PURDUE UNIVERSITY · ECE 364

BASH: LECTURE NOTES

Basic commands

cd - Change Directory

```
cd ~
cd Documents
```

ls - List Directory Contents

```
ls       # list files in current path
ls dir   # list files in dir
ls *.c   # list all files ending in .c
ls foo*  # list all files starting with foo
ls -l    # list files in formatted list (with more info such as date and permissions)
ls -a    # list list all files (including hidden files) in current path
ls --sort=WORD # sort by WORD instead of name: none, extension, size, time, version
```

cp - Copy Files and Directories

```
cp fileA fileB  # copy fileA into fileB
cp *.c dirA     # copy all files matching *.c into dirA
cp -r dirA dirB # copy dirA (recursively) into dirB
```

mv - Move (Rename) Files

```
mv fileA fileB  # move (rename) fileA to fileB
mv *.c dirA     # move all files matching *.c into dirA
mv dirA dirB    # move dirA (recursively) into dirB
#WARNING: mv will overwrite any files it gets as destination. -n will stop it from overwriting
```

echo - Display Text

```
echo "Hello World"      # display Hello World in the console
```

man - Manual

`man` is probably the most useful command in bash. It displays the help text for any command that has it. The help text includes all relevant information about a command and its arguments. Optional arguments and switches are written in [square brackets], required arguments are written by themselves. You will need to know how to use `man` to solve the labs and prelabs. It will also be useful for the exam.

```
$ man command
>--------------------------------------------------------------------------------
  NAME
        cmd - Command name and description

  SYNOPSIS
        cmd [OPTIONAL ARGUMENTS]... [-switches] REQUIRED ARGUMENTS

  DESCRIPTION
        Description of what the command does and how the REQURED ARGUMENTS are
        used.

        -s, --switch
              Switch and description of what it does
```

File Manipulation

cat - Concatenate and Print Files

cat is a special and useful command. Its simplest (and most common) use is to print files to the console (stdout). It can also be used to combine files together and to feed a file into a different bash program (more on that in the next lecture).

```
cat fileA            # display fileA's contents in the console
cat -eT fileA        # display fileA's contents, including invisible (tab/newline) characters
cat fileA fileB fileC # display the contents of fileA, fileB, and fileC, in order.
cat *.txt            # display all files that end in .c, in order.
```

head and tail - Print the Start and End of Files

head is used to print the first n lines of a file to the console. tail is used to print the last n lines.

```
head -n 5 fileA # print the first 5 lines of fileA
head -c 5 fileA # print the first 5 characters of fileA
head -n -5 fileA # print all but the last 5 lines of fileA

tail -n 5 fileA # print the last 5 lines of fileA
tail -c 5 fileA # print the last 5 characters of fileA
tail -n +2 fileA # print all lines of fileA starting at line 2
```

grep - Search Text

grep stands for "Go, RegEx, Print". It lets you search through the contents of files and directories.
WARNING: the -E switch is required for regex to work as expected. Without it bugs may occur.

```
grep -E "foo" fileA        # display all lines that match "foo" in fileA to the console
grep -E "^H" fileA fileB   # display all lines that math the regex "^H" in fileA and fileB
grep -lr -E "foo" dirA     # display all files in directory dirA that contain the word "foo"
                         # in this case, -l stands for list-files, and -r stands for recursive
```

`cut` - Select Sections of a File

When dealing with tabulated data, `cut` allows us to select entire columns of data.

```
cut -f1,5 fileA # select and print columns 1 and 5 of fileA
cut -f2 -d','   # select and print column 2, where fileA is delimited by commas, not tabs
```

`sort` - Sort a File

`sort` will sort all the lines of a file alphabetically

```
sort fileA  # print fileA with its lines sorted alphabetically
```

`wc` - Word Count

```
wc -w fileA  # print the number of words in fileA
wc -l fileA  # print the number of lines in fileA
```

Variables

Bash was designed for text interfaces. All variables and values are by default processed as text strings. Numbers are stored as their string representations.

Assignment and Usage

```
myvar="Hello World"      # myvar now contains the text "Hello World" (no quotes)
mynum=5                  # mynum now contains the character '5' (no quotes)
mylongnum=364            # mylongnum now contains the string "364" (no quotes)

# WARNING: Spaces matter. The following WILL NOT WORK:
brokenvar = Some text
# The equal sign must immediately follow the variable name and the value must follow the equal sign
# If the text string value contains spaces, it must be wrapped in quotes. Like this:
correctvar="Some Text"

# To use a variable's value, the variable name must be preceded by a $ sign, like so: $var
echo $myvar              # display Hello World in the console
Hello World

echo $mynum + $mylongnum # display the text: 5 + 364
5 + 364

mydir=~/Documents
ls $mydir                # list all files in ~/Documents

myfile=$mydir"/fileA"
cat $myfile              # display contents of ~/Documents/fileA
```

Expression (subshell) Assignment

It is often useful to store a command's output in a variable for use later. In such cases, we create a subshell. Subshells are child processes of the calling terminal that act like terminals themselves. You can add any code you want to a subshell and its output will be sent back out to the calling shell (bash process).

```
myfiles=$(cat list_of_files.txt)  # list_of_files.txt contains filenames that we may want to use later
                                  # the $() notation creates a subshell, and runs any code given inside
                                      it.
                                  # after the command is done, its output is sent back out to our calling
                                      statement.
                                  # In this case, it will store it in a variable.

echo $myfiles                     # the variable $myfiles now includes the text from list_of_files.txt
  fileA.py fileB.py fileC.txt

mv $myfiles $someFolder           # that variable can then be used like any other in commands
ls $someFolder
  fileA.py fileB.py fileC.txt

somefiles=$(grep -E -lr "ee364[a-g][0-9]{2}") # list all files that contain ee364 IDs
echo $somefiles                   # "myscript.py anotherfile.py submission.bash ..."
cat $somefiles                    # will output all found files together

# This can be especially useful when we have python files that do a lot of work that we may want to use
    for later
circuits=$(python3 -c "import collectionTasks; print(collectionTasks.getCircuitByComponent('R137'))")
# we now have all relevant circuits in Bash
```

String Manipulation

```
${#string}           # String length
${string:idx}        # Substring strating at idx
${string:idx:len}    # Substring starting at idx of length len
${string//regex/repl} # Replace all matches of regex with the string repl
```

IO Redirection

One of bash's strongest uses is the ability to redirect one command's output into a file or even another command's input. This is achieved using IO redirection.

> - Write Output to File

```
echo "Hello World" > fileA # Writes "Hello World" to fileA. This will overwrite the file if it exists
echo "foo bar" >> fileA    # Appends "foo bar" to fileA. This will not overwrite the file

cat fileA fileB > fileC    # Combine fileA and fileB and store the result in fileC
```

| - Pipe Output to Another Command's Input

The pipe (`|`) lets you take one command's output and feed it to the next command's input. It can be extremely useful when dealing with a lot of data.

```
ls -1 | grep "file[A-Za-z][0-9].csv" # Will return a list of all files matching the pattern in the regex


cat regexlines.txt | grep -f - fileToSearch.txt
echo "ee364[a-g][0-9]\{2\}" | grep -f - fileToSearch.txt

# Above we use the -f switch to tell grep to use a file as its regex input, so grep will search for the
    contents of that file in other files. The file which we will use to input is stdin (specified by the
    dash '-' character above). This lets us input regex to grep from the pipe. Play around with the above
    concepts to familiarize yourself with grep's more powerful features. If regexlines.txt contains more
    than one line, grep will run multiple times using each line of the file as a regex pattern.

This can be combined with subshells and variables:

pattern="file[A-Za-z][0-9].csv"
filesfound=$(ls -1 | grep $pattern)
cat $filesfound | cut -d=',' -f1,3 > allfiles_cols_1_3.csv
```

Evaluation, Conditionals, and Loops

`if [];` then *expr*; else *expr*; fi

```
# If statements in Bash look like this:
if [[ $myvar = "some value" ]]
then
  echo "TRUE"
else
  echo "FALSE"
fi

# Notice that by default bash will compare string values only
# Partial matches can be done too using regex
if [[ $myvar = "ee364.*" ]]
then
  echo "TRUE"
fi

# multiline statements can be combined into a single line using semicolons
if [[ $myvar = "some value" ]]; then echo "TRUE"; fi

# the default comparison mode is for text
if [[ 5 < 364 ]]; then echo "TRUE"; else echo "FALSE"; fi
FALSE

# numerical comparison can be achieved using:
# '-eq': equal, '-ne': not-equal
# '-lt': less-than, '-gt': greater-than
# '-le': less-than-or-equal, '-ge': greater-than-or-equal
if [[ 5 < 364 ]]; then echo "TRUE"; else echo "FALSE"; fi
TRUE
```

`while [[]]; do` *expr*`; done`

```
# while works the same way as if, but in a loop
cond=$(somecommand)
while [[ $cond != "Done" ]]
do
  cond=$(somecommand)
done
```

`for var in $list; do` *expr*`; done`

```
# for loops in bash work similarly to Python, where it's easier to loop over a list of values than over an
    index

cfiles=$(ls *.c)
for f in $cfiles
do
  gcc $f -o ${f//\.c/.o}
done

# when put on one line, for loops can store their outputs in variables

ofiles=$(for f in $cfiles; do echo ${f//\.c/.o}; done)

echo $cfiles  # a.c b.c main.c
echo $ofiles  # a.o b.o main.o

# you can iterate over a range of numbers
for i in {1..5}; do echo i=$i; done
i=1
i=2
i=3
i=4
i=5

# This will not work with variables (try it!). Instead, use the command 'seq'
mynum=5
for i in $(seq 1 $mynum); do echo i=$i; done
i=1
i=2
i=3
i=4
i=5
```

Bash Files

So far we've been running commands in terminal. To reuse scripts, we store them in Bash files. Anything that can run in terminal can be written into a Bash file, and anything that would be outputted to the console in terminal will be printed out when the file is run. The basic structure of a Bash file is shown below

```
#! /bin/bash
echo "Hello World"
myvar=364
echo ECE $myvar
```

Bash files can be run similarly to self-contained programs and can take command-line arguments, just like Bash commands.
The arguments are accessed like variables named by their **argument number**: `$1 $2 $3` are the first three arguments.

`myscript.bash`

```bash
#! /bin/bash

echo "First Argument: $1"
echo "Second Argument: $2"
echo "Third Argument: $3"
```

```
myscript.bash 123 "Hello World" fileA.c
First Argument: 123
Second Argument: Hello World
Third Argument: fileA.c
```

Notes

This should encompass the most common commands you will need to use in Bash. There are many more, but once you've mastered the contents of this document you will be able to be a productive Bash programmer and automate many of your common tasks. The best way to learn is to try, so go through this file and try all the commands, and read their descriptions in `man`, you may find things that are useful but were not covered!