Users & Groups: Unix Account Organization

Unix has user accounts (obviously).

(/etc/passwd has accounts. “passwd” because had hashed passwords; not any more for better security.)

Also “groups”. Sysadmin can define groups and put users into groups. Many-to-many relation. Up to sysadmin what groups mean. Popular usage: one group per project team.

(/etc/group has groups and members.)

‘`id`’ and ‘`groups`’ tell who you are and your groups.

Example: MathLab has two groups per course (students and teachers; teachers only). Also everyone is in default group “cmsusers”.

Permission Flags

Each file is assigned one owning user and one owning group.
(Default: who created it and what's their default group.)

Access permission flags:

- May the owning user read? write? execute?
- May group members read? write? execute?
- May other users read? write? execute?

Notation example: '`rwxr-x---`' = user may do all, group may read and execute, others no access.

"Execute" for regular files: treat as program/script and run it.

My Life Hack

True story: Once upon a time, in an engineering faculty far far away, a sysadmin didn't want students to use a certain standard program. The program enabled students to run multi-player LAN games.

The sysadmin thought this setting would do:

```
-rwxr--r-- 1 root root 56288 Feb 13 1992 xauth
```

I was a friend of some of the students. How did I help them circumvent this?

My Life Hack

True story: Once upon a time, in an engineering faculty far far away, a sysadmin didn't want students to use a certain standard program. The program enabled students to run multi-player LAN games.

The sysadmin thought this setting would do:

```
-rwxr--r-- 1 root root 56288 Feb 13 1992 xauth
```

I was a friend of some of the students. How did I help them circumvent this?

Answer: You can still read, you can copy. You can set your own copy executable!

Permission Flags for Directories

Permission flags for directories have non-obvious meaning:

If you may read: You may see filenames.

If you may write: You may add and delete files.
Does not matter who own those files.

If you may “execute”:

- You may ‘`cd`’ to the directory.
- You may use pathnames that go through the directory.

Popular restriction idiom: ‘`rwx--x--x`’ = people may not discover filenames in your directory, but you may tell selected people selected filenames, then they can access just those files.

Changing Ownership, Permissions

Change permissions (mode):

```
chmod u=rw,g=r,o= path ...
```

Many other notations. `man chmod`

Syscall: `chmod(const char *path, mode_t mode)`

`man 2 chmod`

Change owning user and/or owning group:

```
chown user path ...
chown user:group path ...
chown :group path ...
chgrp group path ...
```

Syscall: `chown(const char *path, uid_t user, gid_t group)`

i-node

File system has a table (array) of “i-nodes”.
“i-node number” = array index
Each file/directory is identified by an i-node, **not filename**.
i-node stores a file/directory’s metadata:

- type: regular file, directory, symbolic link, device, socket. . .
- permissions
- owning user, owning group (both as numerical id’s)
- size
- timestamps
- where is data on disk (if regular file or directory)
- others
- **but not** filename (filenames are in directories)

Can obtain most by ‘`stat`’ command and ‘`stat`’ syscall.

Directory

Directory stores mapping from filenames to i-node numbers.

Exact data structure varies by system. Regardless, use C library functions ‘opendir’, ‘readdir’, ‘closedir’ to access portably.

Logical question: What if two filenames map to the same i-node number?

Directory

Directory stores mapping from filenames to i-node numbers.

Exact data structure varies by system. Regardless, use C library functions ‘opendir’, ‘readdir’, ‘closedir’ to access portably.

Logical question: What if two filenames map to the same i-node number?

Answer: Why not?

Hard Link

When multiple filenames map to the same i-node number

Command ‘`ln`’ can create another filename to have the same i-node number as existing file:

```
ln path 2ndpath
```

We say “creates a hard link”.

Corresponding system call: ‘link’.

The special directories ‘.’ and ‘..’ are implemented this way.

Unfortunately, hard-linking directories disallowed otherwise.

Unlinking

So what happens when you delete a file by filename, but there are other filenames referring to the same i-node number?

The i-node also stores a reference count ("link count"): How many filenames map to this i-node. ('`ls -l`' and '`stat`' display this.)

When you delete, the kernel does:

1. Decrease reference count.
2. If still positive, done.
3. If zero, free up disk space and this i-node.
   (If some processes still have the file open, wait for closing.)

This is why the system call for deleting is called '`unlink`'.

Soft/Symbolic Link (Symlink)

A symlink forwards you to another pathname.

Most system calls follows symlink forwarding. (Can be moar forwarding up to a maximum count.) By extension, most C library functions and programs do this too.

System call '`symlink`' and program '`ln`' can create symlinks:
`ln -s path linkname`

If `path` is relative, relative to the directory `linkname` lives in.

Symlinking to a directory is allowed. (Recall hard-linking to a directory is not.)

'`ls -l`' and '`stat`' show a symlink itself. Add '`-L`' to follow symlink.

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Answer: myhardlink accessible, mysymlink denied

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Answer: myhardlink accessible, mysymlink denied

Suppose now you have access to ‘/dir’.

Exercise 2: I rename ‘/dir/file’ to ‘/dir/stuff’. What happens?

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Answer: myhardlink accessible, mysymlink denied

Suppose now you have access to ‘/dir’.

Exercise 2: I rename ‘/dir/file’ to ‘/dir/stuff’. What happens?

Answer: myhardlink OK, mysymlink broken

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Answer: myhardlink accessible, mysymlink denied

Suppose now you have access to ‘/dir’.

Exercise 2: I rename ‘/dir/file’ to ‘/dir/stuff’. What happens?

Answer: myhardlink OK, mysymlink broken

Exercise 3: I delete ‘/dir/stuff’, create a new file, name it ‘/dir/file’. What happens?

Hard vs Symbolic Links

Suppose:
‘myhardlink’ is hard link to ‘/dir/file’
‘mysymlink’ is symlink to ‘/dir/file’

Exercise 1: What if you don’t have any access to ‘/dir’?

Answer: myhardlink accessible, mysymlink denied

Suppose now you have access to ‘/dir’.

Exercise 2: I rename ‘/dir/file’ to ‘/dir/stuff’. What happens?

Answer: myhardlink OK, mysymlink broken

Exercise 3: I delete ‘/dir/stuff’, create a new file, name it ‘/dir/file’. What happens?

Answer: myhardlink → original file, mysymlink → new file.

File Attributes

System call to get file attributes (“status”):

```
int stat(const char *path, struct stat *statbuf);
```

Returns 0 if success, -1 if error (and sets `errno`).

See ‘`man 2 stat`’ for the fields in `struct stat`. More details in ‘`man inode`’. Same info shown by the `stat` program.

The field `st_mode` has file type and permission flags, e.g., it represents “drwxr-xr-x”.

Bases ten, sixteen, eight, two

We write “26” for twenty six because $2 \times \text{ten}^{\text{one}} + 6 \times \text{ten}^{\text{zero}}$
Decimal, base ten.

Hexadecimal (hex) “1A”: $1 \times \text{sixteen}^{\text{one}} + \text{A} \times \text{sixteen}^{\text{zero}}$
(A = ten, B = eleven,. . .)
C notation: 0x1A or 0x1a

Octal “32”: $3 \times \text{eight}^{\text{one}} + 2 \times \text{eight}^{\text{zero}}$
C notation: 032

Binary “11010”:

$$1 \times \text{two}^{\text{four}} + 1 \times \text{two}^{\text{three}} + 0 \times \text{two}^{\text{two}} + 1 \times \text{two}^{\text{one}} + 0 \times \text{two}^{\text{zero}}$$

Note 3 bits per octal digit, 4 bits per hex digit.

“I use base 10, what is base 4?”

Bitwise Operations

C has bitwise and (&), or (|), not (~), xor (^).
Example using 8-bit unsigned char:
a = 10001001
b = 00000011

`a & b`	`a | b`	`a ^ b`	`~ a`
00000001	10001011	10001010	01110110

Left shift (<<) and right shift (>>):
00011000 << 2 = 01100000
00011000 >> 2 = 00000110

Note how $a \ll k = a \times 2^k$, $a \gg k = \lfloor a/2^k \rfloor$.

Bit Check/Set/Clear/Flip Idioms

How to check/set/clear/flip bit 5 of b:

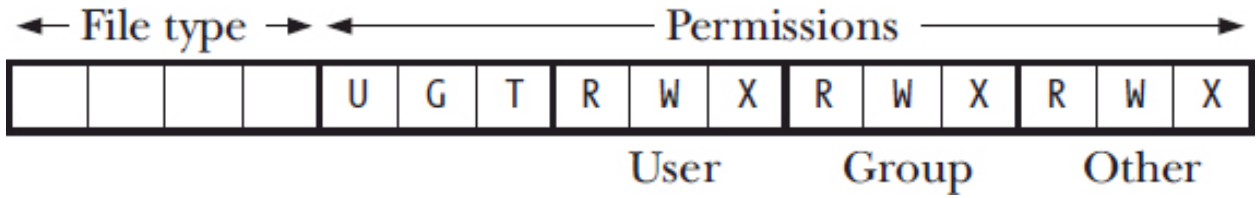
Let m = binary 00100000 = octal 040 = 1 << 5
Bit 5 is on, other bits off.

check	`if (b & m)`
set	`b = b | m`
clear	`b = b & ~m`
flip	`b = b ^ m`

Also `b |= m` etc.

File Type And Permissions Bits

`st_mode` bitwise layout (picture from textbook):



Convenient macros for checking individual bits (`man inode`):
`S_IRUSR` = 0400 = binary 0000 000 100 000 000
`S_IWUSR` = 0200 = binary 0000 000 010 000 000
`S_IXUSR` = 0100 = binary 0000 000 001 000 000
etc.

Sample code: `if (s.st_mode & S_IRUSR) { ... }`

U (set user ID), G (set group ID), T (sticky): Next slide.

Set-UID, Set-GID, Sticky

For directory:

- set-gid: Initial group of new file = directory's group. (Otherwise creator's default group.)
  Use case: Team-wide project directory.

Set-UID, Set-GID, Sticky

For directory:

- set-gid: Initial group of new file = directory's group. (Otherwise creator's default group.)
  Use case: Team-wide project directory.
- sticky: Other users can't delete/rename your files.
  Use case: /tmp is writable by all, everyone can create temp files inside. But you don't want everyone to delete/rename your temp file! /tmp has sticky bit for this.

Set-UID, Set-GID, Sticky

For directory:

- set-gid: Initial group of new file = directory's group. (Otherwise creator's default group.)
  Use case: Team-wide project directory.
- sticky: Other users can't delete/rename your files.
  Use case: /tmp is writable by all, everyone can create temp files inside. But you don't want everyone to delete/rename your temp file! /tmp has sticky bit for this.

For executable file:

- set-uid: Run with file owner's privilege.
  Use case: 'su' and 'sudo' for escalating to sysadmin privilege.

Set-UID, Set-GID, Sticky

For directory:

- set-gid: Initial group of new file = directory's group. (Otherwise creator's default group.)
  Use case: Team-wide project directory.
- sticky: Other users can't delete/rename your files.
  Use case: /tmp is writable by all, everyone can create temp files inside. But you don't want everyone to delete/rename your temp file! /tmp has sticky bit for this.

For executable file:

- set-uid: Run with file owner's privilege.
  Use case: 'su' and 'sudo' for escalating to sysadmin privilege.
- set-gid: Likewise, but group instead of owner.

System Calls for (Low-Level) File I/O

```
int open(const char *path, int flags);
```

`flags`: `O_WRONLY`, `O_RDONLY`, `O_RDWR`, `O_EXCL`, `O_TRUNC`, `O_APPEND`, some others. Can use bitwise-or to combine.
If success, returns "file descriptor", "fd" below.

```
int open(const char *path, int flags, int mode);
```

When `flags` contains `O_CREAT`. `mode` requests initial mode.
(Actual initial mode is further restricted by "umask".)

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, void *buf, size_t count);
```

```
off_t lseek(int fd, off_t offset, int origin);
```

origin: one of `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
Returns new offset from file's beginning (if success).

```
int close(int fd);
```

umask

"umask" limits initial permissions at file creation.
Part of a process's state.

Shell built-in command: `umask`

Syscall: `mode_t umask(mode_t mask)`

Actual initial permissions = (what `open` requests) & ~ umask

umask has 1's for what to **ban**. Some idioms:

- 077: ban rwx for group and others, makes sense on MathLab
- 002: just ban w for others, makes sense in a company

At `open()`, normally request 0666, let umask do the cuts.
Unless highly security-sensitive, then request 0600 right away.

Bridge with High-level stdio.h

If you have file descriptor and wish to use nice high-level stdio.h functions on it:

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

(`mode` is again one of those “r”, “w”, “r+”, “w+”, etc.)

`fclose` does `close(fd)` for you.

If you have `FILE*` and wish to know the file descriptor underneath:

```
int fileno(FILE *stream);
```

File Descriptor

Every process has a “file descriptor table”. File descriptors are array indexes.

Three file descriptors already exist since process creation:
0: standard input
1: standard output
2: standard error

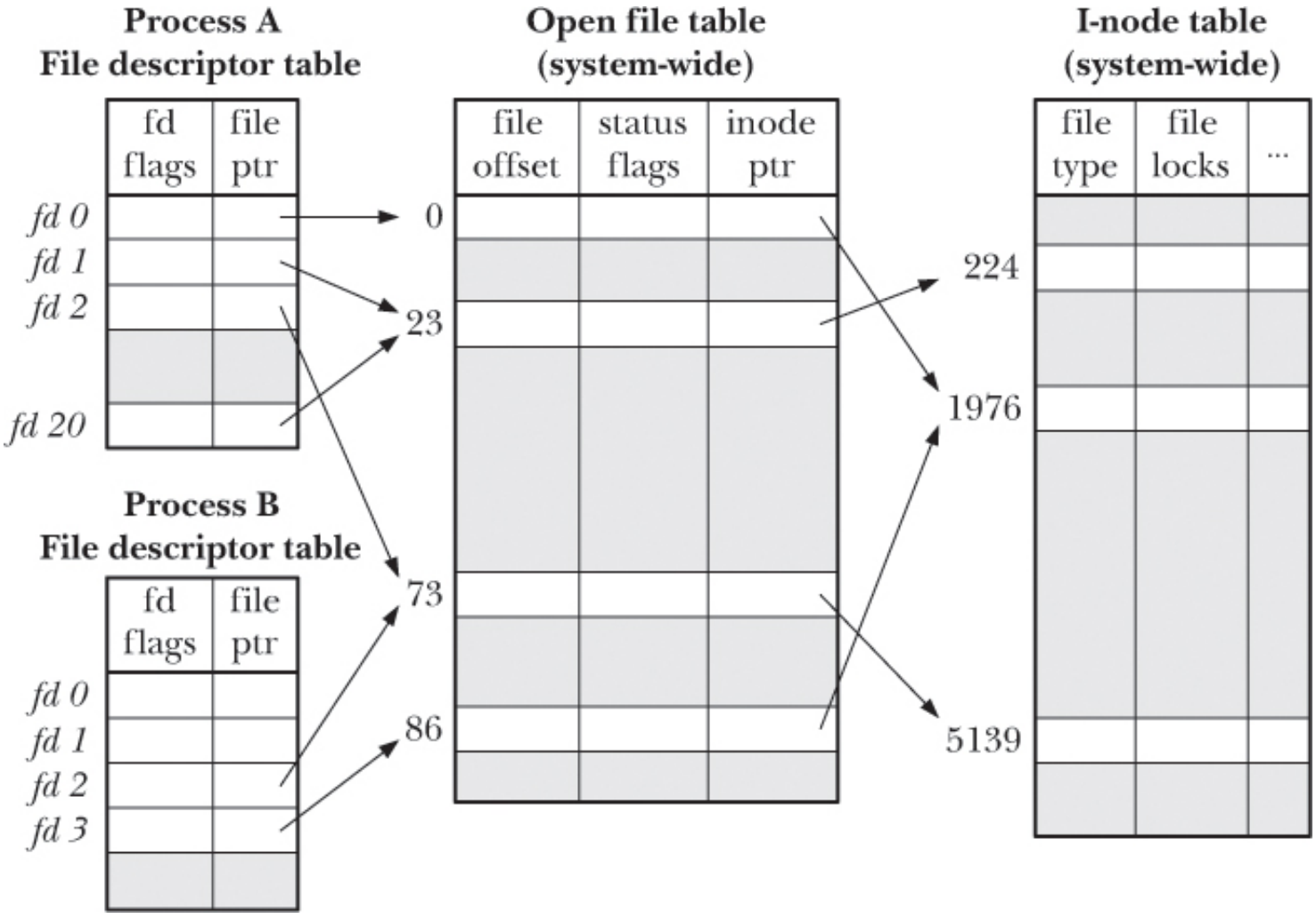
‘open’ (and ‘dup’ later) consumes entries; ‘close’ frees entries.

In fact ‘open’/‘dup’ always consumes the lowest-number free entry. Some programmers rely on this for redirection and pipelining.

File descriptor table is finite. Runs out if too many open’s without closing.

File descriptor table maps to entries in a system-wide “open file table”—the real deal. (Next slide.)

File Descriptor Table → Open File Table



(Picture from textbook.)

How Many-to-One Happens

(OFT = open file table; FDT = file descriptor table)

How two OFT entries refer to same i-node:
Two `open`s (may be same process or different processes)

How two FDs refer to same OFT entry:
`dup` or `dup2` syscalls (next slide)

How two processes have FDs referring to same OFT entry:
When launching child process, FDT cloned by default

Important: file position (“cursor”, current r/w position) in OFT entry, not in FDT.

Two FDs referring to same OFT entry implies: If you read some data via one FD, won’t read it again via the other (unless you seek backwards).

Duplicating File Descriptors

`int dup(int oldfd);`
"Duplicate": Take another FDT entry to refer to the same OFT entry as `oldfd` does. Return new file descriptor.

`int dup2(int oldfd, int newfd);`
Like `dup` but take FDT entry at `newfd`. (If `newfd` was in use, close first.)

We say "duplicate `oldfd` [to `newfd`]".

On previous slide, A's FD 1 and FD 20 refer to the same OPT entry. This could be the result of `dup2(20, 1)`.

Use cases: stdin/stdout/stderr redirection, pipelining.
E.g., shell '`2>&1`' = `dup2(1, 2)`

Demo: offset.c, esp. how "cursor" is shared state.