Revised: 2/22/2020

BASH and Basic Scripting

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Command Input and Output
  - Redirecting Command Output
  - Redirecting Command Input
  - Piping Command Output
    - Filter Commands
- Shell Variables
  - Environment Variables
  - User-Defined Variables
  - Aliases
  - Environment Files
- Shell Scripts
  - Escape Sequences
  - Reading Input
  - Logic and Decisions
  - Loops
  - Special Variables
- Source Code Version Control

Command Input and Output

- The input and output of a command are represented by numeric identifiers called **file descriptors**

- Three basic file descriptors in the BASH shell:

Name	File Descriptor	Usage
Standard Input (stdin)	**0**	Keyboard input
Standard Output (stdout)	**1**	A command's output
Standard Error (stderr)	**2**	A command's error message(s)

Redirecting Command Output

- **Redirection** is the process of writing the output (stdout and/or stderr) of a command to a file instead of to the terminal screen.

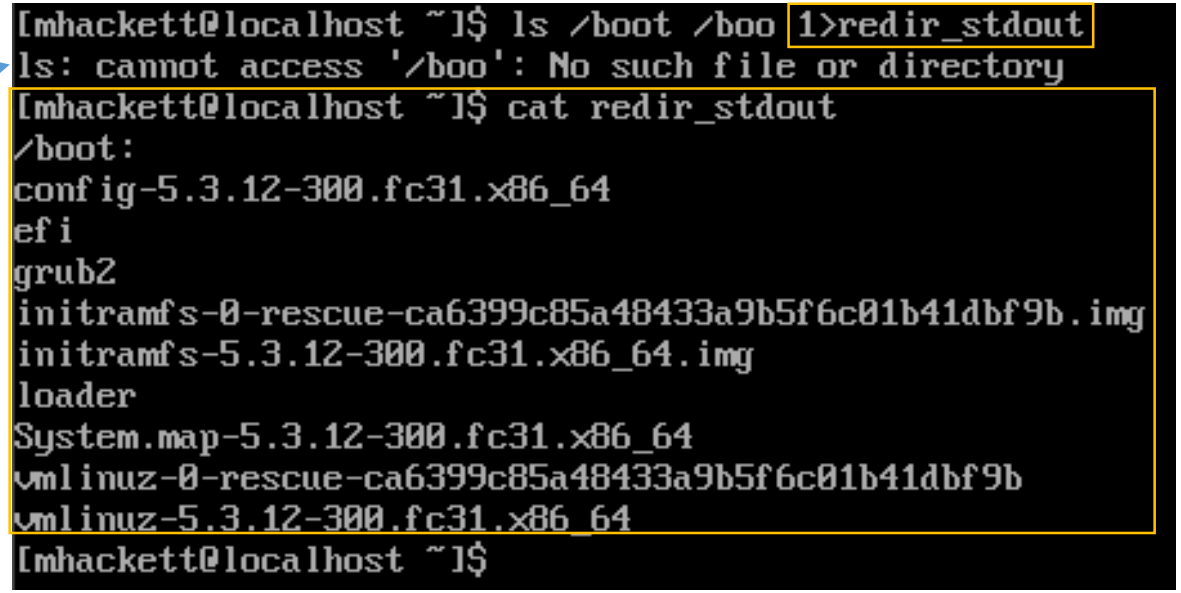
- The shell metacharacter for redirection is the **>** symbol

- The file descriptor is placed before the redirection symbol
  - For example: **1>** will redirect stdout

Redirecting Command Output

- The standard output of **`ls /boot /boo`** is redirected to a file named redir_stdout
- Standard error is still displayed to the terminal.

/boo does not exist



Redirecting Command Output

- The standard error of **`ls /boot /boo`** is redirected to a file named redir_stderr
- Standard output is still displayed to the terminal.

```
[mhackett@localhost ~]$ ls /boot /boo 2>redir_stderr
/boot:
config-5.3.12-300.fc31.x86_64
efi
grub2
initramfs-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b.img
initramfs-5.3.12-300.fc31.x86_64.img
loader
System.map-5.3.12-300.fc31.x86_64
vmlinuz-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b
vmlinuz-5.3.12-300.fc31.x86_64
[mhackett@localhost ~]$ cat redir_stderr
ls: cannot access '/boo': No such file or directory
[mhackett@localhost ~]$
```

Redirecting Command Output

- If the redirection does not have a file descriptor, standard output is assumed.

```
ls /boot /boo >output_file
```

is equivalent to

```
ls /boot /boo 1>output_file
```

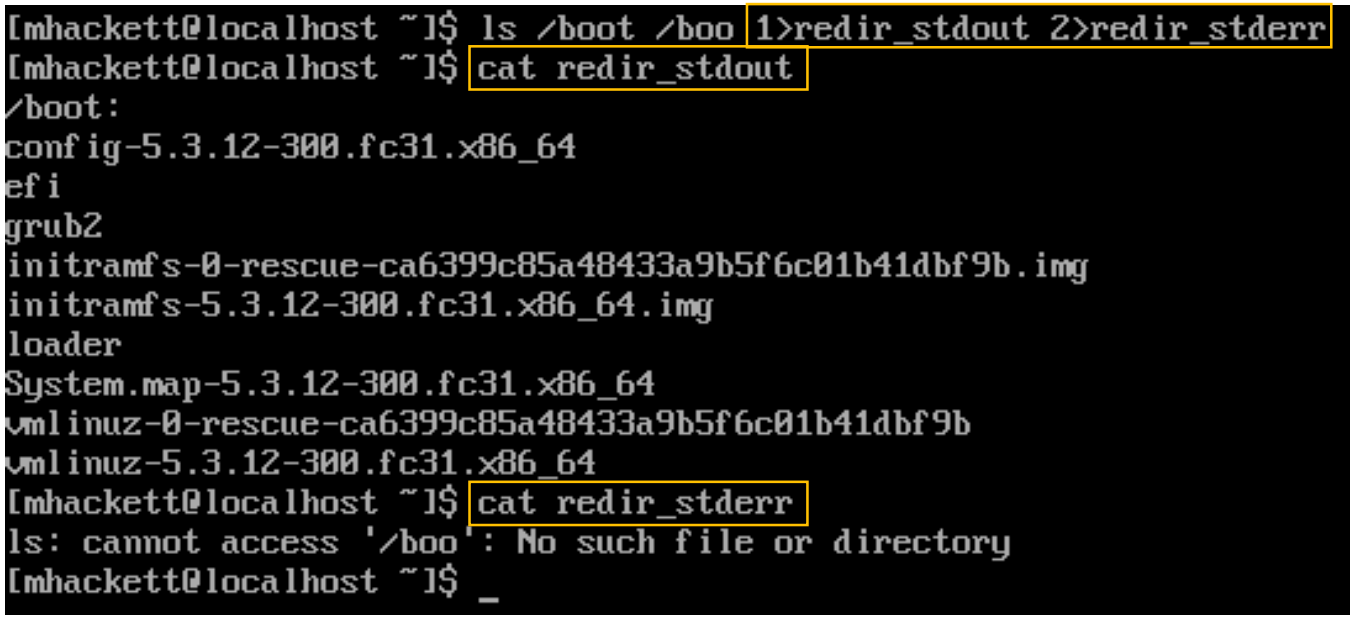
Redirecting Command Output

- If the file stdout or stderr is redirected to *does not* already exist, the file will be created.

- If the file stdout or stderr is redirected to *does* already exist, the existing contents of the file will be **erased**.

Redirecting Command Output

- The standard output of **`ls /boot /boo`** is redirected to a file named redir_stdout *and* the standard error is redirected to a file named redir_stderr



Redirecting Command Output

- To redirect *both* stdout and stderr of **`ls /boot /boo`** to a file named redir_allout:

```
[mhackett@localhost ~]$ ls /boot /boo 1>redir_allout 2>&1
[mhackett@localhost ~]$ cat redir_allout
ls: cannot access '/boo': No such file or directory
/boot:
config-5.3.12-300.fc31.x86_64
efi
grub2
initramfs-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b.img
initramfs-5.3.12-300.fc31.x86_64.img
loader
System.map-5.3.12-300.fc31.x86_64
vmlinuz-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b
vmlinuz-5.3.12-300.fc31.x86_64
[mhackett@localhost ~]$ _
```

Redirecting Command Output

- To redirect *both* stdout and stderr of **`ls /boot /boo`** to a file named redir_allout (**alternative**):

```
[mhackett@localhost ~]$ ls /boot /boo &>redir_allout
[mhackett@localhost ~]$ cat redir_allout
ls: cannot access '/boo': No such file or directory
/boot:
config-5.3.12-300.fc31.x86_64
efi
grub2
initramfs-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b.img
initramfs-5.3.12-300.fc31.x86_64.img
loader
System.map-5.3.12-300.fc31.x86_64
vmlinuz-0-rescue-ca6399c85a48433a9b5f6c01b41dbf9b
vmlinuz-5.3.12-300.fc31.x86_64
[mhackett@localhost ~]$
```

Redirecting Command Output

- The following redirects the standard output of the **`date`** command to a file named date_file.

```
[mhackett@localhost ~]$ date >date_file
[mhackett@localhost ~]$ cat date_file
Sat 15 Feb 2020 02:07:40 PM EST
[mhackett@localhost ~]$ _
```

Redirecting Command Output

- To prevent the original contents of a file from being erased, use **>>** instead of **>**

```
[mhackett@localhost ~]$ date >date_file
[mhackett@localhost ~]$ cat date_file
Sat 15 Feb 2020 02:07:40 PM EST
[mhackett@localhost ~]$ date >>date_file
[mhackett@localhost ~]$ cat date_file
Sat 15 Feb 2020 02:07:40 PM EST
Sat 15 Feb 2020 02:13:11 PM EST
[mhackett@localhost ~]$
```

Redirecting Command Input

- Redirection can also be used to specify input sources (like files) in place of keyboard input
- The shell metacharacter for *input* redirection is the **<** symbol
- No file descriptor is needed since there is only standard input

Redirecting Command Input

- The **`tr`** command is used to **tr**ansform characters of an input source

- This command requires the source to be redirected to standard input

Redirecting Command Input

- Trying to use the **tr** command to change any 2's to X's:

```
[mhackett@localhost ~]$ tr 2 X date_file
tr: extra operand ‘date_file’
Try 'tr --help' for more information.
```

- Does not work because the source needs to be redirected to standard input:

```
[mhackett@localhost ~]$ tr 2 X date_file
tr: extra operand ‘date_file’
Try 'tr --help' for more information.
[mhackett@localhost ~]$ tr 2 X <date_file
Sat 15 Feb X0X0 0X:07:40 PM EST
Sat 15 Feb X0X0 0X:13:11 PM EST
```

Redirecting Command Input

- This command will redirect the output of the **tr** command to a file named tr_date:

```
[mhackett@localhost ~]$ tr 2 X <date_file >tr_date
[mhackett@localhost ~]$ cat tr_date
Sat 15 Feb X0X0 0X:07:40 PM EST
Sat 15 Feb X0X0 0X:13:11 PM EST
[mhackett@localhost ~]$
```

Piping Command Output

- **Piping** is the process of using the output of one command as the input of another command.

- The shell metacharacter for piping is the **|** symbol
  - Usually made using the Shift+\ keys

Piping Command Output

- The following command pipes the output of the **mount** command as the input to the **grep** command

```
[mhackett@localhost ~]$ mount | grep sdb
/dev/sdb1 on /mnt/drive1 type ext3 (rw,relatime,seclabel)
/dev/sdb2 on /mnt/drive2 type ext4 (rw,relatime,seclabel)
/dev/sdb3 on /mnt/drive3 type xfs (rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota
)
[mhackett@localhost ~]$
```

Piping Command Output

- The following command pipes the output of the **`ls`** command as the input to the **`less`** command
  - Useful for stepping through long outputs

```
[mhackett@localhost ~]$ ls -l /usr/bin | less
```

Filter Commands

- A **filter** is a command can accept the standard output of another command as its input.
  - These commands can be used between two pipes
- Commands like **`mount`** and **`ls`** are not filters
  - They do not accept the output of other commands as their inputs
- Commands like **`tr`**, **`grep`**, **`tail`**, **`more`**, and **`less`** are filters
  - They do accept the output of other commands as their inputs

Filter Commands

- The **`sort`** command displays the contents of a file in ascending order, based on the first character in each line.
- Usage:

  `sort` *`filename`*

- The -r option sorts the lines in reverse order.

  `sort -r` *`filename`*

Filter Commands

- The **wc** command displays a **w**ord **c**ount, line count, and character count on the contents of a file.
- Usage:

  wc *filename*

- The -l option displays only a line count.

  wc -l *filename*
- The -w option displays only a word count.

  wc -w *filename*
- The -c option displays only a character count.

  wc -c *filename*

Filter Commands

- The **pr** command displays a file formatted for **pr**inting.
  - It places a date and page number at the top of each page.
- Usage:

  pr *filename*

- The -d option displays the file formatted with double spacing.

  pr -d *filename*

Filter Commands

- The **tee** command sends its input to a specified file as well as to standard output.
  - In essence is writes its input to the file **and** to the terminal screen

- Usage:

  tee *filename*

- The -a option appends the input to the file if the file has existing data that should not be overwritten/erased.

  tee -a *filename*

Filter Commands

- The **`cut`** command displays specific columns of delimited data.
  - A **delimiter** is a character (or characters) that specify individual items or "tokens" of data contained in each line of a file.

- Usage:

  `cut –d`*`delimiter`* **`-f`***`columns`* *`filename`*

- For example:

  `cut –d: -f6,7 somefile.txt`

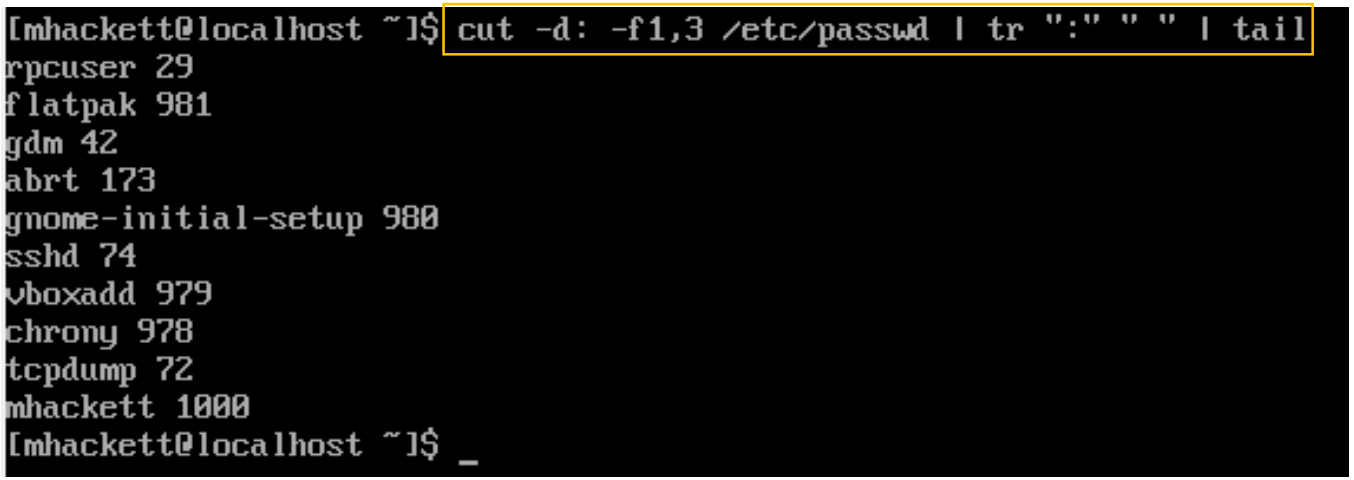
  - Would use a colon as a delimiter between tokens and display only the 6th and 7th tokens from each line.

Filter Commands

```
[mhackett@localhost ~]$ tail /etc/passwd
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
flatpak:x:981:979:User for flatpak system helper:/:/sbin/nologin
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
abrt:x:173:173::/etc/abrt:/sbin/nologin
gnome-initial-setup:x:980:978::/run/gnome-initial-setup/:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
vboxadd:x:979:1::/var/run/vboxadd:/sbin/nologin
chrony:x:978:976::/var/lib/chrony:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
mhackett:x:1000:1000:Michael Hackett:/home/mhackett:/bin/bash
[mhackett@localhost ~]$ cut -d: -f1,3 /etc/passwd | tail
rpcuser:29
flatpak:981
gdm:42
abrt:173
gnome-initial-setup:980
sshd:74
vboxadd:979
chrony:978
tcpdump:72
mhackett:1000
[mhackett@localhost ~]$
```

Filter Commands

- Here, the **cut** command is used to get the 1st and 3rd column from the /etc/passwd file (using : as the delimiter)
  - The **cut** command's output is piped as the input to the **tr** command, which replaces the colons with a space.
  - The **tr** command's output is piped as the input to the **tail** command, which displays only the last 10 lines



Filter Commands

- The **sed** and **awk** commands are powerful tools for manipulating text.
  - An entire 3-credit course could be dedicated just on using **sed** and **awk**

- We'll see only a few examples here

Filter Commands

- The syntax to perform a find an replace operation using sed is:

  sed s/*original/replacement file*

- The following command would replace the first occurrence of “the” with “THE” in each line of the file example.txt and display the results to the terminal.

  sed s/the/THE example.txt

Filter Commands

- The following command would replace the first occurrence of “the” with “THE” in each line of the file example.txt and redirects the output to a file named newfile.txt

```
sed s/the/THE example.txt >newfile.txt
```

- The following command would replace **every** occurrence of “the” with “THE” in each line of the file example.txt and redirects the output to a file named newfile.txt

```
sed s/the/THE/g example.txt >newfile.txt
```

Filter Commands

- The following command would replace every occurrence of “the” with “THE” in any line of the file example.txt that contains the word “love” and redirects the output to a file named newfile.txt

```
sed /love/s/the/THE/g example.txt >newfile.txt
```

- The following command would replace every occurrence of “the” with “THE” on lines 5 through 9 of the file example.txt and redirects the output to a file named newfile.txt

```
sed 5,9s/the/THE/g example.txt >newfile.txt
```

Filter Commands

- The following command would delete any lines that contain the word “the” in the file example.txt and redirects the output to a file named newfile.txt

```
sed /the/d example.txt >newfile.txt
```

Filter Commands

- Unlike sed, awk treats each line of a file as a database record
  - Each token is separated by a space
  - Each token is represented as the variables $1 (the first token), $2 (the second token), etc.

- This command would display the fourth and eighth tokens in each line that contains the word “the” in the file example.txt and redirects the output to newfile.txt

```
awk '/the/ {print $4, $8}' example.txt >newfile.txt
```

Shell Variables

- BASH maintains variables of data/information.
  - A **variable** is an identifier that refers to data stored in the computer's memory
- **Environment variables** are set by the system
  - Variables that are regularly used by processes and programs
- **User-defined variables** are set by the user, and are not used by the system.

Environment Variables

- The **`set`** command is used to display the current values of the environment variables.
  - There are normally a lot of environment variables, so it is advisable to pipe the output of **`set`** to the **`more`** or **`less`** commands:

```
set | less
```

Environment Variables

- One of the most important environment variables is the PATH variable.
  - This specifies the locations where BASH looks for programs to execute.

- To show the current value of an environment variable, the **echo** command is used.
  - The environment variable is preceded by a $

```
echo  $PATH
```

Environment Variables

```
[mhackett@localhost ~]$ echo $PATH
/home/mhackett/.local/bin:/home/mhackett/bin:/usr/share/Modules/bin:/usr/local/bin:/usr/bin:/usr/loc
al/sbin:/usr/sbin
[mhackett@localhost ~]$
```

- On this system, BASH will look in the following locations (in the order listed) for programs to execute:
  - /home/mhackett/.local/bin
  - /home/mhackett/bin
  - /usr/share/Modules/bin
  - /usr/local/bin
  - /usr/bin
  - /usr/local/sbin
  - /usr/sbin

Environment Variables

- Using the **`which echo`** command shows the echo program is in the /usr/bin directory.

- Since /usr/bin is in the PATH, we can issue the **`echo`** command using only its name, instead of needing to provide the absolute path to it
  - Both commands below will display the value of the HOME environment variable

```
echo $HOME
/usr/bin/echo $HOME
```

User-Defined Variables

- New variables can be created by a user
- The syntax for creating a user-defined variable is the variable name, immediately followed by **=**, immediately followed by the value
  - Existing user-defined or environment variables can be changed in the same manner

```
[mhackett@localhost ~]$ MYVAR="My Example Variable"
[mhackett@localhost ~]$ echo $MYVAR
My Example Variable
[mhackett@localhost ~]$
```

User-Defined Variables

- Variables may only contain alphanumeric characters (A-Z, 0-9), hyphens, and underscores
- Cannot start with a number
- Conventionally uses uppercase letters instead of lowercase characters

User-Defined Variables

```
[mhackett@localhost ~]$ MYVAR="My Example Variable"
[mhackett@localhost ~]$ echo $MYVAR
My Example Variable
[mhackett@localhost ~]$
```

- Variables created like the one above are only available to the current shell
  - Most programs run in a *subshell* and would not have access to this variable

User-Defined Variables

- The **`export`** command is used to make variables accessible to subshells

- The **`env`** command is used to display all exported environment and user-defined variables

User-Defined Variables

```
[mhackett@localhost ~]$ MYVAR="My Example Variable"
[mhackett@localhost ~]$ echo $MYVAR
My Example Variable
[mhackett@localhost ~]$ env | grep MYVAR
[mhackett@localhost ~]$ export MYVAR
[mhackett@localhost ~]$ env | grep MYVAR
MYVAR=My Example Variable
[mhackett@localhost ~]$ export MYVAR2="My Other Variable"
[mhackett@localhost ~]$ env | grep MYVAR
MYVAR=My Example Variable
MYVAR2=My Other Variable
[mhackett@localhost ~]$
```

User-Defined Variables

- The **`unset`** command is used to remove a variable

```
[mhackett@localhost ~]$ env | grep MYVAR
MYVAR=My Example Variable
MYVAR2=My Other Variable
[mhackett@localhost ~]$ unset MYVAR2
[mhackett@localhost ~]$ env | grep MYVAR
MYVAR=My Example Variable
```

Aliases

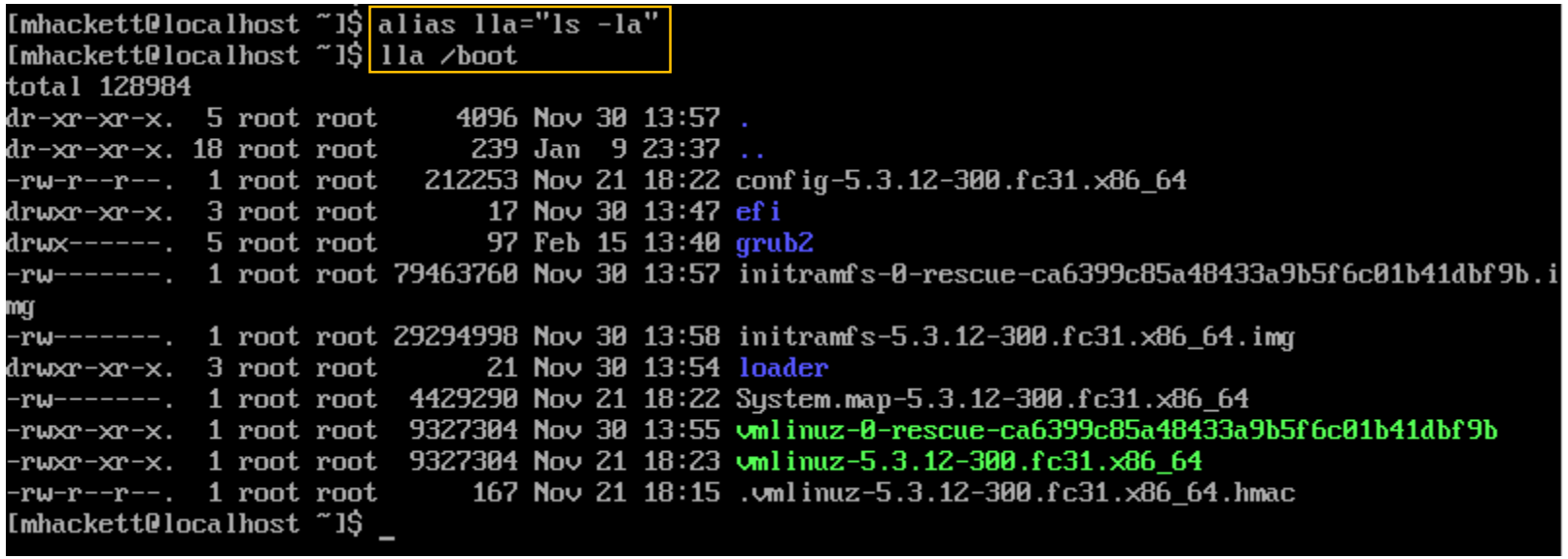
- An **alias** is a special variable that acts as shortcut for a longer command

- The syntax for creating an alias is the **`alias`** command, followed by a space and the alias name, immediately followed by **=**, immediately followed by the command
  - Existing aliases can be changed in the same manner

Aliases

- This command creates an alias called “lla” that, when used, performs a long directory listing that includes hidden files:

```
alias lla="ls -la"
```



Aliases

- Using the **`alias`** command by itself will list all aliases

```
[mhackett@localhost ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias lla='ls -la'
alias ls='ls --color=auto'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias --read-functions --show-ti
lde --show-dot'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
[mhackett@localhost ~]$
```

Aliases

- The **`unalias`** command will remove an alias

```
[mhackett@localhost ~]$ unalias lla
[mhackett@localhost ~]$ lla
bash: lla: command not found...
[mhackett@localhost ~]$ _
```

Environment Files

- When a user exits or logs out of a BASH shell, any variables or aliases created in that shell will be deleted

- **Environment files** contain variables and aliases that will be loaded any time a new BASH shell is started

Environment Files

- Common environment files and the order in which they are read when a new shell starts:

  /etc/profile

  /etc/profile.d/*

  /etc/bashrc

  ~/.bashrc

  ~/.bash_profile

  ~/.bash_login

  ~/.profile

Environment Files

- The /etc/bashrc file contains aliases and environment variables for **all** users when they log in
- The ~/.bashrc file contains aliases and environment variables for that specific user
- These files are read when:
  - The user logs in
  - The user creates a new BASH shell after having logged in

Environment Files

- The /etc/profile file and the files in the /etc/profile.d directory contain aliases and environment variables for **all** users when they log in

- The ~/.bash_profile, ~/.bash_login, and ~/.profile files contain aliases and environment variables for that specific user
  - Normally, only one of these are used
  - If more than one exist, the first one found is used

- These files are read when:
  - The user logs in

Environment Files

- Another environment file used is ~/.bash_logout

- The commands listed in this file are executed when the shell is exited/when the user logs out

Shell Scripts

- A **shell script** is a file that contains executable commands and constructs that can be interpreted and executed by a shell
- Shell scripts are essentially programs that are commonly used to perform administrative Linux tasks
- Any command that can be entered on the command line can also be placed in the shell script

Shell Scripts

- The following an example of a BASH shell script
  - These lines would be contained in a text file

```
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
date
ls /
```

Shell Scripts

- The first line specifies the path to the shell that is to interpret the script's commands
  - Always begins with **#!** which is called "hashpling" or "shebang"

```
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
date
ls /
```

Shell Scripts

- Lines that begin with # (not followed by !) are **comments**
  - These are usually notes that explain and document the purpose of the commands executed in the script

```
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
date
ls /
```

Shell Scripts

- The remaining lines in the script are the commands it will execute
  - Same commands that you would normally enter at the terminal command line

```
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
date
ls /
```

Shell Scripts

- To run a shell script, the user must have read permission to the file
- The following would execute the previous script, if it were contained in a file called example.sh

bash example.sh

```
[mhackett@localhost ~]$ cat example.sh
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
date
ls /
[mhackett@localhost ~]$ bash example.sh
Sun 16 Feb 2020 03:18:32 PM EST
bin   dev  foruser  lib    media  opt   root  sbin  sys  usr
boot  etc  home     lib64  mnt    proc  run   srv   tmp  var
[mhackett@localhost ~]$
```

Shell Scripts

- The **echo** command can be used to print text output

```
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
echo "Today's date is: "
date
ls /
```

Shell Scripts

```
[mhackett@localhost ~]$ cat example.sh
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
echo "Today's date is: "
date
ls /
[mhackett@localhost ~]$ bash example.sh
Today's date is:
Sun 16 Feb 2020 03:23:50 PM EST
bin   dev  foruser  lib    media  opt   root  sbin  sys  usr
boot  etc  home     lib64  mnt    proc  run   srv   tmp  var
[mhackett@localhost ~]$
```

Shell Scripts

- Though this shell script file ended with a **.sh** extension, this is not required
  - .sh is, by convention, the extension for shell scripts

- If the user has executable permissions on the script, the file can be run by executing:

```
./example.sh
```

Shell Scripts

```
[mhackett@localhost ~]$ ./example.sh
-bash: ./example.sh: Permission denied
[mhackett@localhost ~]$ chmod u+x example.sh
[mhackett@localhost ~]$ ./example.sh
Today's date is:
Sun 16 Feb 2020 03:29:37 PM EST
bin   dev  foruser  lib    media  opt   root  sbin  sys  usr
boot  etc  home     lib64  mnt    proc  run   srv   tmp  var
[mhackett@localhost ~]$ _
```

Shell Scripts

- All shell scripts are executed in subshells, which may not have access to all environment or user-defined variables that the script requires
- To run a shell script in the current shell (not in a subshell), use the source command:
  - The . (dot) operator may also be used

    `source ./example.sh`

    or

    `. ./example.sh`

Escape Sequences

- The printed output of the **echo** command can be modified using special characters called **escape sequences**
  - The **-e** option must be used with the **echo** command in order to use **e**scape sequences

- While there are others, we'll only look at the **\c** and **\n** escape sequences:

\c	Prevents a new line from starting after printing
\n	Starts a new line of output

Escape Sequences

```
[mhackett@localhost ~]$ cat example.sh
#!/bin/bash
#This program displays the date and performs a
#listing of the root directory
echo -e "Today's date is: \c"
date
echo -e "Contents of\nthe root folder:"
ls /
[mhackett@localhost ~]$ ./example.sh
Today's date is: Sun 16 Feb 2020 03:48:09 PM EST
Contents of
the root folder:
bin   dev  foruser  lib    media  opt   root  sbin  sys  usr
boot  etc  home     lib64  mnt    proc  run   srv   tmp  var
[mhackett@localhost ~]$
```

Reading Input

- If a script requires input from a user, the **`read`** command is used to read data from standard input
  - The data entered is stored to a variable

```
#!/bin/bash
echo -e "Enter your name: \c"
read NAME
echo "Hello, $NAME!"
```

```
[mhackett@localhost ~]$ cat askname.sh
#!/bin/bash
echo -e "Enter your name: \c"
read NAME
echo "Hello, $NAME!"
[mhackett@localhost ~]$ ./askname.sh
Enter your name: Michael
Hello, Michael!
[mhackett@localhost ~]$
```

Logic and Decisions

- A script is able to choose between executing different sets of instructions based on some condition evaluating to true

- The most common decision structure is an if statement (syntax is shown below, indentation is optional):

```
if this is true
then
   execute these commands
fi
```

Logic and Decisions

- The *this is true* part can be either a command or a test statement
  - A command evaluates to true if the command executes without an error
  - Test statements are shown on the next slide

- The *execute these commands* section contains one or more commands to execute

Logic and Decisions

- A test statement compares two values and evaluates to either true or false
  - A test statement must be enclosed in [ ], with one space after the [and one space before the]

Test Statement	Returns true if
`[ A = B ]`	A is equal to B
`[ A != B ]`	A is not equal to B
`[ A -eq B ]`	A is equal to B (A and B are numbers)
`[ A -ne B ]`	A is not equal to B (A and B are numbers)
`[ A -lt B ]`	A is less than B (A and B are numbers)
`[ A -gt B ]`	A is greater than B (A and B are numbers)

Logic and Decisions

Test Statement	Returns true if
`[ A -le B ]`	A is less than or equal to B (A and B are numbers)
`[ A -ge B ]`	A is greater than or equal to B (A and B are numbers)
`[ -r A ]`	A is a file or directory that exists and is readable
`[ -w A ]`	A is a file or directory that exists and is writable
`[ -x A ]`	A is a file or directory that exists and is executable
`[ -f A ]`	A is a file that exists
`[ -d A ]`	A is a directory that exists

Logic and Decisions

- An if statement can provide alternative commands using an else clause
  - These commands execute in the event the test statement evaluates to false

```
if this is true
then
   execute these commands
else
   execute these commands instead
fi
```

Logic and Decisions

- An if statement can provide alternative test statements using an elif clause

```
if this is true
then
   execute these commands
elif this is true                (Tested only if the preceding if or elif was false)
then
   execute these commands
else                             (Executed only if all preceding tests were false)
   execute these commands instead
fi
```

Logic and Decisions

- Multiple test statements can be combined using special operators

Test Statement	Returns true if
`[ A = B -a C = D ]`	A is equal to B AND C is equal to D
`[ A = B -o C = D ]`	A is equal to B OR C is equal to D
`[ ! A = B ]`	A is NOT equal to B

Logic and Decisions

- Commands can be joined using the && and || operators
- Examples:

  command 1 **&&** *command 2*

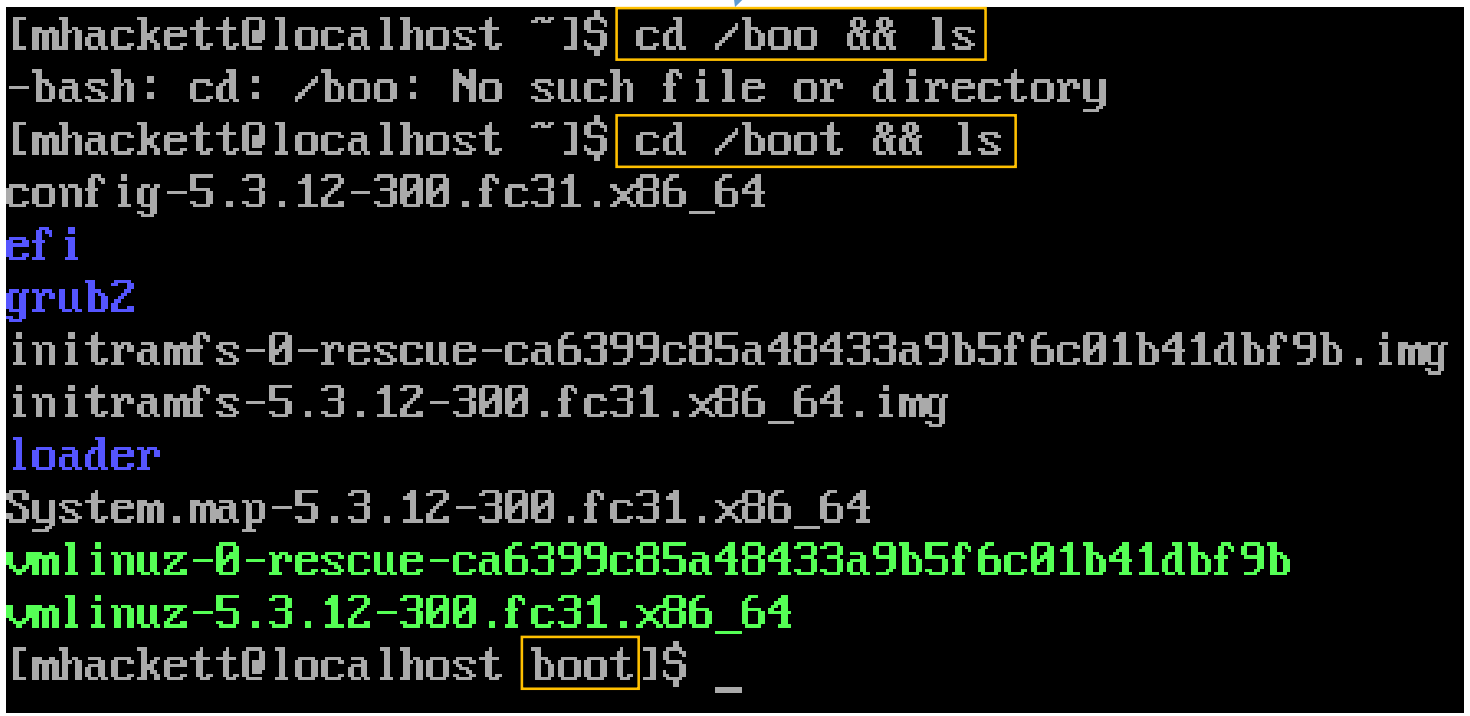
  - Command 2 will execute only if command 1 executed successfully (without an error)

  command 1 **||** *command 2*

  - Command 2 will execute only if command 1 did not execute successfully

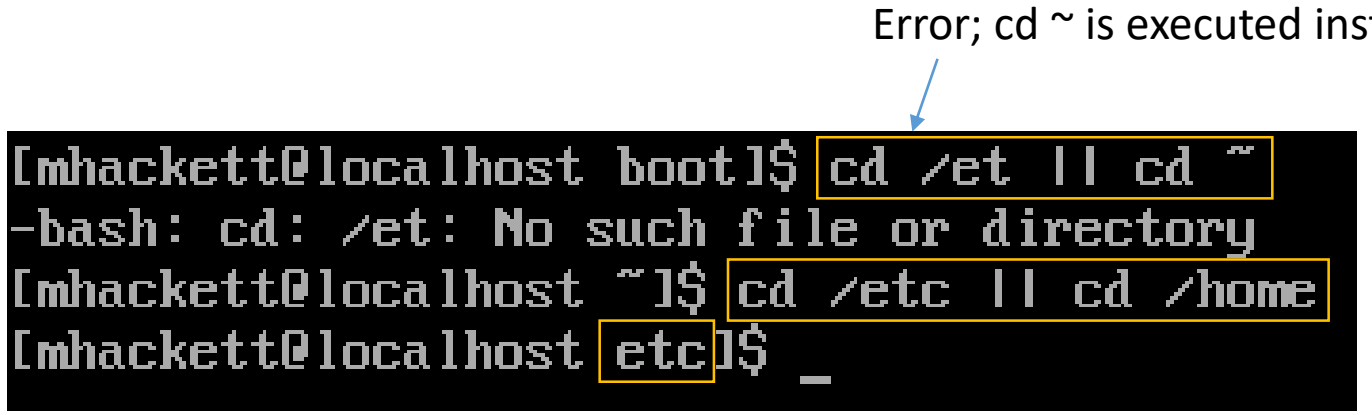
Logic and Decisions

Error; The ls command is not executed



Logic and Decisions

Error; cd ~ is executed instead



Loops

- A script is able to repetitively execute instructions based on:
  - A finite series of values
  - A test statement evaluating to true

- In a *count-controlled loop*, instructions are repeated a certain number of times
  - The loop stops when it has exhausted a list or range of possible values
- In a *sentinel-controlled loop*, instructions are repeated as long as a test statement evaluates to true
  - The loop stops when the test statement evaluates to false

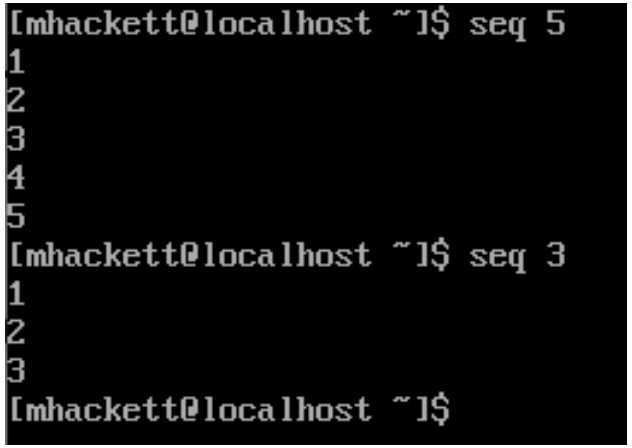
Loops

- **For loops** are count-controlled loops that are often used with the **seq** command
- The syntax for a for loop is shown below:

```
for var in value1, value2, value3...
do
   execute these commands
done
```

Loops

- The **seq** command produces a **seq**uence of numbers, starting at 1 through the value provided to the command



- This can be used by a for loop to iterate through a range of numbers.

Loops

```
[mhackett@localhost ~]$ cat example2.sh
#!/bin/bash
#This program prints the numbers 1 through 10
for NUM in `seq 5`
do
  echo "Number $NUM"
done
[mhackett@localhost ~]$ bash example2.sh
Number 1
Number 2
Number 3
Number 4
Number 5
[mhackett@localhost ~]$ _
```

Loops

- **While loops** are sentinel-controlled loops that repeat as long as its test condition is true
- The syntax for a while loop is shown below:

```
while [ this is true ]
do
   execute these commands
done
```

Loops

```
[mhackett@localhost ~]$ cat example3.sh
#!/bin/bash
#This program prints the numbers 0 through 9
COUNT=0
while [ $COUNT -lt 10 ]
do
  echo "Number $COUNT"
  COUNT=`expr $COUNT + 1`
done
[mhackett@localhost ~]$ bash example3.sh
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
[mhackett@localhost ~]$
```

Loops

- **Until loops** are sentinel-controlled loops that execute as long as its test statement is *false*
  - Sort of an opposite while loop
- The syntax for an until loop is shown below:

```
until [ this is true ]
do
   execute these commands
done
```

Loops

```
[mhackett@localhost ~]$ cat example4.sh
#!/bin/bash
#This program prints the numbers 0 through 9
COUNT=0
until [ $COUNT -eq 10 ]
do
  echo "Number $COUNT"
  COUNT=`expr $COUNT + 1`
done
[mhackett@localhost ~]$ bash example4.sh
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
[mhackett@localhost ~]$
```

Special Variables

- Shell scripts can accept arguments when executed on the command line.
  - These arguments are called **positional parameters**

- Positional parameters are referenced in a shell script using a special variable
  - **$X**, where **X** is a number
  - $0 is the command used, $1 is the first argument, $2 is the second argument, etc.

Special Variables

```
[mhackett@localhost ~]$ cat params.sh
#!/bin/bash
echo The command is $0
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo The fourth argument is $4
[mhackett@localhost ~]$ bash params.sh
The command is params.sh
The first argument is
The second argument is
The third argument is
The fourth argument is
[mhackett@localhost ~]$ bash params.sh testA testB
The command is params.sh
The first argument is testA
The second argument is testB
The third argument is
The fourth argument is
[mhackett@localhost ~]$ bash params.sh testA testB testC testD
The command is params.sh
The first argument is testA
The second argument is testB
The third argument is testC
The fourth argument is testD
[mhackett@localhost ~]$ _
```

Special Variables

- The special variable **$#** is the number of positional parameters

- The special variable **$*** is all of the positional parameters

Special Variables

```
[mhackett@localhost ~]$ cat params.sh
#!/bin/bash
echo The command is $0
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo The fourth argument is $4
echo The number of arguments is $#
echo All arguments: $*
[mhackett@localhost ~]$ bash params.sh testA testB testC testD
The command is params.sh
The first argument is testA
The second argument is testB
The third argument is testC
The fourth argument is testD
The number of arguments is 4
All arguments: testA testB testC testD
[mhackett@localhost ~]$
```

Source Code Version Control

- **Version control software** (VCS) keeps track of changes made to files
  - Most commonly used for tracking and managing source code changes in programs, scripts, and other software projects

- The most popular open source version control system is Git
  - Others include Mercurial, SVN, and CVS

Source Code Version Control

- While Git can be used locally, it is often used to maintain versions of source code files that are worked on collaboratively.

- Programmers can upload (“commit”) their changes to a Git repository that will be accessible to the other programmers.

Source Code Version Control

- A Git **repository** (or simply, “Git repo”) is a directory that contains the files managed by Git and information related to their changes
- Each programmer can **clone** a copy of the repository to their computer
- When a programmer has completed changes, they **add** any new files and **commit** their changes
- The programmer will then **push** their commits/changes to the repository
- Other programmers can **pull** down the commits/changes other programmers have made

Source Code Version Control

- By default, changes are made to the **master branch**
  - A **branch** is a section of a repository that tracks changes to files
- Programmers will create separate branches from the main branch
  - This way, any changes or commits will be isolated in their branch instead of the main branch
  - When ready, the branch can **merge** back into the main branch