LINUX USER GUIDE (IV)

Working on the UNIX command line, part III:

Unix shell interpreters, part I – overview

Cristian Vidrașcu
cristian.vidrascu@info.uaic.ro

March, 2024

Overview

Introduction

On a UNIX system, the command interpreter is a program that performs the same tasks as in MS-DOS or Windows, providing two basic functionalities:

- it takes commands entered by the user, interprets them, and executes them, thus realizing the interface between the user and the operating system;
- provides programming facilities in a specific command language, with which *scripts* can be written, *i.e.* text files containing sequences of UNIX commands.

Remark: in operating systems from the UNIX family, we have several command interpreters (sometimes also called *shells*), such as: `sh` (*Bourne SHell*), `bash` (*Bourne Again SHell*), `csh` (*C SHell*), `ksh` (*Korn SHell*), `ash`, `zsh`, etc.

The main feature that differentiates the interpreters from each other is the command language syntax specific to each of them. In addition, the shells available on UNIX systems are more powerful than the command interpreters in MS-DOS and Windows (`command.com`, respectively `cmd.exe`), offering command languages similar to high-level programming languages in terms of syntax: have alternating and repetitive control structures (like `if`, `case`, `for`, `while`, etc.), which allows complex programs to be written as simple files with sequences of commands (*i.e.*, *scripts*).

We will next present the facilities common to all UNIX shells, which accomplish the first task mentioned above, with explicit references to the syntax used by the `bash` interpreter.

Outline

4 / 26

Basic commands – definition

The basic commands are the individual "components" that can be assembled into compound commands, using several *syntactic composition operators*, which will be described in the next section. But, to begin with, let's remember that in operating systems from the `UNIX` family there are two categories of *basic commands*:

- *Built-in commands*: those that are implemented in command interpreters.
  Examples: `cd`, `help`, etc.
- *External commands*: they are implemented independently (*i.e.*, they are each found in a file, having the same name as the respective command), in:
  - executable files (*i.e.*, executable programs obtained by compiling from source programs written in C or other programming languages).
    Examples: `passwd`, `ls`, etc.
  - text files with sequences of commands, called *scripts*.
    Examples: `.profile`, `.bashrc`, etc.

5 / 26

How to execute basic commands

The general form of launching into execution a simple command, built-in or external:

`UNIX>` *`command_name`* [*`options`*] [*`arguments`*] [*`I/O redirects`*]

Remarks:

- The "`UNIX>` " text is the prompt displayed by the shell, at which it waits for you to type the desired command, followed by pressing the `ENTER` key.
  Note: the text displayed as a prompt is configurable – we'll discuss how later.
- `SPACE` or `TAB` is used as a separator character between the words in the command line.
- Options and/or arguments specified after the command name may be missing (*i.e.*, they are optional, indicated by a pair of brackets '`[...]`'; these do not need to be typed).
- By convention, options are preceded by the '-' character (or '--', for long options).
- The meaning of the arguments depends on the command (*e.g.*, most often they are filenames).
- External commands can also be specified by the full name (*i.e.*, the *absolute or relative path*) of the respective file.
- A longer command (*i.e.*, with many parameters) can be entered on several lines, in which case each line must be terminated with the character '\' followed by `ENTER`, except for the last line (the termination it is done by pressing only the `ENTER` key).
- We will discuss what I/O redirects mean later on in this presentation.

6 / 26

How to execute basic commands (cont.)

Another possibility to launch a simple command, valid only for external commands that are *scripts*, is by calling a specific *shell*:

`UNIX>` `bash` *`script`* [*`options`*] [*`arguments`*] [*`I/O redirects`*]

In this case, the sequence of commands from the file named *script* will be executed, in a **non-interactive** session, by an instance of the shell invoked at the first position on the command line (in this example, the `bash` shell).

Remark: if no script is specified after the invoked shell name in the first position on the command line, a new **interactive** session will be created, controlled by an instance of the specified shell.

And a third possibility, also only for external commands that are scripts, is the following, with two equivalent syntactic forms:

`UNIX>` `.` *`script`* [*`options`*] [*`arguments`*] [*`I/O redirects`*]

`UNIX>` `source` *`script`* [*`options`*] [*`arguments`*] [*`I/O redirects`*]

7 / 26

Execution of basic commands in the background

In the case of the launch forms described on the previous slides, we say that that basic command is running in the *foreground* because the interpreter waits for that command to finish executing and only then redisplays the prompt, giving the user the opportunity to enter a new command to execute.

Another way of executing commands would be in the *background*, that is, the interpreter would not wait for that command to finish executing, but immediately redisplay the prompt, thus giving the user the opportunity to immediately enter a new command for execution.

Syntactically, specifying the execution of a command in the *background* is done by adding the character '`&`' at the end of the command line:

`UNIX>` *`command_name`* `[`*`options`*`]` `[`*`arguments`*`]` `[`*`I/O redirects`*`]` `&`

Note: for external commands that are scripts, '`&`' can be added to the end of any of the three forms of execution specified previously.

I/O redirects

There are three standard logical I/O devices:

- *standard input* (`stdin`), from which input data is read during the execution of a command (through the I/O functions accessing *file descriptor* `0`);
- *standard normal output* (`stdout`), to which output data is written during the execution of a command (through the I/O functions accessing *file descriptor* `1`);
- *standard error output* (`stderr`), to which error messages are written during the execution of a command (through the I/O functions accessing *file descriptor* `2`).

By default, the logical device `stdin` is attached to the physical device keyboard (*i.e.*, the input terminal), and the logical devices `stdout` and `stderr` are attached to the physical device screen (*i.e.*, the output terminal).

However, the command interpreter can "force" a command to receive input data from a specified file instead of reading it from the keyboard during its execution, and to send output data and/or error messages to a specified file instead of displaying them on the screen.

I/O redirects (cont.)

This "forcing" is called *redirecting* the respective I/O stream (or streams), and it is only valid for that execution of the command, and especially without having to make changes to the source code of the command and recompile it!

Specifying I/O redirects is done using the following syntax:

- *redirection of standard input* (`stdin`):
  `UNIX> command_name [parameters] < input_file`
- *redirection of standard normal output* (`stdout`), in *rewrite* vs. *append* mode:
  `UNIX> command_name [parameters] > output_file`
  `UNIX> command_name [parameters] >> output_file`
- *redirection of standard error output* (`stderr`), in *rewrite* vs. *append* mode:
  `UNIX> command_name [parameters] 2> err_output_file`
  `UNIX> command_name [parameters] 2>> err_output_file`

Here's an example, having the effect of concatenating the contents of the first two files into the third:
`UNIX> cat fis1 fis2 > fis3`

Multiple I/O redirects can be specified in a command, and their evaluation is done from left to right. In addition, there is also the syntax `n >&m`, where `n` and `m` are file descriptors. Here is an example:
`UNIX> ls -l .bashrc a_non-existent_file >listing.txt 2>&1`

Compound commands

Outline

Compound commands – definition

Two or more simple commands can be "grouped" into a *compound command* by writing them on a single line (at the command line prompt or in a script), separated by the following syntactic operators for "composing" simple commands:

- the ';' operator – the *sequential execution* of several commands
- the '|' operator – the *parallel, chained execution* of several simple commands
- the '&' operator – the *parallel, unchained execution* of several commands
- the '&&' and '||' operators – the *conditional execution* of commands

The semantics (*i.e.*, the meaning) of each of these composition operators is briefly described below.
Warning: the description focuses on the syntax used by the `bash` shell; in other `UNIX` shells, the syntax of compound commands may differ (*i.e.*, to use other "composition" operators, etc.).
Note: for a detailed description, I recommend consulting the official `bash` shell documentation, available here.

12 / 26

Pipelines

The chaining of several simple commands, in a so-called *chain of commands* (or *pipeline*), is done using the symbol '|' (*i.e.*, the *pipe* character), as follows:
`UNIX>` `[time]` *command_1* | *command_2* | ... | *command_N*

The symbol '|' marks the "connection" of the standard normal output of a command to the standard input of the next command in the pipeline; the communication between the two commands is done through an anonymous communication channel (in this way, the need to communicate via temporary files, as happens in `MS-DOS` or `Windows`, is eliminated).
The `time` keyword has the effect of displaying the execution times of the pipeline.
Note: $N = 1$ is also allowed, *i.e.* a pipeline consisting of a single simple command.

The execution of a pipeline :
All the simple commands in that pipeline are executed simultaneously, in the "same" time (*i.e.*, in parallel, not sequentially one after the other !), each command being executed by a new shell instance. Basically, multiple processes are created, one process for each command in the pipeline.

13 / 26

Pipelines (cont.)

Here are some examples of pipelines:
UNIX> `ls -Al | wc -l`
Outcome: displays the total number of files of any type (*i.e.*, including subdirectories) existing in the current directory.
UNIX> `who | cut -f1 -d" " | sort -u`
Outcome: displays the ordered list of usernames of those logged into the system.
UNIX> `cat /etc/passwd | grep -w so`
Outcome: ?

Demo: see all examples of pipelines in the lab support, available here. The first example also describes the effect of using the `time` keyword.

Extended syntax: instead of any symbol '`|`', in a pipeline with $N \geq 2$, one can use the pair '`|&`', which is an abbreviation for '`2>&1 |`'. This "connects" both standard outputs (the normal one and the error one) of a command to the standard input of the next command in the pipeline.

Sequential execution of several commands

Several commands (simple or pipelines with $N \geq 2$) can be separated by the character '`;`' and will be executed sequentially (*i.e.*, one after the other, in the order in which they appear in command line), by the same instance of that shell.
The syntax used (where each *command_i* is a pipeline of simple commands) :
UNIX> *command_1 ; command_2 ; ... ; command_N*

Equivalently, the N commands can be written each on one line, at the prompter or in a script, like this:
command_1
command_2
...
command_N

Here are two examples:
UNIX> `ls -A ; cd Desktop ; ls -l`
UNIX> `mkdir d1 ; echo "Hello!" > d1/f1.txt ; cd d1 ; stat f1.txt`
The outcome of these commands : ?

Parallel, unchained execution of several commands

Several commands (simple or pipelines with $N \geq 2$) can also be separated by the character '&' and will be executed (almost) simultaneously, being launched in *background* execution (*i.e.*, without waiting for each one to finish in turn and without having the standard I/O streams "chained" via *pipe* as with pipelines). Basically, each command is executed by a new shell instance and runs independently of the others (*i.e.*, without "chaining" between them).
The syntax used (where each *command_i* is a pipeline of simple commands) :
UNIX> *command_1* & *command_2* & . . . & *command_N* [&]

The last command will be run either in the background (if it is terminated with the character '&') or in the foreground (otherwise).

Equivalently, the N commands can be written each on one line, at the prompter (but thus their starting "phase lag" will increase, depending on the speed at which you type them) or in a script, like this:
command_1 &
command_2 &
. . .
command_N [&]

Here is an example:
UNIX> cat /etc/passwd & cat /etc/group &
Outcome: ?

16 / 26

Conditional execution of several commands

The execution of a command may depend on the result of the execution of another.
The syntax used (where *command_1* and *command_2* are pipelines) :

- the '&&' operator – *the conjunction of two commands*:
  UNIX> *command_1* && *command_2*
  The execution of a conjunction: first the first command is executed and then the second command will be executed only if the execution of the first command *completes successfully* (*i.e.*, if *command_1* returns exit code 0).
- the '||' operator – *the disjunction of two commands*:
  UNIX> *command_1* || *command_2*
  The execution of a disjunction: first the first command is executed and then the second command will be executed only if the execution of the first command *ends with an error* (*i.e.*, if *command_1* returns a non-zero exit code).

Remarks: i) One can see the analogy with the short-circuited evaluation of boolean logical expressions.
ii) Sequences consisting of more than two commands separated by '&&' and '||' are also allowed. Since these operators have the same precedence, the associativity from left to right will be applied.

17 / 26

Extended syntax: command lists and compound commands

A *list of commands* is a sequence of one or more pipelines, separated from each other by the operators ';', '&', '&&', or '||', and optionally terminated with one of the characters ';', '&', or <newline>.

The *execution* of a list of commands: the sequence of pipelines is evaluated from left to right, taking into account the order given by the precedence of the operators: '&&' and '||' have the same precedence, higher than that of the operators ';' and '&' which also have the same precedence. Then each pipeline in the list is executed, in the order thus established, applying for each operator its meaning.

Two of the types of *compound commands* are as follows: `(list)` and `{ list; }`.
Note: for a more detailed description of pipelines, command lists and compound commands, I recommend consulting the documentation available here.

Here are some examples:

```
UNIX> cd ; ls -l .bashrc || echo error ; cat non-existent-file && echo ok
UNIX> var=outside ; ps -f ; (ps -f ; var=inside) ; echo $var
UNIX> var=outside ; ps -f ; { ps -f ; var=inside; } ; echo $var
```

The outcome of these commands: ?

Specifying file names

Outline

Specifying individual files

Specifying the name of some file, either as an argument to commands or as the name of an external command, it can be done in three different ways:

- by *absolute path* (*i.e.*, the full name, starting from the root directory)
  Example: `/home/vidrascu/so/file0003.txt`
- by *relative path* to the current working directory (*i.e.*, starting from that directory)
  Example (assuming the current working directory is `/home/vidrascu`) : `so/file0003.txt`
- by *the path relative to a given user's home directory* (*i.e.*, starting from its home directory)
  Example: `~vidrascu/so/file0003.txt`
  Remarks: i) if the username is missing (*e.g.*, `~/so/file0003.txt`), then the current user's home directory is assumed;
  ii) this third mode of specification can only be used on the command line or in scripts, but NOT as arguments to functions called in C programs.

20 / 26

Patterns for multiple file specification

A list of files can be specified as arguments to commands using a single multi-specifier "template", by using the following *special characters*:

- the '*' character : is replaced by any string, including the empty string
  Example: `~vidrascu/so/file*.txt`
- the '?' character : is replaced by any character (exactly one character)
  Example: `~vidrascu/so/file000?.txt`
- the "specified character set" specifier [...] : is replaced by exactly one character, but not by every possible character, but only by those specified between the brackets '[' and ']', in the form of enumeration (separated by ',' or nothing) and/or range (given through the ends of the range, separated by '-')
  Examples: `~vidrascu/so/file000[1,3,579].txt`,
  `~vidrascu/so/file00[0-9][3-9].txt`,
  `~vidrascu/so/file000[1-3,57-9].txt`.

21 / 26

Patterns for multiple file specification (cont.)

(*cont.*)

- the "excluded character set" specifier [^...] : is replaced by exactly one character, but not by any possible character, but only by those in the complement of the set specified between the brackets '[' and ']' similar to above, except that the first character after '[' must be '^' to indicate complementarity
  Example: ~vidrascu/so/file000[^1-3].txt
- the '\' character : is used to inhibit operator interpretation of previous special characters, namely \c (where c is one of the characters '*', '?', '[', ']', '^', '\') will interpret that c character as regular text (*i.e.*, by itself) and not as an operator (*i.e.*, by the "template" associated with it as described above)
  Examples: ce_mai_faci\?.txt , lectie\[lesson].txt.

Important: each such pattern is replaced, in the position on the command line in which it appears, with the list of all existing filenames satisfying that pattern, but only if that list is non-empty. Otherwise, the template remains unchanged after its interpretation.

Note: also *negated templates* can be specified. An example would be: `ls !(*.sh)`

Shell configuration files

Outline

Initializing the shell's interactive sessions

Command interpreters in UNIX can use certain files that store various commands for configuring the interactive sessions of running the respective shell, the names of these files being specific to it.
Thus, in the case of the bash shell, the following files are used ([3]) :

- *initialization files* for bash login sessions :
  - a global script, executed for all users : /etc/profile
  - local scripts, specific to the respective user : ∼/.bash_profile, ∼/.bash_login or ∼/.profile
- *termination files* for bash login sessions – a local script only : ∼/.bash_logout
- *initialization files* for interactive bash sessions, except for login ones :
  - a local script, specific to the respective user : ∼/.bashrc
  - a global script, executed for all users : /etc/bashrc (but it must be called explicitly from the local script)

24 / 26

The history of executed commands

- *history file* of commands entered and executed at the bash command line – a local file named ∼/.bash_history

The contents of this file can be (also) viewed with the command history.
The numbers displayed by this command can be used to repeatedly execute the commands in the history without having to be re-typed. Namely, to invoke a command again, the character '!' is typed at the command line prompt, immediately followed by the number associated with the respective command.

Another useful feature of the bash interpreter, which allows previously typed commands to be invoked, as well as their editing, is to navigate through the history by pressing the UP and DOWN arrow keys at the prompt. After selecting from the history the desired command line, we can move within it with the left-arrow and right-arrow keys to partially modify that line (*e.g.*, to modify, delete or add some parameter of the command) and then press the ENTER key to execute the command thus modified.

25 / 26

Mandatory bibliography

[1] Chapter 2, §2.3 from the book "*Sisteme de operare – manual pentru ID*", by C. Vidrașcu, UAIC Publishing House, 2006. This is available as ebook at the address:

- https://profs.info.uaic.ro/~vidrascu/SO/books/ManualID-SO.pdf

[2] The online support lesson associated with this presentation:

- https://profs.info.uaic.ro/~vidrascu/SO/support-lessons/bash/en/support_lab3.html

Additional bibliography:

[3] The documentation of `bash` shell : `man 1 bash` and "GNU Bash manual"

[4] *Linux Documentation Project Guides* → "Bash Guide for Beginners"

[5] The book "Bash Pocket Reference" (1st edition), by A.Robbins, O'Reilly Media Inc., 2010.