CS197U: A Hands on Introduction to Unix

Lecture 6: Programming & Scripting Part II

Tian Guo

University of Massachusetts Amherst – CICS

Reminders

- Assignment 3 was due today at 3:45pm
- Assignment 4 will be posted soon and be due in two weeks.

That is 3:45 on Oct 15.

- Last time:
  - Many aspects of processing file using awk

Outline

- File Processing Utilities
  - **Awk: Quick reviews and examples**
  - Sed
- Shell Scripting
  - automating command line tasks
- Programming in Linux
  - compilers and accessories

awk - Quick reviews

- Awk by default works in a line-based.
- Value of a column = $1, $2, $3 for column 1, 2, 3, etc
- Special: $0 for the entire line
- Other build-in variables
  - FS: field separator — allow processing files that use separator other than white space.
  - OFS: output field separator — allow reformatting the output
  - NF: number of fields in current line — printing last field
  - NR: number of lines in current file — selecting a range of lines

awk - Quick examples

awk [-F fs] [-v var=value] ['prog' | -f progfile] [file ...]

field separator

variables

script in ‘ ’ or in file

data file

1. Printing 2nd and 4th column separated by comma

data.txt

```
1 10 xyz 100
2 20 abc 200
3 30 def 300
4 40 ade 400
5 50 f2d 500
```

```
awk ‘{print $2 “,” $4}’ data.txt
```

```
awk 'BEGIN{OFS=","} {print $2,$4}' data.txt
```

output:

10,100

20,200

…

2. Printing the last column

```
awk ‘{print $4}’ data.txt
```

```
awk ‘{print $NF}’ data.txt
```

Here NF=4

awk - Quick examples

data.txt

```
C1 C2 C3 C4
1 10 xyz 100
2 20 abc 200
3 30 def 300
4 40 ade 400
5 50 f2d 500
```

3. Printing 4th column except header

```
awk '{if (NR > 1) print $4}’ data.txt
```

```
tail -n+2 data.txt |awk '{print $4}'
```

4. Printing Sum of C2 and C4 and a file header “Sum(C2,C4)”

```
awk 'BEGIN{print "Sum(C2,C4)"}{if (NR > 1) print $1+$4}' data.txt
```

5. Calculating the average of C2

```
awk 'BEGIN{tot2=0; print "Avg(C2)"} {tot2=tot2+$2} END {print tot2/NR}'
```

By default: tot2=0

awk - Quick examples

data.txt

```
C1 C2 C3 C4
1 10 xyz 100

2 20 abc 200
```

6. Counting the num. of lines

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

```
wc -l data.txt | awk ‘{print $1}’
```

7. Removing empty lines

```
awk '{if (NF > 0) print $0}’ data.txt
```

sed - **s**tream **ed**itor (find and replace)

- Transforms lines in a file
  - Mainly useful for finding and replacing strings

```
sed s/expression/replacement/[flags] [file]
```

s=substitute

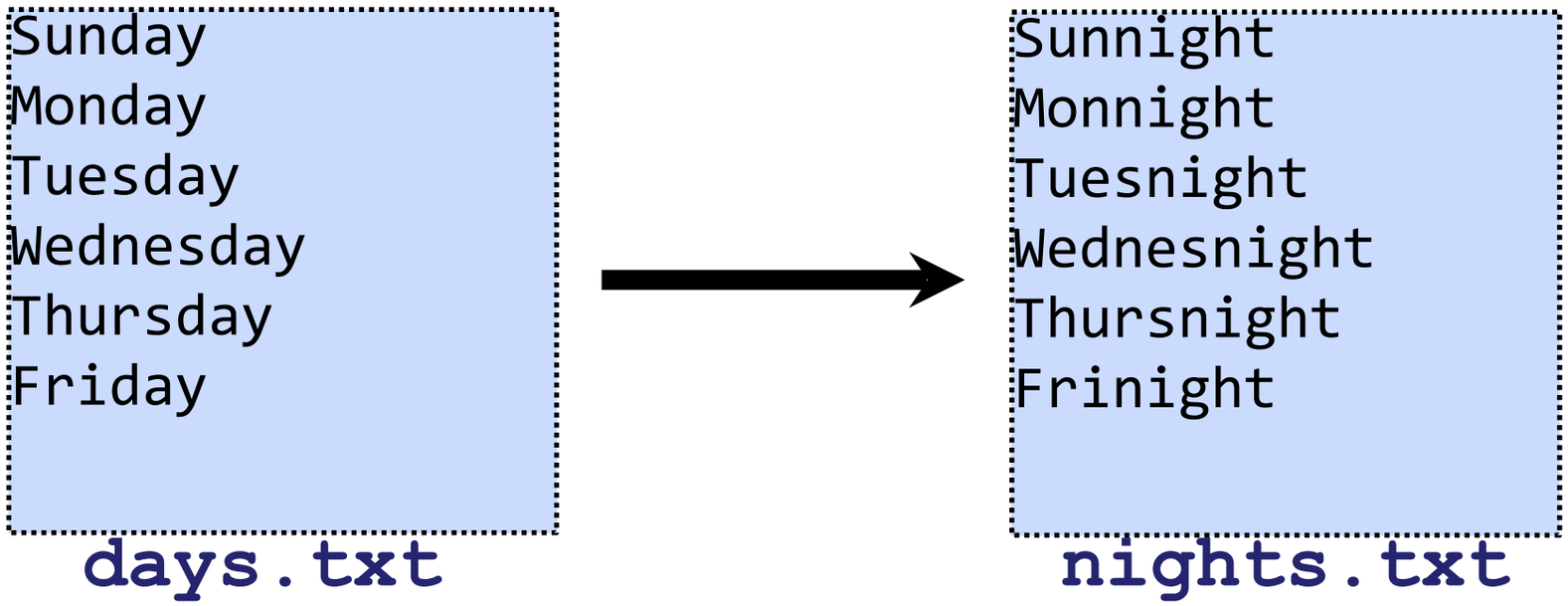
change this

to this

flags:
g = global
(not just first)

sed - **s**tream **ed**itor (find and replace)

- Example 1



Step 1:sed s/day/night/ days.txt

Step 2:sed s/day/night/ days.txt > nights.txt

sed - **s**tream **ed**itor (find and replace)

- Example 2

```
Sunday Sunday
Monday Monday
Tuesday Tuesday
Wednesday Wednesday
Thursday Thursday
Friday Friday
```

days.txt

```
Sunnight Sunnight
Monnight Monnight
Tuesnight Tuesnight
Wednesnight Wednesnight
Thursnight Thursnight
Frinight Frinight
```

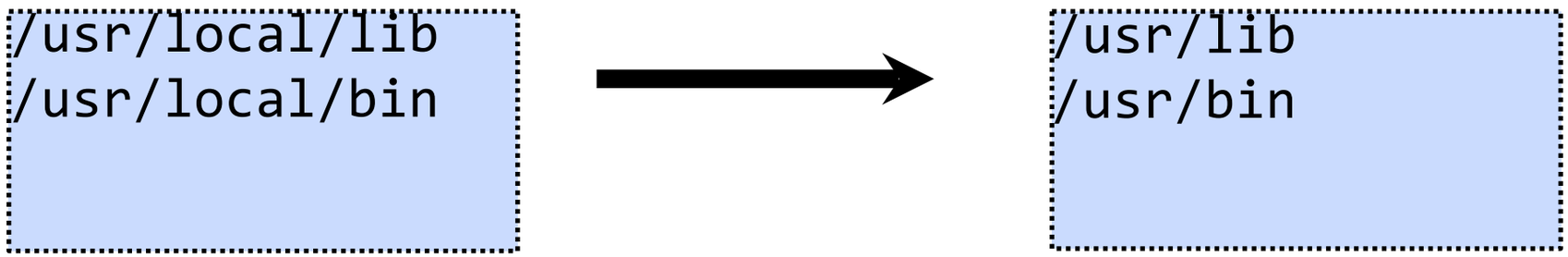
nights.txt

Step 1:sed s/day/night/g days.txt

Step 2:sed s/day/night/g days.txt > nights.txt

sed - **s**tream **ed**itor (find and replace)

- Example 3

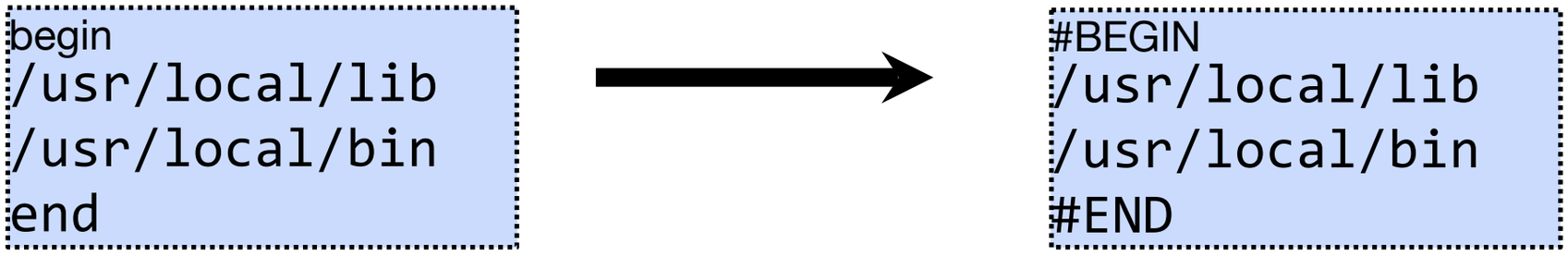


paths.txt **newpaths.txt**

```
sed s/\/usr\/local/\/usr/g paths.txt
    > newpaths.txt
```

sed - **s**tream **ed**itor (find and replace)

- Example 4



paths.txt **newpaths.txt**

step1: sed s/begin/#BEGIN/ paths.txt > tmp.txt

step 2: sed s/end/#END/ tmp.txt > newpaths.txt

```
sed s/begin/#BEGIN/ paths.txt | sed s/end/#END/ > newpaths.txt

sed -e s/begin/#BEGIN/ -e s/end/#END/ paths.txt > newpaths.txt
```

Remember pipes!

- awk, sed, and pretty much any other utility can be used with pipes!
  - Combine commands to do more complicated operations

grep “10-01-2015” file.txt | awk '{print $7}'

No filename needed since input is from grep!

./script.sh | sed s/' '/,/

may not have file name at all if directly processing output of a program

More Resources

- sed and awk are VERY powerful!
  - We have only discussed basic features

- For more information:
  - **`man awk`** or **`man sed`**
  - http://www.grymoire.com/Unix/Awk.html
  - http://www.grymoire.com/Unix/Sed.html
  - google

Outline

- File Processing Utilities
- **Shell Scripting**
  - **automating command line tasks**
- Programming in Linux
  - compilers and accessories

Shell Scripting

- Most shells support some kind of scripting language
  - Script = program that does not need to be compiled to be run
- Can create scripts to automate commands you run regularly
  - Backing up files
  - Compiling programs
  - Anything you can do from the command line...
- Pros: quick to write and run
- Cons: relatively slow, limited library of functions

Steps for Writing a Bash Script

1. Type at least one command in a file
2. Add a line at the top of the file to label it as a script
   - In bash this is:
     - `#!/bin/bash`
3. Use the **chmod** command to make the file executable
4. Run the file

- We will go over each of these steps with examples in the following slides

```
#!/bin/bash

echo “Hello”
```

Output: Hello

Bash shell scripting: Variables

- Declare and assign a variable:
  - `x=123  # no spaces around =`
  - `y='hello world'`
  - `z="$x, $y"`

x = 123 # Note: ERROR when there are spaces between variable x and 123.

if z='$x,$y', what will be written to standard output?

- Print a variable to screen:
  - `echo $a`

- **Warning:** Bash is very particular about syntax
  - Be careful with spaces

Example Script simple.sh

- Use a # symbol at beginning of a line to comment it out

```
#!/bin/bash

# This is a comment
class="CS"
num=197

echo "Hello $class $num"
```

The first line is called shebang; it specifies the which interpreter to execute the script.

Output: Hello CS 197

Running the Script

- First, make the script simple.sh executable
  - `chmod u+x simple.sh`

- Next, run the script from the command line
  - To run `simple.sh` script, use: `./simple.sh`

- Typing `simple.sh` at command line won't work because the files in the current directory are not on your "path"
  - $PATH variable = directories Linux will check when you try to run a program
- To run a program in the current directory, use `./prog_name`

Bash shell scripting: Getting input

- Can read in lines of input from the command line using:
  - **`read varname`**
  - will read one line

```
echo “What is your name?”
read name
echo “Hi $name.”
```

- Can set variables based on output of other commands
  - **`var=`command and arguments`**`**
  - use non-shifted tilde character (called the back tick `)

```
today=`date`
lastline=`tail -n 1 file.txt`
echo $today : $lastline
```

Bash shell scripting: Script arguments

- Can pass command line arguments
  - Reference using $1, $2, $3, ...
    - first, second, third, ... argument

```
#script.sh accepts 2 args

file1=$1
file2=$2

echo “first file: $file1”
echo “second file: $file2”
```

command line arguments

```
elnux>./script.sh f1.txt f2.txt
```

first file: f1.txt
second file: f2.txt

Bash shell scripting: Script arguments

- Lots of special variables:
  - **$0 = script base name**
  - **$# = number of arguments**
  - **$@ = all arguments**
  - **$$ = process ID number**
  - **${#string} = length of $string**
  - ... lots more

```
#script.sh
echo “Script Name: $0”
echo “Num. Parameters: $#”
echo “All parameters are $@”
echo “My process ID: $$”

filename=$0
echo "$filename len: $((${#filename}-2))"
```

```
elnux>./script.sh f1.txt f2.txt
```

```
Script Name: ./script.sh
Num. Parameters: 2
All parameters are f1.txt f2.txt
My process ID: 9616
./script.sh len: 9
```

Bash shell scripting: **`if`** statements

- Can use **`if`**, **`elif`** (else if), and **`else`**
  - Usage:

```
if [ TEST VAR ]     or
if [ VAR1 TEST VAR2 ]
```

  - **`TEST`** = flag for type of test
  - **`VAR`** = variable to be tested
  - The SPACE after [and before] is important
  - End if statement by fi

Flag	**Meaning**
-n	non zero length
-z	zero length
-f	file exists
if [$s=$t]	equal as STRINGS
if [$a -eq $b]	equal as INTEGERS
-lt -gt -le -ge	less/greater than

Bash shell scripting: `if` statement example

```
dir=$1

if [ -d “$dir” ]
then
echo “Reading files in $dir”
else
echo “Not a directory”
fi


if [ -w “$dir/file.txt” ]
then
echo “writable”
elif [ -r “$dir/$file.txt” ]
then
echo “readable”
else
echo “can’t write or read”
fi
```

Flag	Meaning
-d	is a directory
-w	has write permission
-r	has read permission

Bash shell scripting: `if` statement exercise

- Usage:

```
if [ TEST VAR ]     or
if [ VAR1 TEST VAR2 ]
```

Flag	**Meaning**
-n	non zero length
-z	zero length
-f	file exists
if [$s = $t]	equal as STRINGS
if [$a -eq $b]	equal as INTEGERS
-lt -gt -le -ge	less/greater than

- Write an if statement that:
  - checks if the file ‘output.txt’ exists

if [-f “output.txt”]

  - checks if integer variable ‘a’ is equal to integer variable ‘b’

if [$a -eq $b]

  - checks if integer variable ‘a’ is greater than variable ‘b’

if [$a -gt $b]

Bash shell scripting: loops

- Bash supports **`for`**, **`while`**, and **`case`**
  - for loop is more like “for each”

```
for VAR in <list of vars>
do
# for loop body
done

var=0
while [ $var -lt 100 ]
do
# while loop body
var=$((var+1))
done
```

```
#!/bin/bash
for day in Mon Tues Wed Thurs
do
echo $day
done

for x in 1 2 3 4 5 6 7
do
echo “Day $x”
done

for x in {1..7}
do
echo “Day $x”
done
```

Bash shell scripting: loops

- Asterisk (*) can be used within scripts
  - Will match names of files in the current directory

```
#!/bin/bash
for f in *
do
head $f
done

for f2 in *.txt
do
tail $f2
done
```

Will be useful for assignment 4 and 5!

Bash shell scripting: loops exercise

- **SYNTAX:**
- **for VAR in <list of vars>**
- **do**
    - **# for loop body**
- **done**

- Print the numbers 100 through 200

```
for x in {100..200}
do
   echo $x
done
```

- Print the first 4 months of the year

```
for x in Jan Feb March April
do
   echo $x
done
```

More resources

- Nice intros:
  - http://www.panix.com/~elflord/unix/bash-tute.html
  - http://www.doc.ic.ac.uk/~wjk/UnixIntro/Lecture8.html

- Longer references:
  - http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
  - http://tldp.org/LDP/abs/html/

Outline

- File Processing Utilities
- Shell Scripting
  - automating command line tasks
- **Programming in Linux**
  - **compilers and accessories**

Programming in Linux

- Compiled languages - larger programs, optimized performance
  - C, C++, C#
  - Java

- Scripting languages - ease of development, cross platform
  - Shell scripting
  - Python
  - Perl
  - Ruby

C / C++ programming

- `gcc` and `g++` are the main C / C++ compilers
- Usage: `gcc file.c -o execname`
  - If you don't use -o flag, default name is "a.out"
- For projects with multiple files:

```
g++ -c file1.cpp      # produces file1.o
g++ -c file2.cpp      # produces file2.o

# link the .o files into an executable
g++ file1.o file2.o -o execname
```

```
g++ file1.cpp file2.cpp -o execname
```

Building big programs

- If you are installing a program from source code or writing big apps...
- Don’t want to manually compile and link each file and library
- **Makefiles** simplify the build process
  - Contains a list of “build targets”
  - Run `make <target>` to automate compilations steps
- Common procedure when installing from source:
  - `./configure` - checks that you have libraries, prepares configuration files
  - `make` - compiles the main application code
  - `make install` - copies the new executables to their final paths
    - (may require sudo!)

Simple Makefile example

```
# A simple Makefile

myprogram: main.o library.o
	gcc -o myprogram main.o library.o

library.o: library.c library.h header.h
	gcc -c library.c

main.o: main.c header.h
	gcc -c main.c

clean:
	@rm -f myprogram main.o library.o
```

make myprogram
will automatically build the other “targets”

Other languages

- Java
  - Compile to create .class files:
    - javac file.java
    - javac file1.java file2.java file2.java
  - Run: java file (where file.java contains a main method)
- Python
  - type ‘python’ at command prompt to use an interactive interpreter
  - Run script by: python filename.py
- Perl/Ruby
  - Run script by: perl filename.pl or ruby filename.rb