UNIX Shells

- Bourne Shell -- sh
  - Written by At &T
  - A shell commonly used by the administrator when running as root
  - Simpler and faster than C shell scripts
  - Default shell prompt: **$**
- C shell -- csh
  - Developed by Berkley with added extra features
    - history, aliasing, etc
  - Default shell prompt: **%**
- Korn shell -- ksh
  - Developed by David Korn at AT&T
  - Added more features from C shell
  - Default shell prompt: **$**

Linux Shells

- GNU Bourne Again Shell – **bash**
  - The Linux default shell
  - An enhanced Bourne shell
  - The most popular shells used by UNIX and Linux users
  - Default prompt: **$**
- TC shell -- **tcsh**
  - An enhanced but completely compatible version of the Berkeley UNIX C shell, csh(1).
  - Default prompt: **>**

**Note: C shell and tc shell are not installed on CS Lab systems

Which Shell Are You Using?

- shell environmental variable: SHELL
  - `echo $SHELL`
  - `printenv SHELL`

`Note:`

Without arguments, `printenv` will display all the environment variables

In CS lab, the default shell is bash, unless you change it to a different shell

- `chsh` command
  - Run `chsh`, it will prompt you for password, then the login shell you prefer
  - `/etc/shells`
    - Lists all the available shells on the system
    - `chsh` must use one of the shells listed in file `/etc/shells`

Some Useful Files Under $HOME

.bash_login .bash_logout .bash_profile .bashrc (possible .profile)

- .bash_profile/.profile
  - Set env, such as umask, PATH, etc
- .bashrc: set command alias, functions, etc
- bash executes these files in order when login (interactive)

  `/etc/profile` ➔ `~/.bash_profile` ➔ `~/.bash_login` ➔ `~/.profile`

  It stops when a file is found and executed
- bash executes ~/.bash_logout if exists
- When bash started (not through login)

  `/etc/bash.bashrc` ➔ `~/.bashrc`

Interactive Shell Usage

- Type command at the prompt ($, %,or >) with arguments and/or options, this command line is ended with a new line
- The shell reads a line of input (the command line) and parses it : breaking the line into words, called tokens, separated by space or tab
- The first word is either a built-in command, or an executable program located somewhere on the disk

Command Line Processing

- For shell-built-in command, the shell will execute it internally
- For executable program, the shell will search the directories listed in the PATH environmental variable
  - The shell will **`fork`** a new process (a child process) and then execute the program with **`exec`**
  - The shell will report the status of the exiting program when it finishes
  - A prompt will return (appear)

Program Exit Status: ?

- When a command/program terminates, it returns an exit status to the parent process (or parent shell)
- The value of exit status
  - The exit status is a number in `[0-255]`
  - **`0`** – the command was successful in its execution
  - `Non-zero` – the command fails in some way
    - **`127`** – command is not found by the shell
- The special shell variable **`?`** is set to the value of the exit status of the last executed command
  - To check the exit status of the immediate previous executed cmd

    `echo $?`

Multi-commands in a Command Line

- Multiple commands at one line separated by “;”
  - `ls; pwd; date > output`
  - `(ls; pwd; date) > output`

  What’s the difference of the output from the above two command lines?
- Conditional execution of commands
  - `cmd1 && cmd2`
    - cmd2 executes only cmd1 is successful (`$?==0`)

    `find . –name hw5.sh && ./hw5.sh`
  - `cmd1 || cmd2`
    - cmd2 executes if cmd1 fails

```
grep cs390 2>log.err || mail -s “failed” linh@uah.edu <log.err
```

Shell and subshell

- A subshell is a new shell that is executed under the current shell
  - Subshell is a child process of the shell where it is created
  - Sub shell has **no knowledge of the local variables** defined in its parent shell
  - Subshell cannot change variables defined in its parent shell
  - Only environmental variables of current (parent) shell are available to the subshells
- To terminate/exit the subshell using "**exit**" command

Running script/commands

- Normally, a shell script is run by: **`bash script.sh`**
  - it will fork a child process (a subshell) to execute the program
  - The variables in the sub shell will not be available to its parent shell, the local variables defined in the parent's shell are also not available to the subshell
- There are two special programs: dot (.) /source
  - "source" command will run the commands of the script within the current shell
  - The *source* or . (*dot)* command normally is used to execute the above initialization files

```
a) source .bashrc
b)  . .bashrc
```

Shell Variables

- Variable name must start with a letter or _

Two types of variables

- Environmental (global) variables
  - Variables are available to the current shell and its subshells
  - Using command printenv to get the current shell settings
  - Ex: `PATH,HOME,SHELL,LOGNAME,PWD,PS1,PS2, HOSTNAME,USER,` etc.
  - Set/define environment variables
    - Method 1: **`VARNAME="value"; export VARNAME`**
    - Method 2: **`export VARNAME="value"`**
  - More env variables are listed by running "env" or Table 8.1
- Local variables
  - Defined by the user, available only to the current shell , not to its parent nor to its sub shells

The PATH variable

- A colon-separated list of directories
- Used by the shell when searching for the command
- System-wide PATH can be defined in either **`/etc/profile`** **or** **`/etc/bashrc`** **or** **`/etc/bash.bashrc`**
- User-defined path is either in .profile, .bash_profile, *.bashrc* or/and .bash_profile under your `$HOME` directory
  - To add new directory to the existing path

```
PATH=$PATH:~/cs390:~/local/bin:/opt/usr/bin
```

- Check the path value
  - **`printenv PATH (NO $ here)`**
  - **`echo $PATH`**

The Prompt (PS1 & PS2)

- Prompt string settings
  - `\h|\H`: hostname
  - `\u`: username of the current user
  - `\w`: the current working directory
  - `\W`: the basename of the current working directory
  - `\!`: the history number of this prompt
- PS1: primary bash prompt, "**$**" by default for bash, can be reset using:
  - `PS1="[\u@\h \W]\% "`
  - `PS1="[\u@\h]\> "`
  - Really confusing if `PS1="" -`☹
- PS2: secondary bash prompt, ">" by default
  - It's for uncompleted command, or more input is expected
  - Ex: `echo " this is cs390 `↵

Reference Value of Variables

- Prefix a "$" sign to variable name, `$varname` when referencing a variable for its value
- The **echo** command and its options
  - **-n**: suppress newline at the end of a line output
  - **-e**: allows interpretation of the escape sequences, \t,

```
echo "you are so \t nice " ➔you are so \t nice
echo –e "you are so \t nice " ➔ you are so
  nice
```

  - `echo ${varname}` ➔➔ `echo $varname`
    - Use curly bracket for string concatenation
    - `name=${variable}ABC`
- `printf`
  - `printf "The number is %.2f \n" 100`

Quoting

- Used to protect special meta-characters from interpretation and to prevent parameter expansion
  - Special metacharacters that require to be quoted
    - `; & < > | ( ) { } $ \`
- Three types of quoting
  - Matched double quotes: “”
  - Matched single quotes: ‘ ’
  - Back slash: \

- Single quotes
  - Enclosing characters in single quotes preserves the literal meaning of all the characters
  - `echo '$PATH'` ➔ `$PATH`
- Double quotes
  - Enclosing characters within double quotes preserves the literal meaning of all characters except dollar sign ($), backquote (`), and backslash (\).

`echo "here are 5 space:     ."` (five space)

`echo here are 5 space:     .` (only one space displayed!)

- Backslash `\`
  - Preserves the literal meaning of the following character, with the exception of ⟨newline⟩.
  - A backslash preceding a ⟨newline⟩ is treated as a line continuation.

To get the $ printed ➔ `echo It costs me\$500`

To get the \ printed ➔ `echo \\n` → `\n`

Shell Special Variables

- Consisting of a single character
- Preceding $ to access the value stored in the variable

Variable	Meaning
$	The PID of the current shell/process `echo $$`
?	The exit value (exit status or return) of the last executed command, a successful process returns 0, none zero otherwise `echo $?`
!	The PID of the last job put in the background `echo $!`

Shell HISTORY

- Bash will keep track of command history saved in file **`.bash_history`** under your home directory
- The length of the history commands is defined by environment variable **`HISTSIZE`**
  - **`echo $HISTSIZE`** ➔ `1000`
- The size can be reset
  - **`export HISTSIZE=100`**
- The history file will be updated at the time of logout
- History command **`history [n]`**
  - List the last n commands in the shell

Utilization of Shell History

- Short cut to run the previous command
  - `!!`

    The previous command
  - `!123`
    - Execute the command of command # 123
    - The number can be obtained from running “history” command
  - `!name`
    - The first command which matches name
  - `ctrl-r string`
    - Reverse-search for string starting from the end of the history file, press tab key when found the matched one

alias

- Making/giving another name for a command
- Creating alias with shell built-in command **`alias`**
  - Use single or double quotes if there is space in the value
  - The following can be put in file `.bashrc`

```
alias rm='rm -i'
alias ll='ls -l'
alias mv='mv -i'
alias sshzeta='ssh linh@zeta.itsc.uah.edu'
alias cd390='cd cs390_Sp16'
```

  - Multi-commands can be combined into one line separated by ";"

```
alias project1='cd cs390/project1; ls'
```

- Avoid using alias name the same as the system command unless it is intended, such as rm ➔ rm -i
- Deleting aliases: **`unalias name_of_alias`**

Shell I/O Redirection

- Three file descriptors (0, 1, 2) reserved for the terminal
  - 0: stdin; 1: stdout; 2: stderr
- Redirection operators

Operator	What It Does
`< filename`	Redirect input (read input from a file instead of screen/terminal/keyboard
`> filename`	Redirect output to file instead of the terminal screen
`>> filename`	Append output to file
`2> filename`	Redirect stand error output to file
`&> filename`	Redirect output and error (&>> for append)
`1>&2`	Redirect output to where error is going
`2>&1`	Redirect error where output is going

I/O Redirection Examples

- Redirect stdout to a file
  - ls –l > filelist.txt # save result to a file
  - ls –l >> filelist.txt # append the result to the file
  - echo “you are so good” > file
  - cat file2 >> file # append content of file2 to file
  - grep cs390 * > result
- Redirect stdout and stderr to one file
  - grep cs390 * > result 2>&1
  - grep cs390 * &> result
- Separate stdout and stderr to two different files
  - phonelist John Smith > result 2> log.err
- Take input from a file
  - cat <file.txt
  - tr ‘[A-Z]’ ‘[a-z]’ < myfile
  - patch file_original < patch_file (the difference)
  - mail –s “sub” hlin@cs.uah.edu < file.txt

Command Substitution

Assigning the output of a command to a variable

- Two formats
  - Back quotes, the old method: `` `command` ``
    - **`filelist=`ls -1`; echo $filelist` ➔???**
    - **`today=`date "+%m/%d/%y %T" ``**

    Double/single quotes are needed due to the white space b/w %y and %T
  - A set of parentheses preceded with a dollar sign: **`$(command)`**
    - **`filelist=$(ls -1)`**
    - **`today=$(date "+%m/%d/%y %T")`**
- Examples of different quoting schemes
  - `cal=$(cal),` or `cal=`cal``
  - `echo $cal`
  - `echo `$cal``
  - `echo `cal``
  - `echo "$cal"`

Parameter Expansion

```
foo='this is a test'
```

- **$foo ←→ ${foo}**
- **${variable#pattern}**
  - Delete the shortest matching part from LEFT and return the rest
  - `${foo#t*is}` ➔ `is a test`
- **${variable##pattern}**
  - Delete the longest matching part from LEFT and return the rest
  - `${foo##t*is}` ➔ `a test`
- **${variable%pattern}**
  - Matches the smallest trailing portion to pattern and removes it
  - `${foo%t*st}` ➔ `this is a`
- **${variable%%pattern}**
  - Matches the longest trailing portion to pattern and remove it
  - `${foo%%t*st}` ➔ `Empty output!`

String Substitution

```
foo='this is a test'
```

- **${var/pattern/string}**
  - Replace pattern with string: `${foo/is/IS}` ➔
- **${var^}; ${var^pattern}**
  - Capitalize the first letter or the first occurrence of pattern
- **${var^^}; ${var^^pattern}**
  - Convert every letter in var or matched pattern to uppercase
- **${var,} or ${var,pattern}**
  - Convert the first letter or the first occurrence of pattern to lowercase letter.
- **${var,,} or ${var,,pattern}**
  - **Convert every letter in var or every letter matched pattern to lowercase**

NOTE: The pattern is a "single character", or one of the set, such as [b-d]

Parameter Testing (set or unset)

- **`${foo:-bar}`**
  - If `foo` exists and is not null, return **`$foo`**, otherwise return `bar`
  - no modification to variable `foo`
- **`${foo:=bar}`**
  - If variable `foo` exists and is not null, return **`$foo`**,
  - otherwise set **`foo`** to bar and return its value (bar)
- **`${foo:+bar}`**
  - If foo exists and is not null, return bar,
  - otherwise substitute nothing (return null)
- **`${foo:?message}`**
  - If variable foo exists and is not null, return its value ($foo)
  - otherwise print the message

```
hlin@dakota:~> fruit=${fruit:-apple}
hlin@dakota:~> echo $fruit
apple
hlin@dakota:~> echo ${fruit:-pear}
apple
```

```
hlin@dakota:~> echo ${fruit:-apple}
apple
hlin@dakota:~> echo ${fruit:-pear}
pear
hlin@dakota:~> echo ${fruit:=apple}
apple
hlin@dakota:~> echo ${fruit:-pear}
apple
```

Substring

- **`${string:offset}`**
  - Position starts with zero
  - Returns the substring starting from offset
- **`${string:offset:len}`**

  returns the substring starting from offset for `len` long
  - Position starts with zero
  - For negative offset, the position is taken from the end of the string
    - Need to leave a white space b/w ":" and "-" sign

    `echo ${day: -3:2}` ➔ `DA if day="MONDAY"`