Bash Command Line:

Sean al-Baroudi

October 19, 2017

Basics of Bash Shell:

Beginnings:

- **A shell** is a macro processor that ultimately runs commands to achieve some user end.
- **Macro:** An expandable set of expressions via text substitution; used for referencing, reuse and modularizing sets of commands.
- Each program has an input, output and error stream (stdin - 0, stdout - 1, stderr - 2). Each command can be run in **sequence or parallel**, and can **communicate or run independently.** or in parallel (asynchronously or not). For every BASH expression that is typed into command line, an execution tree is set up.
- **Terminology:**
  - Field: a unit of text that is the result of a shell expansion.
  - Job: A set of processes chained together in a pipeline.
  - Pipeline: A set of processes in sequence, in which output from one is fed into the next one.
  - Operator: Control or Redirection types exist.
  - Signal: An interrupt mechanism in which the kernel communicates with a process.
  - Token: A parser construct; word or operator
  - Word: A basic unit of characters that represents some object (operator, input, command, name....etc).
  -
- **Shell Operational Loop:**
  1. Read input from file.
  2. Break input into words and ops. Use quoting to impose meaning.
  3. Tokens are parsed into sets of commands.
  4. Perform macro/shell expansions (paths, links, etc).
  5. Set up redirections and pipelining (if any)
  6. Run Exec on the chain of commands.
  7. Wait for other branched shell processes to finish, gather return codes and return result to stdout/stderr.

Objects in Shell Environment:

Variables, System and User Parameters:

- A variable is a named parameter.
- A parameter is some entity that represents information.
- Variables are set by assigning:

  $$var =< somevalue >$$

  $$\$somevalue$$

  Variables are dereferenced and then string substituted into expressions by using the $ operator in front.
- You can *unset* variables using the said keyword.
- Varables initialized with no value are **given the null string as a value, by default.**
- Positional Parameters: Are assigned to shell arguements that trail a command. For every expression fed into a shell, the arguments behind a command will be numbered. They are referenced by number:

  $$\$0, 1, 2...$$

  Much like the argv construct of C.
- Arrays: Typical notation, constructed with:

  $$varname = [1, 2, 3, 4...]$$

- Special Parameters: These can be dereferenced (and substituted)., but never set.
  - $\$*$: Expands all positional parameters.
  - $\$-$: Gives a list of all flags provided to the last command.
  - $0: the name of the last command.
  - $?: the exit status of the last terminated command.
  - $$: Expand the PID of a shell.
  - $_ Expands absolute pathname used to invoke shell.

Shell Syntax, Quoting:

- Typical syntax for a command is as follows:

  $$cmd < options >< arguements > [pipeorredirect]$$

- Parsed items are either comments, or tokens in which meaning is imposed. Tokens can be operators, reserved words, arguements or metacharacters. The syntax is bracket
- Quoting is important, as it narrows the meaning of what is enclosed. and directionally sensetive (order matters).
- **Quoting Types:**
  1. **Back Quote ‘:** Used for command substitution. Use $(*cmd*) instead.
  2. **Escape Character:** A backslash to escape metadata characters.
  3. **Single Quote (’):** Preserves literal value of each character between the quotes. Cannot escape single quote in a single quote pair!
  4. **Double Quotes (”):** Some metadata characters still maintain special meanings; most characters are still considered literally. Special characters that are excepted include:

     $$\$ \text{ ‘ } \backslash$$

     Backslash is excepted for some metadata charcters like quotes or newline.
  5. Comment (#): obvious.

Shell Expansions:

- Brace Expansion: Braces {...} are used to represent a sequence (number or letter). They can be prefixed and suffixed with strings and numbers on either side. Some examples of usage:

  $$prefix\{x..y..jump\}suffix$$

  $$prefix\{A..F\}\{A..F\}\{1..9\}suffix$$

  Brace expansions can’t be done for variables; as dereference ($) is done after brace expansion.
  Braces are also used to separate expanded entities, or use the multiplication rule for iterating name possibilities.
- Tilde Expansion (”∼”): Represents user’s home directory.
- Shell Parameter Expansion ($): **A critically important operator (!).** Used for **parameter expansion (dereferencing) , command substitution or arithmetic expression.**
- For parameter expansion, we may operate and iterate on sets of things, rather than just dereference a variable name. Generally, the {} brackets are used to enclose such expressions.
- ***Parameter Tests and Cutting:*** This is done with the colon (:) modifier. Some usages are below:

  $$\{param : -word\}$$

  If param is unset, substitute word instead.

  $$\{param := word\}$$

  If param is unset, set it equal to word (used for defaulting)

  $$\{param :?word\}$$

  If param not set, write word to stderr and stdin (used for warnings).

$$\{string : 1 : 5\}$$

Cut out substring in positions 1 to 5

$$\{param : 4 : -4\}$$

Cut out substring from 4 in, to 4 from the end.

$$\{!prefix*\}$$

Return a list of every prefix that matchs the above pattern (*) can be anything aftewards.

- **Command Substitutions $(command):** Instead of the command pumping output to stdout or err and returning an exit code in line the expression, return the stdout in place of the command name.
- **Arithmetic Expressions:** Integer arithmetic can be performed, and then substituted in place using the following syntax:

$$\$((expr))$$

-

Shell Commands:

- A Command involves a program, its arguements, and support elements the shell provides it, such as input and output streams.
- In general, all commands give an exit code, and this is used to determine the final output value of the command expression that was fed into the shell.
- **Pipelines:** Are used when the stdout/err of one command is fed into the stdin of another command; they are a form of redirection.
- (Anonymous) Pipe (cmd1 | cmd2): This causes cmd1 stdout → cmd2 stdin.
- Other Pipe (cmd1 | & cmd2): This redirects stdout and err as well.
- Pipes are considered to be **anonymous** (the file buffers they are represented by generally are not searchable).
- **Lists:** In general, command line expressions are structured as a series of pipelines connected by operators. These together form a **list.**
- Semi-colon (cmd1 ; cmd2): finish the first command, and execute the second after the first one has completed. Ignore the return code.
- Ampersand (cmd1 & cmd2): Run cmd1 in the background (separate shell process), and then run command2 in the current user shell. The results of cmd1 still print to users current shell.
- AND (cmd1 && cmd2): if cmd1 is successful, run cmd2. Else, short circuit.
- XOR (cmd1 || cmd2) If cmd1 fails, cmd2 will be executed
-

1 More Advanced Concepts:

- **File System Objects:** Everything is a file. Even directories. Metadata at the front of each file allows for types and distinctions. In the end, a file system boils down to blobs of data on a hard disk with header metadata, that "represents" our filesystem. The different flavours of "files" are below, and constitute the working objects of a file system:
  - Block Device (buff): This represents a device driver, in which information streams from it. It of course appears as a file. A buffer is a file that stores a block of information, which is read by the system. Information may stay here for extended periods of time until it is sent/recieved.
  - Character Device (nobuff): Allows the shell/os direct access to read/write from a hardware device. Blocks of information are not stored in a buffer to be written later. There may be latency if the device is busy (Block Alignment).
  - Directory: A structure that links files together in a user defined hiearchy.
  - Regular File: A file that stores information, with no special features.
  - Symbolic Link:
  - Pipe (Unnamed): A pipe is an InterProcess Communication tool, that allows processes running to send information to one another. An unnamed pipe is a file that lasts as long as the sending process is running. It is then deleted.
  - Pipe (Named): These pipes lay around the operating system after the sending process has terminated.
  - Socket: A pipe between processes on different machines. Note: you can use a socket between two processes on the same machine; its a more global concept that incorporates the OSI model of communication (Internet).
  - Door: ?
- **Weak Quote ":** This restricts the interpretation of what is encapsulated by the quote. Expressions ($\$EXP$) are still substituted.
- **Strong Quote ':** The interpretation is purely literal (no special meaning or substitutions).
- **Quote/Escape a Character:** Use the \ symbol for this.
- **File Permission Types:** Each file has a owner (user), and two groups it is associated with (group and other). This allows for more fine grained control over how the file is accessed. Standard Usage: a user will own the file, a small group of people will be given special access, and "others" will have restricted access. Note: Each user has their own special group (which is often just uid=gid as created by the admin. Also, gid=0 represents the admin of the server).
- Keyboard Signals:
-

Redirection:

- For each running program in BASH, three default file descriptors are provided (0,1,2). Others can be set up if necessary.
- Redirection involves changing the data streams to go to non-standard places.
- **Basic Operators:**
- **Pipe (cmd1 | cmd2):** Maps the stdout of cmd1 to the stdin of cmd2.
- Input Redirect (cmd1 $N <$ file): redirects the data stream N of cmd1 from a file. If N is not present, N=0 and stdin is assumed.
- Output Redirect (cmd1 $N >$ file): redirects the datastream N of cmd1 into a file. File is created if it does not exist. If N is not present, N=1 and stdout is assumed.
- **Doubled/Specific Operators:**
- ReadWrite to File (cmd <> file) cmd stdout sent to file, that is open for both reading and writing.
- Appendto (cmd >> file): cmd stdout written to a file in which we append.
- HereDocument (cmd << EndWord): this opens up an input line and pumps in text into cmd stdin buffer, until EndWord (on its own, with no spaces) is detected in the user input stream.
- Duplicate File Descriptor (cmd m> &n): the path of *file descriptor m is overwritten with the path of file descriptor n.*
- Create fd (for bash to pass on) (exec N> file).
- Close a fd (for bash) (exec N> &−)
- **Syntax for Changing the streams of a Single Command:**

$$\langle cmd--Options\rangle\ \ 0<\ \ filein\ \ 1>\ \ fileout\ \ 2>errfile$$

  This is the longform to redirect every default stream for a given command. The angle brackets are present only for delimitation. Note that the 0 and 1 are redundant here.
- **Example: Swap file descriptors 0 and 2:**

```
exec 3<> somefile
cmd 3>&2 2>&0 1>&3
```

  You must use exec to set up a new bash file descriptor, which will be passed to the command when it is executed. You need to create a file for fd3 to actually point to.
-

Most General Conception of a Command Line Expression:

Fill this in later.

2 Basic Commands:

These consist of simple, file system and file, and commands that move around.

- **man** $<progname>$: Manual for a command

- **cd** $< directory >$: Change directory. ".." is previous directory, "." is pwd. " " is the home folder, "/" is root.
- **touch** [filename]: makes a file.
- **file** $< filename >$: Descriptive information on the type of file
- **chmod** [entity][op][mode]: for user (u), group (g), other (o) or all (a) implement read (r) write (w) or execute (x) modes. Ops are + and -.
- **cat** [options] [file list] : Concatinates files and writes them to standard output. -n lists line numbers, and "-" for a file name reads from standard output.
- **less** [options] [filepath]: A sophisticated program for examining files; loads as needed. Use this to read in parts of a file for examination, or to pipe to other commands. Options:
  - -b Num: load in a specific number of kilobytes.
  - -n: Suppress Line numbers.
  - -N: add line numbers this are not transferred with a piping operation.
  - -e: exit when EOL is hit twice.
- **head** [options] [file1] [file2] ...: simple command for displaying the top lines of a file. "-n" is the number of lines to display. "-q" suppresses header separators.
- **tail** [options] [file1] [file2] ...: same as head, but from the end of the file.
-
- **split** [options] [file] [prefix]: split files into chunks, and redirect the chunks somewhere. "-l num" splits based on file lines. "-b num" splits on byte size. "-n num" generates N sized chunks from the input file. "-e" avoids chunks of size zero.
- **test:**
- kill [options] [pids]: This command sends signals to processes; typically used to stop, kill or continue processes. Options are as follows:
  - –list -l: List every signal you can send, in a nice table.
  - -s: Specify a numerical code for a specific signal
  - -9: kill every process you can kill, matching the PIDs specified
  - -SIGSTOP -SIGKILL: stop or start a process; useful to pause a computationally intensive thing.
- **tree**: list the contents of a directory, with a nice printout format.
- mv
- **uname** -a: Prints all of the system information
- whereis
- locate
- ls : prints directory information.
  - -a lists all (including hidden directories)
  - -l lists all information.
  - -R recurse subfolders
  - -r reverse orderingq
- cp [options] [dest] [sources...]: The copy command. Be aware of the following options:
  - -u: only copy a file if it an updated version of what is seen at dest.
  - -nc: don't clobber files; if exists, don't copy
  - -t [dest]: list the destination explicitly, so you can list sources afterwards.
  - -R: recursively copy each source.
  - -i: interactive; prompt before overwrites
  - -v: verbose. Write whats going on as the work is being done.
- alias
-
- **diff:** [-shortopt] [-longopts] [file A] [fileB]. This works on looking at the differences of files. Note: As directories are files, it can also look at the differences in structure between two subdirectories!
  - -y: Display differences side by side
  - -q: just report if two files are different (Y or N)
  - (!) –suppress-common-lines: just output differences.
  - -r: recursively contrast two directories.
- diff also has a lot of options for formatting outputs, and context around what differences are reported.
- **sort** [options] file1 file2 filen: Sort will take a given input source (user) or a list of files as input. These are then appended back to back, and the entire contents is sorted by some ordering (alphabetic, numeric, byte, etc...). An "item" is separated by a new line in the file/source (so Enter).
  - *blank*: sort will accept input from the keyboard. Press CTRL+D to terminate
  - -r: will reverse the ordering.
  - -n: numeric sort
  - -R: random sorting.
  - -c: check if input sorted already.
  - -u: eliminate non-unique elements.
- split [options] [inputfile] [prefix]: splits a file up into pre-defined chunks:

prefix : A name provided, which is numbered for every file chunk.
  - -l N: put N lines per file.
  - -n N: specify the number of chunk you want directly.
  - -a N: specify length of suffices
  - **cut** [options] [filesq]: extracts columns of data from a set of files, and prints them to standard output.
    - -d : Select column delimiter. use single quotes to encase.
    - -f : fields. Allows you to select a range or list of columns to display.
    - –output-delimiter: change the delimiter when displaying output. Use single quotes to encase it.
    - -c: characters: select a range of characters instead.
  - Note: If multiple files given, results will be appended below each other, for each file.
- split

3 Complex/SuperPower Commands:

- **find** *[path]* [expressions: *Option Test Action* $< querytrings >$] : Search for files in the directory tree. Find does not follow symbolic links by default. A set of expressions is applied to each file, with the search string given by user. Files that come up true are added to the set of "found" files. Expressions consist of:
  Tests: that are run on the files that are considered; tests return true or false, and true entries are gathered up. Tests allow us to apply constraints to what we are searching.
  Actions: are performed after the search is done.
  Global Options: Affect the entire search process, (such as depth of search).
  Common find options:
  - -name: search file names specifically
  - -type ¡arg¿: consider the type of file. For example: *f* is regular files, whereas *b* is a block device.
  - +-n•
  - **ps** [options]: Display information from a selected set of processes. Information will be outputted in a table format. There are many options, which are catagorized by Unix or BSD types. UNIX requires a dash "-". Options need to be specified in ordered catagories, below:
    - **Simple Process Selection:**
    - -A -e: List every available process
    - **Process Selection by List:**
    - -p [pidlist]: select only processes in the list
    - -u [userlist]: filter by particular users/
    - **Output Format Control:**
    - -f: Full format listing
    - -F: Extra full format listing
    - -s: Signal format
    - -X: Register format
    - -u: user oriented format
    - -v: virtual memory format
    - **Output Modifiers:**
    - -f: ASCII art forest
    - -h: no header
    - **Thread Display:**
    - -m show threads after processes.
    - **Standard Usages:**
    - -wax: list locations of proceses.
    - -aux : lists even more information.
  - awk: Is a command with its own language, useful for parsing lines of data. To separate columns of output reliably, simply remember the following:

```
<data source> | awk ' {print $col1, ..., $coln} '
```

  - grep [options] pattern [files]: GREP - Global Regular Expression Print. A program that can search and exclude search patterns from files, and displays the matching output. Lots of options, here are some below:
    - -i: ignore case
    - -c: count number of occurences
    - -r: read files in a directory, recursively

        * PATTERN: Can be just some quoted words, or a regex expression.
    - lsof
    - crontab

Complex Examples:

1. Find all Chrome Processes and kill them:

   ```
   ps -A | grep chrom | awk ' {print $1} '| while read -r line;
   do kill -9 "$line"; done
   ```

2.
3.
4.
5.
6.
7.

Other Things not covered:

Coprocesses, Looping statements, if statements, Positional parameter more than 2 digits use braces?

References

[1] https://www.gnu.org/software/bash/manual/bash.html#Coprocesses

[2] http://www.thegeekstuff.com/2010/06/bash-shell-brace-expansion/

[3] http://linuxcommand.org/lc3_adv_redirection.php

[4] http://www.catonmat.net/download/bash-redirections-cheat-sheet.pdf

[5] http://www.catonmat.net/download/bash-redirections-cheat-sheet.pdf

[6] https://stackoverflow.com/questions/7082001/file-descriptors

[7] http://tldp.org/LDP/abs/html/io-redirection.html

[8] https://www.quora.com/Can-you-recommend-books-on-inter-process-communication-using-Bash?share=1

[9] https://en.wikipedia.org/wiki/Unix_signal#Sending_signals

[10] http://www.linux-admins.net/2010/12/lpi-101-certification-practice-test.html