UNIX AND LINIX IN INFOCOMMUNICATION
Week 3

O. Sadov

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.

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.

Utilities

All commands typed on the command line or executed in a command file are either commands built into the interpreter or external executable files. The set of built-in commands is quite small, which is determined by the basic concept of UNIX — the system should consist of small programs that perform fairly simple well-defined functions that communicate with each other via a standard interface.

A rich set of standard utilities is a good old tradition for UNIX-like systems. The shell and the traditional set of UNIX utilities, is a POSIX standard.

As we discussed earlier, we have different branches of development of UNIX-like systems with different types of utilities:

- BSD-like dating back to the original UNIX implementations;
- SYSV based systems;
- GNU utilities.

Some command syntax was changed by the USL with the introduction of SYSV, but on most commercial UNIX a set of older commands was still

included for compatibility with earlier BSD-based versions of UNIX from the same vendor. GNU utilities often combine both syntaxes and add their own enhancements to traditional utilities. And now the GNU toolkit has become the de facto standard.

Executable files on UNIX-like systems do not have any file name extension requirements as they do on Windows. The utility executable can have any name, but must have execute permission for the user who wants to run it.

A standard utility can have options, argument of options, and operands. Command line arguments of programs are mainly parsed by the getopt() function, which actually determines the form of the parameters when the command is invoked. This is an example of utility's synopsys description:

```
utility_name[-a][-b][-c option_argument] \
                        [-d|-e][-f[option_argument]][operand...]
```

1. The utility in the example is named `utility_name`. It is followed by options, option-arguments, and operands. The arguments that consist of <hyphen-minus> characters ('-') and single letters or digits, such as 'a', are known as "options" (or, historically, "flags"). Certain options are followed by an "option-argument", as shown with [-c option_argument]. The arguments following the last options and option-arguments are named "operands".

   The GNU `getopt()` function supports so-called long parameters, which start with two dashes and can use the full or abbreviated parameter name:

   ```
   utility_name --help
   ```

2. Option-arguments are shown separated from their options by <blank> characters, except when the option-argument is enclosed in the '[' and ']' notation to indicate that it is optional.

   In GNU getopt's long options also may be used the 'equal' sign between option and option-argument:

   ```
   utility_name --option argument --option=argument
   ```

3. When a utility has only a few permissible options, they are sometimes shown individually, as in the example. Utilities with many flags generally show all of the individual flags (that do not take option-arguments) grouped, as in:

```
utility_name [-abcDxyz][-p arg][operand]
```

   Utilities with very complex arguments may be shown as follows:

```
utility_name [options][operands]
```

4. Arguments or option-arguments enclosed in the '[' and ']' notation are optional and can be omitted. Conforming applications shall not include the '[' and ']' symbols in data submitted to the utility.

5. Arguments separated by the '|' (<vertical-line>) bar notation are mutually-exclusive.

```
utility_name [-a|b] [operand...]
```

   Alternatively, mutually-exclusive options and operands may be listed with multiple synopsis lines. For example:

```
utility_name [-a] [-b] [operand...]
utility_name [-a] [-c option_argument] [operand...]
```

6. Ellipses ("...") are used to denote that one or more occurrences of an operand are allowed.

System manuals

The easiest way to get information about the use of a command is with the -h option or --help for GNU long options.

Also since its inception, UNIX has come with an extensive set of documentation. Some information is often found in the `/usr/doc` or `/usr/local/doc` or /usr/share/doc directories as text files.

But the cornerstone of the Unix help system is the man command. And the man in this case is not about gender — it is just an abbreviation for manual.

```
man man
```

The man command has been traditional on UNIX since its inception, was created in the teletype era and still works great on all types of equipment. And in the synopsis of the man command, we see two worlds, two UNIX utility systems: BSD and SYSV.

And at first we can see the different command syntaxes:

```
SYSV~--- \cmd{man [-t] [-s section] name}
GNU, BSD~--- \cmd{man [-t] [section] name}
```

Parts of the man page are more or less the same:
NAME, SYNOPSIS, DESCRIPTIONS, FILE, SEE ALSO, DIAGNOSTIC, BUGS

The minus S option with some integer parameter of man command in the SYSV variant denotes a section of real paper manuals supplied with the OS by vendor. For GNU/BSD flavors, use only the section number. Section numbers are also different.

Let's look for example to well known C-languge function 'printf' manual page. But

```
man printf
```

"man printf" shows us the man page for the shell command, not the C function. To see the manual for the C printf function, we must run:

```
man 3 printf
```

To view a list of printf-related manual pages, we must run:
whatis — search the whatis database (created by makewhatis) for complete words
and
'man -k' or 'apropos' — search the whatis database for strings.

The databases should be created by the 'makewhatis' program, which is usually started at night by the cron service. If you have a freshly installed system and want to run any command related to the whatis database, you posibly need to start the makewhatis program manually.

OK — let's look to real man page:

```
zless /usr/share/man/man1/man.1.gz
```

Troff(short for "typesetter roff")/nroff(newer "roff") is an implementation of a text formatting program, traditional for UNIX systems, using a plain text file with markup. It is ideologically based on the RUNOFF MIT program, developed in 1964, and after a series of source code losses and rewrites, a C-based implementation was re-implemented in 1975. Under the name Troff, it was accepted for use on the UNIX system and, of course, into the AT&T patent department.

See – http://manpages.bsd.lv/history.html

The main advantage of this tool was portability and the ability to generate printouts for various devices, from a common ASCII printer to high-quality typographic photosetters. Creating new technologies such as PostScript printers simply adds the appropriate output drivers for the markup renderer. Compared to the now better known WYSIWYG systems, such systems have better portability between platforms and higher typing quality. Moreover, such systems are more focused on the programmatic creation of printed documents without human intervention.

Let's take a look at an example of manually rendering the man page that was hidden under the hood of the man command:

```
$ zcat /usr/share/man/man1/man.1.gz | tbl | eqn -Tascii | \
                                              nroff -man | less
```

It's just a software pipeline in which we unpack the compressed TROFF source, go through the TROFF preprocessor for tables and math equations, pass a troff variant named 'nroff' to output a text terminal, and finally pass a text pager/viewer named 'less'.

What is it viewer? In the TTY interface, the man command seems like a good one — when you run it, you get paper manuals that you can combine into a book, put on a shelf, and reread as needed. On a full-screen terminal — before, Ctrl-S(stop)/Ctrl-Q(repeat) was enough for viewing, because at first the terminals were connected at low speed (9600 bits per second for ex.), and now special programs were used — viewers.

And in fact, when you run the "man" command, you see just a viewer's

interface, in most cases it is "`less`". We will discuss viewers further, but in a nutshell, the most commonly used of them the '`less`' handles the UP/DOWN keys normally, exiting a program — the 'q' key means 'quit'.

Well. It looks great, but the man-style documentation representation has certain limitations: it is only a textual representation without any useful functions invented after that time — for example, impossibility of using hypertext links.

To improve it, the GNU Project has created an information system that also works on all types of alphanumeric terminals, but with hypertext support. For many GNU utilities, the corresponding help files are in info format, and the man pages recommend that you refer to `info`.

Info has its own user interface and the best way to learn it is to simply run the '`info info`' command. The internal source of "info" is the text markup files in texinfo format. From such files are generated text files for viewing on terminals, and also by the TEX typesetting system generated documentation for printing.

TEX is another typesetting system (or "formatting system") that was developed and mostly written by Donald Knuth, released in 1978. TEX and its LATEX extension are very popular in the scientific world as a means of typing complex mathematical formulas.

OK. But the inability to use graphic illustrations and any kind of multimedia context remained relevant. And in the past, almost every commercial UNIX system vendor created their own help system, which includes both hypertext support and graphics for ex., and worked in the X Window System.

But now with the advent of HTML (yet another text markup language), such reference information began to be provided in this format directly on the system or on WWW servers. Often these HTML pages or print-ready PDF versions are simply generated from some content oriented markup such as DocBook XML.