

Intermediate Linux

Presented by Wesley Brashear

February 17, 2023



High Performance
Research Computing
DIVISION OF RESEARCH



JOIN OUR STUDENT CLUSTER TEAM!

What is it?

The Student Cluster Competition (SCC) is a High Performance Computing (HPC) multi-disciplinary experience integrated within the HPC community's biggest gathering, the Supercomputing Conference (SC23).

In this real-time, non-stop, 48-hour challenge, teams of undergraduate students assemble small clusters on the exhibit floor and race to complete real-world workloads across a series of applications all while staying under a power cap.

For more information about SCC23, please visit <https://bit.ly/scc23>

When is it?

Sun, Nov 12 – Fri, Nov 17, 2023



Where is it?

Denver, Colorado, USA



How to Apply?

https://bit.ly/hprc_scc23

Who can apply?

TAMU Undergraduate students*

*Undergraduate in the Fall of 2023

Application Deadline: March 1, 2023

What else can I look forward to?

- Attend Supercomputing 23 (SC23) with registration, travel and lodging covered.
- Learn about high performance computing and its scientific applications from industry experts.
- Participate in Mentor–Protégé Matching and the Job Fair at SC23 and learn from some of the top minds in industry.
- Working with supervisors and vendor partners to design and build a winning cluster from commercially available components as part of preparation for the competition.



Overview

- Text Processing
- Text Processing Practice
- Bash Scripting
- Bash Scripting Practice
- bc
- Setting Up an Environment

Practice Files

- Login to Grace now
 - ssh (enable X11 forwarding) or through the portal
- Change to your SCRATCH directory

```
cd $SCRATCH
```

- Copy the practice files and change directories

```
cp -r /scratch/training/Intermediate_Linux .  
cd Intermediate_Linux
```

Text Processing

Students will be able to use vi (vim) to view, edit and save text files. Students will also be able to understand common uses of stream editors, sed, awk, and grep.

Text Processing - GUI

- Files can be edited through the Files tab on the Open OnDemand Portal
- If you access Grace through terminal with X11 enabled:

```
gedit filename
```

```
emacs filename
```

vi editor

- `vi filename` # opens (creates) a file using vi
- `vi -R filename` # opens a file using vi in read-only mode
- `view filename` # same as `vi -R filename`

Two modes

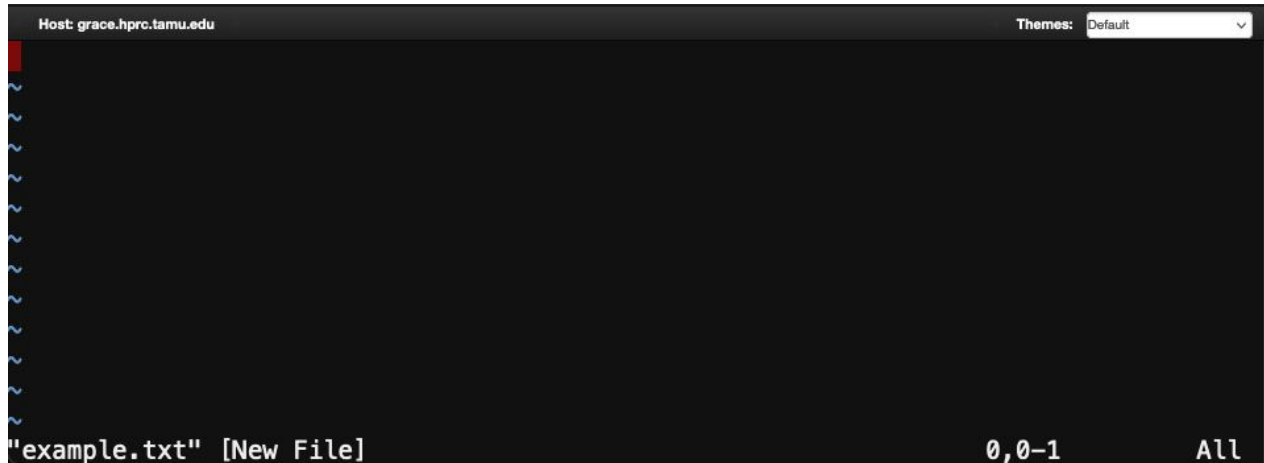
- insert mode
 - for typing in text
 - all keystrokes are interpreted as text
 - `i` command initiates insert mode
- command mode
 - for navigating the file and editing
 - all keystrokes are interpreted as commands
 - `Esc` returns the user to command mode

vi editor - Practice

- Create a file:

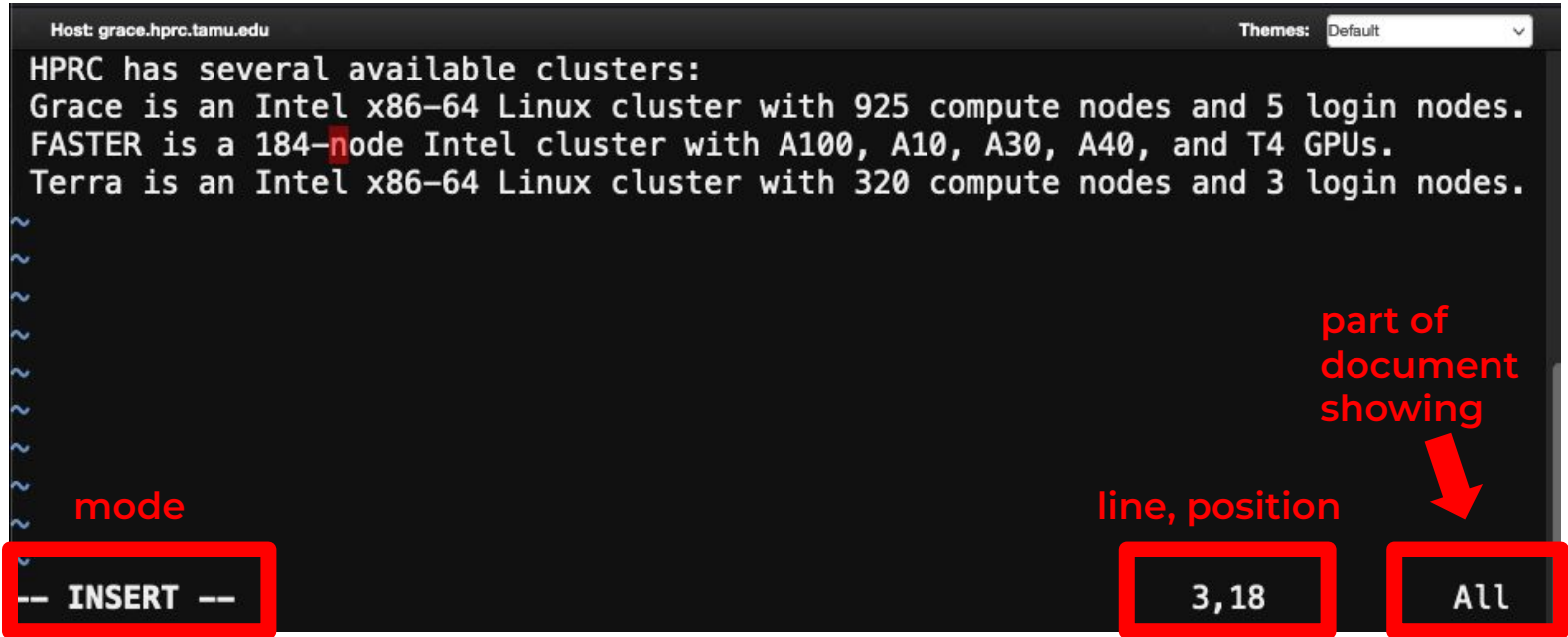
```
vi example.txt
```

- vi starts in command mode:



vi editor - Practice

- press `i` to enter insert mode
- Type a few sentences over several lines:



The screenshot shows the vi editor interface. At the top, it displays "Host: grace.hprc.tamu.edu" and "Themes: Default". The main text area contains the following text:
HPRC has several available clusters:
Grace is an Intel x86-64 Linux cluster with 925 compute nodes and 5 login nodes.
FASTER is a 184-node Intel cluster with A100, A10, A30, A40, and T4 GPUs.
Terra is an Intel x86-64 Linux cluster with 320 compute nodes and 3 login nodes.
The editor status bar at the bottom shows "-- INSERT --", "3,18", and "All". Red annotations highlight these elements: "mode" points to "-- INSERT --", "line, position" points to "3,18", and "part of document showing" points to "All".

vi commands

To exit a file or save press Esc

- **ZZ** or **:wq** or **:x** save the file and exit
- **:w** *filename* - save the file with the name filename
- **:w!** force save
- **:q** or **:q!** quit without saving
- **:q** quits a file when there have been no changes
- **:q!** quits the file regardless of changes

Try writing something, then close the file you created!

vi commands

Open the file you created in vi to practice these commands

Moving around in the file

- **h**, **l** (or space), **j** and **k** - left, right, down and up
- **G** move to end of file
- **nG** go to line n
- **CTRL + f** Scroll down a full screen
- **CTRL + b** scroll up a full screen
- **0** (zero) Move to start of current line
- **w** move forward one word
- **b** move back one word
- **e** move to the end of the word

vi commands

Open the file you created in vi to practice these commands

Commands that take you into insert mode

- **i** insert text to the left of the cursor
- **I** insert text at the beginning of the line
- **a** insert text to the right of the cursor
- **A** insert text at the end of the line
- **o** open a line below the cursor
- **O** open a line above the cursor
- **R** overwrite text to the right of the cursor
- **cw** change a word with new text - the cursor must be at the beginning of the word

vi commands - Practice

- Open a new file named "HelloWorld.sh"

- Change vi to insert mode and type:

```
#!/bin/bash  
echo "Hello World"
```

- Then save and exit the file.

- Type `ls` - Do you see the file you created?

- Type `more HelloWorld.sh` - You should see:

```
#!/bin/bash  
echo "Hello World"
```

- Type `bash HelloWorld.sh` - You should see:

```
Hello World
```

vi commands

Editor commands that keep you in command mode

- **x** delete a single character at the cursor
- **dd** delete the entire current a line
- **n~~dd~~** delete n lines
- **dw** delete a word
- **dG** delete to the end of the file
- **D** delete to the end of the line
- **ra** replace current character with a (a = character, number, etc.)
- **u** undo last command (only 1 undo on most unix machines. Most new versions of vi (vim) have multiple undo and redo (Ctrl-r) capability)
- **n~~yy~~** yank n (n is a number) lines to memory
- **p** (lowercase p) put the yanked lines below the cursor
- **P** (uppercase P) put the yanked lines above the cursor

vi commands

Miscellaneous commands

- `/name` search forward for *name*
- `?name` search backward for *name*
- `:1,$ s/pattern1/pattern2/g`
 - from line 1 to the bottom find and substitute pattern1 for pattern2
 - you could also use `:% s/pattern1/pattern2/g`
 - % and 1,\$ mean the entire file
 - the **g** means that all occurrences of pattern1 will be substituted in a line and not just the first one
- `:e filename` exits to the file filename
- `ma` marks that line and stores the position in the variable a
- `:'a,. y x` yanks the lines between the mark a and where the cursor is (.) and stores it in the variable x
- `:pu x` puts the lines stored in x into the file where the cursor is
- `:r filename` read file named filename and insert after current line
- `:set all` lists all of the settings
- `:set number` displays line numbers

vi commands - Practice

- Open the file named “environment.txt”: `vi environment.txt`
- Search for “OMP_NUM_THREADS”: `/OMP_NUM_THREADS`
- Remove the highlight from the search: `:noh`
- Replace the “1” after the equals sign to “2”:

```
# move cursor over "1"
x      # deletes char at current position
a      # open insert mode after cursor
2      # type in the new value
ESC    # exit insert mode
```


vi commands - Practice

- Search backward for “PWD=” `?PWD=`

- Change the path to contain your username:

```
cw      # Change word (erase current word and go into insert mode)
yourUserName # Type in new variable name
ESC     # Exit insert mode
```

- Search backward again for “USER=” and delete this line without entering insert mode:

```
?USER=  # search backwards for “USER=”
dd      # delete this line
```

vi commands - Practice

- Wait! Undo that last delete

```
u # undo last dd command
```

- Copy that line and the next two lines and paste them on the end of the file:

```
3yy # "3 lines yanked" should appear on the bottom left  
G # Move the cursor to the bottom of the file  
p # lowercase p pastes the yanked lines below the cursor
```

vi commands - Practice

- Move the cursor to the beginning of the file

```
1 G # Navigate to first line
```

- Search for all occurrences of “username” and replace with “hprc_user”

```
:1,$ s/username/hprc_user/g
```

- List all of the vi settings: `:set all`

- Show the line numbers in vi: `:set number`

- Save the document and exit: `:x` or `:wq`

GNU sed - Stream editor (sed)

A stream editor is used to perform basic transformations on text read from a file or a pipe.

- Useful one-line scripts for sed: <http://sed.sourceforge.net/sed1line.txt>
- Online manual: <https://www.gnu.org/software/sed/manual/>
- Common uses
 - **sed -n '4,6p' filename**
 - print out line 4 to line 6 (without -n, all lines will be printed and lines 4 to 6 will be printed twice)
 - **sed '2,4d' filename**
 - delete line 2 to line 4 (output will contain lines 1, 5-10)
 - **sed '3,\$d' filename**
 - delete line 3 to the last line of the file (output will contain lines 1 and 2)

Try these commands on the file labeled "sed_example.txt"

GNU sed - Stream editor

- **s** substitute
 - `sed 's/pattern1/pattern2/g' filename`
 - find `pattern1` and replace it with `pattern2` for all instances of `pattern1`, output is set to `stdout` (`g` can be left off to only replace the first instance, or can be replaced with a number to replace that many instances)
 - `sed 's/pattern1/pattern2/g' filename > filename2`
 - output is set to `filename2`
 - `sed -i 's/pattern1/pattern2/g' filename`
 - modifies the file in-place (changes the original file)
 - `sed 's/^/pattern1/' filename`
 - insert `pattern1` at the beginning of each line of a file
 - `sed 's/$/pattern1/' filename`
 - insert `pattern1` at the end of each line of a file

sed - Practice

- In this practice we will use the file **sed_ex02.txt**.
- Make the following changes to the text file using sed commands.
- **cp sed_ex02.txt tmp.txt** create a copy of the file to work on.
- Replace all instances of “vegetables“ with “cheese” even if it appears more than once in a line.
- Delete the 1st line of the file.
- Delete the now 2nd line of the file.
- Delete the now fourth line of the file.
- Print lines 1 to 4 and save to a file output.txt.
- Examine the contents of output.txt

GNU Awk

awk is used to search files for lines (or other units of text) that contain certain patterns and then do something (print, manipulate, etc).

```
awk options '/search pattern/ {action}' input-file > output-file
```

- Delimiters (Field Separator, FS)
 - Default is white space
- Search patterns
 - **awk** *'/pattern/' filename*
- Variables
 - fields are stored in variables based on the FS
 - \$0 the entire line
 - \$1 1st field
 - \$2 2nd field

shopping_list.txt

peach	fruit	8
tomato	vegetable	5
zucchini	vegetable	4

\$1 \$2 \$3

GNU Awk

- Variables
 - NR number of records
 - NF Number of Fields in a record
 - RS Specifies the record separator
 - FS Specifies the field separator
 - OFS Specifies the Output field separator
 - ORS Specifies the Output record separator

- Print statement

- `awk '/pattern/ {print $0}' filename`
- `awk '/pattern/ {print $1 "," $2}' filename>outputfilename.txt`

- `printf` statement for more control over the print format

- https://www.gnu.org/software/gawk/manual/html_node/Printf-Examples.html

- Pre-processing/Post-processing

- BEGIN

```
awk 'BEGIN {print "Shopping List"} { print $1, $2 }' sample.txt
```

- END

```
awk 'END { print NR }' sample.txt
```


GNU AWK - Practice

- Print the 3rd column of the tmp.txt file created during the last practice.
- What does it say?

Searching File Contents - grep

grep *search-pattern filename* - searches the file *filename* for the pattern *search-pattern* and shows the results on the screen (prints the results to standard out).

- **grep Energy run1.out**
 - searches the file run1.out for the word Energy
 - grep is **case sensitive** unless you use the **-i** flag
- **grep Energy *.out**
 - searches all files in that end in .out
- **grep "Total Energy" */*.out**
 - You must use **quotes** when you have blank spaces. This example searches for Total Energy in every file that ends in .out in each directory of the current directory
- **grep -R "Total Energy" Project1**
 - Searches **recursively** all files under Project1 for the pattern Total Energy

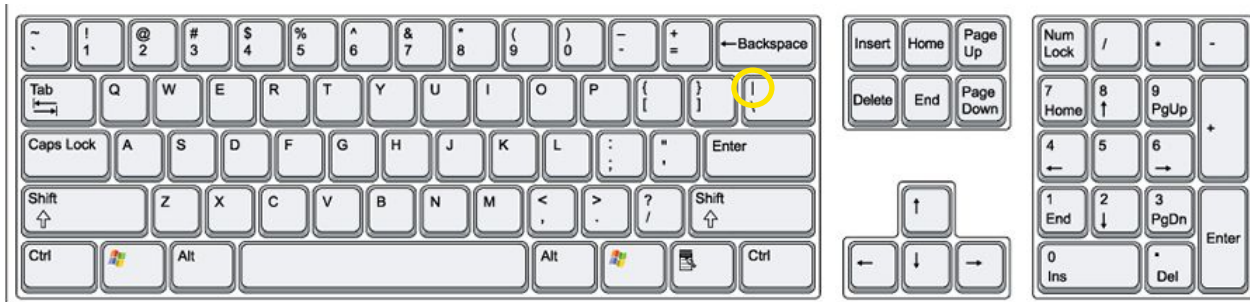
Searching File Contents - grep

- **grep -A N 'search' filename**
 - Outputs N lines after each line containing the search term.
- **grep -B N 'search' filename**
 - Outputs N lines before each line containing the search term.
- **grep -v 'search' filename**
 - Outputs lines that do not contain the search term.

Searching File Contents - egrep

egrep *'pattern1|pattern2|etc'* filename

- searches the file filename for **all patterns** (pattern1, pattern2, etc) and prints the results to the screen.
- The | character is called a **pipe** and is normally located above the return key on the keyboard.
- egrep 'Energy|Enthalpy' *.out
 - searches for the word Energy or Enthalpy in every file that ends in .out in the current directory.



grep & egrep Hands on Practice

- Use grep to count the number of lines containing either 'min' or 'max' in the file ex03.txt.
- Using only grep, print only the line after the line containing 'love' in the file ex02.txt

Bash Scripting

Learning Objective:

Understand conditions, loops and write bash scripts for simple tasks

Basic Shell Scripting

A shell script is a text file that contains one or more linux commands that can be run as a single batch of commands.

Shell scripts can be used to automate routine tasks.

It is good practice to name shell scripts with: **.sh**

```
#!/bin/bash ← shebang, indicates the shell
# ===== script header
# Description, Revision history, License

# VARIABLE ASSIGNMENT
CURRENTUSER=$(whoami)
# SHOW MESSAGES
grep $CURRENTUSER /etc/passwd
```

} body

The shell ignores blank and commented-out lines.

It is a good practice for developers to include info about the script in the header.

To store the output of a command in a variable, use `MYVAR=$(command)`

To run the script

- run with **bash script.sh**
- add executable permission to the script file (**chmod u+x script.sh**) and run with **./script.sh**

Basic Constructs for Bash Scripting

- Conditionals:

If something is true do something and if it is false, do something else

```
if < some test >
then
  commands
elif < some test >
then
  other commands
else
  commands
fi
```

```
#!/bin/bash
#

i=1
if [ $i -eq 1 ] ; then
  echo i is equal to 1
else
  echo i does not equal 1
  echo i equals $i
fi
```


Integer Comparison Operators

<code>-eq</code>	is equal to	<code>if ["\$a" -eq "\$b"]</code>
<code>-ne</code>	is not equal to	<code>if ["\$a" -ne "\$b"]</code>
<code>-gt</code>	is greater than	<code>if ["\$a" -gt "\$b"]</code>
<code>-ge</code>	is greater than or equal to	<code>if ["\$a" -ge "\$b"]</code>
<code>-lt</code>	is less than	<code>if ["\$a" -lt "\$b"]</code>
<code>-le</code>	is less than or equal to	<code>if ["\$a" -le "\$b"]</code>
<code><</code>	is less than (within double parentheses)	<code>(("\$a" < "\$b"))</code>
<code><=</code>	is less than or equal to (within double parentheses)	<code>(("\$a" <= "\$b"))</code>
<code>></code>	is greater than (within double parentheses)	<code>(("\$a" > "\$b"))</code>
<code>>=</code>	is greater than or equal to (within double parentheses)	<code>(("\$a" >= "\$b"))</code>

String Comparison Operator

`==` True if `$a` starts with an "z" (pattern matching).

True if `$a` is equal to `z*` (literal matching).

File globbing and word splitting take place.

True if `$a` is equal to `z*` (literal matching).

```
[[ $a == z* ]]
```

```
[[ $a == "z*" ]]
```

```
[ $a == z* ]
```

```
[ "$a" == "z*" ]
```

`!=` is not equal to

```
[ "$a" != "$b" ]
```

`<` is less than, in ASCII alphabetical order

```
if [[ "$a" < "$b" ]] or  
if [ "$a" \< "$b" ]
```

`>` is greater than, in ASCII alphabetical order

```
if [[ "$a" > "$b" ]] or  
if [ "$a" \> "$b" ]
```

`-z` string is null, that is, has zero length

```
if [ -z "$s" ]
```

`-n` string is not null

```
if [ -n "$s" ]
```

Basic Constructs for Bash Scripting

Case Constructs

```
case var in
  case1)
    <commands>
    ;;
  case2)
    <commands>
    ;;
  *)
    commands ;;
esac
```

* symbol defines the default case, usually in the final pattern.

```
#!/bin/bash
#
month='June'
case $month in
  Jan)
    mnum='01'
    ;;
  Feb)
    mnum='02'
    ;;
  ...
  Dec)
    mnum='12'
    ;;
esac
```

Practice: Conditionals

- Create a shell script that checks if the current day on the system belongs to the first, middle or last part of the month.
- If it is within the first 10 days of a month, print (echo) "We are within the first 10 days of the month."
- Otherwise, check if it is less than or equal to 20 and echo "We are within the middle 10 days of the month."
- If none of the previous conditions are met, return "We are within the last few days of the month."
- Hint: To obtain current day of month, use the command `date +%d` (there is a space before +)

Bash Scripts with input outside the script

- Command line arguments

```
#!/bin/bash
# shell script exercise
my_name=$1
echo "Howdy $my_name"
```

```
[user@host ~]bash my_name.sh Amy
[user@host ~]Howdy Amy
```

- Read input during script execution

```
#!/bin/bash
# Ask the user for emails
read -p "Username: " uservar
read -sp "Password(hidden): " passvar
echo
echo Thank you $uservar we now have your info
```

```
[user@host ~]bash info.sh
Username: Stan
Password(hidden):
[user@host ~]Thank you
Stan we now have your info
```

- Accept data that has been redirected into the Bash script via STDIN

Basic Constructs for Bash Scripting

Loops: Do something over and over until a specific condition changes and then stop

```
while [ <some test> ] ; do
    <commands>
done
```

```
for var in <list> ; do
    <commands>
done
```

```
#!/bin/bash
#
i=1
while [ $i -le 100 ] ; do
    echo i equals $i
    ((i++))
done

#!/bin/bash
for file in *.log ; do
    head -n1 $file
done
```

Practice: Loops

- Write a simple number-guessing game in a script called `guess.sh`. When the script is launched, a random number between 1 and 10 is generated and stored in the variable `RANDOMNUM`. The script then will expect input from the user. If the guess is incorrect, it will continue to ask the user for an input until the user guesses the number correctly. (Hint: to generate a random number between 1 and 10, use the command `shuf -i1-10 -n1`)
- Change the permissions on files end with `.sh` to `755` using a for loop or run **`bash guess.sh`**

bc - Basic Calculator

- bc is a command line calculator which can be useful for quick calculations.
 - Allows for arithmetic operations in bash scripts.

- Addition/subtraction Example: `echo "13+3" | bc`

- Exponential Example: `echo "10^3" | bc`

- Assign value of calculation to variable example: `x=`echo "12+5" | bc`
echo $x`

- Variables can be used in calculations:
 - What value do you get? `echo "$x+100" | bc`

bc - Practice

- Create a bash script that defines a variable i as 1.
- Then add a loop that adds $(i+2)$ to i , 12 times.
 - Print i each step of the loop.
- What is the final number?

Customizing the Environment

Bash Environment Variables

- Environment variables store information that is used across different processes in a Linux system.
- Use all caps for Bash Environment variable. **A-Z 0-9 _**
- Use lowercase for the variables that you create. **a-z 0-9 _**
 - **HOME** Pathname of current user's home directory
 - **PATH** The search path for commands.
- Use the **echo** command to see the contents of a variable

```
echo $HOME
```

The Search PATH

- The shell uses the **PATH** environment variable to locate commands typed at the command line
- The value of PATH is a colon separated list of full directory names
- The PATH is searched from left to right. If the command is not found in any of the listed directories, the shell returns an error message
- If multiple commands with the same name exist in more than one location, the first instance found according to the PATH variable will be executed.

```
echo $PATH
```

```
/usr/lib64/qt-3.3/bin:/sw/local/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/usr/lpp/mmfs/bin:/home/netid/local/bin
```

- Add a directory to the PATH for the current Linux session

```
export PATH=$PATH:/home/netid/bin
```

Customizing the Environment

Two important files for customizing your Bash Shell environment

- **.bashrc** (pronounced dot bashrc)
 - contains aliases, shell variables, paths, etc.
 - executed (sourced) upon starting a non-login shell.
- **.bash_profile** (dot bash_profile)
 - also can contain aliases, shell variables, paths, etc
 - normally used for terminal settings
 - executed (sourced) upon login
 - if **.bash_profile** doesn't exist, the system looks for **.profile** (dot profile)
- **. .bashrc** (or **source .bashrc**)
 - Executes the commands in the **.bashrc** file

.bash_profile file contents

```
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/.local/bin:$HOME/bin
export PATH

# Personal aliases
alias h="history|more"
alias m="more"

# User specific functions
function cc() { awk -f cc.awk "$@">"$@".cc ; }
```

A line that begins with a # is a comment

Enable settings in .bashrc

Syntax to set a global variable:

```
export var_name=value
```

Specify PATH for all sessions

Add personal aliases

Syntax to create a function:

```
function name() { command ; }
```

If you type **cc test** at the prompt, the following command will be executed:

```
awk -f cc.awk test.log > test.cc
```

Practice 4: Alias and \$PATH

- Add a new alias in your `.bash_profile` under your home directory named **simple** that executes the command: `echo I succeeded in creating a simple alias`
- Activate your new alias
- Type **simple** at the prompt to use your new alias
- Make a directory named **myapps** in your home directory
- Create a file (*your choice of a name*) in **myapps** with the following content:
 - `echo I succeeded in adding myapps to my path`
- Change the permissions of *filename* to allow execution (replace *filename* with the name that you used)
- Run *filename* by typing *filename* (you should get an error message)
- Add **myapps** directory to your PATH with **export** in your current session
- Run *filename* by typing: *filename*

Solution: Alias and \$PATH

- Add a new alias in your `.bash_profile` file named **simple** that executes the command:
echo I succeeded in created a simple alias

```
echo 'alias simple="echo I succeeded in creating a simple alias" '>> .bash_profile
```

- active your new alias

```
. .bash_profile
```

- Type **simple** at the prompt to use your new alias

```
simple
```

- Make a directory named **myapps** in your home directory

```
cd  
mkdir myapps
```

- Create a file (your choice of a name) in myapps with the following content:
 - echo I succeeded in adding myapps to my path

```
cd myapps  
echo "echo I succeeded in adding my apps to my path" >> filename
```


Solution: Alias and \$PATH

- Change the permissions of *filename* to allow execution (replace *filename* with the name that you used)

```
chmod u+x filename
```

- Run *filename* by typing *filename* (you should get an error message)

```
filename
```

```
bash: filename: command not found
```

- Add **myapps** directory to your PATH with **export** in your current session

```
export PATH=$PATH:/home/username/myapps/
```

- Run *filename* by typing: *filename*

```
filename
```



**HIGH PERFORMANCE
RESEARCH COMPUTING**
TEXAS A&M UNIVERSITY

<https://hprc.tamu.edu>

HPRC Helpdesk:

help@hprc.tamu.edu

Phone: 979-845-0219

Help us help you. Please include details in your request for support, such as, Cluster (Faster, Grace, Terra, ViDaL), NetID (UserID), Job information (Job id(s), Location of your jobfile, input/output files, Application, Module(s) loaded, Error messages, etc), and Steps you have taken, so we can reproduce the problem.