UNIX AND LINIX IN INFOCOMMUNICATION
Week 3

O. Sadov

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.

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 \colorbox{red}{!=} 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!

Under the hood — about streams numbers

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.

Under the hood — differences in redirection/pipelining in Windows

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.

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.