Command Line Interface Techniques

EECS 268

Dr. Douglas Niehaus
Jared Straub

Overview

- Command line use is not emphasized by the GUI centric approach used by many applications and systems for many reasons
  - GUIs are easier for beginners
  - GUIs present a large amount of information that can be helpful to users figuring things out
  - Command Line Interface (CLI) a common term
- For casual users, GUIS are often the best choice
- For professional users, GUIs often impose limits on productivity and power that can be annoying
- CLI is often preferred by professional SWE and System Administrators as a way to overcome these limits in a variety of situations

Overview

- Limitations imposed by GUIs
  - By presenting all options, they add complexity to each *specific* use, since *all* uses are supported
  - Clicking makes user feel busy but it is often not the fastest way to accomplish a given actions
    - Particularly well-known actions
  - Script languages used to create one-time capabilities, or long-lived commands cannot easily use GUI capabilities
    - CLI tools are usually easy to use in scripts
- As a result of these and other limitations many SWE prefer the CLI for many activities
  - You should at least know how to use it

Overview

- Other motivations for CLI use
  - Remote access (SSH) using command line is lower BW than remote GUI or remote Desktop
    - Remote GUI/Desktop over a slow connection is at least painful and may be impossible
  - In many systems the command line interface is a superset of the GUI
    - GUIs are often wrappers for CLI
  - Many systems do not provide GUIs for all tasks
    - Embedded systems of many varieties
    - Routers, Machine Control, etc
    - GUIs built for them often emit CLI commands

Overview

- In other cases, a variety of basic SWE activities are generally faster using CLI
- Time is the most valuable SWE resource
- Cumulative time saving on activities done many times every day for many years can add up
- Difference between adequate and great SWE is based partly in the ability to have a higher sustained productivity
  - Continuous development of more efficient methods is a big part of that
  - Scripting and CLI use as part of developing a personal set of tools and methods is a big part of this for many SWE professionals

GUI/CLI Design Patterns

- There are two common ways to design software that can robustly support both CLI and GUI access
- One way emphasizes *library* or *module* layers that provide powerful high-level interfaces for all application semantics
  - GUI an CLI tools are then independently implemented *thin wrappers* for library routine calls
- Another uses the *CLI* implementation as the primary *interface* for all application capabilities
  - GUI is then an independent application which constructs and emits CLI command strings
- Both are often fine for a given application, each is the best approach for some applications

Remainder of Presentation

- Directory Hierarchy Navigation and Use
- File Access and Manipulation
- Command Argument Completion and Regular Expressions
- Combining Commands: Pipes
- CLI Context
  - Command Path, Environment Variables, History
- Combining Commands: Basic Shell and Python Scripts as Utility Commands
- Process Management
- Remote Machine Access
- Convention
  - "$ **fred** A B C": command "fred" invoked with arguments "A B C" at command prompt "$"

Directory Hierarchies

- CLI assumes that the user is always at a specific location within the overall directory hierarchy (DH)
  - Rooted a ”/” and branching from there to arbitrary levels
- Current Working Directory (CWD)
  - Current location within directory hierarchy
  - **pwd** command prints CWD path
  - “.” symbolizes CWD “..” symbolizes parent(CWD)
- Movement within DH - **cd** command changes CWD
  - No argument or “~” - Home Directory of user
  - **cd** *path* – change CWD to specified path
    - Absolute Path: starts with “/”
    - Relative Path: starts with “.” or “..”

Directory Hierarchies

- List contents of a directory
  - "$ **ls** ." or "$ ls" lists names of CWD contents
  - "$ **ls** -l path" for long form information path
    - If directory, then contents of it
  - "$ **ls** -d path" to list properties of directory itself
- Create and Delete Directories
  - "$ **mkdir** path" - make end element of path
  - "$ **mkdir** -p path" - make all missing parts of path
  - "$ **rmdir** path " - delete last path element
    - Must me empty
  - "$**rm** -fr path" - remove last element and everything underneath it, recursively
- Disk Use: "$ **du** ." - size of each file and dir under "."

Directory Hierarchies

- BASH – Born Again Shell
  - Bourne shell was original UNIX shell
  - Stupid Programmer Joke
- Provides commands keeping a stack of directories
  - Useful when bouncing around to different locations using a single prompt
  - Try “M-x manual-entry” in Emacs on “pushd”
    - Search for specific information
  - “$ **pushd** path” - pushes path on stack
    - Switches top two elements with no argument
  - “$ **dirs**” - prints stack
  - “ **popd**” - pops top element making second CWD
  - Numeric arguments and several variations listed in manual page entries

Directory Hierarchies

- Bash Prompt customization can help
  - Mine: export PS1="\h:\W\$ "
    - Defined in ~/.bashrc
    - Use "$. ~/.bashrc" to invoke new version
    - Prompt: "machine CWD$"
- Try setting yours using "[]" around pair instead of "$"
- Leave a space after prompt or not and try it out
- Many possible components "M-x manual-entry bash"
  - Search for "Prompting"
- I like s short prompt, and a machine name as I log into several machines simultaneously
- Others like more information – customize as you like

Files

- “$ **file** path” - tells you type of object specified
  - Try it out on a range of file types
  - Some are text (readable) and some binary
- Reading Files
  - “$ **cat** path” - display contents
  - “$ **more** path” - display contents with some control
  - “$ **less** path” - display contents with more control
- Copy Files
  - “$ **cp** path1 path2” - copy path1 to path2
  - “$ **cp** -R path1 path2” - recursively copy
  - “$ **rsync** -av path1 path2” - smarter copy
  - “$ **tar** -cf foo.tar path” - create archive “foo.tar” of path and DH under it

Files

- File Permissions
  - Three levels: Owner, Group, Other
    - Three bits in each: read, write, execute
  - “$ **ls** -l path” to see owner, group, permissions
  - Often expressed as OCTAL numbers
    - “777” = “rwxrwxrwx”
    - “664” = “rw-rw-r--”
    - “444” = “r—r--r--”
  - Execute for dirs is overloaded to mean “usable as a component of a bigger path”
  - Leading (4th) component has special property bits
    - Try Google on “UNIX File Permissions”

Files

- Modifying properties
  - “$ **chown** fred path” - change owner of path to fred
  - “$ **chgrp** sue path” - change group of path to sue
  - “$ **chmod** 664 path” - change permissions of path
- File Content processing
  - '$ **grep** “reg-expr” path'
    - Look at each line of “path” and print those containing instances of “reg-expr” regular expression
  - **awk** – tool supporting rule (predicate-action pairs) per-line processing of files
  - “$ **diff** path1 path2” - print per-line differences between path1 and path2

Files

- File Content processing
  - “$ **sed** -e'expr' path” - stream editor applying the specified expression to file in path
  - One common use is substitution: 's/e1/e2/g”
    - This says substitute every instance of expression e1 with e2, globally
    - Without “g” at the end it would do it only first time
    - Older style syntax from “ed” editor
    - Also used in vim, IIRC
  - “$ **sort** path” - sort contents of path line by line
    - Arguments can modify type of sort and data from each line used
- Other useful content commands: **head**, **tail**

Files

- Finding Files
  - "$ **find** . -print" - print paths of all files under CWD
  - Use grep and a pipe, explained later to look at a subset but "-name" option is similar
  - "-type x" where x is "f" or "d" can distinguish files and directories
  - Very powerful tool, including a "-exec" option for executing scripts on each file qualifying
  - Expressions are evaluated left to right

CLI Arguments

- Path arguments on CLI are Regular Expressions
  - Literals: “fred”, or “../sue”, or “./joe/john”
- All standard RE elements are also legal
  - “$ **ls** -l *.cc” - list all elements in “.” ending in “.cc”
  - All basic RE elements supported, with dialect roughly the same as vim or emacs
- “$ **ls** [abc].txt” - “a.txt” or “b.txt” or “c.txt”
- “$ **ls** fred.*”
  - All files with base name fred, regardless of what follows the “.”
- “**ls** */fred.*”
  - All files with base name “fred” in any directory within CWD, but only at first level, not all levels

CLI Arguments

- TAB completion also can speed you up a lot
  - BASH is always watching you
- Command Path
  - Start typing a command name and hit Tab when partially done
  - It may complete for you if BASH is sure
  - If not, either it does not exist, or too many choices
    - Try “$ **g**<TAB>” - too many choices
    - Try “$ **gc**<TAB>” - 14 choices
  - They are not really in the local directory but more on that a bit later
  - Good way to figure out the name of a command partially remembered

CLI Arguments

- File arguments are treated similarly
  - Completed if BASH is sure a unique choice exits
  - Not if several choices exist or BASH is clueless
- Naming directories in hierarchies so that frequently used components have “easy unique prefixes” can make typing paths at the command line *fast*
- Remember that many tasks are done many many times every day
  - Typing path names (CLI)
  - Clicking through path components (GUI)
- Basic principle of optimization is reduce overhead of those operations done most frequently

Redirection and Pipes

- UNIX philosophy for CLI use is to create a large number of commands useful for one kind of activity
  - Which are written according to the “pipe” convention” to promote us in combinations
- Standard I/O ports
  - STDIN – all commands read STDIN
    - Keyboard by default
  - STDOUT – all commands write to STDOUT
    - Screen by default
  - STDERR – can be used separately for “error” output to distinguish it from “regular” output
- Key point is user can “redirect” standard I/O ports

Redirection and Pipes

- Redirection
  - "< path" - redirection STDIN to come from a path
  - "> path" - redirects STDOUT to go to a path
  - ">> path" - appends to the file at path
  - "$ **cmd1** | **cmd2**" - redirect STDOUT of cmd1 to STDIN of cmd2
    - Thus, cmd2 processes output of cmd1
    - No limit on number of pipe stages
- Example: find largest uses of disk space under you HOME directory when your quota is exceeded
  - "$ **du** ~ | **sort** -nr >hogs"
  - What is this doing? Try to describe it in words, look up command options, give it a try

Redirection and Pipes

- “$ **make** > make.out”
  - Redirects STDOU to “make.out” to preserve output for later analysis
  - Errors still come to STDERR making analysis bogus
  - “$ **make** >& make.out”
    - Redirects BOTH STDOUT and STDERR to make.out
    - Ugly obscure syntax... just remember it like I do
    - “**tail** -f make.out” lets you watch output as it is produced, but you have to watch for the end as it just sits there

Redirection and Pipes

- “$ **sort** -nr <in_file >out_file”
  - Reads lines from “in_file” and outputs them in sorted order into “out_file”
- “tee” outputs to a file and to STDOUT
- Try:
  - “$ **du** ~ | **tee** out1 | **sort** -nr >out2”
  - “$ **head** out1”
  - “$ **head** out2”
  - “$ **sort** -nr <out1 >out3”
  - “$ **diff** out1 out3”

CLI Context

- Commands are type at the CLI in the *context* of the user typing them, which provides an *environment* within which the command is interpreted
- Environment is defined by the environment variables
  - "$ **env**" lists them
  - "echo $HOME" prints the one named HOME
    - The echo command echoes its arguments
    - "$<env-var>" tells Bach to look for an environment variable named <env-var> and substitute its variable
- PATH variable defines and ordered set of directories which are searched for command names
  - Set members separated by colons (:)
  - First match found is used
  - $HOME/bin often put first for per-user commands

CLI Context

- "~/.bashrc" file used to provide customized definitions of environment variables, add to existing ones, or create new ones
- "$ **which** emacs" outputs path of file named "emacs" used as the command
- If you create a Python script named "emacs", put it in $HOME/bin, with $HOME/bin first on your $PATH, what will happen when you type:
  - "$ **emacs**"
- Bash records a history of all commands
  - "$ **history**"
  - **Up-arrow** and **Down-arrow** keys let you browse
  - "$ **!**<cmd-number>" lets you repeat a command

CLI Context

- “$ **!**<reg-ex>” also lets you repeat the first match to the <reg-ex> while searching back in history
- Command line editing
  - Emacs commands apply to command line text
  - Movement (C-a, C-e), cut (C-k), paste (C-y), etc
  - Type at any point to insert text
  - Return when cursor anywhere on line executes it
- “$ **alias** cmd='string'” is the simplest way for you to define your own commands
  - Often used to create short versions of a frequently used command or specify default options
    - Try “$ **alias**” to see if you have any defined

CLI Context

- Defining common and convenient commands in "~/.bashrc"

```
alias ls=ls
alias ll='ls -l'
alias pd=pushd
alias pp=popd
alias h=history
```

Scripts

- BASH script is a file containing a set of statements
- Conditional and looping statements exist
  - You can look them up
  - “$ **man** bash” or “M-x manual-entry” in **emacs**
- Straight linear code is quite common
  - Each statement can be just as if typed at CLI
- Invocation
  - “$ **bash** path” interprets contents of path as a set of Bash statements
  - “#!/bin/bash” in first line of script specifies interpreter, similar to Python, and permits giving execute permission to script file directly

Scripts

- Bash script arguments
  - $1, $2, etc – each argument by position
  - $* - all arguments
  - $0 – command name element
- Bash is basically a full power programming language
  - Competing with all the others: Perl, Python, Ruby
  - Except for simple straight-line scripts I would generally prefer to write in Python
    - YMMV
- Python: os.system() routine takes a string as arg
  - Interpreted as if it were type at CLI, pipes too
  - Python wrappers (GUI) can often do little except build CLI strings, using existing commands
  - os.popen and variants worth some research as they also support controlling STDIN/STDOUT

Scripts

- Create and example script or two
  - echo “hello World”
  - echo $*
  - Call some other programs, whatever
- Create $HOME/bin
- Add it to your PATH in your .bashrc
  - “$. ~/.bashrc” remember, to use it in current shell

    export PATH=$PATH:$HOME/bin
  - Export keyword ensures definition exported to all subsequently executed shells
- Recall how you were able to call PDB from Emacs
- Consider things you do a lot or things you have trouble remembering as candidates for implementation as personal commands using scripts

Processes

- Each CLI command runs as separate child process
  - Just like every line in a Makefile
- Parent Bash process waits until child completes
  - Then give prompt
- Ampersand (&) tells the Parent not to wait
  - Child now running “in the background”
  - “$ **emacs**” vs “$ **emacs** &”
- Sometimes when you forget &, you can stop the current job (C-z), which gives you the prompt
  - Then use “$ **bg**” to make job continue running in the background
  - Corresponding “**fg**” command exists
  - Learn more of job control on Bash man page

Processes

- All processes have a unique identifier – PID (integer)
  - Echoed when a process starts in background
  - “$ **echo** hello &”
  - Job number in square braces, PID after
  - Finishes at once, but job completion announcement echoed at next RETURN
- List all your active processes
  - “$ **ps**” - same user and terminal as invocation
- List all processes in system
  - “$ **ps** aux” - includes User ID of owner
  - Pipe into grep to find all processes owned by your login ID: “$**ps** aux | **grep** <your-login>”

Processes

- “$ **ps** alx” also interesting
- Pipe output into **head** or **more** to see top line giving title to each column
  - You can look up any you like, but some are particularly interesting
  - PID – Process ID
  - PPID – Parent process PID
  - COMMAND – executable process is running
  - %CPU – percentage of CPU consumed by process
  - %MEM – percentage of memory used
  - UID – user ID of process owner
  - USER – login name of process owner

Processes

- Sometimes a process goes crazy
  - Firefox 3.6, but not 3.6
  - “$ **ps** aux | **grep** firefox” to find PID
- “$ **kill** -9 PID” - to kill it
  - Technically it sends Signal with integer value 9
  - “$ **kill** -signal name” - uses signal name
  - “$ **kill** -l” - lists all signal names and numbers
- “$ **top**” - gives summary system state and continuously updated active process list
  - Sorted by %CPU
  - Note cumulated CPU TIME column

Remote Access

- Secure Shell (**ssh**)
  - “$ **ssh** mach” - login into mach as current user
  - “$ **ssh** -l other mach” - log into mach as user “other” instead
    - Or, “$ **ssh** other@mach” is another form
  - “$ **ssh** -p X mach” log into mach using non-standard port X, standard SSH port == 22
- Secure Copy
  - “$ **scp** source-path destination-path”
  - Source and destination paths can be local or remote “user@mach:path”
- **Screen** – powerful tool to manage, detach and reattach, several shell or other remote sessions on a single remote machine using a single connection

Conclusion

- The CLI is a powerful and versatile tool giving access to thousands of commands
- CLI is faster for some purposes than GUI methods
- CLI provides some capabilities GUIs do not
- GUIs provide some capabilitie CLI does not
  - GUI diff wrappers ar worth checking out
- CLI is good to know
- Life long learning of new commands and refining your methods is strongly relevant
- Your preferences and methods should suit you
- Learn from other people's methods