

COMP 206 – Introduction to Software Systems Linux Intro

Lecture 3

Sept 10th, 2018

Today's Outline

- More important Shell commands
- Important concepts of the shell: standard IO, redirection, pipes and shell variables
- How to create our first shell programs

More commands to know

- echo: copy input to output (why is this needed?)
- grep: filter input based on a pattern
- cut: break up a line into its fields
- tr: translate inputs to outputs
- sort: sort inputs, then output
- ps: display running processes (once)
- top: display the running processes (continuous) and resource usage
- ssh: remotely connect to another computer

Examples to try together

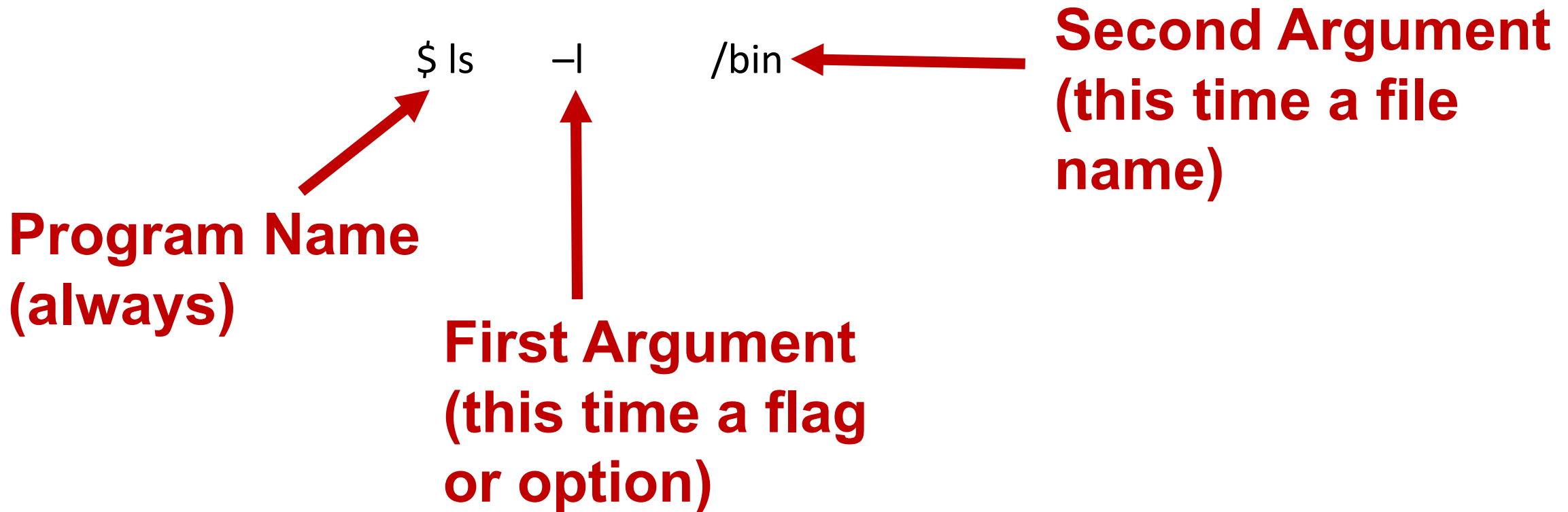
- `$ echo hello world`
- `$ cut -f3 -d" "`
- `$ tr [a-z] [A-Z]`
- `$ sort`

There are thousands of commands, which do you need?

- Those listed here form the core for 206 knowledge and testing
- To use Linux in real life, you don't need to memorize command, but rather have the skill to find and learn new commands as needed
- Tools to help with this:
 - `man` – A linux command that shows the manual page for other commands!
 - The "--help" argument (e.g., "`$ ls --help`")

Recall: Command-line arguments

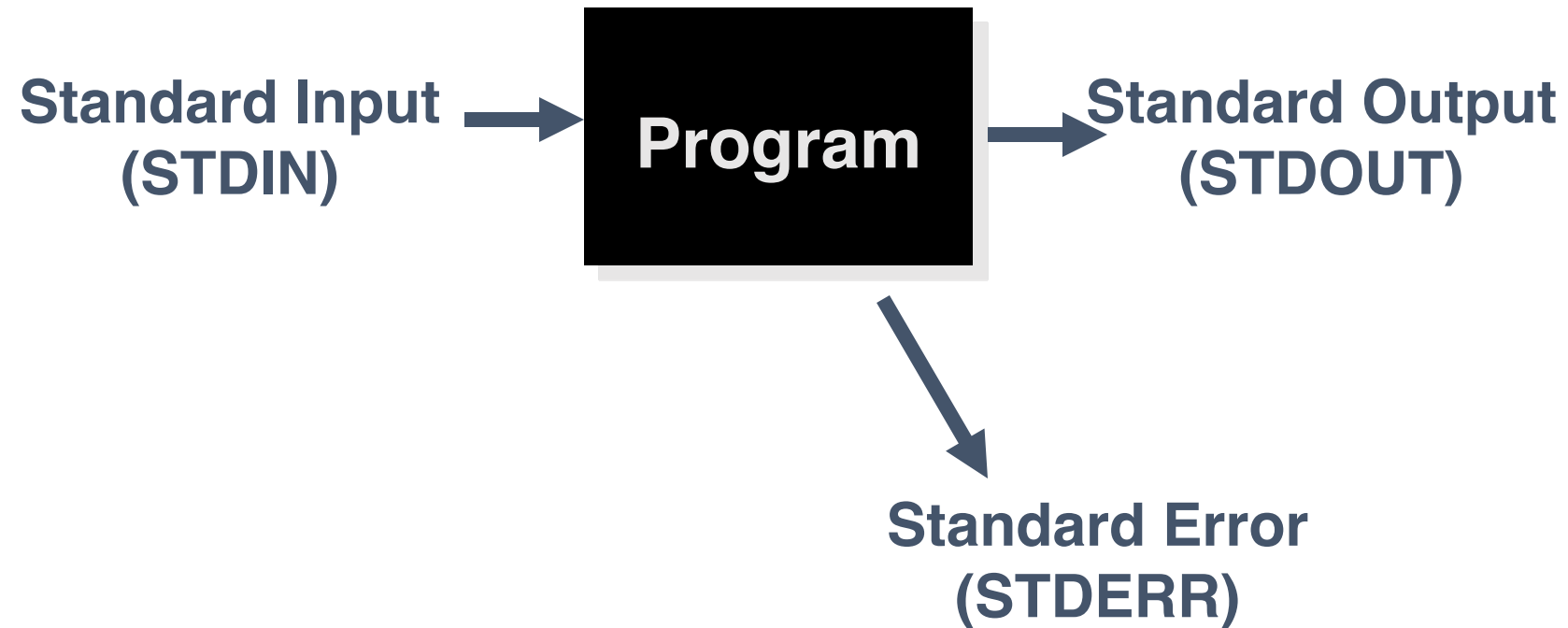
- Type the program's name followed command-line arguments, the shell executes this program, feeding the options you specified:



Other sources of input

- Arguments are given once, parsed by the program as it begins to modify its operation, but it is impossible to change or add to them once we press enter
- Many programs require input during their operation. We saw this for `tr`, `cut` and `sort`. Other uses include e.g., a text-editor, the shell itself, a username/password entry dialog.
- Linux calls this type of input “standard input” and gives us flexible options for where our programs get standard input. By changing these, we can start to build up more complex commands and programs.

Programs and Standard I/O



Defaults for I/O

- When a shell runs a program for you:
 - standard input is your keyboard.
 - standard output is your screen/window.
 - standard error is your screen/window.

Terminating Standard Input

- If standard input is your keyboard, you can type stuff in that goes to a program.
- To end the input you press Ctrl-D (^D) on a line by itself, this ends the input *stream*.
- The shell is a program that reads from standard input.
- What happens when you give the shell ^D?

Input Redirection

- The shell can attach things other than your keyboard to standard input.
- A file, using the “<” operator:
 - E.g. “\$ grep pattern < search_file.txt”.
 - The contents of the file are fed to a program as if you typed it).
- The output of another command, using the “|” operator:
 - E.g. “\$ ls | grep”
 - The output of another program is fed as input as if you typed it. Note that both programs can run simultaneously and continue to transmit information over their “pipe”. E.g.,

Output Redirection

- The shell can attach things other than your screen to standard output (or stderr).
- A file, using the “>” operator:
 - E.g., `$ ls > file_info.txt`
 - The output of a program is stored in file
- The input of another program, using a pipe as we have seen on the previous slide

Input redirection

- To tell the shell to get standard input from a file, use the “<” character:

```
$ sort < nums.txt
```

- The command above would sort the lines in the file `nums` and send the result to `stdout`.

You can do both!

```
$ sort < nums > sortednums
```

```
$ tr a-z A-Z < letter > rudeletter
```

Output and Output Append

- The command `$ ls > foo.txt` will create a new file named foo (deleting any existing file named foo).
- If you use `>>`, the output will be appended, leaving any contents that were in the file previously and adding the new output at the end:

```
$ ls /etc >> foo.txt
```

```
$ ls /usr >> foo.txt
```

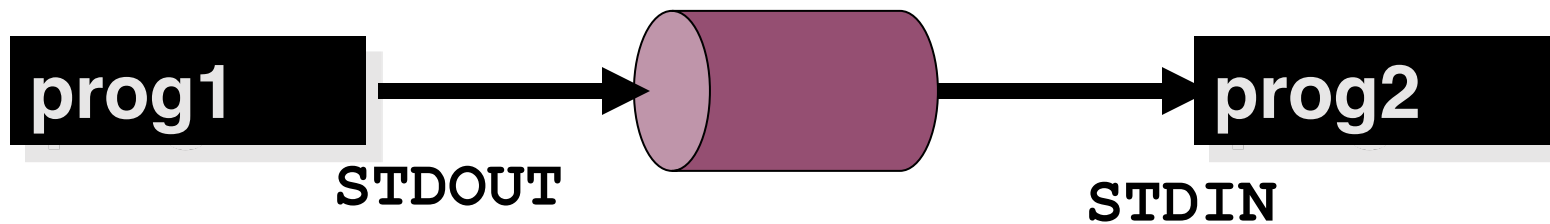
- `>>` still creates the file if it wasn't there previously

What about the errors?

- Sometimes we do not want them ending up in our re-direct file, so the default behavior of “>” is that errors stay on the terminal and only standard output enters the file.
- However, the shell is powerful and we can decide on what goes where:
 - “1>” means to send standard output only (same as “>”)
 - “2>” means to send standard error only (std out might stay on terminal)
 - “&>” means to send both standard error and output
- It's OK to add both “1>” and “2>” giving different files. Do not name the same file or you only get standard out.

Pipes

- A pipe is a holder for a stream of data.
- A pipe can be used to hold the output of one program and feed it to the input of another.



Asking for a pipe

- Separate 2 commands with the “|” character.
- The shell does all the work!

```
$ ls | sort
```

```
$ ls | sort > sortedls
```

Building commands

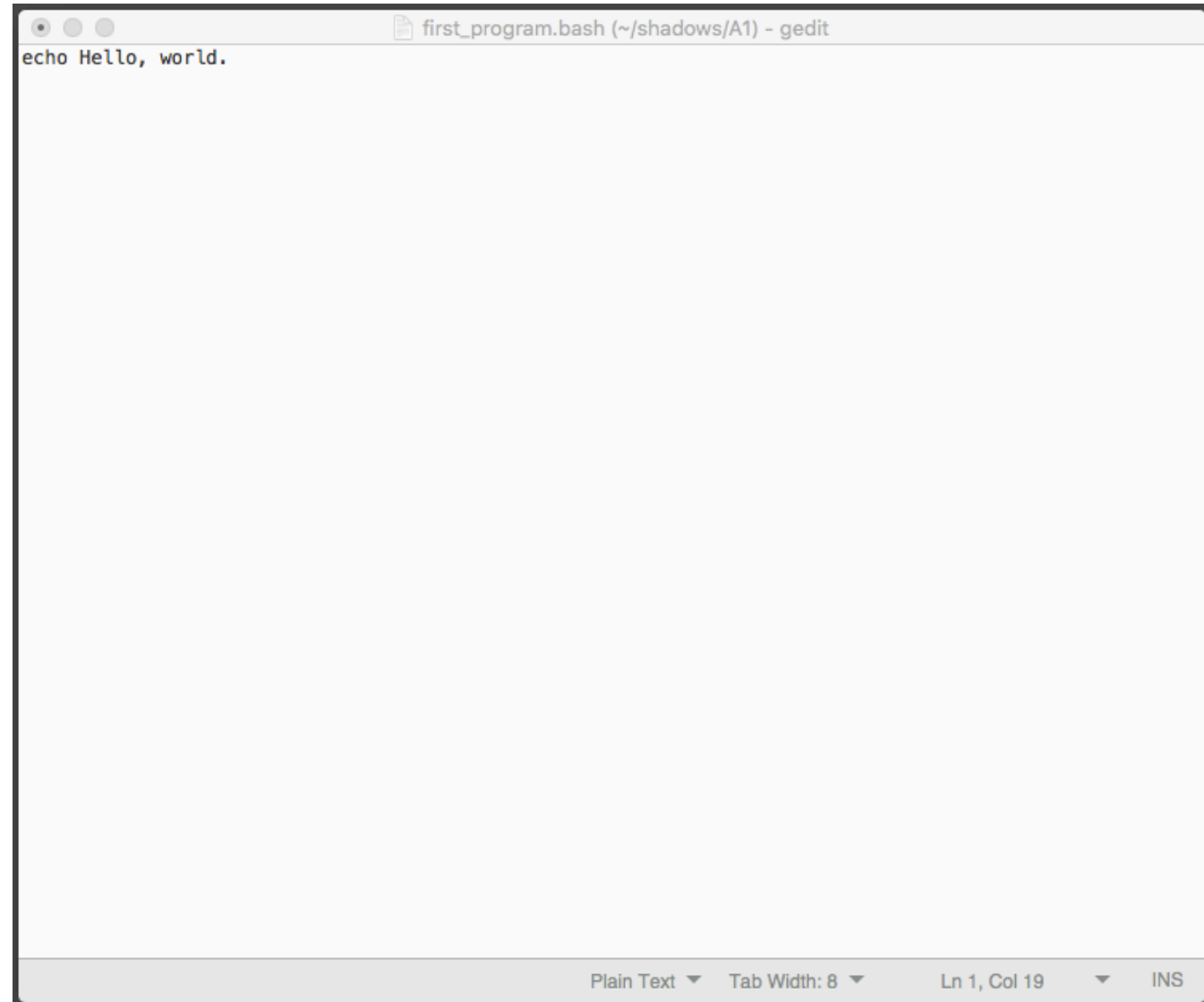
- You can string together a series of unix commands to do something new!
- Exercises:
 - List all files in the current directory but only use upper case letters.
 - Find all the image files (.png) in your home directory and sort them alphabetically by file name

Going beyond one liners

- Now we can create some powerful single shell command lines.
- All shell programming can be done as single lines, using the “;” operator, which is equivalent to pressing enter.
- This gets to be a big mess quickly. Good to try for fun, but don’t do it for Assignments or in the work place!
 - Instead, we can put many lines together in a file to create a shell program

How to create a Shell program

- Shell programs are written in text files, named ".bash" by convention
- Enter the contents "echo Hello World" into a text editor, save the file as "first_program.bash", and you have successfully written a shell program!
- Any series of commands you can type into the terminal one by one can also form a shell program. Start a new line whenever you would press enter.

A screenshot of a gedit text editor window. The title bar at the top reads "first_program.bash (~/shadows/A1) - gedit". The main text area contains the command "echo Hello, world." on a single line. The status bar at the bottom shows "Plain Text", "Tab Width: 8", "Ln 1, Col 19", and "INS".

```
first_program.bash (~/shadows/A1) - gedit
echo Hello, world.
```

How to run our shell programs

- Run a shell program in one of several ways:
 - `$ bash first_program.bash`
 - `$. first_program.bash`
 - `$ source first_program.bash`

A Note about Text Editing in Linux

- It is possible to continue with the terminal, as many good editors operate entirely inside the shell's window:
 - Vim
 - Emacs
 - Pico
 - Nano
- Many people feel more comfortable using a window-based editor with the usual graphical menus file-save, edit, mouse interaction etc. Some popular ones for COMP 206 students:
 - Gedit
 - Sublime
 - VS Code
 - Eclipse
- There is no rule and we will not check or ask you to know any specific editor. Just try a few and see what you're comfortable with

The first line of a bash program

- How did the shell know that our program was written in bash even when we ran it with "source" or "." syntax (and what if it had instead been written in CSH?)
- In our first example, this was assumed because our terminal was already using the bash, so things worked easily
- It is proper shell programming practice to add a shabang (a.k.a. sharp-bang, hash-bang, pound-bang) sequence to indicate which shell we intend:
- `#!/bin/bash` is the standard first line for every bash script

The simplest shell programming style

- We have already seen. It involves simply listing several shell commands in order, perhaps with the input/output redirection and pipe operators to make them work together:

```
ls > dir_contents.txt  
grep elephant dir_contents.txt > animals.txt  
grep hippo dir_contents.txt >> animals.txt  
echo "I found the following animals:"  
cat animals.txt
```

- Note that we are using files on the disk to hold temporary data, as if they were variables. It gets us started, but we can go much further!

Shell Variables

- The shell keeps track of a set of parameter names and values.
 - Assignment just with equals: `# my_var=Hello`
- Some of these are special parameters determine the behavior of the shell. Others are simply to be used to build program logic.
- We can access these variables and use them in many ways when composing shell programs:
 - Access with the dollar sign character: `# echo $my_var`

Shell Variables

NOTE: I will use the symbol "#" to mean your command-prompt for the next small section. This is to avoid confusion with "\$" that has a special meaning for shell variables. In a minute we will see how to swap the prompt.

- The shell keeps track of a set of parameter names and values.
 - Assignment just with equals: `# my_var=Hello`
- Some of these are special parameters determine the behavior of the shell. Others are simply to be used to build program logic.
- We can access these variables and use them in many ways when composing shell programs:
 - Access with the dollar sign character: `# echo $my_var`

Displaying Shell Variables

- Prefix the name of a shell variable with "\$".

- The **echo** command will do:

```
# echo $HOME
```

```
# echo $PATH
```

- You can use these variables on any command line:

```
# ls -al $HOME
```

```
# var=ls
```

```
# $var -al $HOME
```

```
# options=-al
```

```
# $var $options $HOME
```

Setting Shell Variables: Details

- Variable with an assignment command is a shell *builtin* command:

```
# HOME=/etc
```

```
# PATH=/usr/bin:/usr/etc:/sbin
```

- There cannot be any spaces in the variable name, between the variable name and the equals, between the equals and the value, or within the value.
- However, it is possible to use values with spaces by enclosing them in quotes:

```
# NEWVAR="blah blah blah"
```

set command (shell builtin)

- The **set** command with no parameters will print out a list of all the shell variables.
- You'll probably get a pretty long list...
- Depending on your shell, you might get other stuff as well...

Shell Variables with special meaning

| | |
|-----------------|--|
| PWD | <i>current working directory</i> |
| PATH | <i>list of places to look for commands</i> |
| HOME | <i>home directory of user</i> |
| MAIL | <i>where your email is stored</i> |
| TERM | <i>what kind of terminal you have</i> |
| HISTFILE | <i>where your command history is saved</i> |
| PS1 | <i>the string to be used as the command prompt</i> |

Example \$PS1

- The **PS1** shell variable is your command line prompt. It's a string.
- By changing PS1 you can change the prompt.
- E.g.
 - `# PS1="Next command: "`
 - `# PS1="# "`

Fancy **bash** prompts

Bash supports some fancy stuff in the prompt string:

`\t` is replaced by the current time

`\w` is replaced by the current directory

`\h` is replaced by the hostname

`\u` is replaced by the username

`\n` is replaced by a newline

Example **bash** prompt

```
===== [foo.cs.rpi.edu] - 22:43:17 =====  
/cs/hollind/introunix echo $PS1  
===== [\h] - \t =====\n\w
```

Capturing command output in a variable

- We saw that “>” stored the command’s output in a file
- Sometimes we want to skip the filesystem (efficiency of memory vs disk) and re-use the output within our BASH program
- The back-tick operator allows this:
 - Format: `# variable=`command``
 - Anything that `# command` alone would output to the terminal is now stored as the value of variable. Access it with `$variable`. (of course this name is just an example, we can pick any other, e.g. `# daves_var=`comand``)
 - A nearly equivalent syntax is `# variable=$(command)`

Example Backup Program:

```
1  #!/bin/bash
2
3  # This bash script is used to backup a user's home directory to /tmp/.
4
5  user=$(whoami)
6  input=/home/$user
7  output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz
8
9  tar -czf $output $input 2> /dev/null
10 echo "Backup of $input completed! Details about the output backup file:"
11 ls -l $output
```

Math

- The shell can do simple arithmetic.
- Enclose your computation in `$((computation))`
- You can use this in quite flexible ways:
 - `# a=$((3+5))`
 - `# echo There are $((60*60)) seconds in an hour`

Summary:

- In Linux, the Shell is the user-space program we interact with by typing commands, one by one or as a program.
 - The shell allows running other tool programs such as `cd`, `ls`, `cat`, `pwd`, `find`, `grep`, text editors and many more.
 - The shell allows for more complex behaviors by chaining tool programs through redirection and pipes and using variables
- Next lecture we will finish off the last topics in shell programming and begin to think about writing our own systems programs in C.

Exercises

- Read “\$man bash”. Most things we discussed so far can be found there in a different format. Note that there are some advanced features on the man page that we will not cover, so don't be distracted.
- Try out an online tutorial or two. I like this one:
 - <https://linuxconfig.org/bash-scripting-tutorial-for-beginners>
- Go for it and dive in to Assignment 1. Good luck!