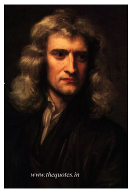
UNIX AND LINIX
IN INFOCOMMUNICATION

O. Sadov

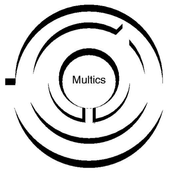
1.09.2020

History

If I have seen further than others,
it is by standing upon the shoulders of giants
Isaac Newton



This is a well-known saying. But we must understand that many of these giants have failed. But such fails can be a good lesson for new developers.



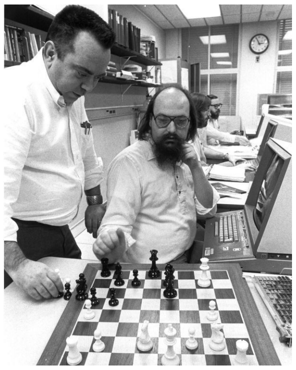
One such giant was the MULTICS project. The development of Multics began in 1965 as a research project by MIT, General Electric and Bell Labs to create a time-sharing, multiprocessing and multiuser interactive operating system. After several years of development, the enthusiasm of the developers decreased more and more as the system became more and more complex and the prospects for completion of development became less and less.

Bell Labs pulled out of the project in 1969; but some of the people who worked on it got a lot of experience. Among them were Ken Thompson and Dennis Ritchie of Bell Labs, the inventors of the UNIX OS.

It's funny, but the history of Unix systems is closely related to computer games. It started in 1969 when Ken Thompson discovered an old PDP-7 computer in a dark corner of the lab and wanted to use it to play Space Travel game.

There was little to do — an operating system had to be written to run it. And he did it at 1970. It was originally a single-tasking OS written in assembly language that was loaded from paper tapes and called UNICS as opposed to the complexity of MULTICS.

And then the team of Ken Thompson and Dennis Ritchie received a new DEC PDP11 computer to develop a word processing system for the Bell Labs patent department. For the first three months the machine sat in a corner, enumerating all the closed Knight's tours on a 8 × 8 chess board — just because the hard drive wasn't shipped with a super new computer. This time could be used to choose a programming language, because it was a computer with a completely new architecture and programs written on PDP7 assembler was not useful for it. And most interesting was the concept of another project used in some R&D projects including MULTICS — BCPL. It was a high-level programming language focused on portability. Most of this language was written in the language itself, and only a small machine-dependent part was written in assembly. To support a new machine, only 1/5 of the compiler code needed to be rewritten, which usually took 2-5 man-months.



Thompson used the same concept when writing his simplified successor to BCPL, language B. This language was not very expressive and effective on the PDP11. In 1972, Ritchie started to improve B, which resulted in creating a new language C. In 1973, the UNIX kernel was refactored in C language to follow the same concept of portability — most of the code was machine independent.

The system was distributed in source code among universities for a nominal fee, which served as an explosive growth in its popularity in the 80s. Almost all the developers of new computer systems since this period have used UNIX as the base platform for their new developments. One of the most famous of these is the Berkeley Software Distribution (BSD) developed at the University of California, Berkeley based on Unix version 6 with its own copyleft license. And most of the hardware vendors of the 1980s used BSD as the base OS for their new computers.

UNIX was not a significant part of AT&T Bell Laboratories' business, and it was not a problem for them. But in the 1980s Bell Labs split into several companies as a result of an antitrust lawsuit against AT&T. The new UNIX System Laboratories company was created and the new UNIX System V specification was developed. UNIX was the main business of this company, and they were very aggressive in pushing the new standard in the market. And that has been the cause of UNIX wars against non-commercial developers including BSD.

The commercialization of the UNIX system market and the move to a closed development and distribution model has led to an alternative movement to develop a set of programs similar to the set of utilities standard in UNIX — the GNU (self-referential abbreviation "GNU is Not Unix") project. In 1991, a Finnish student Linus Torvalds created his own operating system kernel, which is compatible with the UNIX OS, called Linux. The Linux kernel, combined with the set of utilities from the GNU project, served as the basis for creating a complete operating system, comparable in capabilities to commercial UNIX systems, and usually even superior to them.

Open and Free

Initially, when computer R&D projects were mostly just university research, they were open source and free of charge as a common scientific research result. In addition, scientists were very interested in widespread dissemination of this result, because their reputation directly depended on the prominence of their scientific work.

Commercialization has changed this world to a closed and paid model. Not completely closed, because standardization is very important for government agencies and corporate consumers to protect investments and prevent vendor locking. As a consequence, many open standards have been created by committees and organizations:POSIX, ANSI, ISO.

And openness was a serious weapon in business competition. For example, some well-known companies, including DEC, IBM, HP, Siemens, Bull and others created the Open Software Foundation (OSF) to fight SUN and AT&T during the so-called “Unix wars”. The POSIX subsystem (which was actually just a description of UNIX-like systems standards) was included in Microsoft Windows NT because in the 1980s the US federal government required compatibility with this open standard for government contracts.

This openness is very important because we get more compatible systems from different vendors, ideally without undocumented features. As a result, we have a computing infrastructure with higher efficiency and lower cost of ownership.

But for some people, especially in the scientific world, this is not enough. And in 1985, Richard Stallman from MIT published the GNU Manifesto where he announced the GNU Project. The main goal of this project was to create a UNIX-like OS with a full set of UNIX utilities from completely free software. The Free Software Foundation (FSF) was formed to support these activities.

But what is this freedom in the computer world and how is it different with openness? This difference is most accurately described in licenses. In the proprietary world, the most widely used are so-called copyright licenses, which usually restrict certain user rights. Even with legal access to the source code, the copyright holder can harass the consumer, as we saw in the USL versus BSD or SCO versus IBM lawsuits.

In contrast, the free software world uses copyleft licenses. The most famous permissive licenses for free software were published in the late 1980s. Two of them, named BSD and MIT — the educational institutions in which they were created — look almost the same and give us the following basic rights:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).

- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

The most famous projects released under these licenses are BSD Unix and the MIT X-Window graphics system. Such licenses do not restrict the use of closed derivative projects and their proprietarization. To avoid this, the GNU General Public License (GNU GPL or GPL) was developed. One important limitation added to the fundamental freedoms of this license (run, learn, share and modify software) is the limitation for closing. Any derivative work must be distributed under the same or equivalent license terms.

It's interesting, but this license does not completely restrict the creation of proprietary closed applications using GPL licensed software. For example, the OS kernel or shared libraries, simply because they are not included directly in the proprietary application code.

We now have many free and open source licenses approved by the Open Source Initiative:
`https://opensource.org/licenses`

Another challenge for the free and open world is patent lawsuits. For example, in 2007, Microsoft threatened to sue Linux companies like Red Hat over patent violations. To solve this problem GPLv3 license has been created and some activities such as the Open Invention Network (OIN), which have a defensive patent pool and community of patent non-aggression which enables freedom of action in Linux.

Main Concepts

At the top level, UNIX-like systems can be very convenient for common users, and they may not even know they are using this type of OS. For example, currently the most commonly used operating systems are Linux-based Android systems and UNIX-based Apple systems, in which the user only sees the user friendly graphical UI.

But beginners who are just starting to learn UNIX-like systems for administration or development sometimes complain about their complexity. Don't be afraid — actually such systems are based on a fairly simple concepts.

There are only three things (three and a half to be exact) you need to know to be comfortable with any UNIX-like system:

1) Users

2) Files

3) Processes

3.5) Terminal lines

Users

Users is not very well known to modern users only because we now have a lot of computer devices with personal access. UNIX was created at a time when computers were expensive rarity and a single computer was used by many users. As a consequence, from the beginning, UNIX had strong security policy and restrictions on permissions for users.

And now on UNIX-like systems, we have dozens of users and groups, even if hidden by an autologin machinery. And most of them are so-called pseudo-users, which are needed to start system services. As we will see later, they are required by architecture, since it is on the permissions of users and groups that the system is built to control access to system resources (processes and files).

If we are talking about ordinary users, they can log in with a username and password and interact with the applications installed on the system. Each user has full permissions only in their home directory and limited access rights to files and directories outside of it. This can be viewed as foolproof — common users cannot destroy anything on the system just because they do not have such permissions. Moreover, they cannot view another user's home directory or protected system files and directories. To perform system administration tasks, the system has a special superuser (generally called "root") with extra-permissions.

At the system level, each user or group looks like an integer number: a user identifier (UID) and a group identifier (GID).

Files

Files are the next important thing for UNIX-like systems. Almost all system resources look like files, including devices and even processes on some systems. And the basic concepts have been the same since the beginning of the UNIX era. We have a hierarchical file system with single root directory. All resources, including file systems existing on devices or external network resources, are attached to this file system in separate directories — this operation is called "mount". On the other hand, you can access a device (real or virtual) as a stream of bytes and work with it like a regular file. All files and directories are owned by users (real or pseudo) and groups, and read, write, and execute access to them is controlled by permissions.

Processes

A process is a program launched from an executable file. Each process belongs to a user and a group. The relationship between the owners of processes and resources determines the access rights according to the resource permissions. All processes live in a hierarchical system based on parent-child relationships. There is an initial process on the system called "`init`" that is started at boot. All system services are started from this initial process.

There are fundamentally two types of processes in Linux:

- Foreground processes (also referred to as interactive processes) — these are initialized and controlled through a terminal session. In other words, there has to be a user connected to the system to start such processes; they haven't started automatically as part of the system functions/services.

- Background processes (also referred to as non-interactive/automatic processes) — are processes not connected to a terminal; they don't expect any user input. System services are always background processes.

And finally — interactive foreground processes must be attached to the terminal session through the terminal line. At the time of the creation of UNIX, a TTY (teletype) device (originally developed in the 19th century), was the primary communication channel between the user and the computer. It was a very simple interface that worked with a stream of bytes encoded according to the ASCII character set. The connection is made via an serial interface (for example RS232) with a fixed set of connection speeds.

This interface is still the main user interface for UNIX-like systems. This interface is still the main user interface for UNIX-like systems. The implementation of each new form of user interaction, such as full-screen terminals, graphics systems, and network connections, all started with the implementation of a simple TTY command line interface. Moreover, as we will see, this interface abstraction gives us a very powerful and flexible mechanism for communication between programs, possibly without human intervention.

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

Kernel

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 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.

Shell

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.

Utilities

The shell is a very effective glue for utilities in multitasking operating systems. The most commonly used utilities 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.

Libraries

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 Distro 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 Build-Root, 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 system installation or configuration. 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
```

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

```
$ TEST=123
```

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

```
$ set TEST=123
```

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

```
$ echo $TEST
123
```

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

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

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
```

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:
```

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
```

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

```
$ echo $TEST
456
```

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
...
```

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
```

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
```

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

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

We can remove the setting of variables using the “unset” command:

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

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

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

```
$ env | grep TEST
$
```

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. 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. 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 or telnet, 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
- `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 manuals
- `PAGER` is the command used by `man` to view something(e.g. man pags.

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.

OS as a bunch of interupt handlers

We talked about the kernel as some kind of magic program that loads at boot and keeps running while your computer is on. But what exactly does this program do? To understand this, we need to look at the evolution of software in computer systems. At the beginning, when they had just created computer programs, the question quickly arose — how to reuse what was written?

So, the well-known Pareto law works — when you need something new, most likely, it is already 80% written by someone, and you only need to complete or redo only 20%. You can of course use the good old "cut and paste" method, but this is often a source of errors and bugs, especially on large and complex systems.

To solve this problem, some of the most commonly used functions have been compiled into so-called software libraries, which are usually just archives of object files of such compiled programs. You can then link some of these object files to your executable and use functions developed and tested by others in your program.

But with the development of the computer industry and the widespread use of computer systems, the question arises about a more stable software core for working with external devices and system resources, for regulating access to them for programs and users. And the solution came in the form of interrupts.

What is it interrupt? Accordingly Dijkstra "It was a great invention, but also a Box of Pandora" (E.W. Dijkstra EWD 1303). A more formal interrupt is the processor's response to an event that requires attention from the software. Historically, interrupts were first created to solve problems detected by hardware, for example. arithmetic overflow implemented for UNIVAC I (1951).

Interrupts are widely used to handle arithmetic errors, improper memory access, and finally, to work with devices. Technically, interrupts are simply events that switch the processor from executing the current program to the execution of the so-called interrupt handler program, the address of which is placed in a special table placed in a fixed memory area — the table of interrupt vectors. For each type of interrupt, a special position in this table is used with the address of the corresponding handler program. When an interrupt request is generated, the processor saves the current state of the processor registers and switches the program counter to this handler. At the end of the handler program, a special machine instruction is executed — the exit from interrupt and the processor state before the interrupt was called is restored.

But the real breakthrough came when software interrupts were invented. They were first implemented as a debugging feature called "transfer trapping" in the IBM 704 (1955). But gradually it became clear that this allows you to implement the functionality of the processor with a virtual instruction set. For example, we can only call one machine command to open a file. It must be a software interrupt command.

There are different commands for different computer architectures: INT on Intel, SVC/SWI on ARM, TRAP on SPARC, etc. On the PDP 11, we had more than one software interrupt in the instruction set — EMT for DEC OS and SYS used by UNIX.

But they all do only one thing — we jump to the vector interrupt and run the corresponding handler program.

Such a command from the programming API side looks like a function called "system call". And when we call such a function from our program, it looks like a single machine instruction in assembler, but it can run many other functions and interact with many system services. For example, when we execute the system call "open a file" that is hosted on a network file server, we are actually searching the filesystem tree by calling the VFS function, redirecting to the corresponding network file system driver, sending network requests by the network TCP/IP stack drivers through a network device connected to the file server's network, interacting with the server side of the network file system on the server, viewing our file on the server's local file system using its own VFS, and then accessing it through the hard disk device driver that placed this file.

And in fact, all this black magic is done with a single command, which may look like a virtual command to "open a file" on your processor. In general, an OS is basically just a set of interrupt handlers that implement such a set of virtual instructions specific to this OS and interact with the hardware. For example, as we will see in the future, with a system timer to switch the processor context while a multitasking system is running.

`https://people.cs.clemson.edu/~mark/interrupts.html`
`https://virtualirfan.com/history-of-interrupts`

Input/Output Redirection

The standard design pattern for UNIX commands is to read information from the standard input stream (by default — the keyboard of the current terminal), write to standard output (by default — terminal screen), and redirect errors to standard error stream (also the terminal screen), unless specified in the command parameters anything else. These defaults settings can be changed by the shell.

The command ends with a sign "greater than" (>) and the file name, means redirecting standard output to that file. The application code does not change, but the data it sends to the screen will be placed in this file.

And the command ends with a "less than" sign (<) and a file name, which means redirecting standard input to that file. All data that the application expects from the keyboard is read from the file.

Double "greater than" (>>) means appended to the output file.

Number two with "greater than" means redirecting standard error to a file. By default, stderr also prints to the screen and in this way we can separate this stream from stdin. And finally, such a magic formula:

```
prog 2>&1
```

This means stdout and stderr are combined into one stream. You may use it with other redirection, for example:

```
prog > file 2>&1
```

This means to redirect standard output to one "file", both standard output and standard error streams. But keep in mind — such combinations are not equivalent: `prog > file 2>&1` != `prog 2>&1 > file`

In the second case, you first concatenate the streams and then split again by redirecting stdout to the selected file. In this case, only the stdout file will be put into the file, stderr will be displayed on the screen. The order of the redirection operations is important!

So The question is: what are we missing in terms of symmetry? It's obvious — double "less than" sign <<).

And this combination also exists! But what can this combination mean? Append something to standard input? But this is nonsense. Actually this combination is used for the so-called "Here-document".

```
prog <<END_LABEL
.....
END_LABEL
```

After the double "less than" some label is placed (END_LABEL in our case) and all text from the next line to END_LABEL is sent to the program's standard input, as if from the keyboard.

Be careful, in some older shells this sequence of commands expects exactly what you wrote. And if you just wrote a space before END_LABEL for beauty, the shell will only wait for the same character string with a leading space. And if this line is not found, the redirection from "here document" continues to the end of file and may be the source of some unclear errors.

And finally, the pipelines. They are created with a pipe symbol "|" placed between commands. This means connecting the standard output from the first command to the standard input of the second command. After that, all the data that the first command by default sending to the screen will be sent to the pipe, from which the second command will be read as from the keyboard.

Programs designed in this redirection and pipelining paradigm are very easy to implement and test, but such powerful interprocess communication tools help us create very complex combinations of interacting programs. For example as such:

```
prog1 args1...<file1|prog2 args2...|...|progN argsN...>file2
```

The first program receives data from the file by redirecting stdin, sends the result of the work to the pipeline through stdout and after a long way through the chain of filters in the end the last command sends the results to stdout which is redirected to the result file.

Shell Settings

The shell is customizable.

As you will see, most UNIX commands have very short names – just two characters for the most common commands. This is because the developers wanted to shorten the printing time on TTYs, but are still very useful for the CLI work with nowadays. And we have a very useful tool for making shorter commands from long sentences — it's called aliases:

```
alias ll='ls -al'
alias
```

Et voila — now you only have a two letter command that runs the longer command.

And you can 'unalias' this:

```
unalias ll
alias
```

But after logging out of the shell session or restarting the system, all these settings and variable settings will be lost.

But you can put these settings in init files. These are common shell scripts where you may setup what you want:

```
/etc/profile - system defaults
```

Files for the first shell session that starts at login:

`Bourne shell`:	~/.profile
`Bash`:	~/.bash_profile
`C-Shell`:	~/.login

And initialization files for secondary shells:

`/etc/bashrc`:	system defaults
`Bash`:	~/.bashrc
`C-Shell`:	~/.cshrc

Keystrokes

A few words about keyboard shortcuts. They are actually very useful for command line work. Let's take a look at them:

erase	erase single character	[Ctrl]-[H] or [Ctrl]-[?] (or [Backspace], [Delete])
werase	erase word	[Ctrl]-[W]
kill	erase complete line	[Ctrl]-[U].
	This can be very useful when you enter something wrong on an invisible line, such as when entering a password.	
rprnt	renew the output	[Ctrl]-[R]
intr	Kill current process.	[Ctrl]-[C]
	In fact, these strange settings for the [Delete] key were used by some older UNIX. And many were very confused when, when trying to delete incorrectly entered characters, they killed the executable application.	
quit	Kill the current process with dump	[Ctrl]-[\]
	Kill the current process, but with a memory dump. Such a dump can be used to analyze the internal state of programs by the debugger. It can be created in the system automatically during a program crash, if you have configured your system accordingly, or like this – by [Ctrl]-[\] keystroke to analyze state, for example, a frozen program.	
stop	Stop a current process	[Ctrl]-[S]
start	Continue a previously paused process	[Ctrl]-[Q]
	Continue a previously paused process. And if the program seems to be frozen, first try pressing [Ctrl]-[Q] to resume the process. Perhaps you accidentally pressed [Ctrl]-[S].	
eof	End of file mark	[Ctrl]-[D]
	Can be used to complete input of something.	
susp	stops the current process and disconnects it from the current terminal line	[Ctrl]-[Z]
	As you probably know, this is the EOF mark on Windows systems. But on UNIX-like systems, it stops the current process and disconnects it from the current terminal line. After that, the execution of this process can be continued in the foreground or in the background.	

Keystrokes, KSH/Bash keyboard shortcuts

[ESC]-[ESC] or [Tab]: Auto-complete files and folder names.

This is very useful for dealing with UNIX-like file systems with very deep hierarchical nesting. As we will see later, three levels of nesting is a common place for such systems. Of course, we can use file management interfaces like graphical file managers or text file managers like Midnight Commander (mc), reminiscent of Norton Commander. But as we can see, in most cases, the autocompletion mechanism makes navigating the file system faster and can be easier if you know what you are looking for.

To use this machinery, you just need to start typing what you want (command name, file path or environment variable name), press [Tab] and the shell will try to help you complete the word. If it finds an unambiguous match, the shell will simply complete what it started. And if we have many variants, Shell will print them and wait for new characters to appear from us to unambiguously start the line. For example:

```
$ ec[tab]ho $TE[Tab]RM
xterm-256color
$ ls /u[tab]sr/l[Tab]
lib/ lib32/ lib64/ libexec/ libx32/ local/
o[Tab]cal/
bin etc games include lib man sbin share src
$
```

`[Ctrl]-[P]` — Go to the previous command on "history"
`[Ctrl]-[N]` — Go to the next command on "history"
`[Ctrl]-[F]` — Move cursor forward one symbol
`[Ctrl]-[B]` — Move cursor backward one symbol
`[Meta]-[F]` — Move cursor forward one word
`[Meta]-[B]` — Move cursor backward one word
`[Ctrl]-[A]` — Go to the beginning of the line
`[Ctrl]-[E]` — Go to the end of the line
`[Ctrl]-[L]` — Clears the Screen, similar to the "clear" command
`[Ctrl]-[R]` — Let's you search through previously used commands
`[Ctrl]-[K]` — Clear the line after the cursor

Looks more or less clear except for the Meta key. The Meta key was a modifier key on certain keyboards, for example Sun Microsystems keyboards.

And this key used in other programs – Emacs text editor for ex. On keyboards that lack a physical Meta key (common PC keyboard for ex.), its functionality may be invoked by other keys such as the Alt key or Escape. But keep in mind the main difference between such keys — the Alt key is also a key modifier and must be pressed at the same time as the modified key, but ESC generally is a real ASCII character (27/hex 0x1B/ oct 033) and is sent sequentially before the next key of the combination.

Another key point is that the origins of these key combinations are different. The second is just the defaults for those specific shell and can be changed using the shell settings. But the first one is the TTY driver settings. And if we want to change such keyboard shortcuts, for example, so that the Delete key does not interrupt the process, we can do this by asking the OS kernel to change the parameter of the corresponding driver. As we will see later, this can be done with the "stty" utility.