Architecture et Systèmes

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

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]`.

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 . . .

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).