Architecture et Systèmes

Stefan Schwoon

Cours L3, 2015/16, ENS Cachan

Users in Unix

Modern (multi-tasking) systems such as Unix administrate multiple *users*:

“natural persons”

“virtual users” (administrator, various “deaemons”)

Goal of user administration: share resources while protecting privacy, security, etc.

Practical aspects: Passwords, home directory, . . .

Here: relationship with other concepts (processes, files, etc)

User and group identifiers

Each user has a numerical identifier (user id, or uid).

Each user also belongs to one *primary* group and possibly multiple other groups.
Each group also has a numerical identifier (gid).

Group memberships are usually determined by the administrator.

See also: command id, files /etc/passwd, /etc/group

Users/groups and processes

Each process has several attributes relating to users and groups:

- its effective user ID;
- its effective group ID;
- a list of supplementary group IDs.

These determine most of the privileges that a user has with respect to accessing files etc.

C functions: `getuid`, `getgid`, `getgroups`

A process normally inherits its IDs from its parent process.
Execeptions: login, setuid (see later)

Role of user/group ids

Effective user ID:

governs right to send signals to other processes

for root/admin (user 0): restrictions for many kernel calls lifted

Effective user ID/suppplementary group IDs:

govern read/execute access to files

Effective user ID/group ID:

governs IDs used on files created by process

We come back to these points when discussing the file system.

Input/output in Unix

Input/output operations transfer data between memory and other processes or external devices.

In Unix/Posix, input/output is done through files.

A file is an abstract concept that has at least a *read* and a *write* operation (and possibly others).

Examples of files

Files are a fairly general concept in Unix, goes beyond the standard concept of “something on the hard drive”.

Different implementations, e.g.:

- files on hard disk;
- temporary buffers (console, pipes);
- kernel structures (procfs);
- network sockets, external devices.

Aspects of input/output

System-wide storage of data:

file system

access rights

physical organisation of hard drive

Process-level management:

functions to access/create/etc files

functions to manipulate data stored in files

Unix file system

Unix manages a file system for storing data beyond the life of individual processes.

The file system is a tree-like data structure.

> Non-leafs are called directories.
>
> Leafs can be ordinary files, special devices, symbolic links, . . .
>
> Some nodes can be mapped to other file systems, e.g. stored on hard drives, USB sticks, etc. These are called mount points.

See `mount` for a list of “mounted” devices. Each device is managed by a driver.

Organisation of a file system

Entries in the file system are referenced by a path:

absolute path: starting with `/`, path of directories starting at root, separated by slashes

relative path: interpreted relative to the *current directory* attribute of a process, can be changed by `chdir` (`cd` in the shell).

Note: `.` means the current directory, `..` the directory above.

Attention: Entries in the file system must be distinguished from *inodes*.

Inodes

(origin of term unknown, possibly *index nodes*)

data structure typically used in Unix for permanent files, e.g., hard disk

device partitioned into logical blocks of a fixed, chosen size

a set of these blocks is reserved for storing inodes

an inode contains information about a file:
type, owner, group, access rights, number of pointers to the file, block numbers where data is stored, . . . , but not the name.

Relation between files and inodes

An inode represents a block of data on disk;
a file is a named reference to an inode.

In general: many-to-one relation from files to inodes
(but often one-to-one, except for directories).

The `ls -i` command lists the inode number of files;
`stat` displays information about the inode associated with a file.

Inodes and the directory structure

Directories are special inodes that contain a list of files/directories:

- their names
- their inode numbers

The system of files on a disk forms a tree-like structure (a DAG).

Note that the name of a file is not stored in the file's inode but in the directory containing it.

Indeed, the same file (= inode) can be referenced by multiple directory entries (see `ln` command, "hard" links).

File is physically removed (the inode is freed) when the last link to it is lost (hence `unlink`(2) for removing a file).

Users/groups and files

Each file (more precisely: each inode) belongs to some user and some group.

By default, the process that created the file imparts its effective uid and gid on the file.

Access rights on a file are governed by the uid and gid of a file:

- 9 “ugo” bits: (user,group,others) $\times$ (read,write,execute)
- 3 other bits: setuid, setgid, sticky (see later)

Shell commands: `chmod`, `chown`, `chgrp`, `umask`

Access rights: example

Suppose a process opens a file for reading. When does it have the permission?

> If the process uid equals the file owner, the file must be readable for its owner.
>
> Else, if one of the process gids equals the file's group, the file must be readable for its group.
>
> Else, the file must be readable for "others".

For write: analogous; execute bit: file may be used with the `exec` family of functions.

Access rights on directories

rwx bits have a different meaning on directories:

- read: can read the list of files in that directory
- write: can create, rename, delete files in that directory
- execute: can recover the inode data for the files in that directory (`stat`)

Setuid and setgid

Function of the setuid/setgid bits on *files*:

When file is executed, set the effective user/group id of the child to that of the file owner and group.

The original uid/gid of the parent process that created the child is saved in the *real uid/gid*.

C functions: `getuid`, `geteuid`

Function of setgid bit on *directories*:

Files created in the directory will carry the gid of the directory (and not of the process creating the file).

Process-level management of files

Each process has its set of file descriptors.

The file descriptors owned by a process (at a given moment) are the files to which the process can input/output.

Open files are created by Unix functions such as `creat`, `open`, `pipe`, which return file descriptors to the caller.

`open`(2) opens a file, e.g. on the disk.

`fork` duplicates a process *and its file descriptors*.

`dup` duplicates a file descriptor within the same process.

`close` removes a file descriptor from the process.

Open file table

File descriptors within a process refer to a table of “open files”, maintained by the kernel.

An open file is a data structure that permits access to a file:
type, access mode, position in file, buffered data, . . .

> This is a system-wide structure (there is only one of it).
>
> Multiple open file entries may access the same data (but, e.g., with different access modes, positions, . . .).
>
> An open-file entry persists as long as there is a file descriptor in some process that references it.

Standard file descriptors

Three file descriptors in each process are special:

- 0 is the “standard input” (e.g., `getch` or `scanf` use it)
- 1 is the “standard output” (e.g., `printf` uses it)
- 2 is the “standard error” (often points to same as standard output)

For processes running on the terminal, standard input is (usually) from keyboard and standard output is the screen, which are device files.

Can be changed by ‘redirecting’ output to file (e.g., done by shell before launching the process).

Creating file descriptors

`open`: open a file in the file system. Examples:

- `open("myfile",O_RDONLY)`: open file in read-only mode (alternatives: `O_WRONLY`, `O_RDWR`)
- `open("myfile",O_WRONLY | O_CREAT)`: open for write, create file if it does not exist
- `open("myfile",O_WRONLY | O_CREAT | O_TRUNC,0666)`: as before, but discard previous file contents, and set permissions - see later

`creat`: shorthand for open with `O_WRONLY`, `O_CREAT`, and `O_TRUNC`

Creating file descriptors

`dup` and `dup2`: duplicate file descriptors

- `g = dup(f)`: create a fresh descriptor `g` that behaves like `f`
- `dup2(f,g)`: make `g` behave like `f`, close old file descriptor behind `g` first if necessary

`pipe`: create unidirectional data channel

```
int p[2]; pipe(p);
```

After this, data written into `p[1]` can be read from `p[0]`.

Reading and writing

Data is read/written to a file with `read`/`write`(2).

Certain files (e.g. files on disk) are random-access, i.e.

- data is read or written at the *current position*;
- the position is changed upon read/write or a *seek* operation.

A file may be opened in a certain access mode (read-write, read-only, . . .)

Notes on read/write

Syntax `read/write(fd,p,n)`: read/write *n* bytes starting at address *p* from/to file *f*.

In general, it is advisable to check the return values of `read` and `write`.

Both `read` and `write` return the number of bytes actually read or written. (This may happen due to non-error conditions.) Return value of -1 means error, more information in `errno` variable.

`read` returning 0 means end-of-file.

`read` blocks if no data currently present but other processes may yet write to file.

More on pipes

A pipe is usually created to enable two processes to communicate.

Reading on pipe either returns data that has been written, or blocks until data arrives, or fails if all file descriptors for writing on the pipe have been closed.

Writing on pipe results in `SIGPIPE` if all reading descriptors have been closed.

Shell usage: suppose user types `cmd1 | cmd2`

> Parent opens pipe, forks *twice*, closes both pipe ends, then waits for both children.
>
> First child closes reading end of pipe, redirects standard output to writing end (using dup2), then execs `cmd1`.
>
> Second child closes writing end of pipe, redirects standard input to reading end, then execs `cmd2`.

Changing position inside a file

`lseek` changes the position of the read/write head in a file. The next read/write is from the position determined by this operation.

Not available on all file types, e.g. pipes!

Syntax: `lseek(f,p,m)`, where `m` is one of `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.

`SEEK_SET`: set position in *f* to *p* (start at 0)

`SEEK_CUR`: advance current position in *f* by *p*

`SEEK_END`: set position to end of file plus *p*

I/O: C standard vs ANSI

C typically provides two families of functions for I/O:

`open`, `write`, `read`, . . .

- System calls, defined by POSIX standard (may not exist on other OS)
- work on *file descriptors* (0, 1, 2, . . .)
- unbuffered I/O

`fopen`, `printf`, `scanf`, . . .

- Defined by ANSI-C standard (exist in (practically) all C implementations)
- work on *streams* (stdin, stdout, stderr, . . .)
- buffered I/O: flush using `fflush(3)` or by newline (or: use `setvbuf(3)`)

Mixing these two may produce strange effects . . .