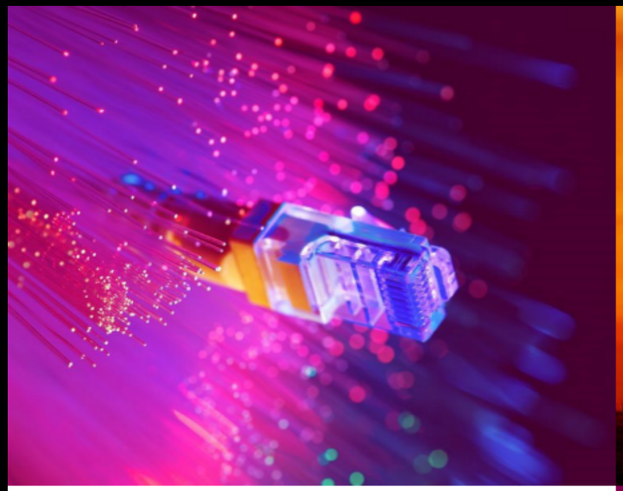

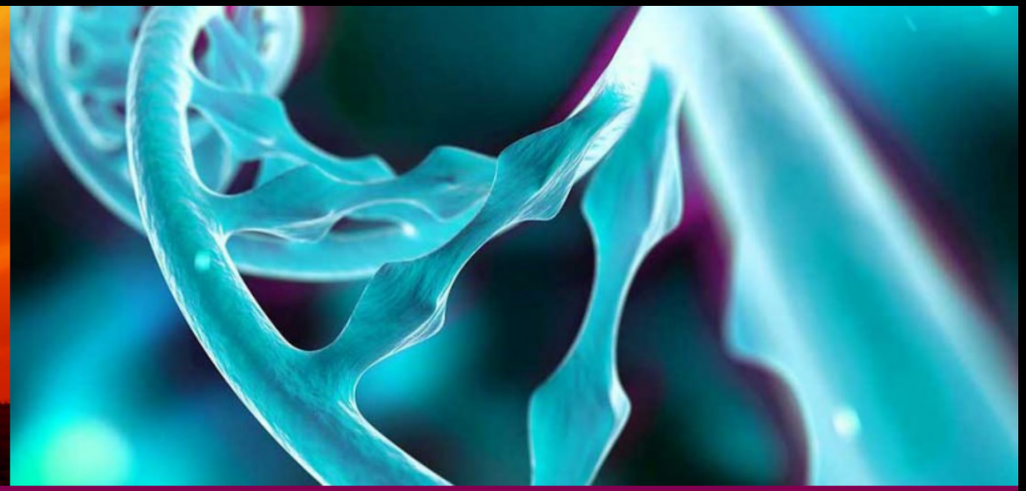

Working with bash — shell internals

CSC Training, 2019-12

CSC – Finnish expertise in ICT for research, education and public administration

What is a shell?

- At its base, a shell is simply a macro processor that executes commands.
- A Unix shell is both a command interpreter and a programming language.
  - As a command interpreter, the shell provides the user interface to the rich set of utilities.
  - The programming language features allow these utilities to be combined.
- Shells may be used interactively or non-interactively. In interactive mode, they accept input typed from the keyboard. When executing non-interactively, shells execute commands read from a file.
- There are a multitude of different shells available.
  - Bourne Again shell — bash — is probably dominant.

Shell operation — a seven-stroke cycle engine

- When the shell reads input, it proceeds through a sequence of operations. Roughly speaking, the shell:
  - reads input and divides it into words and operators; then
  - parses these tokens into commands and other constructs, removes the special meaning of certain words or characters, expands others, redirects input and output as needed; and finally
  - executes the specified command, waits for the command’s exit status, and makes that exit status available for further inspection or processing.
- A command is just a sequence of words; the first word generally specifies a command to be executed, with the rest of the words being that command’s arguments.

Readline — bash bare essentials

- Readline is the library that handles reading input when using an interactive shell.
  - By default, the line editing commands are similar to those of emacs. A vi-style editing interface is also available.
  - Readline can be configured to taste with `~/.inputrc` file.
- Move to the start/end of the line: `C-a` / `C-e`.
- Move a word forward/backward: `M-f` / `M-b`.
- Kill rest of the line: `C-k`.
  - Killed text is copied into *yank buffer* and can be inserted back with `C-y`.
- To clear the screen while retaining the current line, type `C-l`.

Prompting — the shell expects something from you

- When executing interactively, bash displays the primary prompt PS1 when it is ready to read a command, and the secondary prompt PS2 when it needs more input to complete a command.
  - Bash allows these prompt strings to be customized by inserting a number of backslash-escaped special characters:

```
$ PS1="(\!) \h:\W\$ "
```

\! = history number of this command
\h = hostname up to the first .
\W = the basename of current working directory
\$ = if effective UID is 0, a #, otherwise a $

Completing — avoid typing (and errors)

- Pressing the `TAB` key attempts to perform completion on the text before cursor.
- Bash attempts completion treating the text as:
  - a variable (if the text begins with `$`),
  - username (if the text begins with `~`),
  - hostname (if the text begins with `@`), or
  - a command (including aliases and functions) from `$PATH`.
  - If none of these produces a match, filename completion is attempted.
- To just see possible completions hit `M-?` and to insert all possible completions hit `M-*`.

History — more ways to avoid typing

- The shell provides access to the command history, the list of commands previously typed. The value of `HISTSIZE` variable is used as the number of commands to save in a history list.
  - On startup, the history is initialized from the file named by the variable `HISTFILE`, and when shell exits the last `$HISTSIZE` lines are saved to that file.
- To access the history you use `UP` and `DOWN` keys.
- You can also search the history by pressing `C-r`.
- To view the complete history, use `history` command.

History expansion — avoid typing with a twist

- History expansions introduce words from the history list, making it easy to repeat commands, or fix errors in previous commands quickly.
  - `!n` refer to command line *n*.
  - `!-n` refer to the current command line minus *n*.
  - `!!` refer to previous command.
  - `!string` refer to the most recent command starting with *string*.
  - `!?string?` refer to the most recent command containing *string*.
  - `^string1^string2^` Repeat the last command, replacing *string1* with *string2*.

```
$ for i in *; do echi $i; done
$ ^echi^echo^
```

Quoting — Take it literally

- Quoting is used to remove the special meaning of certain characters or words to the shell.
- There are three quoting mechanisms: the escape character `\`, single quotes `''`, and double quotes `""`.
  - The escape character preserves the literal value of the next character.
  - Single quotes preserve the literal value of each character within the quotes.
  - Double quotes preserve the literal value of each character within the quotes, with the exception of `$`, `` ` ``, and `\`.

```
$ echo \$HOME
$ echo '$HOME'
$ echo "I'm \$HOME"
```

Simple command expansion — I press enter and…?

- When a command is executed, the shell performs the following actions, from left to right:
  1. Variable assignments and redirections are stripped away and saved for later processing.
  2. Any words left are *expanded*. If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.
  3. Redirections are performed.
  4. Variable assignments undergo expansions and quote removal.
  5. Command is executed.

```
$ LANG=fi_FI.utf8 ls -ld ~/*
```

Expansion — Making a short story long

- Expansion is a process of exchanging a word with one (or more) another word, with certain rules.
- There are seven kinds of expansions, performed in the following order:
  1. Brace expansion
  2. Tilde expansion
  3. Parameter and variable expansion
  4. Arithmetic expansion
  5. Command substitution
  6. Word splitting
  7. Pathname expansion

Brace expansion — Make strings happen

- Brace expansion is a mechanism by which arbitrary strings may be generated. It is of form `[preamble]{str[,str,…]}[postscript]`.

```
$ mkdir -p exp/{jan,feb,mar,apr}/run{1..3}-data
```

- Brace expansions may be nested.

```
$ touch exp/{{jan,mar},feb/{1,2}}/skip
```

Tilde expansion — A way to home

- Tilde expansion (almost) always expands to (some) user's home directory `$HOME`.

```
$ echo ~
$ echo ~root
```

- Tilde expansion can also be used to refer to current working directory `$PWD`, or previous working directory `$OLDPWD`.

```
$ cd ~/Documents
$ cd /tmp
$ echo ~
$ echo ~+
$ cd ~-
```

Filename expansion — the wildcards

- The shell scans the command line for the characters `*`, `?`, and `[`. If one of these is found, the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of file names matching the pattern.
  - `*` matches any string, including an empty string.
  - `?` matches any single character.
  - `[…]` matches any one of the enclosed characters. If the first character following the `[` is `!` or `^` then any charater not enclosed is matched.

```
$ ls -d ?e*
$ ls -d *[cw]*
$ echo {/usr,}/bin/t[!r]*
```

Shell builtin commands — some of them

`bg [jobspec]`

`cd [dir]`

`echo [args]`

`export [name]`

`fg [jobspec]`

`history`

`jobs [jobspec]`

`kill [pid | jobspec]`

`pwd`

`unalias [name]`

`unset [name]`

`times`
: Show accumulated user and system times for the shell.

`type [name]`
: Indicate how each *name* would be interpreted.

`ulimit [limit]`
: Control the resources available to the shell.

`umask [mode]`
: The user file-creating mask is set to *mode*.

Signals — there’s a trap

- Signals are asynchronous notifications that are sent to processes when certain events occur.
- `trap` allows you to catch signals and execute code when they occur.
  - Option `-l` lists all available signals, and option `-p` lists all signals currently trapped.

```
$ trap "echo kukkuu" SIGUSR1
```

```
# Run something important, no Ctrl-C allowed.
trap "" SIGINT
important_command

# Less important stuff from here on out, Ctrl-C allowed.
trap - SIGINT
not_so_important_command
```