Bash Shell Scripting for Helix and Biowulf

staff@hpc.nih.gov

October 3 & 4, 2017

This Presentation Online

https://hpc.nih.gov/training

*nix Prerequisite

- It is essential that you know something about UNIX or Linux
- Introduction to Linux (Helix Systems)
- **https://hpc.nih.gov/training**
- Linux Tutorial at TACC
- Linux Tutorial at TLDP

Guide To This Presentation

- These lines are narrative information
- Beige boxes are interactive terminal sessions:

```
$ echo Today is $(date)
```

- Gray boxes are file/script contents:

```
This is the first line of this file.
This is the second line.
```

- White boxes are commands:

```
echo Today is $(date)
```

Terminal Emulators

- PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/)
- iTerm2 (http://iterm2.com/)
- FastX (http://www.starnet.com/fastx)
- NoMachine (https://www.nomachine.com/)

Connect to Helix

- open terminal window on desktop, laptop

```
ssh user@helix.nih.gov
```

- DON'T USE 'user'. Replace with YOUR login.

Copy Examples

- Create a working directory

```
mkdir bash_class
```

- Copy example files to your own directory

```
cd bash_class
cp -pR /data/classes/bash/* .
```

HISTORY AND INTRODUCTION

BASH

- Bash is a shell, like Bourne, Korn, and C
- Written in 1989 by Brian Fox the first employee of the FSF
- Default shell for most Linux distributions, MacOS, BSDs and many other platforms
- Bash has incorporated many of the “cool” features of it's predecessors as well as added many cool features of its own eg.
- Bash-Completion package; tab-completion feature

Online References for More Bash Information

- http://www.gnu.org/software/bash/manual/
- http://www.tldp.org/
- http://wiki.bash-hackers.org/

Bash on Helix and Biowulf

- Both Helix and Biowulf run RHEL v6, Bash v4.1.2

- Verify your version of bash with the following
- $bash --version

You might be using Bash already

```
$ ssh user@helix.nih.gov
...
Last login: Wed Aug 01 23:45:67 2019 from
123.456.78.90
$ echo $SHELL
/bin/bash
```

If not, just start a new shell:

```
$ bash
```

What shell are you running?

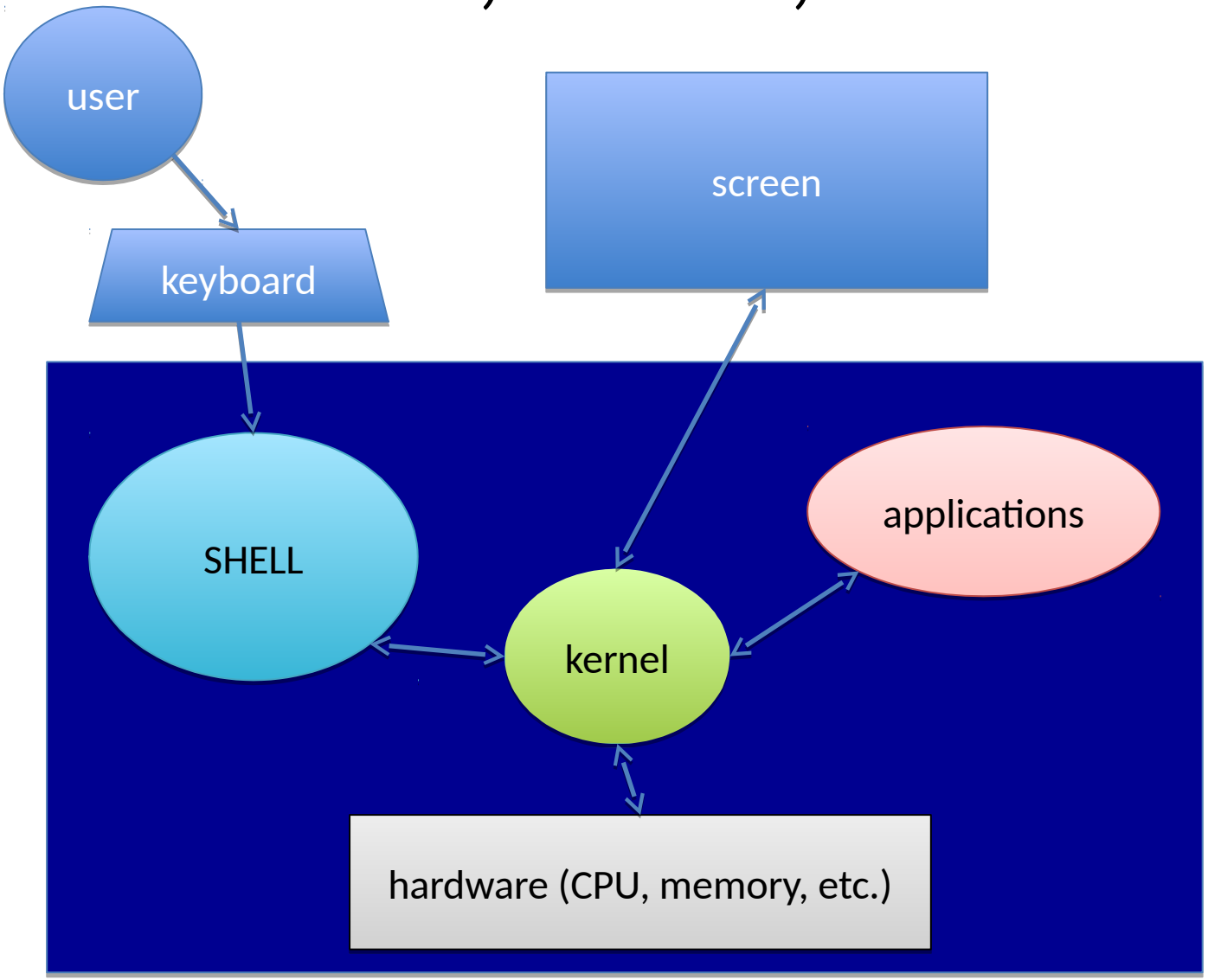
- You should be running bash:

```
$ echo $0
$ echo $SHELL
-bash
```

- Maybe you're running something else?

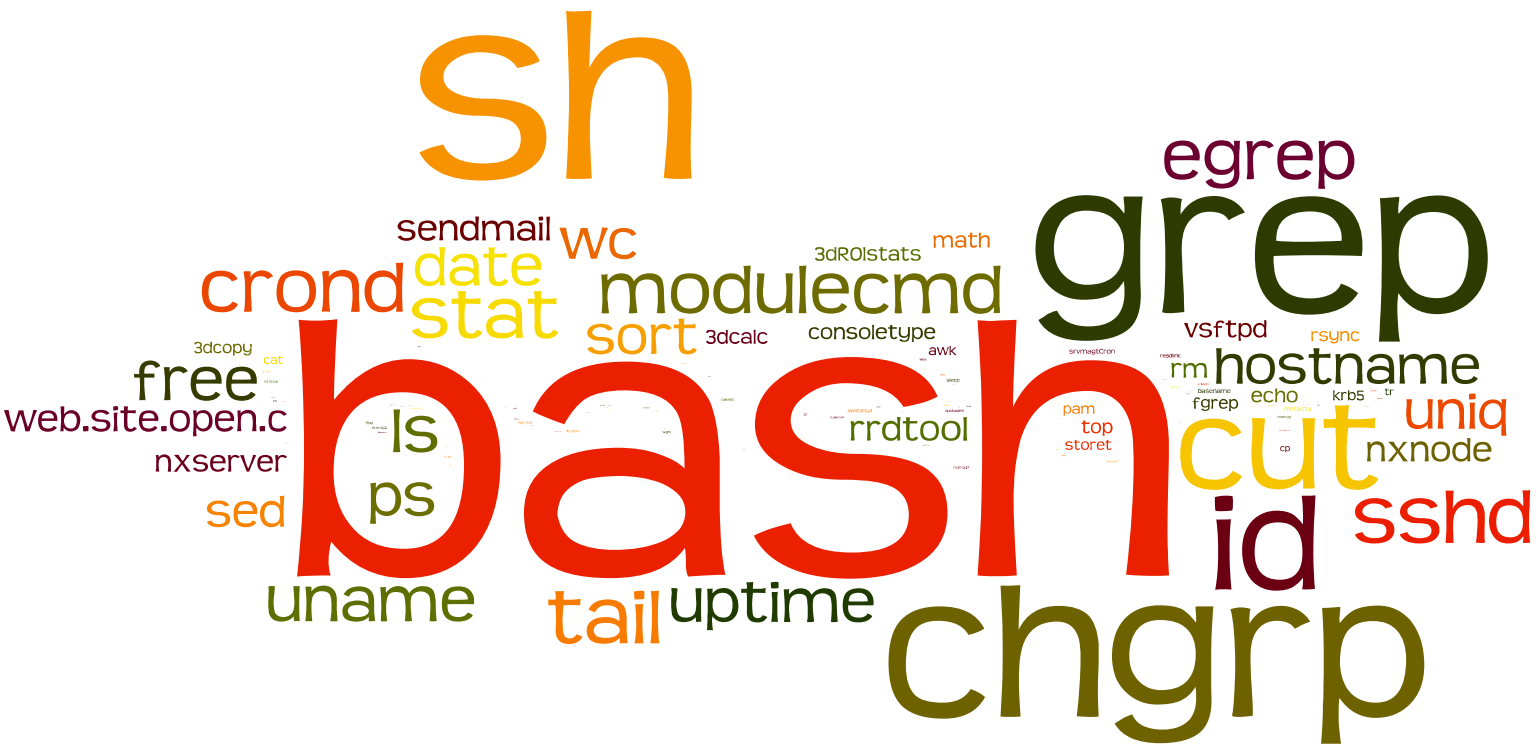
```
$ echo $0
-ksh
-csh
-tcsh
-zsh
```

Shell, Kernel, What?



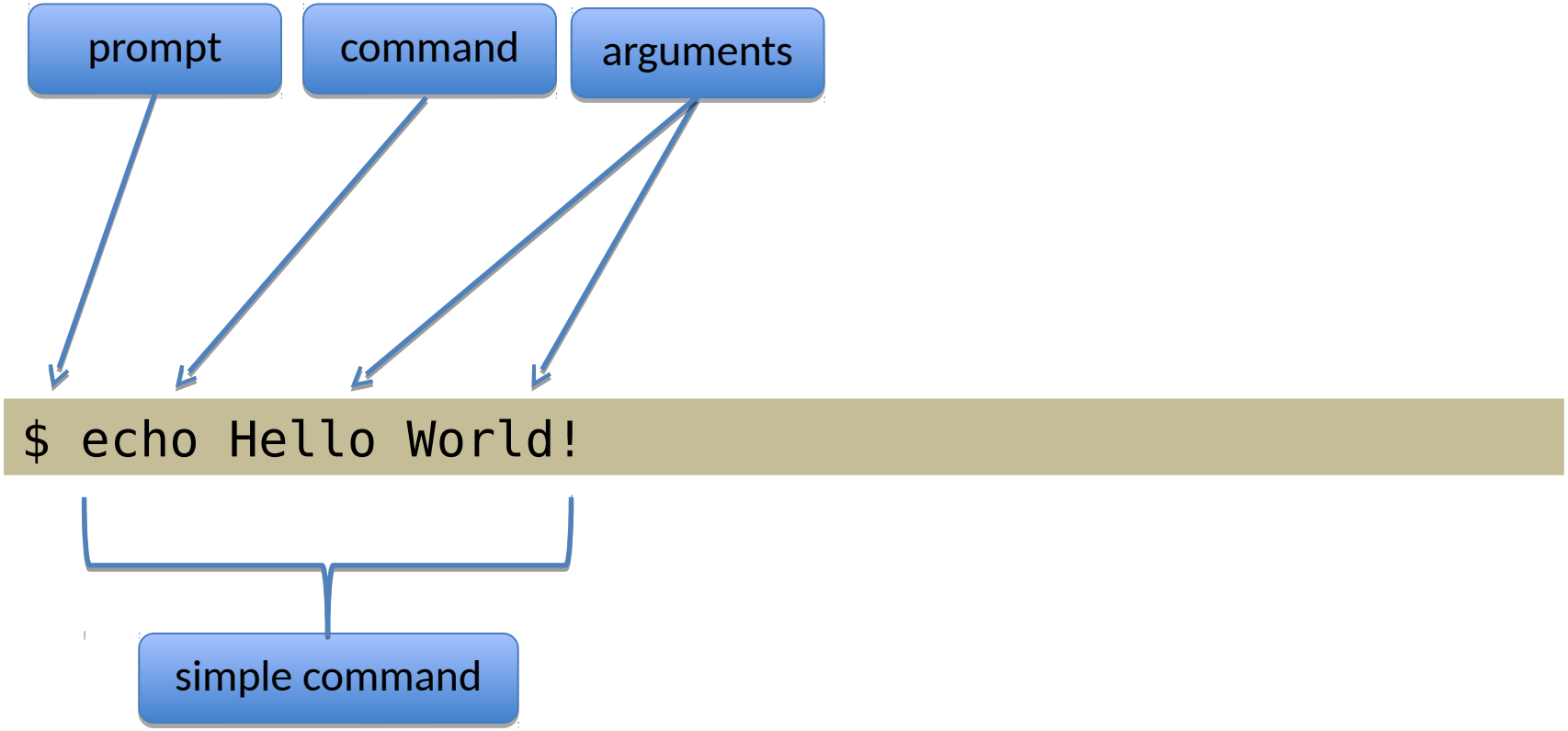
Elemental Bash

- Bash is a “command processor”
- Bash interprets characters typed on the command line and *tells* the kernel what programs to use and how to run them
- AKA command line interpreter (CLI)



POTPOURRI OF COMMANDS

Simple Command



Essential *nix Commands

cd	ls	pwd	file	wc	find	du
chmod	touch	mv	cp	rm	cut	paste
sort	split	cat	grep	head	tail	less
more	sed	awk	diff	comm	ln	
mkdir	rmdir	df	pushd	popd		
date	exit	ssh	rsh	printenv	time	echo
ps	jobs	[CTRL-c]	[CTRL-z]	[CTRL-d]	top	kill
apropos	info	man	help	type		

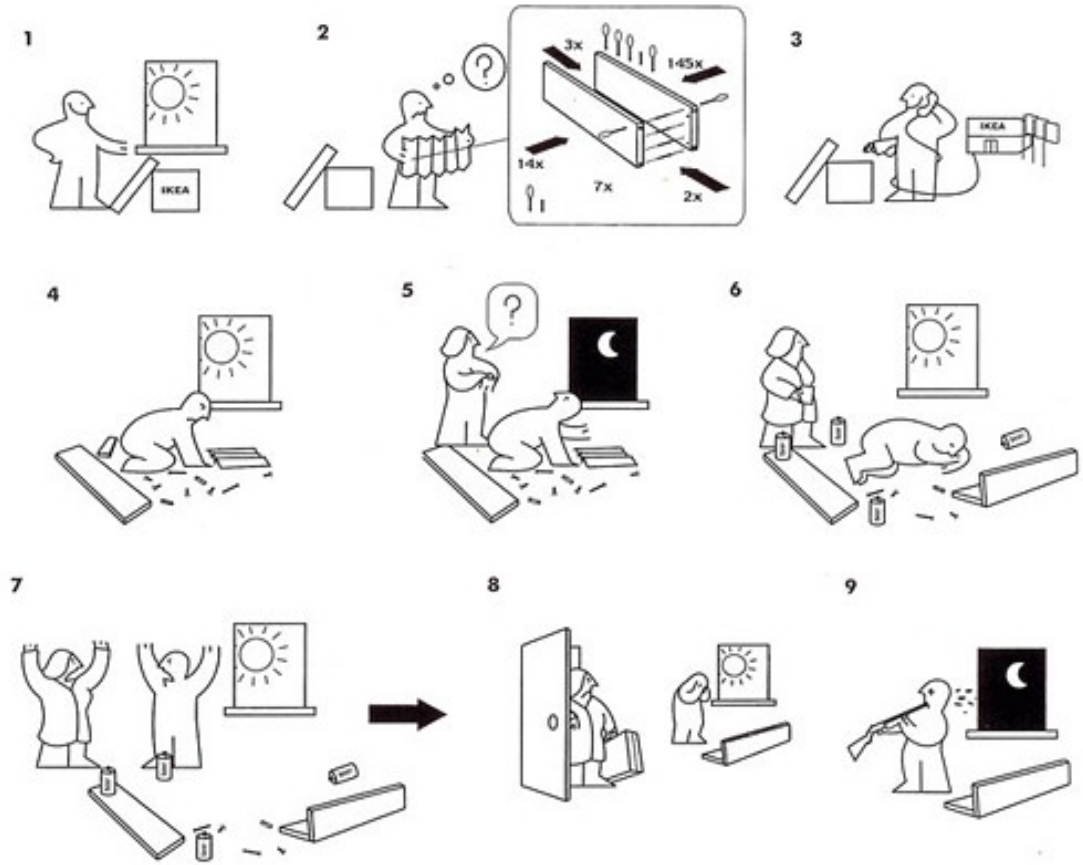
Comprehensive List

- https://hpc.nih.gov/training/handouts/BashScripting_LinuxCommands.pdf

Exercise:

- Type a command that lists all the files in your /home directory and shows file sizes and access permissions

- Make a directory, go into it, and create a file, edit the file, then delete the file



DOCUMENTATION

Documentation

- [google]
- apropos
- info
- man
- help
- type

Documentation: `apropos`

- `apropos` is high level index of commands

```
$ apropos sleep
Time::HiRes(3pm)          - High resolution alarm,
sleep, gettimeofday, interval timers
sleep(1)                  - suspend execution for an
interval of time
sleep(3)                  - suspend thread execution
for an interval measured in seconds
usleep(3)                 - suspend thread execution
for an interval measured in microseconds
```

Documentation: `info`

- `info` is a comprehensive commandline documentation viewer
- displays various documentation formats

```
$ info kill
$ info diff
$ info read
$ info strftime
$ info info
```

Documentation: man

```
$ man pwd
$ man diff
$ man head
```

Documentation: type

- type is a builtin that displays the type of a word

```
$ type -t rmdir
file
$ type -t if
keyword
$ type -t echo
builtin
$ type -t module
function
$ type -t ls
alias
```

builtin

- Bash has built-in commands (`builtin`)
- `echo`, `exit`, `hash`, `printf`

Documentation: `builtin`

- Documentation can be seen with `help`

```
$ help
```

- Specifics with `help [builtin]`

```
$ help pwd
```

Non-Bash Commands

- Found in `/bin`, `/usr/bin`, and `/usr/local/bin`
- Some overlap between Bash `builtin` and external executables

```
$ help time
$ man time
```

```
$ help pwd
$ man pwd
```

SCRIPTS

Why write a script?

- When one-liners are not enough
- In need of quick and dirty prototypes
- Maintain library of functional tools for re-use
- Provide glue to string other apps together
- Batch system submission to one of the world's fastest super computers

What is in a Script?

- Commands
- Variables
- Functions
- Loops
- Conditional statements
- Comments and documentation
- Options and settings

My Very First Script

- create a script and inspect its contents:

```
echo 'echo Hello World!' > hello_world.sh
cat hello_world.sh
```

- call it with bash:

```
bash hello_world.sh
```

- results:

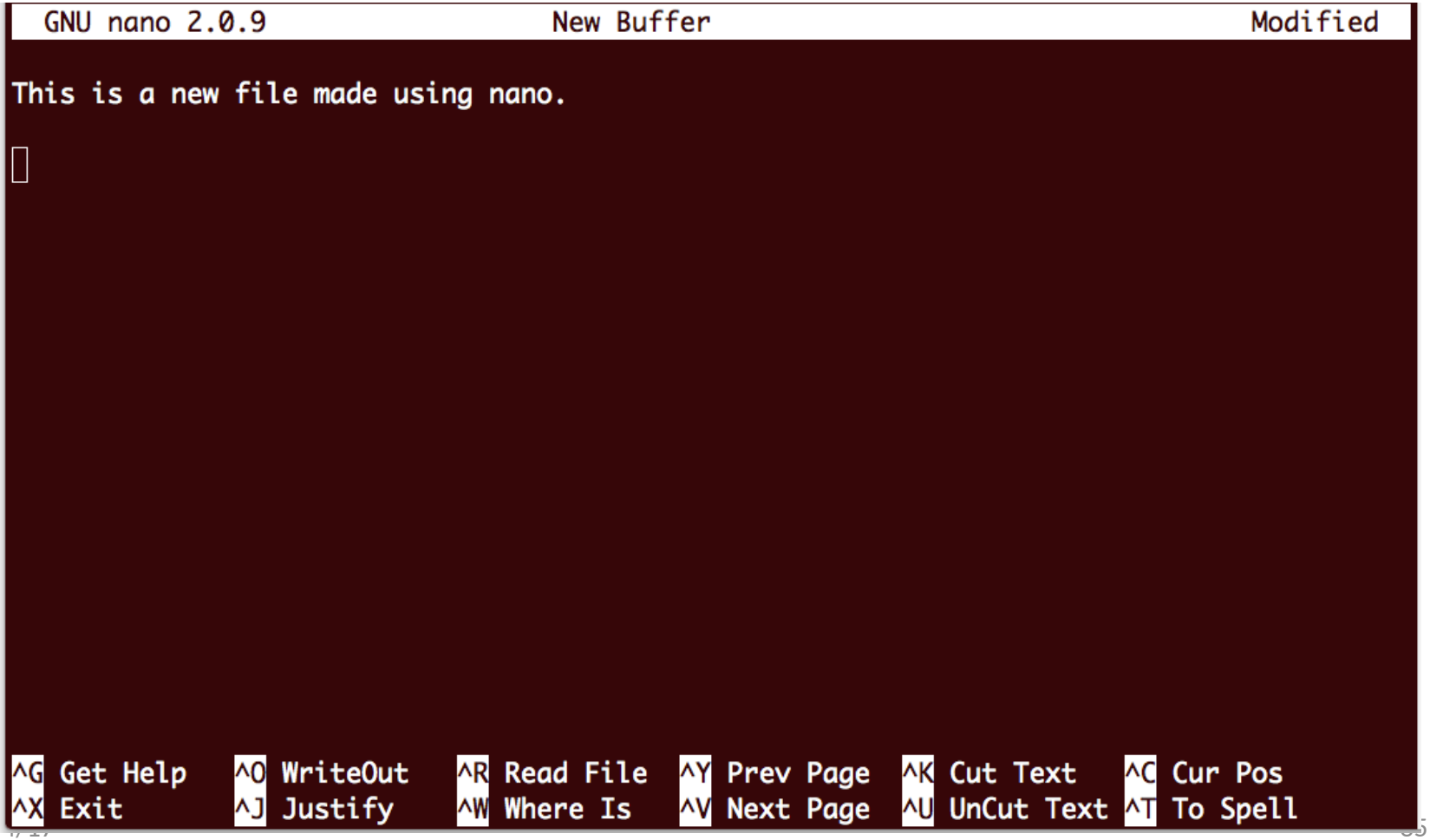
```
$ bash hello_world.sh
Hello World!
```

Multiline file creation

- create a multiline script:

```
$ cat << ALLDONE > hello_world_multiline.sh
> echo Hello World!
> echo Yet another line.
> echo This is super exciting!
> ALLDONE
$
$ bash hello_world_multiline.sh
Hello World!
Yet another line.
This is super exciting!
$
```

`nano` file editor



Editors: Do you have a favorite?

- **`SciTE`**
- **`gedit`**
- **`nano`** and **`pico`**
- **`vi`**, **`vim`**, and **`gvim`**
- **`emacs`**
- Use **`dos2unix`** if text file created on Windows machine

Execute with a Shebang!

- If bash is the default shell, making it executable allows it to be run directly

```
chmod +x hello_world.sh
./hello_world.sh
```

- Adding a shebang specifies the scripting language:

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

Debugging

- Call bash with **`–x –v`**

```
bash –x –v hello_world.sh
```

```
#!/bin/bash –x -v
echo Hello World!
```

- **`–x`** displays commands and their results
- **`–v`** displays everything, even comments and spaces

Exercise:

- Create a script to run a simple command

  (for example use builtin commands such as: `echo, ls, date, sleep...`)
- Run the script

SETTING THE ENVIRONMENT

Exercise:

- Type a command that lists all the files in your /home directory and shows file sizes and access permissions

- Make a directory, go into it, and create a file, edit the file, then delete the file

Environment

- A set of variables and functions recognized by the kernel and used by most programs
- Not all variables are environment variables, must be `exported`
- Initially set by **startup files**
- `printenv` displays variables and values in the environment
- `set` displays ALL variable

Variables

- A *variable* is a construct to hold information, assigned to a *word*
- Variables are initially *undefined*
- Variables have *a key-value pair format*
- Variables play critical roles shell scripts

Set a variable: Key-Value Pair KVP

- Formally done using `declare`

```
declare myvar=100
```

- Very simple, no spaces

```
myvar=10
```

Set a variable

- Examine value with `echo` and `$`:

```
$ echo $myvar
10
```

- This is actually called **parameter expansion**

export

- `export` is used to set an environment variable:

```
MYENVVAR=10
export MYENVVAR
printenv MYENVVAR
```

- You can do it on a single line

```
export MYENVVAR=10
```

`unset` a variable

- `unset` is used for this

```
unset myvar
```

- Can be used for environment variables

```
$ echo $HOME
/home/user
$ unset HOME
$ echo $HOME

$ export HOME=/home/user
```

Bash Variables

- `$HOME` = **`/home/[user]`** = **`~`**
- `$PWD` = current working directory
- `$PATH` = list of filepaths to look for commands
- `$CDPATH` = list of filepaths to look for directories
- `$TMPDIR` = temporary directory (`/tmp`)
- `$RANDOM` = random number
- Many, many others…

printenv

```
$ printenv
HOSTNAME=helix.nih.gov
TERM=xterm
SHELL=/bin/bash
HISTSIZE=500
SSH_CLIENT=96.231.6.99 52018 22
SSH_TTY=/dev/pts/274
HISTFILESIZE=500
USER=student1
```

Exercise:

- Set an environment variable
- Look at all your environment variables

ALIASES AND FUNCTIONS

Aliases

- A few aliases are set by default

```
$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto’
alias edit='nano'
```

- Can be added or deleted

```
$ unalias ls
$ alias ls='ls –CF’
```

- Remove all aliases

```
$ unalias -a
```

Aliases

- Aliases belong to the shell, but *NOT* the environment
- Aliases can *NOT* be propagated to subshells
- Limited use in scripts
- Only useful for interactive sessions

Functions: What are they?

- functions are a defined **set** of commands assigned to a word

```
function status { date; uptime; who | grep $USER; checkquota; }
```

```
$ status
Thu Aug 18 14:06:09 EDT 2016
 14:06:09 up 51 days,  7:54, 271 users,  load average: 1.12, 0.91, 0.86
user pts/128      2013-10-17 10:52 (128.231.77.30)
Mount                        Used      Quota  Percent    Files    Limit
/data:                    92.7 GB   100.0 GB   92.72%   233046  6225917
/home:                     2.2 GB     8.0 GB   27.48%     5510      n/a
```

Functions

- display what functions are set:

```
declare -F
```

- display code for function:

```
declare -f status
```

```
status ()
{
    date;
    uptime;
    who | grep --color $USER;
    checkquota
}
```

Functions

- functions can propagate to child shells using `export`

```
export -f status
```

Functions

- `unset` deletes function

```
unset status
```

Functions

- local variables can be set using the "`local`" keyword within a function

Exercise:

- Find out what aliases you are currently using
- Do you like them? Do you want something different?
- What functions are available to you?

LOGIN

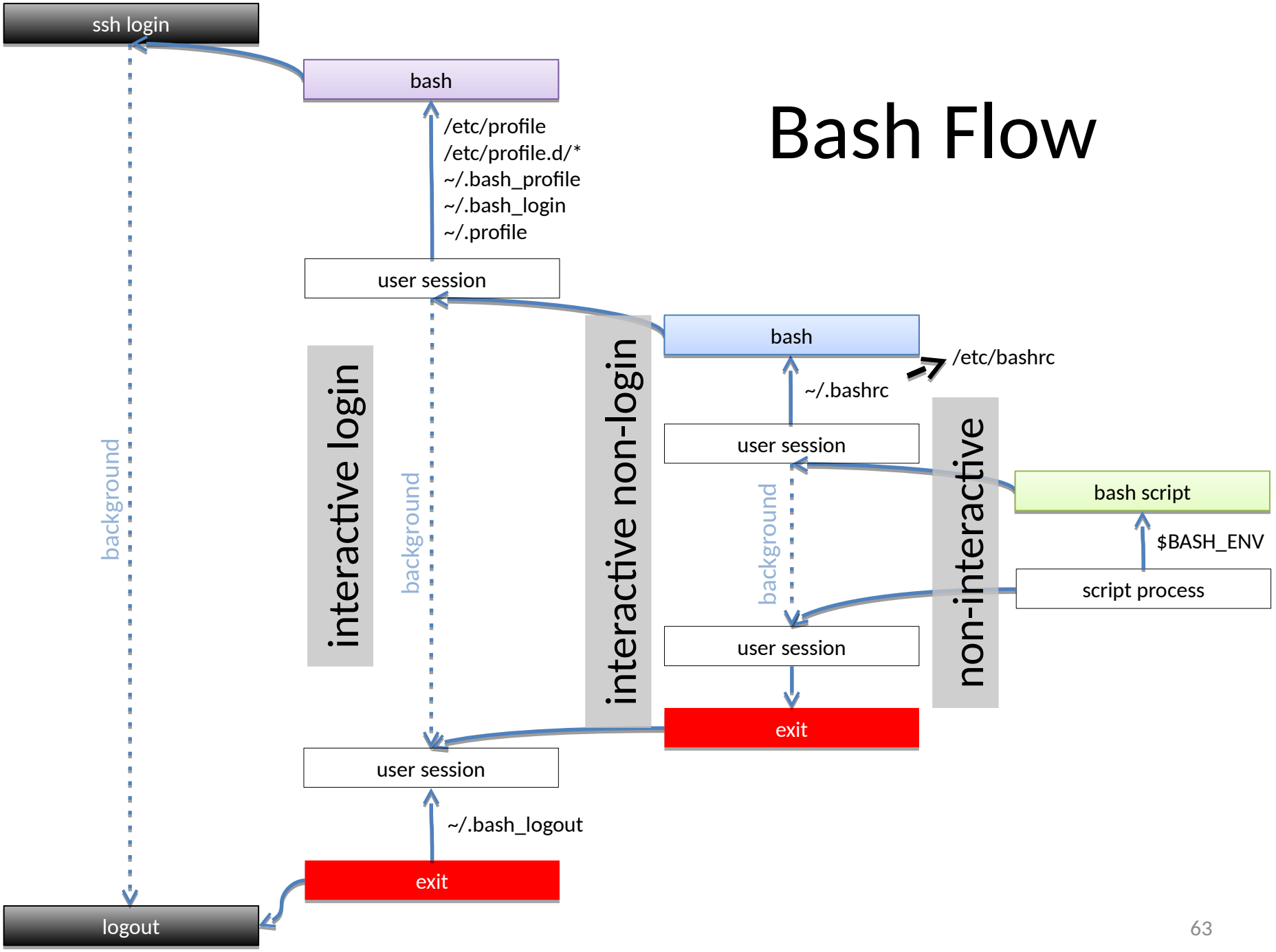
Logging In

- `ssh` is the default login client

```
$ ssh $USER@helix.nih.gov
```

- what happens next?

Bash Flow


ssh login
bash
/etc/profile
/etc/profile.d/*
~/.bash_profile
~/.bash_login
~/.profile
user session
interactive login
background
background
interactive non-login
bash
/etc/bashrc
~/.bashrc
user session
background
non-interactive
bash script
$BASH_ENV
script process
user session
exit
user session
~/.bash_logout
exit
logout

Logging In

- Interactive login shell (`ssh` from somewhere else)

`/etc/profile`
`~/.bash_profile`

`~/.bash_logout` (when exiting)

- The startup files are `sourced`, not executed

`source` and `.`

- `source` executes a file in the current shell and preserves changes to the environment
- `.` is the same as `source`

~/.bash_profile

```
cat ~/.bash_profile
```

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

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
```

Non-Login Shell

- Interactive non-login shell (calling bash from the commandline)
- The retains/inherits environment from login shell

  `~/.bashrc`

- Shell levels seen with $SHLVL

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
```

~/.bashrc

```
cat ~/.bashrc
```

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
   . /etc/bashrc
fi

# User specific aliases and functions
```

Non-Interactive Shell

- From a script
- Retains environment from login shell

  `$BASH_ENV` (if set)

- Set to a file like `~/.bashrc`

SIMPLE COMMANDS

Definitions

Command

word

Simple command / process

command arg1 arg2 …

Pipeline / Job

simple command 1 | simple command 2 |& …

List

pipeline 1 ; pipeline 2 ; …

Some command examples

- What is the current time and date?

```
date
```

- Where are you?

```
pwd
```

Process

- A *process* is an executing instance of a simple command
- Can be seen using `ps` command
- Has a unique id (*process id*, or *pid*)
- Belongs to a process group

top command

```
top - 15:51:30 up 5 days, 19:16, 240 users,  load average: 15.40, 14.51, 14.77
Tasks: 4930 total,  16 running, 4897 sleeping,  17 stopped,   0 zombie
Cpu(s):  5.0%us,  1.6%sy,  3.9%ni, 89.4%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  1058786896k total, 969744396k used, 89042500k free,    87800k buffers
Swap: 67108856k total,     2736k used, 67106120k free, 650786452k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
77108 wenxiao   39  19 63.8g  63g 1696 R 100.0  6.3 104:15.91 getAlignmentSta
98656 guptaas   39  19 4868m 1.2g  18m R 100.5  0.1   1287:38 MathKernel
254799 wenxiao  20   0 13.1g  13g 1664 R 100.0  1.3   3:48.60 getAlignmentSta
52986 wenxiao   39  19 77.0g  76g 1696 R 100.0  7.6 104:19.32 getAlignmentSta
59585 lobkovsa  39  19 24616  11m 1136 R 100.0  0.0 193:29.44 org_level_corr_
60824 hex2      39  19 28.9g  28g 4100 R 100.0  2.8   7483:20 R
245039 lobkovsa 20   0 24616  11m 1136 R 100.0  0.0  29:18.36 org_level_corr_
245092 lobkovsa 20   0 24616  11m 1136 R 100.0  0.0  29:06.75 org_level_corr_
254829 wenxiao  20   0 12.1g  11g 1664 R 100.0  1.1   3:39.88 getAlignmentSta
10184 rdmorris  20   0  531m 282m 4640 R 99.9  0.0   0:43.47 R
245080 lobkovsa 20   0 24616  11m 1136 R 99.6  0.0  29:13.30 org_level_corr_
252981 javiergc 20   0 9204m 465m  55m S 99.6  0.0   9:12.76 MATLAB
179412 sedavis  39  19 63624 7632 2876 S 11.3  0.0 442:10.21 ssh
```

ps command

```
[root@helix ~]# ps -u rdmorris -f --forest
UID         PID   PPID  C STIME TTY          TIME CMD
rdmorris 242197 242128  0 Dec12 ?        00:00:00 sshd: rdmorris@pts/107
rdmorris 242198 242197  0 Dec12 pts/107  00:00:00  \_ -bash
rdmorris 114343 114296  0 Dec12 ?        00:00:02 sshd: rdmorris@pts/251
rdmorris 114344 114343  0 Dec12 pts/251  00:00:00  \_ -bash
rdmorris  55737 114344  0 12:26 pts/251  00:00:00      \_ /bin/bash /home/rdmorris/MascotTools/M
rdmorris  55738  55737 10 12:26 pts/251  00:22:01          \_ /usr/local/R-2.13-64/lib64/R/bin/e
rdmorris  46893  46840  0 Dec12 ?        00:00:09 sshd: rdmorris@pts/112
rdmorris  46915  46893  0 Dec12 pts/112  00:00:00  \_ -bash
```

wazzup

```
[hooverdm@helix ~]$ wazzup pchines -c
  PPID     PID USER         ELAPSED %CPU %MEM COMMAND
 13575   13582 pchines  3-22:58:11  0.0  0.0 sshd
 13582   13676 pchines  3-22:58:10  0.0  0.0  \_ bash
 13676  131290 pchines       01:45 99.6  0.0       \_ MATLAB
     1  135226 pchines       01:02  0.1  0.0 gam_server
     1  134568 pchines       01:07  0.1  0.0 gconfd-2
     1  134566 pchines       01:07  0.0  0.0 dbus-daemon
     1  134565 pchines       01:07  0.0  0.0 dbus-launch
```

`wazzup` and `watch`

- `wazzup` is a wrapper script for

```
$ ps -u you --forest --width=2000 -o
ppid,pid,user,etime,pcpu,pmem,{args,comm}
```

- `watch` repeats a command every 2 seconds
- Useful for testing applications and methods

history

- The history command shows old commands run:

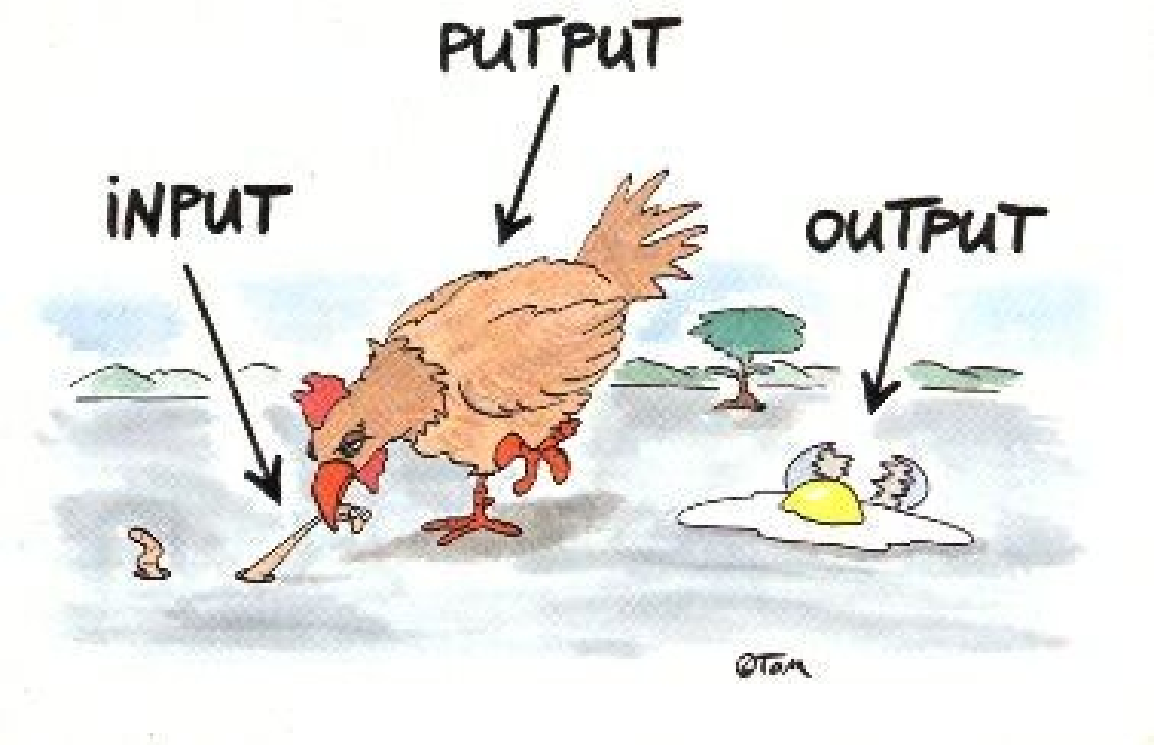
```
$ history 10
  594  12:04  pwd
  595  12:04  ll
  596  12:06  cat ~/.bash_script_exports
  597  12:06  vi ~/.bash_script_exports
  598  12:34  jobs
  599  12:34  ls -ltra
  600  12:34  tail run_tophat.out
  601  12:34  cat run_tophat.err
  602  12:34  wazzup
  603  13:54  history 10
```

history

- HISTTIMEFORMAT controls display format

```
$ export HISTTIMEFORMAT='%F %T  '
$ history 10
  596  2014-10-16 12:06:22  cat ~/.bash_script_exports
  597  2014-10-16 12:06:31  vi ~/.bash_script_exports
  598  2014-10-16 12:34:13  jobs
  599  2014-10-16 12:34:15  ls -ltra
  600  2014-10-16 12:34:21  tail run_tophat.out
  601  2014-10-16 12:34:24  cat run_tophat.err
  602  2014-10-16 12:34:28  wazzup
  603  2014-10-16 13:54:38  history 10
  604  2014-10-16 13:56:14  export HISTTIMEFORMAT='%F %T  '
  605  2014-10-16 13:56:18  history 10
```

http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x1712.htm



INPUT AND OUTPUT

Redirection

- Every process has three file descriptors (file handles): STDIN (0), STDOUT (1), STDERR (2)
- Content can be redirected

```
cmd < x.in
```

Redirect file descriptor 0 from STDIN to x.in

```
cmd > x.out
```

Redirect file descriptor 1 from STDOUT to x.out

```
cmd 1> x.out 2> x.err
```

Redirect file descriptor 1 from STDOUT to x.out, file descriptor 2 from STDERR to x.err

Combine STDOUT and STDERR

```
cmd 2>&1
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT)

- Ordering is important

Correct:

```
cmd > x.out 2>&1
```

Redirect file descriptor 1 from STDOUT to filename x.out, then redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (x.out)

Incorrect:

```
cmd 2>&1 > x.out
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT), then redirect file descriptor 1 from STDOUT to filename x.out

Redirection to a File

- Use better syntax instead – these all do the same thing:

```
cmd > x.out 2>&1
```

```
cmd 1> x.out 2> x.out
```

WRONG!

```
cmd &> x.out
```

```
cmd >& x.out
```

Redirection

- Appending to a file

```
cmd >> x.out
```
Append STDOUT to x.out

```
cmd 1>> x.out 2>&1
```
Combine STDOUT and STDERR, append to x.out

```
cmd &>> x.out
```
Alternative for concatenation and appendation

~~`cmd >>& x.out`~~
WRONG!

Exit Status

- A process returns an exit status (0-255)
- 0 = success (almost always)
- 1 = general error, 2-255 = specific error
- Stored in $? parameter

```
cat /var/audit
```

```
cat: /var/audit: Permission denied
```

```
echo $?
```

```
1
```

Brace Expansions

- Brace expansion { , , }

```
echo {bilbo,frodo,gandalf}
```

```
bilbo frodo gandalf
```

```
echo {0,1,2,3,4,5,6,7,8,9}
```

```
0 1 2 3 4 5 6 7 8 9
```

Bash Wrap Up on Day 2

Bash Shell Scripts What Exactly are They?

- Contain a series of commands
- The interpreter executes the commands in the script
- Commands that you run on the command line can be put in script
- Shell scripts are great for automating task

Variables

- Variables are basically storage locations that have a name
- You can think of variables as Name-Value pairs
- Syntax

  VARIABLE=”value”

  note no spaces before nor after the equal sign when assigning your variable.

Variables

- Variables are as most things in *nix case sensitive
- By convention variables are written in upper case
- To use a variable add the $ sign in front of the variable name

  eg. ${PATH}
- You man put variables in curly braces which are optional unless you want to append to the variable…

  lets look at fav_editor.sh

Tests

- Basic syntax for test

  [condition we want to test for]

  eg.

  $ [-e /etc/passwd]
- After entering this test condition evaluate the response by checking the exit status.
- Recall that an exit status of zero implies that the condition was met and there are “no”errors. If the condition was not met then a number other than zero would be returned.

Using the if Statement in Combination with Tests to Make Decisions in your Scripts

- Basic syntax

  if [test condition is true]

  then

    execute a command

    execute another command

    execute as many commands as you like

    fi

The if with else Combination

- It is possible to perform an action if the tested condition is not true

  Basic syntax

  if [test condition is true]

  then

  execute this command

  else

  execute this other command over here!

  fi

The if with elif and else

- Basic syntax

  if [the condition being tested evaluates to true]

  then

  execute this command

  elif [this other test condition is true]

  then execute this other command

  else

  execute this here command

The “for” Loop

- The for loop is a good option for performing an action on a list of items.
- The first line in a for loop contains the word “for” followed by a variable_name then list of items
- The second line in a for loop contains the word “do” which is followed by the command actions you want to take on the list of items
- End the for loop with the word “done”

Positional Parameters

- Positional parameters are parameters that “contain” the content of the commandline

  $my_script.sh parameter_in_position1

  $0 refers to the script itself

  $1 first parameter after the script

  $2 second parameter after the script
- Positional parameters rang from $0 - $9

Exit Status aka Return Code aka Exit Code

- All commands in bash return an exit status
- Exit codes are integers that range from 0 – 255
- By convention an exit status of 0 represents success
- A non zero exit status indicates an error condition
- Exit status maybe used for error checking
- Information on exit status for a particular command can usually be found in the man pages

Check your status on exit with $?

- The $? variable contains the return code of the previously executed command

  eg.

  $uptime

  $echo $?

What can you do with the exit status? Error Handling?

- One can use the exit status of a command to make a decision or take a different action
- That is if the exit status is good go one way if not go in another direction

  see the Bachelorette?

  Check the ping_status script

&& and ||

- && = AND
- Commands following the && will proceed only if the exit code of the previous command is zero “0”

  eg.

  mkdir /media/lego && cp ninjago.mp4 /media/lego
- || = OR

  cp ninjago.mp4 /media/lego || cp ninjago.mp4 /movies

Chaining Commands together with the semicolon

- You can chain commands together by separating them with the semicolon
- The semicolon does not check the exit status
- The commands following the semicolon will always get executed
- Separating commands with the semicolon approximates executing each command on a new line

Using the “exit” command in your shell script

- One can explicitly define the return code/exit status of ones script

  exit 0

  exit 5

  exit x; where x < 255
- The default value is that of the last command in the script if the exit code is not explicitly stated
- The exit command terminates the script

More on Functions

- Create with the function keyword within a script

  function name_of_function () {

  #code block goes here

  #this code is executed when the function is called

  }

How do we call our function in a script?

- To call a function in a script simply list the name of the function

  eg.

  ```
  #!/bin/bash
  function hello() {
          echo “Hello!”
  }
  hello
  ```

- Note one function may be used to call another function

Using Positional Parameters when making a function call within a script

- Functions will accept parameters just like scripts can
- The first parameter is stored in $1
- The second in $2 etc…
- All parameters are stored in $@
- Recall $0 is the script itself, not the function

  see The Three Stooges

Typical Script Components

- Shebang
- Comments/header information
- Global Variables
- Defined Functions
- Main contents
- Exit status

Wildcards: Jokers Wild

- A wildcard is a character or string used for pattern matching (match files or directories)
- Wildcards are often called globbs
- Globbing expands the wildcard pattern into a list of file and or directories
- Wildcards can be used with commands such as ls, rm, cp…
- The two main wildcards are * and ?
- * matches zero or more characters
- ? matches a single character

Putting Characters in Groups/Classes

- Birds of a feature
- Use the square brackets [] to define a character class
- Match characters between the brackets

  eg. [aeiou] used to match any single vowel
- [!] Matches any character NOT included between the brackets.
- [!aeiou]* will match words not starting with a vowel

Class Ranges

- Use a hyphen to separate the range of characters in a class

  eg. [a-g]* will match files that start with a,b,c,d,e,f or g
- [3-9]* will match files that start with 3,4,5,6,7,8 or 9

Predefined Named Character Classes

- [[:alpha:]]
- [[:alnum:]]
- [[:digit:]]
- [[:lower:]]
- [[:upper:]]
- [[:space:]]

  Wild cards maybe used in shell scripts :)

The Case Statements

- To create a “case” statement start with the word “case” followed by an expression/variable end the line with the word “in”

  case $VAR in

- Next list a pattern you want to test against the variable ending with a parentheses

  pattern-A)

  #command to the executed

  ;;

Case

Pattern-B)

#command to be executed if pattern matches pattern-B

;;

- End the case statement with “esac”

see Ferrari vs. Bugatti

while loops

- A while loop is a loop that repeats a series of commands as long as a condition is true
- The condition could be a test such as if a variable has a certain value
- Checking for the existence of a file
- Checking any command and it's exit status

Syntax of a while loop

- $while [condition is true]
  do
      #execute the commands
  done

  see forever

Questions? Comments?

staff@hpc.nih.gov