UNIX AND LINIX IN INFOCOMMUNICATION
Week 4

O. Sadov

Utilities

All commands typed on the command line or executed in a command file are either commands built into the interpreter or external executable files. The set of built-in commands is quite small, which is determined by the basic concept of UNIX - the system should consist of small programs that perform fairly simple well-defined functions that communicate with each other via a standard interface.

A rich set of standard utilities is a good old tradition for UNIX-like systems. The shell and the traditional set of UNIX utilities, is a POSIX standard.

As we discussed earlier, we have different branches of development of UNIX-like systems with different types of utilities:

- BSD-like dating back to the original UNIX implementations;
- SYSV based systems;
- GNU utilities.

Some command syntax was changed by the USL with the introduction of SYSV, but on most commercial UNIX a set of older commands was still included for compatibility with earlier BSD-based versions of UNIX from the same vendor. GNU utilities often combine both syntaxes and add their own enhancements to traditional utilities. And now the GNU toolkit has become the de facto standard.

Executable files on UNIX-like systems do not have any file name extension requirements as they do on Windows. The utility executable can have any name, but must have execute permission for the user who wants to run it.

A standard utility can have options, argument of options, and operands. Command line arguments of programs are mainly parsed by the getopt() function, which actually determines the form of the parameters when the command is invoked. This is an example of utility's synopsys description:

```
utility_name[-a][-b][-c option_argument] \
                        [-d|-e][-f[option_argument]][operand...]
```

1. The utility in the example is named `utility_name`. It is followed by options, option-arguments, and operands. The arguments that consist

of <hyphen-minus> characters ('-') and single letters or digits, such as 'a', are known as "options" (or, historically, "flags"). Certain options are followed by an "option-argument", as shown with [-c option_argument]. The arguments following the last options and option-arguments are named "operands".

The GNU `getopt()` function supports so-called long parameters, which start with two dashes and can use the full or abbreviated parameter name:

```
utility_name --help
```

2. Option-arguments are shown separated from their options by <blank> characters, except when the option-argument is enclosed in the '[' and ']' notation to indicate that it is optional.

   In GNU getopt's long options also may be used the 'equal' sign between option and option-argument:

   ```
   utility_name --option argument --option=argument
   ```

3. When a utility has only a few permissible options, they are sometimes shown individually, as in the example. Utilities with many flags generally show all of the individual flags (that do not take option-arguments) grouped, as in:

   ```
   utility_name [-abcDxyz][-p arg][operand]
   ```

   Utilities with very complex arguments may be shown as follows:

   ```
   utility_name [options][operands]
   ```

4. Arguments or option-arguments enclosed in the '[' and ']' notation are optional and can be omitted. Conforming applications shall not include the '[' and ']' symbols in data submitted to the utility.

5. Arguments separated by the '|' (<vertical-line>) bar notation are mutually-exclusive.

   ```
   utility_name [-a|b] [operand...]
   ```

Alternatively, mutually-exclusive options and operands may be listed with multiple synopsis lines. For example:

```
utility_name [-a] [-b] [operand...]
utility_name [-a] [-c option_argument] [operand...]
```

6. Ellipses ("...") are used to denote that one or more occurrences of an operand are allowed.

System manuals

The easiest way to get information about the use of a command is with the -h option or --help for GNU long options.

Also since its inception, UNIX has come with an extensive set of documentation. Some information is often found in the /usr/doc or /usr/local/doc or /usr/share/doc directories as text files.

But the cornerstone of the Unix help system is the man command. And the man in this case is not about gender — it is just an abbreviation for manual.

```
man man
```

The man command has been traditional on UNIX since its inception, was created in the teletype era and still works great on all types of equipment. And in the synopsis of the man command, we see two worlds, two UNIX utility systems: BSD and SYSV.

And at first we can see the different command syntaxes:

```
SYSV~--- \cmd{man [-t] [-s section] name}
GNU, BSD~--- \cmd{man [-t] [section] name}
```

Parts of the man page are more or less the same:
NAME, SYNOPSIS, DESCRIPTIONS, FILE, SEE ALSO, DIAGNOSTIC, BUGS

The minus S option with some integer parameter of man command in the SYSV variant denotes a section of real paper manuals supplied with the OS by vendor. For GNU/BSD flavors, use only the section number. Section

numbers are also different.

Let's look for example to well known C-languge function 'printf' manual page. But

```
man printf
```

"man printf" shows us the man page for the shell command, not the C function. To see the manual for the C printf function, we must run:

```
man 3 printf
```

To view a list of printf-related manual pages, we must run:
whatis — search the whatis database (created by makewhatis) for complete words
and
'man -k' or 'apropos' — search the whatis database for strings.

The databases should be created by the 'makewhatis' program, which is usually started at night by the cron service. If you have a freshly installed system and want to run any command related to the whatis database, you posibly need to start the makewhatis program manually.

OK — let's look to real man page:

```
zless /usr/share/man/man1/man.1.gz
```

Troff(short for "typesetter roff")/nroff(newer "roff") is an implementation of a text formatting program, traditional for UNIX systems, using a plain text file with markup. It is ideologically based on the RUNOFF MIT program, developed in 1964, and after a series of source code losses and rewrites, a C-based implementation was re-implemented in 1975. Under the name Troff, it was accepted for use on the UNIX system and, of course, into the AT&T patent department.

See – http://manpages.bsd.lv/history.html

The main advantage of this tool was portability and the ability to generate printouts for various devices, from a common ASCII printer to high-quality typographic photosetters. Creating new technologies such as PostScript printers simply adds the appropriate output drivers for the markup renderer. Compared to the now better known WYSIWYG systems, such systems have

better portability between platforms and higher typing quality. Moreover, such systems are more focused on the programmatic creation of printed documents without human intervention.

Let's take a look at an example of manually rendering the man page that was hidden under the hood of the man command:

```
$ zcat /usr/share/man/man1/man.1.gz | tbl | eqn -Tascii | \
                                              nroff -man | less
```

It's just a software pipeline in which we unpack the compressed TROFF source, go through the TROFF preprocessor for tables and math equations, pass a troff variant named 'nroff' to output a text terminal, and finally pass a text pager/viewer named 'less'.

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.

And in fact, when you run the "`man`" command, you see just a viewer's interface, in most cases it is "`less`". We will discuss viewers further, but in a nutshell, the most commonly used of them the '`less`' handles the UP/DOWN keys normally, exiting a program — the 'q' key means 'quit'.

Well. It looks great, but the man-style documentation representation has certain limitations: it is only a textual representation without any useful functions invented after that time — for example, impossibility of using hypertext links.

To improve it, the GNU Project has created an information system that also works on all types of alphanumeric terminals, but with hypertext support. For many GNU utilities, the corresponding help files are in info format, and the man pages recommend that you refer to `info`.

Info has its own user interface and the best way to learn it is to simply run the '`info info`' command. The internal source of "info" is the text markup files in texinfo format. From such files are generated text files for viewing on terminals, and also by the TEX typesetting system generated documentation for printing.

TEX is another typesetting system (or "formatting system") that was developed and mostly written by Donald Knuth, released in 1978. TEX and its LATEX extension are very popular in the scientific world as a means of typing complex mathematical formulas.

OK. But the inability to use graphic illustrations and any kind of multimedia context remained relevant. And in the past, almost every commercial UNIX system vendor created their own help system, which includes both hypertext support and graphics for ex., and worked in the X Window System.

But now with the advent of HTML (yet another text markup language), such reference information began to be provided in this format directly on the system or on WWW servers. Often these HTML pages or print-ready PDF versions are simply generated from some content oriented markup such as DocBook XML.

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`