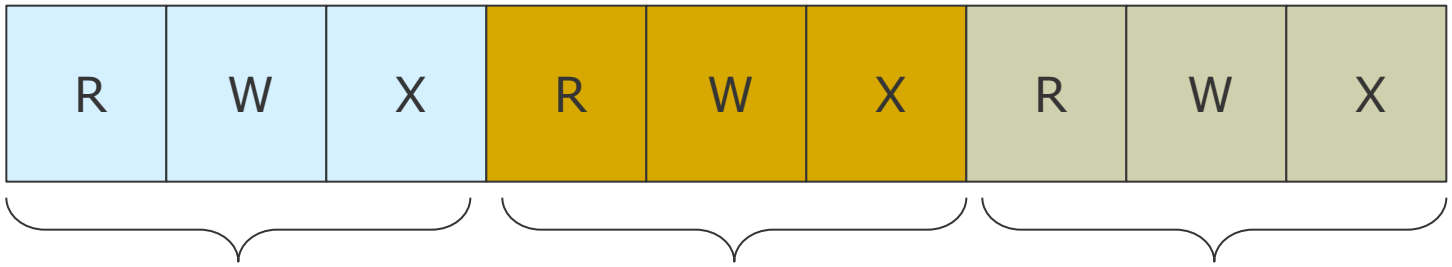
Protection and Access Lists

- Associate each file and directory with access list
  - Lists who is authorized to access the file
  - For each person, lists the mode in which access is authorized (e.g., read/write/execute/append/delete/list)
- Problem with access list: length
- Solution: (Unix-style) condensed version of the access list
  - (u=user) owner - user who created the file
  - (g=group) group - a set of users who are sharing the file and need similar access
  - (o=other) universe - all other users

Access Lists Example

- UNIX - 3 fields of length 3 bits are used.
- User categories:
  - user(u),group(g),others(o)
- Access bits:
  - read(r), write(w), execute(x)
- The change mode (chmod) command:
  - chmod go+rw myfile

Access Control sys/stat.h



S_IRUSR	S_IRGRP	S_IROTH
S_IWUSR	S_IWGRP	S_IWOTH
S_IXUSR	S_IXGRP	S_IXOTH
S_IRWXU	S_IRWXG	S_IRWXO

S_ISUID – set user ID on execution
S_ISGID – set group ID on execution

Can use these in flags supplied to open, or with struct stat.st_mode to check, e.g., if (st.st_mode&S_IRUSR==0)…

File Access Example

```
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char** argv) {
   int fd;
   mode_t fdmode = (S_IRUSR | S_IWUSR |
                    S_IRGRP | S_IROTH);

   fd = open("f.txt", O_RDWR | O_CREAT, fdmode);
   if (fd == -1)
       perror("Failed to open f.txt");
}
```


File Access Example

```
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char** argv) {
   int fd;
   mode_t fdmode = (S_IRUSR | S_IWUSR |
                    S_IRGRP | S_IROTH);

   fd = open("f.txt", O_RDWR | O_CREAT, fdmode);
   if (fd == -1)
       perror("Failed t
}
```

User: read, write
Group: read
Other: read

Open file in current directory for reading & writing. Overwrites any existing file.

It's not the be-all and end-all

- Unix's traditional access control is a compromise, not the holy grail
- Other systems have more flexible control
  - e.g, Andrew File System has ACLs
  - web sites do various things (e.g., permissions based on friend or friend-of-friend relationships on social networks)

File system API (a sample)

- **The basics:** `open(), close(), read(), write()`
- **Permissions:** `chmod(), chown()`
- **Metadata:** `stat(), fstat(), lstat()`
- **Directories:** `mkdir(), rmdir(), opendir(), closedir()`
- **Links:** `link(), unlink(), symlink()`

stat

- Meta-information about a file
- E.g., Modification and access time
- Kind of file (e.g. Directory | regular file?)
- Support for symbolic (“soft”) links

stat versions

- Three flavors
  - `int stat(const char *path, struct stat *buf);`
  - `int fstat(int filedes, struct stat *buf);`
  - `int lstat(const char *path, struct stat *buf);`
- `stat` and `fstat` are the same, except stat works on pathnames and `fstat` on file descriptors
- `lstat` is the same too, except for a symlink:
  - `stat`, `fstat` return info about the referenced file
  - `lstat` returns info about the symlink itself

What stat gives you

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* mode and 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 filesystem I/O */
    blkcnt_t  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 status change */
};
```


Stat macros (st_mode)

- S_ISREG(m) is it a regular file?
- S_ISDIR(m) directory?
- S_ISCHR(m) character device?
- S_ISBLK(m) block device?
- S_ISFIFO(m) FIFO (named pipe)?
- S_ISLNK(m) symbolic link?*
- S_ISSOCK(m) socket?*

*(Not in POSIX.1-1996.)

stat example

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

int main(int argc, char** argv) {
  struct stat s;
  stat(argv[1], &s);
  printf("inode:   %10d\n", s.st_ino);
  printf("# links: %10d\n", s.st_nlink);
  printf("size:    %10d\n", s.st_size);
}
```

Let's try it out...

File system API (a sample)

- **The basics:** `open(), close(), read(), write()`
- **Permissions:** `chmod(), chown()`
- **Metadata:** `stat(), fstat(), lstat()`
- **Directories:** `mkdir(), rmdir(), opendir(), closedir()`
- **Links:** `link(), unlink(), symlink()`

Directories

```
#include <stdio.h>
#include <dirent.h>

int main(int argc, char** argv) {
  struct dirent *e;
  DIR *d = opendir(argv[1]);

  while((e = readdir(d)) != NULL)
      printf("%10d  %s\n", e->d_ino, e->d_name);

  closedir(d);
}
```

Q: What pieces of information does e contain?

Let's try it out...

Take note...

- `readdir` will return "." and ".."
- `readdir` returns a pointer to a static structure
  - i.e., not threadsafe, not recursive-safe
- Various other functions to move around in the directory file (`rewinddir`, `seekdir`, `telldir`)

File system API (a sample)

- **The basics:** `open(), close(), read(), write()`
- **Permissions:** `chmod(), chown()`
- **Metadata:** `stat(), fstat(), lstat()`
- **Directories:** `mkdir(), rmdir(), opendir(), closedir()`
- **Links:** `link(), unlink(), symlink()`

Links, hard and soft

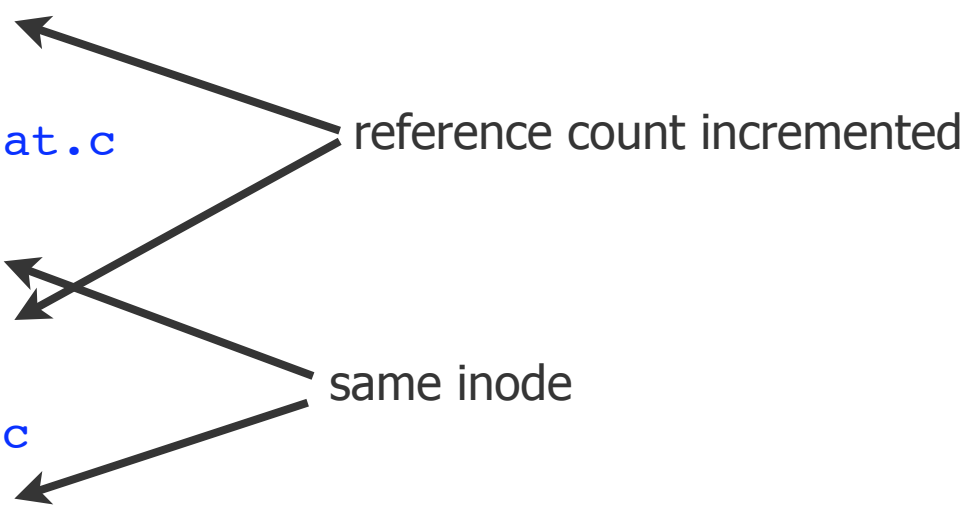
- `int link(const char *path1, const char *path2);`
  - adds a directory entry
  - increments reference count in file's inode
- `int unlink(const char *path);`
  - removes dir entry, decrements ref count
  - if ref count = 0, deletes file
- `int symlink(const char *path1, const char *path2);`
  - adds a directory entry
  - creates new file for the soft link

A question.

- What's the syscall to delete a file?
- **`unlink`**!

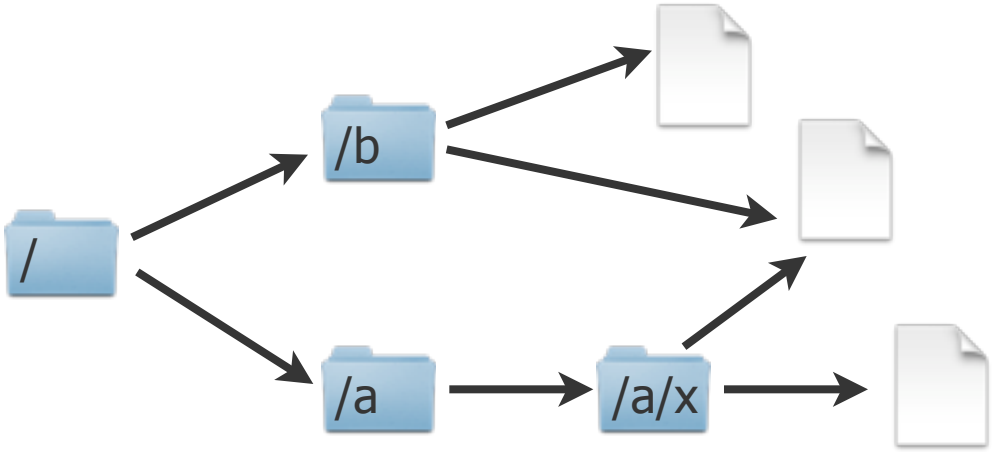
Let's try linking some stuff...

```
$ ./stat stat.c
inode:      9846273
# links:          1
size:           326
$ ln stat.c also_stat.c
$ ./stat stat.c
inode:      9846273
# links:          2
size:           326
$ ./stat also_stat.c
inode:      9846273
# links:          2
size:           326
$ ln . thisdir
ln: .: Is a directory
```

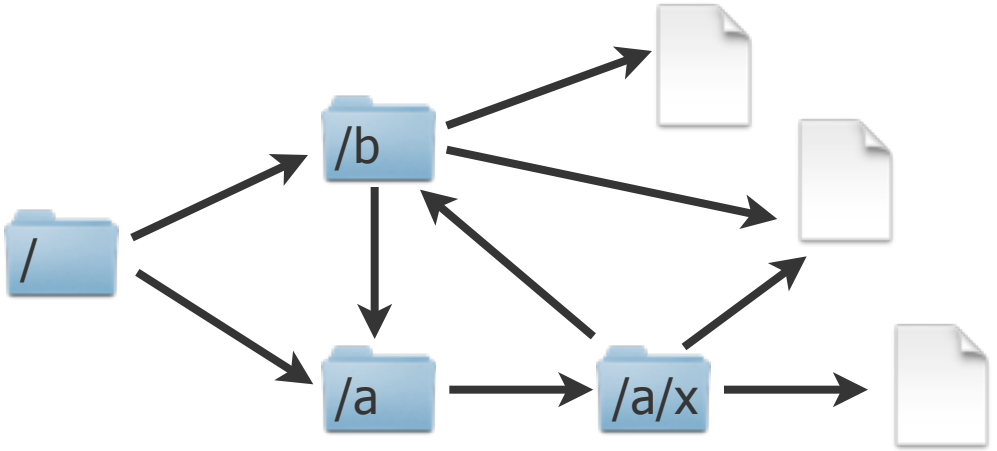


Why can't I add a link to a dir?

Actual situation: FS is directed acyclic graph

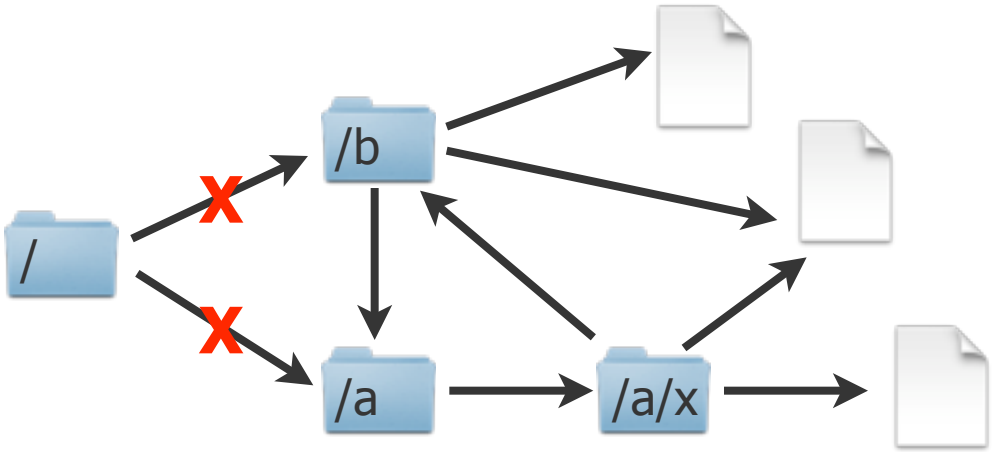


Situation if we could add links to directories: FS contains cycles



Cycles would be confusing

Suppose I unlink /b and /a...



- What's the ref count at /b (or /a)?
- When will they be deleted?
- Answer: ref count would be 1. The directories are unreachable (disconnected from the root of the FS) but would never be removed because ref count > 0.
- Note this problem is similar to garbage collection in prog. languages with automatic memory management. FS avoids it by avoiding cycles in the reference graph.

One last question...

- What would be a good name for this function?

```
int ____________(char* f) {
  struct stat s1, s2;
  stat(f, &s1);
  lstat(f, &s2);
  return s1.st_ino != s2.st_ino;
}
```


One last question...

- What would be a good name for this function?

```
int ____________(char* f) {
  struct stat s1, s2;
  stat(f, &s1);
  lstat(f, &s2);
  return s1.st_ino != s2.st_ino;
}
```

One last question...

- What would be a good name for this function?

```
int is_symlink(char* f) {
  struct stat s1, s2;
  stat(f, &s1);
  lstat(f, &s2);
  return s1.st_ino != s2.st_ino;
}
```