UNIX AND LINIX IN INFOCOMMUNICATION
Week 4

O. Sadov

Some utils

OK. But you can get useful information not only from the 'info' utility.

OS variant

Ok, we just logged in. First, let's try to determine which part of the UNIX-like universe we are in.

Uname

`uname` — print system information, in most simple case — just name of kernel. With "`all`" flag we will get more information. And for what needs can such information be used, besides simple curiosity? The answer is simple — it can be used to create portable applications or some kind of administrative scripts for various types of UNIX-like systems. You can use it in your installation or shell configuration scripts to select different binaries and system utilities according to your specific computer architecture and OS.

This works well for good old UNIX systems that are very vendor dependent. But on Linux systems, '`uname`' will only display the Linux kernel name, possibly with the kernel version. And as we know, we will have many different Linux distributions, which can be very different from each other. And how can we adapt to this diversity?

One of the possibilities is the lsb_release command:
`lsb_release` — provides certain LSB (Linux Standard Base) and distribution-specific information. The Linux Standard Base (LSB) is a joint project by several Linux distributions under the organizational structure of the Linux Foundation to standardize the software system structure, including the Filesystem Hierarchy Standard used in the Linux kernel. The LSB is based on the POSIX specification, the Single UNIX Specification (SUS), and several other open standards, but extends them in certain areas.

Date

Good. We get information about "where". Let's try to figure out "when".

`date` — print the system date and time. What time? The current time of our time zone. We can check the time in a different time zone, for example, Greenwich Mean Time (GMT):

```
$ TZ=GMT date
```

We can also set the current computer time:

```
$ man date
...
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
...
```

Also you can choose a different output format for time and date using the '`+`' option:

```
date [OPTION]... [+FORMAT]
```

and use this command to convert from different time representations using the '`-date`' option. You can find more details on the man page.

And of course we can see the calendar:

```
$ man cal
...
cal - displays a calendar
...
```

For example, the calendar for the first year of the UNIX epoch:

```
$ cal 1970
```

Users information commands

Okay — 'what', 'when', but what about 'who'? As we discussed earlier — we know that users and groups are just some magic numbers. Let's look at the user's info:

```
id - print real and effective user and group IDs
```

But also we have yet another command:

```
logname - print user's login name
```

For what needs can we use this command if we already have an 'id' command? First of all: as far as we remember, we have different users, moreover, different types of users. Let's look... This is a regular user session. The username is "user". Let's look to 'id' command. And this is the session of the root user. He, as we remember, is the superuser. And we see absolutely another result of the id command. But "logname" will show the same result in both cases, just because we are logged in with the user named "user" in both sessions, and then switched to superuser with the "sudo" or "su" command. This can be important in some cases, and you can use this command to determine the real user ID.

Multiuser environment

As we recall, a UNIX-like system is a multi-user environment, and we have many utilities for working with such a system.

who — show who is logged on

finger — user information lookup program — more informative command including user downtime. At this point, it can be understood that a particular user is still sitting at his workplace or has left for coffee. Moreover, we can see the user's status on other computers. But in this case, you must understand that this is a client-server application. You must have a server part on the computer that you requested, and you need the appropriate privileges.

If you find the required user in the list of computer users, you can send him a message manually or from the program using the "write" command:

```
write - send a message to another user
```

Just enter something and finish your message with EOF (^D as we remember). In this command, we can select the terminal line to write.

Terminal line control

And we can get our current terminal line using the 'tty' command:

```
tty - print the file name of the terminal connected
      to standard input
```

We also have '`stty`' command to print and change terminal line settings. With the option '`-a|-all`' we can get all the current driver settings of this terminal line. And then we can change these settings with this command. For example, the previously discussed setting of the Delete key to interrupt a program on some older UNIX systems.

Another note about older UNIX systems is that stty on such systems may not have the '`-F`' option. But we still have the option to select the device — just by redirecting stdin:

```
stty < /dev/tty0
```

Processes

Process

We've discussed the users, and then it's appropriate to talk about another of the three whales of UNIX-like systems — processes. We can get information about the processes by running the "`ps`" (process status) command. In this case, we again see two worlds — two systems SYSV and BSD:

```
SYSV: ps [-efl]
BSD: ps [-][alx]
```

What about GNU? As we can see, GNU ps supports both sets of options with some long options.

By default `ps` without options shows only process started by me and connected to my current terminal line.

To get the status for all processes, we must use:

```
SYSV: ps -ef
BSD: ps ax
```

And we can get more information about the processes using the "long" options:

```
SYSV: ps -l
BSD: ps l
```

What information about the processes can we see?

`UID` — effective user ID. A process can have a different identifier than the user running the application because, as we will see later, there is a mechanism in UNIX-like systems to change the identifier on the run.

`PID` — a number representing the unique identifier of the process.

`PPID` — parent process ID. As we remember, we have a hierarchical system of processes, and each process has its own parent. We can see this hierarchy for example by such options set:

```
ps axjf | less
```

or just by command:

```
pstree
```

`PRI` — priority of the process. Higher number means lower priority. But, as we will discuss later, we cannot change the priority, because this value is dynamically changed by the process scheduler. And we can only send recommendations to the scheduler using the 'nice' (NI) parameter:

`NI` — can be set with 'nice' and 'renice' commands

`TTY` — controlling tty (terminal).

`CMD` — and the command.

And also a very useful (especially if the system hangs) command '`top`', which dynamically displays information about processes, sorted accordingly by the use of system resources — memory and CPU time.

`nice` — run a program with modified scheduling priority. 'Nice' value is just an integer. The smallest number means the highest priority. The nice's

range can be different on different systems and you should look at the "man nice" on your system. In the case of Linux nice value — between -20 and 19. Only the superuser can increase the priority, the normal user can just decrease the default, which can be seen by invoking the "nice" command with no arguments.

For example:

```
nice -n 19 command args...
```

means execution of the command with the lowest priority. This can be useful for reducing the activity of non-interactive applications, such as the backup process, which can slow down the interactive response of the system.

renice — alter priority of running processes by PID. In this case, you may not use the '-n' option — just a 'nice' number. For example:

```
renice 19 PID...
```

Jobs

At the Shell level, we can use the 'jobs' mechanism.

The easiest way to start a new background job is to use the ampersand (&):

```
gedit &
xeyes
```

Once the command is running, we can disconnect from the terminal line and pause it by pressing '^Z'. As we can see, the eyes do not move now.

We can see background and suspended jobs using the jobs command:

```
jobs
```

In the first position of the jobs list, we see the job number. We can use this job number with a percent sign in front of it.

A suspended task, we can switch it to the background execution mode. By default — current job:

```
bg [%jobN] - resume suspended job jobN in the background
```

and reattach the background job to the terminal line by bringing it to the foreground:

```
fg [%jobN] - resume suspended job jobN in the foreground
```

After that we can interrupt the foreground job by pressing '^C'.

Signals

Another way to terminate a process is with the 'kill' command:

```
kill %job
```

also you can kill the process by PID number:

```
kill [-s sigspec | -n signum | -sigspec] [pid | jobspec] ...
```

But in some cases 'kill' does not work — for example, if the process is frozen. We can fix this problem by calling another kill, just because kill is actually sending a signal to the process, and we just have to choose a different signal.

```
kill -l -- list of signals
```

15) SIGTERM – generic signal used to cause program termination (default kill)

2) SIGINT — "program interrupt" (INTR key — usually Ctrl-C)

9) SIGKILL — immediate program termination (cannot be blocked, handled or ignored)

1) SIGHUP — terminal line is disconnected (often used for daemons config rereading)

3) SIGQUIT — core dump process (QUIT key – usually C-\)

Offline execution

When you execute a Unix job in the background (using `&`, `bg` command), and logout from the session, your process will get killed. We can avoid this using nohup command:

`nohup` — run a command immune to hangups, with output to '`nohup.out`'

Another very useful program is '`screen`' — it's screen manager with VT100/ANSI terminal emulation which supports multi-screen session support with offline execution. In fact, you can run some long running commands on multiple screen sessions and after disconnecting from this terminal line with your hands or after breaking connecting. After that, you can reconnect to this screen and you will see that all processes are still running.

Later execution and scheduled commands

Another possibility of offline executing commands is later execution and scheduled commands.

`at`, `batch`, `atq`, `atrm` — queue, examine or delete jobs for later execution

`crontab` — maintain crontab files for individual users

Files

Finally, let's discuss the third pillar that holds the whole UNIX world — files.

To facilitate the work of users who are not familiar to working with the command line, there are a number of free file management interfaces, for example, Midnight Commander (mc), reminiscent of Norton Commander, or graphical file managers, reminiscent of MS Windows Explorer. But we'll see how we can work with files and directories from the CLI or scripts.

First, let's take a look at some of the symbols that have special meaning in the file path:

```
/ - root directory and directory separator
. - current directory
```

```
.. - parent directory
~/ - home directory
```

As we can see, UNIX uses a slash (/) as the directory separator, and Windows uses a backslash (\). This is interesting because early versions of Microsoft's MSDOS operating systems did not support subdirectories just because it was just a clone of CP/M OS from Digital Research. It was a small OS for 8-bit microcomputers. It was a small OS for 8-bit microcomputers without disk storage or with a small floppy disk. Usually there were only a few dozen files on a floppy disk, and only a flat file system with one directory per file system was supported.

And at first, Microsoft MSDOS operating systems didn't support subdirectories. Only when developing its own "multiuser" OS — OS Xenix based on UNIX, Microsoft implemented a hierarchical file system and ported it to the "single user" MSDOS. But at that point the forward slash was already taken — it was used as a standard CP/M command option marker, like a 'dash' in UNIX commands. And Microsoft choose a 'backslash' as a directory marker.

OK. As we remember, we have a hierarchical file system with a single root directory and for newbies, this file system hierarchy can seem too complex. They say, "When we install some software on Windows, we have separate directories for each product, and it's too easy for us to find something, but on your system we don't know where we can find something".

But in fact, in UNIX-like systems, we have a very clear and stable standard for file system hierarchy, which is reflected, for example, in the corresponding Linux specification: https://refspecs.linuxfoundation.org/fhs.shtml

In fact, we have three main levels with a repeating directory structure. At the first level, we have directories like this: (`http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-5/`). In the '`/usr`' and '`/local`' directories we see again: (`http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-5/`).

And, as I said, devices in UNIX-like systems look like files, but as special files placed in a special directory '`/dev`': (`http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-5/`). Usually each such file is just a rabbit hole in the

OS kernel. When working with a pseudo-file in this directory, we see this device as a stream of bytes and work with it as with a regular file.

We may also have many other secret paths to the kernel, such as `/proc` and `/sys`. For example, we can see:

- `/proc/cpuinfo`
- `/proc/meminfo`
- `/proc/interrupts`

File Commands

UNIX tools support a standard set of commands for working with files and directories:

- `ls` — list directory contents. Let's look in '`man ls`'. We can simply specify files and directories as arguments and view the listing in different ways according to the options.

  Ok, let's take a look at our current directory — it's just '`ls`' without arguments. As we remember, after logging in, this is the home directory.

  ```
  ls
  ```

  We see some directories, but we don't see, for example, shell startup files. No problem, let's run:

  ```
  ls -a
  ```

  We can see the shell startup files and more — the directories "`.`" ("point") (current) and "`..`" ("double point") (top level) are also visible. Because that means "all" files and directories, including hidden ones. Hidden files in UNIX are just a naming convention — names must begin with a period. It is not an attribute as it is on Microsoft systems. Initially it was just a trick in the '`ls`' utility to hide the current and top directories, and then it came to be used as a naming convention to hide any file or directory.

Also we can see directory listing recursively:

```
ls -R
```

Another very important option is the "long list" ("`-l`"):

```
ls -l
```

We see a table with information about the file/directory in the corresponding lines.

  - The first column is the file attribute. The first letter is the file type: "`-`" ("dash") is a regular file, "`d`" is a directory, and so on. Then we can see read, write, and execute permissions for three user groups: owner, owner group, and everyone else. Once again, we see the difference between UNIX and Microsoft. In the first case it is an attribute, in the second case executability is just a naming convention: '.com', '.exe', '.bat'.
  - Some mystery column that we will discuss later.
  - Then we can see owner and owner group, size of file, time of modification and the name of file.

- `pwd` — print name of current/working directory
- `cd` — change directory. Without arguments – to home firectory.
- `cp` — copy files and directories. Most interesting option is '`-a|--archive`' with create recursive archive copy with preserving of permissions, timestamps, etc...
- `mv` — move (rename) files and directories.
- `rm` — remove files or directories.

  ```
  rm -rf ...
  ```

  means recursive delete without asking for confirmation.
- `mkdir` — make directories. If any parent directory does not exist, you will receive an error message:

```
mkdir a/b/c
mkdir: cannot create directory 'a/b/c': No such file or directory
```

To avoid this, use the `-p` option:

```
mkdir -p a/b/c
```

- `rmdir` — remove empty directories. If directory is not empty, you will receive an error message. Nowadays, running 'rm -rf something...' is sufficient in this case. But in the old days, when '`rm`' did not have a recursive option, to clean up non-empty directories, you had to create a shell script with 'rm's in each subdirectory and the corresponding 'rmdir's.

- `ln` — make links between files. Links are a very specific file type in UNIX and we will discuss them in more detail. If we look at the man page for the 'ln' command, we see a command very similar to 'cp'. But let's take a closer look:

```
cat > a
ln -s a b
ln a c
cat b; cat c
```

At the moment everything looks like a regular copy of the file, but let's try to change something in the one of them:

```
cat >> c
cat a; cat b
```

Wow, all the other linked files have changed too! We are just looking at the same file from different points, and changing one of them will change all the others. And in this they all seem to be alike. But let's try to delete the original file:

```
rm a
cat b; cat c
```

In the first case, we can still see the contents of the original file, but in the second case, we see an error message. Simply because the first

is a so-called hard link and the second is symbolic. We can see the difference between the two in the long ls list:

```
ls ?
```

And we can restore access to the content for the symbolic link by simply recreating the original file:

```
ln b a
cat c
```

Another difference between them is the impossibility of creating a hard link between different file systems and the possibility of such a linking for soft links. For more details on internal linking details, see the corresponding lecture. “Under the Hood”

Permissions

And finally, let’s discuss file permissions. As we remember, we have the owner user, the owner group and all the others, as well as read, write and execute permissions for such user classes. And we have the appropriate command to change these permissions:

```
chmod [-R] [ugoa][-+=](rwx)
```

And as we understand it, permissions are just a bit field. As far as we understand, permissions are just a bitfield and in some cases it might be more useful to set them in octal mode — see for information on this.

“Under the Hood”

You can also change the owner and group for a file or directory by command ‘chown’.

```
man chown - change file owner and group
```

But keep in mind — for security reasons, only a privileged user (superuser root) can change the owner of a file. The common owner of a file can change the group of a file to any group that owner is a member of:

```
chown :group file...
chgrp group file...
```

Text Viewers

As we remember, UNIX was originally created to automate the work of the patent office, has a rich set of tools for working with text data. But what is text? Generally it is just a collection of bytes encoded according to some encoding table, originally ASCII. In a text file, you will not see any special formatting like bold text, italics, images, etc. — just text data. And this is the main communication format for UNIX utilities since the 1970s.

As you know, Microsoft operating systems have different modes of working with files — text and binary. In UNIX, all files are the same, and we have no difference between text and binary data. See details in ("Under the Hood").

Concatenating and spliting

The first creature that helps us work with text files is the "`cat`". Not a real "`cat`", but an abbreviation for concatenation. With no arguments, cat simply copies standard input to standard output. And as we understand it, we can just redirect the output to a file, and this will be the easiest way to create a new file:

```
cat > file
```

When we add filenames as arguments to our command, this will be a real concatenation — they will all be sent to the output. And if we redirect them to a file, we get all these files concatenated into an output file.

```
cat f1 f2... > all
```

If we can combine something, we must be able to split it. And we have two utilities for different types of breakdowns:

- `tee` — read from standard input and write to standard output and files
- `split` — split a file into fixed-size pieces

Text viewers and editors

What is it viewer? In the TTY interface, the man command seems like a good one — when you run it, you get paper manuals that you can combine into a book, put on a shelf, and reread as needed. On a full-screen terminal — before, Ctrl-S (stop)/Ctrl-Q(repeat) was enough for viewing, because at first the terminals were connected at low speed (9600 bits per second for ex.), and now special programs were used — viewers. Unlike text editors, viewers does not need to read the entire file before starting, resulting in faster load times with large files.

Historically, the first viewer was the "more" pager developed for the BSD project in 1978 by Daniel Halbert, a graduate student at the University of California, Berkeley. The command-syntax is:

```
more [options] [file_name]
```

If no file name is provided, 'more' looks for input from standard input.

Once 'more' has obtained input (file or stdin), it displays as much as can fit on the current screen and waits for user input. The most common methods of navigating through a file are `Enter`, which advances the output by one line, and `Space`, which advances the output by one screen. When 'more' reaches the end of a file (100%) it exits. You can exit from "more" by pressing the "`q`" key and the "`h`" key will display help. In the 'more' utility you can search with regular expressions using the '`slash`' or the '`+/`' option. And you can search again by typing just a slash without regexp. Regexp is a very important part of UNIX culture and is used in many other programs and programming environments: (`http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-6`)

The 'main' limitation of the more utility is only forward movement in the text. To solve this problem, an improved 'more' called '`less`' was developed. The "`less`" utility buffers standard input, and we can navigate forward and backward through the buffer, for example. using the cursor keys or the `PgUp/PgDown` keys. A reverse search with a question mark is possible.