Operating systems. Lecture 2

Michał Goliński

2018-10-09

Introduction

Recall

- Basic definitions
  - Operating system
  - Virtual memory
  - Types of OS kernels
- Booting process
  - BIOS, MBR
  - UEFI

Questions?

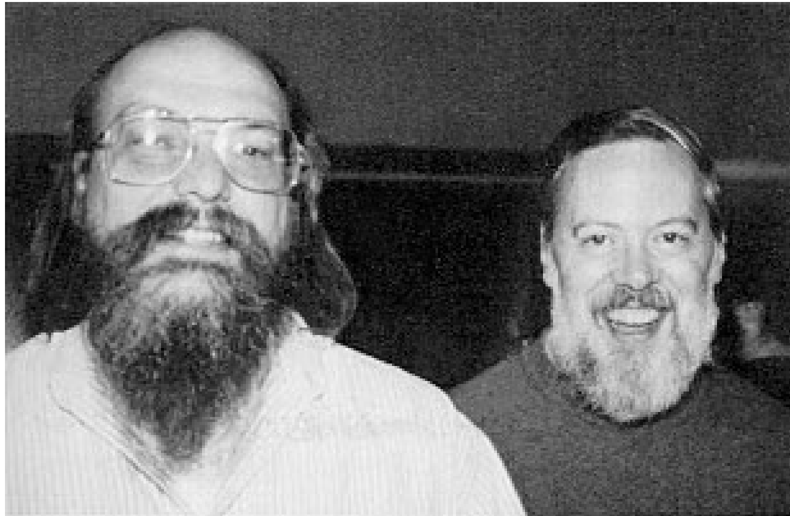
Plan for today

- History of Unix-like systems
- Introduction to Linux
- Introduction to `bash`
- Getting help in `bash`
- Managing files in `bash`

History

History of Unix-like systems

- 1960s – Multics (MIT, AT&T Bell Labs, General Electric)
- 1960s/1970s – UNIX (Ken Thompson, Dennis Ritchie)



PDP-7

History of UNIX-like systems

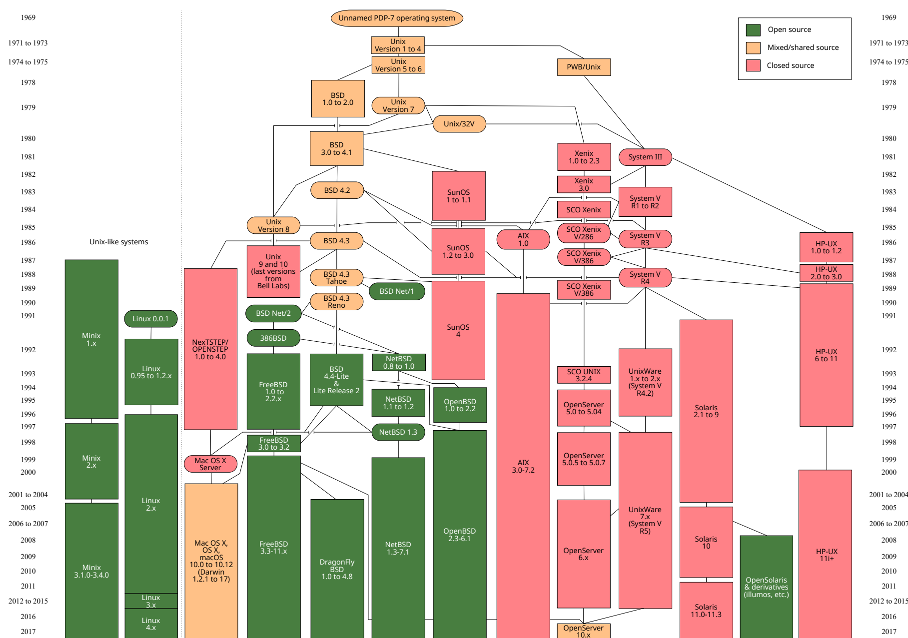
- 1970s, 1980s – popularization, standardization and commerciallization of UNIX
- 1983 – the GNU Project is stared

- 1988 – the first version of the POSIX standard

History of Unix-like systems

- 1991 – the first version of a Unix clone – Linux (Linux is not Unix)
- 1992 – the license is fixed as GPLv2
- 1996 – version 2.0, supporting many processors
- 2003 – version 2.6, new *scheduler*, much better with multiprocessor machines, preemptive kernel, rewriting code to not depend on the so-called Big Kernel Lock
- if the version scheme had not been changed, the current kernel (4.18) would be 2.6.78

Genealogy



OS market today

- PC – Windows 87%, macOS 10%, Linux 3%
- smartphones – Android (Linux) 72%, iOS – 26%
- web servers – Unix/Linux - >70%, Windows <30%
- Top 500 – 500 Linux

Introduction to Linux

Distribution

Linux distribution – complete software system containing the Linux kernel as well as user programs from many different sources. Most distributions have an aim, e.g.,

- friendly for beginners
- tailored for older computers
- tailored for routers
- suitable for servers
- for power users
- commercial, with paid support

GNU

The GNU Project – started in the 1980s by Richard Stallman project of creating an operating system with open and free source code

. . .

Project started with writing free clones of popular Unix tools, leaving creating the kernel for later. The GNU kernel (Hurd) is barely working even today, but many user programs were created, often better than originals. These are used by most Linux distrubutions. That's why we call them GNU/Linux.

. . .

Linux itself *is not* a part of the GNU Project.

GNU GPL

GNU General Public License – free software license. It gives the user of a program the right to get get the program's source code from the authors for free. The user can further modify and distribute it, but only under GPL.

. . .

Moreover, if a program is derived from a GPL licensed work (legally a very grey area), the whole program has to be GPL-licensed (GPL is *viral*)

Most important distributions

- **Debian** – servers
- **Ubuntu** – for beginners, servers, commercial support
- **Red Hat Linux** – workstations, servers, commercial support
- **CentOS** – workstations, servers
- **Fedora** – workstations, servers, emphasis on free software
- **OpenSUSE** – PCs, commercial support
- **Arch Linux** – for power users, *rolling release*

Installing Linux in a virtual machine

Virtual machine is a program that simulates a computer. It allows to install an operating system (guest) without worries about damaging the installed system (host). VirtualBox is a very good free virtual machine. We will install Manjaro KDE. Installation media can be download from here.

Introduction to `bash`

Unix philosphy

Guidelines for Unix programmers:

- Write programs that do one thing and do it well.
- Write programs to work together
- Write programs to handle text streams, because that is a universal interface.

Everything is a file

This means that the system exposes devices and interfaces as special types of files, for which normal operation have special meaning, e.g.,

- disks – allow for reading and writing disk images
- network connections – allows sending and receiving information through the network
- the `/proc` directory – allows getting statistics and information from a running system by reading text files
- the `/proc/sys` directory – allows to change some parameters of a running kernel

Terminal

Why would you even need text mode?

- repeating the same commands for different files
- batch processing
- text mode program is often easier to write than a graphical one
- toolbox of cooperating programs

Shell

Shell — command line interpreter, usually run at the last stage of logging into a system (unless a graphical shell was requested). It waits for user commands and runs requested programs with given arguments.

. . .

Historically important shells:

- Bourne shell – `sh`
- C shell – `csh`
- `tcsh`
- *Bourne-Again shell* – `bash`

Command's anatomy

```
$ <command> <argument1> <argument2> ...
```

The first character ($) is the so called *prompt*. Under default settings it is displayed by the shell. We use the convention that `$` means the command is to be entered as ordinary user (nothing prevents to use them as root). When the command is preceded by `#` it means that it should be entered as root (and probably won't work otherwise).

Command's anatomy cont.

`<command>` is looked up in the following order

- among defined aliases
- among defined functions
- among shell builtin commands
- among executables in consecutive directories of the environment variable `$PATH`

. . .

The search is case sensitive. If a command is not found, shell returns an error.

Command's anatomy cont.

Arguments are arbitrary strings separated by spaces (or other whitespace). The meaning of the arguments depends on specific command. In most programming languages arguments are passed as an array of strings during execution. In principle the order of arguments matters.

Environment variables

Besides arguments a running command also gets implicitly a list of so-called environment variables. User can change them at will and they can also influence how a program works. We will explore this mechanism later.

Arguments and variables in C

The following program in C prints its arguments and environment variables:

```
#include <stdio.h>
int main(int argc, char *argv[], char *env[]) {
    for (int i = 0; i < argc; ++i) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    for (int i = 0; env[i] != NULL; ++i) {
```

```
        printf("Environment variable: %s\n", env[i]);
    }
    return 0;
}
```

Spaces in arguments

Sometimes we want to pass an argument that itself contains a space. We can't do it like this:

```
$ cat Nowy plik tekstowy.txt
```

To force `bash` not to split arguments on a space we can use any of the following methods:

```
$ cat Nowy\ plik\ tekstowy.txt
$ cat "Nowy plik tekstowy.txt"
$ cat 'Nowy plik tekstowy.txt'
```

Completion

Of course typing all the commands can be frustrating and is error-prone. Thankfully shell can complete commands and filenames in arguments. We use Tab for completion. If the completion is ambiguous, using Tab twice gives all the possibilities:

```
$ abc<Tab><Tab>
abcdiff         abcechobounds   abcstitcher
abcecho         abcls           abctree
```

Completion

Completion mechanism is flexible and extendable. For some commands (unfortunately not for all) there are completion scheme that do more than filenames.

Search in history

A very useful function allows for searching in commands that have already be entered. Just type Ctrl+R for the magic to happen. Repeat to go back in history. Press Enter to confirm, press Ctrl+C to abort.

Globbing

Shell is very good when working with multiple files. The globbing mechanism allows to easily operate on them with out typing. Globs are strings that contain *metacharacters*. `Bash`

understands the following metacharacters:

- `*` – stands for any string, even an empty one
- `?` – stands for any one character
- `[]` – stands for any indicated character

Globbing cont.

Examples:

- `*.txt` – all files with names ending with `.txt`
- `a*b?c` – all files with names starting with *a*, ending with *b*, any single character nad *c*
- `?.py` – all files with four character names of which last three are *.py*
- `[abc]*` – all files with names starting with *a*, *b* or *c*

. . .

When shell encounters a glob it replaces it with a list of files that match it, as if we typed it ourselves. Otherwise it leaves the glob as is.

Command options

Authors of command line tools make sure that there is no need to type a lot. That's why some sensible defaults are usually in place. The most common way to change the default action chosen by the programmer are *options*. The POSIX standard defines functions (`getopt`, `getopt_long`) which are widely used in command line programs and hence give a uniform mechanism for processing options.

Command options cont.

- Options are arguments that start with a hyphen (`-`).
- Ordering of options usually does not matter but options are case-sensitive.
- Short options are preceded by one hyphen and can be joined together (e.g., `-l -R` is equivalent to `-lR`).
- Long options are preceded by two hyphens and cannot be joined together (e.g., `--recursive`).
- Both short and long option can take additional parameters (e.g., `--width 40`).

Other common conventions

- most programs use long options ang give short options as contractions for most often used ones.
- A short help for a program can be displayed by using the options `-h` or `--help`.
- The version of a program can be displayed by `-V` or `--version`.

- More verbose output is often turned on by `-v` (`-vv` for even more details).

Examples

```
$ ls
$ ls -a
$ ls --all
$ ls -A
$ ls -lR
$ ls -Rlh
$ ls --color=always
$ rm -file_starting_with_a_hyphen.txt
$ rm -- -file_starting_with_a_hyphen.txt
$ ssh -p 23 golinski@lts.wmi.amu.edu.pl
```

Useful key combinations

- Ctrl+C – send an interrupt to the running process (usually killing it)
- Ctrl+R – search in history
- ↑, ↓ – recall previous/next commands
- ←, → – move the cursor
- Ctrl+D – send an end of file character
- Ctrl+K – delete everything from the cursor to the end of line

Help and documentation

Builtin help

As we have already seen many commands give a short help with options `--help` or `-h`, e.g.,

```
$ cp --help
$ mpv --help
```

Pager

Pager is a program that helps when we want to display more text than possible on one screen. The most often used is called `less`. Its basic usage should be intuitive, more can be learned from the help (press h to view it). to quit `less` press q.

The older help viewer: `man`

The command `man` displays the pages of the so-called *manual*. In most cases it is enough to just search for the command that we are interested in. We will also find manual pages

for the C library functions and Unix system function (for programmers). Sometimes more than one page matches the query, then one can add the manual *section* to refine the search. We can search the text using /, we close the manual using q and we can get more help by pressing h.

```
$ man man
$ man printf
$ man 3 printf
```

Searching the manual: `apropos`

To search the manual (keywords only, not full-text) you can use `apropos`:

```
$ apropos printf
```

It can be useful if we have only a rough idea about what are we looking for.

The newer help viewer: `info`

A series weaknes of `man` is the necessity to write all the help in one document, which leads to some very long manual pages (e.g., `gcc`). The `info` program displayes a documentation that can resemble a book, not a page. In particular it can be divided into chapters and can follow hyperlinks.

```
$ info info
$ info elinks
$ info gcc
```

`info` cont.

Using `info` is a bit more complicated than `man`, mainly because it has to allow to navigate the text:

- `n` – previous page
- `p` – next page
- `l` – go back
- `H` – display short help
- `h` – display `info` documentation for `info`
- `q` – quit

Managing files in the shell

Current directory

The shell keeps track of the so-called *current directory* (also: *current working directory*), i.e., the specific directory in the filesystem it believes to be in any moment. This corresponds to the directory displayed in a window of a graphical file manager. This is the directory used by relative paths.

Absolute path

Absolute path is the filename together with all the directories from the root of the hierarchy, e.g.,

`/home/michal/Dropbox/Adiunkt/Dydaktyka/DSOP-EN/Lectures/lecture02.md`

Under Linux there is no direct equivalent of the windows drives `C:`, `D:` etc., all drives are *mounted* in a single directory tree rooted at `/`. This means that different directories may in fact live on different drives (even over the network).

Relative path

Relative path is a path starting in the current directory.

```
Dropbox/Adiunkt/Dydaktyka/DSOP-EN/Lectures/lecture02.md
DSOP-EN/Lectures/lecture02.md
./DSOP-EN/Lectures/lecture02.md
../DSOP-EN/Lectures/lecture02.md
```

The special directories `.` and `..` present in any directory mean: this directory and the parent directory, respectively.

Changing the current directory: `cd`

To change the current directory use the built-in `cd` command:

```
$ cd /
$ cd /home/michal
$ cd ~ #~ denotes the home directory
$ cd ..
$ cd ../DSOP-EN
$ cd #without arguments goes to the home directory
$ cd - #goes to the previous directory
```

Print the current directory: `pwd`

To display the current directory you can use the `pwd` command (usually a shell built-in):

```
$ pwd
/home/michal/Dropbox
```

The directory stack

Two commands: `pushd` and `popd` manage the directory stack. It allows to go back in the history of current directories (like in a browser).

```
$ pushd /
/ ~
$ pushd /etc
/etc / ~
$ popd
/ ~
$ popd
~
$ popd
bash: popd: directory stack empty
```

Listing directories: `ls`

To list the content of a directory you can use the `ls` command:

```
$ ls                # lists the current directory
$ ls /home/michal   # lists the specified directory
$ ls Dropbox        # lists the specified directory
$ ls *txt           # lists the given files (or directories)
```

Common options

- `-R` – list subdirectories recursively
- `-l` – print more information
- `-a` – show also hidden files (i.e., starting with `.`)
- `-h` – use the suffixes *K, M, G* when displaying size

Copying files: `cp`

To copy a file or directory we use the `cp` command:

```
$ cp file.txt new_file.txt
$ cp file1.txt file2.txt directory
$ cp -R directory other_directory
```

```
$ cp directory/* directory/.* other_directory
```

Common options

- `-R` – copy directories recursively
- `-v` – print the names of the files that are being copied
- `-n` – do not overwrite existing files
- `-i` – ask before overwriting

Moving files: `mv`

To move a file or directory we use the `mv` command:

```
$ mv file.txt nowy_file.txt
$ mv file1.txt file2.txt directory
$ mv directory other_directory
$ mv directory/* directory/.* other_directory
```

Common options

- `-n` – do not overwrite existing files
- `-i` – ask before overwriting

Removing files: `rm`

To delete a file we use the `rm` command:

```
$ rm file.txt file2.txt
$ rm -d empty_directory
$ rm -r directory
```

. . .

There is no trash. All files are deleted permanently.

Common options

- `-R`, `-r` – remove directories recursively
- `-f` – do not ask, igonore nonexisting files
- `-d` – remove empty directories
- `-i` – ask before every removal

Creating directories: `mkdir`

To create a new directory we use the `mkdir` directory:

```
$ mkdir new_directory
$ mkdir -p new_directory/new_directory
```

Common options

- `-p` – create parent directories if necessary

Removing directories: `rmdir`

To remove an empty directory one can use `rmdir`, in practice more often one would use `rm -d` or `rm -r`.

Links

Link are pointer to existing existing file (cf. a shortcut under Windows). Opening a link is equivalent to opening the linked file.

So-called hard links are just different names for existing files – there is no difference between them, deleting the original file does not erase data as long as a hard link exists. Soft (symbolic) links contain just the path of the original file. Deleting the original file deletes data and leaves a dangling link. For directories only symbolic links can be created.

Creating links: `ln`

To create a new links we use the `ln` command:

```
$ ln target link_name # creates a hard link
$ ln target           # creates a hard link with the same name
↪  in the current directory
$ ln target link_name # creates a hard link in an existing
↪  directory
$ ln -s cel link_name # creates soft links
```

Common options

- `-s` – create soft (symbolic) links

File permissions

Every file under Linux is owned by a user and by a group. We have three groups of permissions: for the owner, for the group and for everybody else. In each group there are three permissions:

- `read` – possibility to read the contents of a file, listing a directory
- `write` – possibility to change the contents of a file, adding/deleting files in a directory
- `execute` – possibility to run a file as a program, accessing files in a directory

File permissions cont.

Execute is tricky for directories. If `r` is set but `x` is not, then one can get names of files in a directory but no other info about the files (size, owner etc.).

File permissions cont.

Permissions are not additive. For each access only one group of permissions is taken into account. E.g., if the owner tries to read a file and owner is not permitted to read, then access is denied even though everybody else can read the file.

File permissions cont.

Besides symbolic notation (`rwx`) it is common to use octal notation. We get permissions in octal by adding 4, 2 and 1 depending on whether the given file has the read, write, execute permissions respectively. Three octal digits give permission for owner, group and everybody else.

. . .

For example `743` gives `rwx` for the owner, `r--` for the group and `-wx` for everybody else.

Changing permissions: `chmod`

The `chmod` command in symbolic notation allows to add or remove specific permissions:

```
$ chmod u+rw file  #add owner the permissions to read and write
$ chmod g-x file   #remove the group the permission to execute
$ chmod o=rx file  #give other the permission to read and
 ↪  execute
$ chmod a=rwx file #give all permissions to all
```

Changing permissions cont.

Using the octal notation `chmod` can just fix permissions:

```
$ chmod 644 file      #give permissions rw-r--r--
$ chmod 755 directory #give permissions rwxr-xr-x
```

Change modification time: `touch`

The `touch` command allows to change the last access and modification time for file (default to the current time). It will be useful for us, as nonexistent files are created.

The tool is useful when using `make` if we want to force recompilation of a file without changing its contents.