# Operating systems. Lecture 3

Michał Goliński

2018-10-16

## Introduction

### Recall

- History
  - Unix
  - GNU
  - Linux
- `bash`
  - `man`, `info`
  - `cd`
  - `ls`, `cp`, `mv` etc.

### Questions?

### Plan for today

- VFS and filesystems overview
- processing text

## Filesystems

### Empty disk

A new disk (hard drive, SSD) is today an array of empty cells called sectors. Sectors are numbered with consecutive numbers. The disk can only read and write one sector at a time. Sectors used to be 512B, but more and more disks use 4kB sectors.

. . .

Historically sectors were addressed by three numbers (cylinder, head and sector) but this is obsolete today.

. . .

Physically there is no empty space – every sector on a brand new disk would give random data when read (at least in principle).

## Problems

To use a disk we need to solve several problems

- How to store files and directories?
- How to prevent one person from accessing another person's data?
- Which sectors are free and which are taken?

. . .

Files and filesystems are the solution to these problems.

## Filesystem

File is a logical way in which OS handles data on drives.

Filesystem (or "file system")is a logical structure that allows one to write data to disk in an organized way. Most popular filesystems allow to create directory structures, store meta-datada associated with files (ownership, timestamps) and sometimes even error correction in face of hardware errors.

A good filesystem should be relatively fast, should not consume too much space in addition to file data and should gracefully handle different errors.

## Mounting filesystems

In order to use a filesystem from a drive it has to be *mounted*. In Windows drives are usually mounted at `C:`, `D:` etc. In Unix all driver are mounted in a single directory structure at `/`. You can mount a drive at any directory. By convention drives are often mounted as subdirectories of `/media`, but often in other places.

. . .

To mount a drive we use the `mount` command. The command by itself will give a list of mounted filesystems.

## Unmounting filesystems

Before a drive can be removed a filesystem must be *unmounted*. Unmounting makes sure that all the data is really written to disk and all the data structures associated with files are properly closed. Removing a drive without proper unmounting can lead to file data not being written properly and, in extreme cases, to irreperable damage to the filesystem.

## Most popular filesystems

- FAT, FAT16, FAT32, exFAT – older and relatively simple filesystems having roots in DOS

- NTFS – a modern Windows filesystem
- ext2/3/4 – most popular Linux filesystems (consecutive versions)
- ZFS, Btrfs – modern filesystems with interesting possibilities, mainly for servers
- ISO 9660, UDF – filesystems used on optical drives
- HFS+, APFS – filesystems used by Apple

A more complete table with properties can be found on Wikipedia.

### Filesystems – a programmer's perspective

Using a filesystem is usually completely transparent to programers. To use a file a program has to *open* it, it can then read and write data. In the end a file should be *closed* to indicate that a program is done with a file. Specifics of accessing different filesystem are concealed by the operating system (the filesystem driver).

### Formatting drives

Before a disk can be used it has to be formatted – a special program has to create the data structures for a given filesystem. Formatting erases all the files from the disk, but it does not usually overwrite the file data. With special software and a bit of patience the data can (theoretically) be recovered. These programs bypass the filesystem driver and under Linux they are called `mkfs.ext4`, `mkfs.vfat` etc.

### Repairing filesystems

Sometimes filesystems becomes corrupted (disconnecting without unmounting, power failure, disk defect). Special programs are often able to help. They also bypass the filesystem driver and try to mend the on-disk data structures that comprise the filesystem. Under Linux these programs are called `fsck.ext4`, `fsck.vfat` etc.

### File recovery

When a file is being deleted its contents is not usually erased, only the space occupied by the file is marked as unused. Special software can be used to recover the contents of deleted files (or parts thereof). This software would also need to bypass the filesystem driver and interact with the drive itself. A good idea when an important file was accidentally deleted is to read the filesystem in read-only mode to have a better chance that the contents of the file will not be overwritten.

## Virtual filesystem

### VFS

Virtual File System is a layer that sits on top of other operating systems in Linux. VFS gives a uniform interface the user programs interact with. To support a new filesystem one needs

to implement functions required by the VFS. If some filesystem supports functionality outside of the scope of the VFS, then this functionality will not be accessible through the usual interface.

To implement a new filesystem one has implement (in C) several functions which carry out the specific operations. In particular one has to implement mounting, unmounting, reading and writing of directory entries and files.

### Superblock

Mounting a filesystem creates in kernel a so-called *superblock* in memory. Filesystem driver puts in the superblock pointers to functions that manipulate the given filesystem (mostly reading and writing inodes to/from disk).

### Inode

Inodes represent filesystem objects (files and directories) in a running kernel. To implement a filesystem some of the functions acting on inodes have to be defined. They are directly connected to action performed on files (opening, creating, deleting, manipulating permissions and times).

### File

Finally an open file is represented by an in-memory structure that contains a list of functions that are usually performed on contents of a file (reading, writing, flushing or closing).

### Synthetic filesystem

One does not need actual on-disk structure to implement the functionality of a filesystem. For example a running Linux kernel has a filesystem mounted at `/proc` that has contains directories with information on running processes. Similarly a filesystem at `/sys` contains information about the attached hardware and the kernel. By writing to these directories one may change the way the kernel is running.

### FUSE

Writing a filesystem driver used to be a pretty complicated task, in particular it required a good knowledge of C. Filesystems-in-userspace allows to write drivers by writing normal programs. One still has to implement the functionality, but the program can be written in any language, e.g., in Python. Many different FUSE based filesystems were implemented.

### Examples

- Mount a ZIP and read files without decompressing
- Keep a directory in an encrypted file
- Mount a remote directory (over SSH) and access it like local files
- Toy filesystems: wikipedia, reddit

# Input and output of commands

## Standard input and output

Every program under Unix when started has three open files:

- standard input
- standard output
- standard error output

. . .

Standard input is normally read from the keyboard. Both outputs are displayed on the screen.

## Redirection

One can redirect input and output to/from files using the (quite intuitive) operators < and >:

```
$ command <stdin.txt >stdout.txt 2>stderr.txt
```

The output file will be created if necessary and overwritten if it exists. By using the » operator we can append to an existing file.

## Outputs

Standard output is meant for normal output of the program, i.e., calculation results, while the error output is meant for important messages to the user. Even though both outputs are normally mixed, they can be easily separated if needed. One can join them together by the following construction:

```
$ command 2>&1
```

## Filename as argument

Many basic tools used in Unix obey the convention that the input is read from a file if some filename is given as argument. If no filename is given on the command line, the standard input is being used.

## Pipes

Pipe is the shell mechanism for creating a more complicated commands from simple ones. It connects the standard output of one program with the standard input of another one. Pipe is created by typing |.

```
$ command1 | command2
```

Pipes use very little memory or disk space: there is only a small buffer associated with a pipe. Commands will be paused if there is nothing to read or no place to write.

# `coreutils`

## Description

`coreutils` is a package of basic tools used in Unix. Proficiency in using them is absolutely fundamental for efficient usage of the shell. Implementing most of them should be very easy once knows a bit of C.

### `echo`

The `echo` builtin command simply prints its arguments to the standard output. This is sometimes useful. We will be using it in scripts to print messages.

```
$ echo 1 2 3
$ echo -n 1 2 3
$ echo -e '\e[0;31mTHIS IS RED\e[0m'
```

### `cat`

The `cat` program copies the standard input to standard output. Also (because of the aforementioned filename convention) print contents of file to stdout.

```
$ cat
$ cat file.txt
$ cat -n file.txt
```

### `tac`

The `tac` program reverses the order of lines of its input.

```
$ tac 1 2 3
$ cat 1 2 3 | tac
```

### `od`

The `od` program prints the standard input in octal/binary/hexadecimal.

```
$ echo Michał Goliński | od
$ echo Michał Goliński | od -x
$ echo Michał Goliński | od -c
```

### nl

This program copies input to output with added line numbers.

### base64

This command can code binary data into strings that are safe to transfer over the network.

```
$ echo Michał Goliński | base64
$ echo Michał Goliński | base64 | base64 -d
```

### head

The `head` program outputs the beginning of the given file/standard input. By default it is 10 lines. One can further influence what "the beginning" means.

```
$ ls -lR / | head
$ ls -lR / | head -n 5
```

### tail

The `tail` program outputs the end of its input, by default 10 lines. It is very convenient to, say strip the header off a file.

```
$ ls -lR /sys | tail
$ ls -lR /sys | tail -n 10
$ ls -lR /sys | tail -n +10
```

### split

The `split` program splits its input into equal chunks (by default each 1000 lines).

```
$ split large_file
$ split -n 10 large_file
```

### csplit

The `csplit` (*context split*) program splits the files into parts delimited by a given pattern. This way one can, e.g., split the books into chapters etc. The pattern is usually a string or a regular expression.

```
$ csplit book.txt '/Intro/' '/Chapter/' '{7}' '/Epilogue/'
```

### wc

The `wc` command shows the number of lines, words and bytes in the given files.

## Calculating hashes

The programs `md5sum`, `sha1sum`, `sha512sum` calculate and check checksums (MD5, SHA-1, SHA-2) for files.

### sort

The `sort` program sorts the given files lexicographically. The order (collation) is given by the `LC_COLLATE` environment variable.

```
$ ls | sort
$ ls | sort -r
$ ls -l | sort -n -k 5 -r
```

### shuf

The `shuf` program shuffles the lines of input using the Fisher-Yates algorithm. Every permutation is equally probable.

```
$ ls | shuf
$ shuf -i 1-50
```

### uniq

The `uniq` deletes consecutive identical lines. Usually the file is sorted before.

```
$ ls -R /sys | sort | uniq -c
```

### `cut`

The `cut` program allows to cut interesting fields from tabular data.

```
$ cut -d ' ' -f -5,8,10
```

### `paste`

The `paste` program accepts two files as argument and writes one file from the first, a `TAB` character and a line from the second one.

### `join`

The `join` program allows to join two files on a common field (like in SQL). Both files must be sorted with respect to the requested field.

### `tr`

The `tr` program translated one character set into another.

```
$ tr A-Z a-z
$ tr -s ' '
$ tr -d 0-9
```

## Regular expressions

Regular expression is a flexible searching mechanism employing metacharacters, similar to globbing. There are many regexp engines which, although similar, differ in details and in strength. One of the weaker ones (but still very popular) are POSIX regular expressions.

## POSIX-Extended regular expressions (ERE)

Some metacharacters:

- `.` – any single character
- `[abc]` – any single character from the set
- `^` – start of line
- `$` – end of line
- `*` – repeat the previous element any number of times (also zero)
- `(  )` – a group
- `\1`, `\2` – first group, second group etc.
- `[:upper:]` – capital letter

### Examples

- `^www\..*\.pl$`
- `.*@.*\..*`
- `[:digit:]{11}`
- `(.)(.)(.)\2\1`

## `grep`

The GNU `grep` program output only the lines that match the given regular expression. One can use POSIX-Basic or POSIX-Extended Regular Expression.

### Most important options

- `-E` – use ERE instead of the default BRE
- `-F` – search for a string (not a regexp)
- `-R` – search recursively in files
- `-v` – iprint lines **not matching** the regexp
- `-i` – ignore case
- `-c` – print only the number of matches for a file
- `--color=always` – print the matched fragment in color
- `-C 1` – print also the context (lines around the match)

## `sed`

The GNU `sed` program (*stream editor*) is a complex but useful tool used t edit files using regular expressions. In fact any Turing machine can be implemented as a sequence of `sed` commands.

```
$ sed -ri 's/([^ ]*) Nowak/\1 Kowalska/' *
```

## `tee`

The `tee` program copies the standard input both to the standard output and to a given file.

## `pv`

The `pv` program copies the standard input to the standard output and writes the transfer speed to the standard error output.

## `less`

The `less` program is a pager: displays long output on a single screen. Allows to scroll and search. `less` is used by default to display `man` pages.

## Archiving

Archiving is bundling files data and metadata in a single file (not necessarily compresed). The single file is easier to copy and distribute. Linux standard archiving tool is `tar`.

## Compression

Standard compressor that are usually used under Linux are *stream compressors* – they can easily compress data as it flows through a pipe. That's why they can only compress a file at a time. This complement `tar` very well. Under Linux one can often encounter files with extensions `tar.gz`, `tar.bz2`, `tar.xz` etc. These are `tar` archives compressed with `gzip`, `bzip2` and `xz` respectively. They can be decompressed automatically by `tar` or by the programs `gunzip`, `bunzip2` and `unxz`. The programs `zcat`, `bzcat` and `xzcat` decompress a strem to the standard output.

## Both steps at the same time

The formats `zip`, `rar` and `7z` link archiving and compression. In most cases this does not lead to problems but sometimes one may lose some metadata. ZIP files can also have problems with non-ASCII filenames as many tools do not use UTF-8 properly.