UNIX AND LINIX IN INFOCOMMUNICATION
Week 2

O. Sadov

Components

The main design principle used in UNIX-like systems is the "KISS" principle. KISS, an acronym for "keep it stupid simple" or more officially "keep it short and simple", is a design principle noted by the U.S. Navy in 1960. The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

And from the very beginning UNIX was a very flexible modular system. The basic set of components from which UNIX-like systems are built is:

- Kernel
- Shell
- Utilities
- Libraries

The kernel is the first bunch of OS code that is loaded onto your computer's memory and run for execution. This program launches all processes on the system, handles interactions between system resources, and still live while your system is running. The kernel runs with the highest privileges and has access to all system resources. All processes in the system operate in user space and interact with such resources and among themselves, sending requests to the kernel using special software functions named "system calls". And the kernel handles such requests according to the permissions between the processes and resources.

Under the hood — kernel as a set of interrupt handlers

But if we have a kernel, it seems reasonable to have a shell around it. And we have this one. Oh, sorry, not one – now there are many shells dating back to the first Unix shell by Ken Thompson, introduced in 1971. Actually, the shell is the first and most important communicator between human, programs, and kernel. Generally it's just a program that is launched when the user logs in. It listens for standard input (usually from the keyboard) and sends the output of commands to standard output (usually to the screen).

The best-known implementation of the UNIX shell is the Bourn shell, developed by Stephen Bourne at Bell Labs in 1979 and included as the default

interpreter for the Unix version 7 release distributed to colleges and universities. Supported environment variables, redirecting input/output streams, program pipes and powerful scripting. All modern shells (and not only UNIX shells) inherit these capabilities from this implementation.

The shell is a very effective glue for utilities in multitasking operating systems. The most commonly used utilities were developed early in the life of UNIX. There are tools for working with users, groups, files, processes. Since UNIX was originally created to automate the work of the patent department, it has a rich set of tools for processing text files and streams. The most commonly used design pattern for UNIX utilities is the filter between standard input and standard output. An arbitrary number of such programs can be combined into a so-called software pipeline that uses Shell program pipes for interprocess communication.

Each such utility can be very simple, but together they can be a very powerful compound program that fits on a single command line. Doug McIlroy, head of the Bell Labs Computing Sciences Research Center, and the inventor of Unix pipes, described the Unix philosophy as follows: "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."

Currently, the most commonly used utilities are those of the GNU project, which were created after the commercialization of UNIX. In most cases, they are richer in their capabilities and more complex parameters than the classic SYSV set of utilities that you see in commercial UNIXes.

On small embedded systems, you might see a systems like "busybox" that looks like a single binary with many faces built from a configurable modular library. It may look like a fully featured set of UNIX-style utilities, including a shell and a text editor. And during the build phase, you can choose exactly the features you need to reduce the size of the application.

All utilities and shells are built on top of software libraries. They can be dynamically or statically linked. In the first case, we have more flexibility for updates and customization and we get a set of applications that take up less disk/memory space in total. In the second case, we have a solid state application that is less dependent on the overall system configuration and can be more platform independent. And as I said earlier in the first case, you can use libraries with "viral licenses" (like the GPL) to write proprietary applications, but in the second case, you cannot.

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.