UNIX AND LINIX IN INFOCOMMUNICATION Week 2

O. Sadov

Your system

The best advantage of free systems is their availability. You can download many kinds of Linux or UNIX systems for free. You can distribute such systems, downloaded from the Internet, and use them to create your own customized solutions using a huge number of already existing components.

For example, for educational purposes we use our own Linux distribution called NauLinux. This title is the abbreviated title of the Russian translation of the original Scientific Linux project — "Nauchnyi Linux". We are adding many software packages for working software-defined networks and quantum cryptography emulating and are using this new distributions to simulate various complex systems in educational or research projects. This distribution is free and is used by students to create their own solutions.

Scientific Linux on which we are based is also free. It was created at Fermilab for use in high energy physics and has focused from the beginning on creating specialized flavours optimized for the needs of laboratories and universities.

This, in turn, is based on Red Hat Enterprise Linux (RHEL), a commercial Linux distribution widely used in the industry. You will be charged for using the binaries for that distribution and getting support from the vendor. But the sources from this distro are still free, and anyone can download it and rebuild their own distro.

The source of RHEL, in turn, is the Fedora experimental project, developed by the community with the support of Red Hat. Leveraging large communities of skilled and motivated users lowers the cost of testing, development, and support for the company. And this is an example of the profitable use of the Free and Open Source model by a commercial company.

There are many free BSD OS variants, currently FreeBSD, NetBSD, OpenBSD, DragonFly BSD. They all have their own specifics and their own kernels with incompatible system calls. This is a consequence of the fact that the development of these systems is driven by communities in which disagreements arise from time to time, and they are divided according to different interests regarding the development of systems.

On the Linux project, we still have one and only one benevolent dictator, Linus Torvalds. As a result, we still have a single main kernel development thread published on `kernel.org`. While many other experimental Linux kernel flavours are also being developed, not all of them are accepted into the mainstream. In turn, on the basis of this single kernel, the development of various OS distributions is made, often with some changes from the distribution vendors.

The most commonly used free Linux distributions are:

- community driven Debian project
- Ubuntu Ditro based on Debian and developing by Canonical company
- The Fedora Project on which RHEL development is based
- and Centos — another free RHEL respin

Gentoo, Arch, Alpine and many other Linux distributions are also well known. Many projects are focused on embedded systems, such as BuildRoot, Bitbake, OpenWRT, OpenEmbedded and others.

You can usually use them in different ways - install on your computer (including coexistence with other systems such as Windows, with the ability to dual boot), boot from a live image or network server without installing the OS to a local hard drive, run as a container or virtual machine on your local computer or network cloud, etc. You can find detailed information on installing and configuring such systems in the documentation of the respective projects.

Moreover, you can simply use online services to access UNIX or Linux systems, for example:

- `https://skn.noip.me/pdp11/pdp11.html` — PDP-11 emulator with UNIX
- `https://bellard.org/jslinux` — a lot of online Linux'es

When you log in, you will be asked for a user and password. Depending on your system configuration, after logging in, you will have access to a graphical interface or text console. In both cases, you will have access to the Shell command line interface, which we are most interested in.

The user password is set by the 'root' superuser during installation or system configuration, and this password can be changed by the user himself or by the superuser for any user using the '`passwd`' command.

Command Interpreter

The first characters that you can see at the beginning of a line when you log in and access the command line interface is the so-called Shell prompt. This prompt asks you to enter the commands. In the simplest case, in the Bourne shell, it's just a dollar sign for a regular user and a hash sign for a superuser. In modern shells, this can be a complex construct with host and user names, current directory, and so on. But the meaning of the dollar and hash signs is still the same.

The Shell as a command interpreter that provides a compact and powerful means of interacting with the kernel and OS utilities. Despite the many powerful graphical interfaces provided on UNIX-like systems, the command line is still the most important communication channel for interacting with them.

All commands typed on a line can be used in command files executed by the shell, and vice versa. Actions performed in the command interpreter can then be surrounded by a graphical UI, and the details of their execution, thus, will be hidden from the end user.

Each time a user logs into the system, he finds himself in the environment of the so-called home interpreter of the user, which performs configuration actions for the user session and subsequently carries out interactive communication with the user. Leaving the user session ends the work of the interpreter and processes spawned from it. Any user can be assigned any of the interpreters existing in the system, or an interpreter of his own design. At the moment, there is a whole set of command interpreters that can be a user shell and a command files executor:

- `sh` is the Bourne-Shell, the historical and conceptual ancestor of all other shells, developed by Stephen Bourne at AT&T Bell Labs and included as the default shell for Version 7 of Unix.

- `csh` — C-Shell, an interpreter developed at the University of Berkeley by Bill Joy for the 3BSD system with a C-like control statement syntax. It has advanced interactive tools, task management tools, but the command file syntax was different from Bourne-Shell.

- `ksh` — Korn-Shell, an interpreter developed by David Korn and comes standard with SYSV. Compatible with Bourne-Shell, includes command line editing tools. The toolkit provided by Korn-Shell has been fixed as a command language standard in POSIX P1003.2.

In addition to the above shells that were standardly supplied with commercial systems, a number of interpreters were developed, which are freely distributed in source codes:

- `bash` — Bourne-Again-Shell, quite compatible with Bourne-Shell, including both interactive tools offered in C-Shell and command line editing.
- `tcsh` — Tenex-C-Shell, a further development of the C-Shell with an extended interactive interface and slightly improved scripting machinery.
- `zsh` — Z-Shell, includes all the developments of Bourne-Again-Shell and Tenex-C-Shell, as well as some of their significant extensions (however, not as popular as bash and tcsh).
- `pdksh` — Public-Domain-Korn-Shell, freely redistributable analogue of Korn-Shell with some additions.

There are many “small” shells often used in embedded or mobile systems such as ash, dash, busybox.

Environment variables

The operating system supports a special kind of resource called environment variables. These variables are a NAME/VALUE pair. The name can start with a letter and be composed of letters, numbers, and underscores. Variable values have only one type — character strings. Names and values are case sensitive. And, as we'll see, variables can be global and local, just like in regular programming languages.

To get the value of a variable on the Shell, precede the variable name with a dollar sign (“$”):

```
$ echo $USER
guest
```

mverb If the variable is not set, an empty string is returned.

The assignment operator ("=") is used to set the value of a variable (in the case of Bourne-like shells):

```
$ TEST=123
```

mverb or the built-in 'set' operator (in the case of C-like Shells):

```
$ set TEST=123
```

mverb You can check your settings by calling the '`echo`' command, which simply sends its own arguments to stdout.

```
$ echo $TEST
123
```

mverb

The '`set`' command with no arguments lists the values of all variables set in the environment:

```
$ set
COLUMNS=197
CVS_RSH=ssh
DIRSTACK=()
....
```

mverb

Shell commands can be combined into command files called scripts, where the first line, in a special kind of comment, specifies the shell to execute the set. For example, let's create a file called `test` in a text editor or just by command "`cat`" with the following content:

```
#!/bin/sh

echo TEST:
echo $TEST
```

mverb This program will print the text message "TEST:" and the value of the `TEST` variable, if this one specified, to standard output. You can run it from the command line by passing it as a parameter to the command interpreter:

```
$ sh test
TEST:
```

mverb

We didn't see the value of the variable. But why? Because we started a new shell process to run the script. And we have not set this variable in the context of this new shell. Let's do it. Okay, let's expand our definition to the environment space of all processes started from our current shell:

```
$ export TEST=456
$ sh test
TEST:
456
```

mverb And, as we can see, the value of the variable in our current SHELL has also changed:

```
$ echo $TEST
456
```

mverb

The `export` operation is the globalization of our variable. You can get the settings for such global exported variables for a session by calling the interpreter builtin command "`env`", in the case of Bourne-like interpreters (`sh`, `ksh`, `bash`, `zsh`, `pdksh`. . .), and "`printenv`" in the case of the C-Shell style (`csh`, `tcsh`. . .):

```
$ env
HOSTNAME=myhost
TERM=xterm
SHELL=/bin/bash
...
```

mverb And this is our first example of using the command pipeline — we just look at only the `TEST` variable in the full set:

```
$ env | grep TEST
TEST=456
```

mverb

Variables can be local to a given process or global to a session. You can set local values for variables by preceding command calls:

```
$ TEST=789 sh -c 'echo $TEST'
789
```

mverb But, as we can see, our global settings are the same:

```
$ echo $TEST
456
$ sh /tmp/test
TEST:
456
```

mverb

We can remove the setting of variables using the "`unset`" command:

```
$ unset TEST
$ echo $TEST
```

mverb As we can see, this variable has also been removed from the list of global environment variables:

```
$ sh /tmp/test
TEST:

$ env | grep TEST
$
```

mverb And there is no "`unexport`" command — just only "`unset`" command.

Finally, as with traditional programming languages, we can use shell scripts such as libraries that can be run from an interactive shell session or from another shell script to set variables for top-level processes. Let's try to write another script in which we simply set the `TEST` variable.

```
$ cat > /tmp/test_set
TEST=qwe
$ source /tmp/test_set
$ echo $TEST
```

```
qwe
```

But for our first script, this variable is still invisible:

```
$ sh /tmp/test
TEST:
$
```

Why? Just because it is not exported. Let's export:

```
$ cat > /tmp/test_exp
export TEST=asd
```

And run our test script again:

```
$ sh /tmp/test
TEST:
asd
```

As we can see, in our main shell session, the variable has changed too:

```
$ echo $TEST
asd
```

System variables

Environment variables are one of the simplest mechanisms for interprocess communication. Let's discuss some of the most commonly used system variables that are predefined for use by many programs.

The most basic ones are:

- `USER` is user name.
- `HOME` is the home directory.
- `SHELL` is the user's home shell.
- `PS1` for shell-like or promt for cshell-like shells is the primary shell prompt, printed interactively to standard output.

- PS2 is a secondary prompt that is issued interactively to standard output when a line feed is entered in an incomplete command.

All of these variables are more or less self-explanatory, but some commonly used variables are not that simple, especially in terms of security and usability:

PATH

PATH is a list of directories to look for when searching for executable files. The origin of this idea comes from the Multics project. This is a colon-separated list of directories. The csh path is initialized by setting the variable PATH with a list of space-separated directory names enclosed in parentheses. As you probably know, on Windows you have the same environment variable but with fields separated by semicolons.

But this is not only the difference between UNIX-like and Windows PATH. On Windows, you have a default directory to search for executable files — the current directory. But on UNIX or Linux, not. But why? It seems so useful. And long ago it was normal to have a PATH that started with dot, the current directory.

But let's imagine this situation: a user with a name, for ex. "badguy", has downloaded many movies in his home directory to your university lab computer and filled up an entire disk. You are a very novice sysadmin and do not know about disk quotas or any another magic that can help you avoid this situation, but you know how to run disk analyzer to find the source of the problem.

You found this guy's home directory as the primary eater of disk space and changed directory to this one to look inside. You call the standard command "ls" for listing of files or directories in badguy's home directory:

```
$ cd ~badguy/
$ ls -l
```

But he's a really bad guy — he created an executable named "ls" in his home directory and wrote in it only one command:

```
rm -rf /
```

Which means — delete all files and directories in the entire file system, starting from the root directory, without any questions or confirmations. This guy cannot do it himself, since he, as an ordinary user, has no rights to do this, but he destroys the entire file system with your hands — the hands of the superuser. Because of this, it is a bad idea to include a dot in your search `PATH`, especially if you are a superuser.

IFS

`IFS` — Input field Separators. The IFS variable can be set to indicate what characters separate input words — `space`, `tab` or `new line` by default. This feature can be useful for some tricky shell scripts. However, the feature can cause unexpected damage. By setting `IFS` to use slash sign as a separator, an attacker could cause a shell file or program to execute unexpected commands. Fortunately, most modern versions of the shell will reset their `IFS` value to a normal set of characters when invoked.

TERM

`TERM` is the type of terminal used. The environment variable `TERM` should normally contain the type name of the terminal, console or display-device type you are using. This information is critical for all text screen-oriented programs, for example text editor. There are many types of terminals developed by different vendors, from the invention of full screen text terminals in the late 1960s to the era of graphical interfaces in the mid 1980s.

It doesn't look very important now, but in some cases, when you use one or another tool to access a UNIX/Linux system remotely, you may have strange problems. In most cases, access to the text interface is used, for example, via SSH, telnet or serial line, since it requires less traffic. But in some combinations of client programs and server operating systems, you may see completely inappropriate behavior of full-screen text applications such as a text editor. This could be incorrect display of the editor screen or incorrect response to keystrokes. And this is a consequence of incorrect terminal settings. The easiest way to solve this problem is to set the `TERM` variable. Just try setting them to type “ansi” or “vt100”, because these are the most commonly used types of terminal emulation in these clients.

Other variables related to terminal settings:

- `LINES` is the number of lines to fit on the screen.
- `COLUMNS` — the number of characters that fit in the column.

And the variables related to editing:

- `EDITOR` is the default editor because there are many editors for UNIX-like systems.
- `VISUAL` — command line editing mode.

Some settings to personalize your environment:

- `LANG` is the language setting.
- `TZ` is the time zone.

And some examples of application specific settings:

- `DISPLAY` points to an X-Window Server for connecting graphical applications to the user interface.
- `LPDEST` is the default printer, if this variable is not set, system settings are used.
- `MANPATH` is a list of directories to look for when searching for system manual files
- `PAGER` is the command used by `man` to view something, manual pages for example.

Special symbols of Shell

In addition to the dollar sign ($), which was used to retrieve a value from a variable by name, we have seen some other special characters earlier: space and tab as word separators, hash (#) as a line comment. We also have many other special characters.

Newline and semicolon (;) are command separators.

The ampersand (&) can also look like a command separator. But if you use this sign, the command written before it will run in the background (that is, asynchronously as a separate process), so the next command does not wait for completion.

Double quoted string ("STRING") means partial quoting. This disables the interpretation of word separator characters within STRING — the entire string with spaces appears as one parameter to the command.

Single quoted string ('STRING') interpreted as a full quoting. Such quoting preserves all special characters within STRING. This is a stronger form of quoting than double quoting.

Backslash (\) is a quoting mechanism for single characters.

The backquotes (`) indicates command substitution. This construct makes the output of the command available as part of the command line. For example to assign to a variable. In POSIX-compliant environments, another form of command (dollar and parentheses; $(...)).) can be used.

And in the special characters of the shell, we can see some of the characters that we will see in the regular expressions. They are used to compose a query to find text data. It is a very important part of the UNIX culture, borrowed by many programming languages to form such search patterns. These are asterisk (*) and a question signs (?).

Asterisk used when a filename is expected. It matches any filename except those starting with a dot (or any part of a filename, except the initial dot).

Question symbol used when a filename is expected, it matches any single character.

The set of characters in square brackets means — any of this set. The same with an exclamation sign — none of this set.