ITP20004 – Open-Source Software Labs

Project Management

Charmgil Hong

charmgil@handong.edu

Spring, 2023

Handong Global University

Announcements

- Weekly schedule

Week	Mon		Week	Thur	
1	Course overview, motivation, administrivia		1	CPR: C Programming Reinforcement - Functions	
2	Computer organization and Linux environment (1)		2	CPR: C Programming Reinforcement - Strings	
3	Computer organization and Linux environment (2)		3	CPR: C Programming Reinforcement - User-defined types, and memory allocation	
4	Basic Linux commands + Writing code on Linux (vim)		4	Getting started with Linux / Hands-on Linux command-line tools	
5	More Linux commands		5	CPR: C Programming Reinforcement - Understanding compilation and build process	
6	Project management (1)	**Proj 1 출제**	6	Project management (2)	
7	-		7	Project: BASIC interpreter	Project 1
8	-	Tuesday night: Midterm exar	8	Project QnA (Optional)	
9	CPR: C Programming Reinforcement - Accessing files and		9	Debugging with GDB + Unit testing with gtest	**Project 1 due: Week#9 Saturday**
10	Shell script			Shell script	
11	Code review GNU utilities	**Proj 2 출제**	12	Writing an application in C	
12	Github and open-source community			Using Github	**Project 2 due: Week#12 Saturday**
13	Computer network basics		10	Linux network commands	
14	Project: Text-based Game		14	Socket programming	
15	Project: Multi-user game	**Proj 3 출제**	15	Project: Multi-user game	
16	Final exam		16		**Project 3 due: Week#16 Saturday**

Announcements

- For each lab
  - Before a lab, every student submits a pre-lab report (worksheet-type assignment) – **individual work**
  - After a lab, each team sees and reports to the TA with the results – **team work**

- Post-lab report session (lab #7)
  - Tuesday 9pm

Announcements

- Team assignment for Weeks 6-11

학번	이름	학번	이름	학번	이름	학번	이름
21	쉘튼	21	조유진	18	박현우	21	서준예
20	나예원	20	김가현	18	임건호	22	이온유
18	송민준	20	이준형	22	곽철호	19	이지명
20	송산	19	유건민	20	이상현	20	정성호
21	김연희	17	김홍찬	21	송영은	21	이선환
20	방석민	18	현승준	18	정현준	20	비보시놉아잣
18	조성민	20	유승준	18	최정겸	21	사우지아유인
20	윤예람	22	이채연	20	김승환	22	반대준
22	윤유원	22	황찬영	21	최지안	22	서종현
18	마석재	20	정지원	18	김두환	20	김유겸

Agenda

- Task #1 – Shell scripts

Task #1 – Shell Scripts

- Let us write some shell scripts
  - ***script1.sh*** – Make an upside down pyramid
    - prints out a pyramid of height *n*; the value of *n* is declared within the script
    - Assume *n* is ranging between 1 and 12
    - If *n* is smaller than 1, print nothing
    - If *n* is larger than 12, print out an error message (do not print a pyramid)
    - `E.g., n = 3`

```
*****
 ***
  *
```

`n = 5`

```
*********
 *******
  *****
   ***
    *
```

Task #1 – Shell Scripts

- Let us write some shell scripts
  - ***script2.sh*** – Write a guessing game script that generates a random number between 1 and 100 at the script starts, and asks the user to guess the number
    - The script iterates by taking a number from the user and telling whether the user input was lower or higher than the random number
    - The iteration stops when the user enters the same value as the random number

    - `$RANDOM` generates a random number
    - `num=$((RANDOM%200+100))` sets variable `num` to a random number between 100 and 200

Task #1 – Shell Scripts

- Let us write some shell scripts
  - ***script2.sh*** – Write a guessing game script that generates a random number between 1 and 100 at the script starts, and asks the user to guess the number
    - E.g.,

```
Guess a number (1-100)?
17
Lower
31
Lower
72
Higher
59
Correct! You pick the right number!
```

Task #1 – Shell Scripts

- Let us write some shell scripts
  - ***script3.sh*** – Write a shell script that takes an integer *n*, **via a command line argument**, finds and prints out the *n*-th Fibonacci number **using an iterative approach**
    - Assume Fibonacci(0) = 0 and Fibonacci(1) = 1 are the base cases
    - Fibonacci(*i*) = Fibonacci(*i*-2) + Fibonacci(*i*-1)
    - Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
    - E.g., `$ script3.sh 4`
      `fib(4)=3`
      `$ script3.sh 7`
      `fib(7)=13`

Task #1 – Shell Scripts

- Let us write some shell scripts
  - ***script4.sh*** – Write a shell script that takes a directory name (absolute path), examines all files in that directory, and counts the total number of lines that all the files contain
    - Assume all files are text file
      - Example: `/home/lsso/files`
    - *E.g.*, `$ script4.sh /home/lsso/files`
      `10`

Task #1 – Shell Scripts

- Hints: Feel free to access and use the snippets from https://linuxhint.com/30_bash_script_examples/

Post-lab report

- 9:00pm, Tuesday, May 9, 2023
  - At NTH 413

Shell Scripts

- Linux shells are script language interpreters, as well as command line interfaces
  - Shell script: file (text file) containing a series of controls and commands to be run by the shell
  - Shells allow us to
    - Program (combine/reuse) existing commands with command control mechanisms (if, for, while, switch)
    - Set and use our own variables

First Shell Scripts

- `hello.sh`

```
#!/bin/bash          ← tells Linux that the file is to be executed by /bin/bash
# This is a comment
echo Hello World!     # This is a comment, too
```

  - `#!/bin/bash`: shebang - a character sequence consisting of the characters number sign and exclamation mark (#!) at the beginning of a script
  - "#" character is a comment marker (as in many scripting languages)
    - The shebang line is usually ignored by the shell script interpreter

First Shell Scripts

- How to make it runnable?

```
$ chmod 755 hello.sh
./hello.sh
Hello World!
```

  - Make your script file executable & run it

First Shell Scripts

- `hello.sh`

```
#!/bin/bash
# This is a comment
echo Hello World!    # This is a comment, too
```

  - `echo Hello World!`
    - "echo" is a Linux built-in that takes multiple arguments and print them on the screen
  - What if we run the following lines?
    - `echo Hello World!`
    - `echo Hello        World!`
    - `echo "Hello        World!"`
    - `echo Hello * World!`
    - `echo "Hello * World!"`

Variables

```
#!/bin/bash
MY_MESSAGE="Hello World!"
echo $MY_MESSAGE
```

```
#!/bin/bash
echo What is your name?
read NAME
echo "Hello $NAME, How are you doing?"
```

Variable Scopes

```
#!/bin/bash
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

- Execution
  - `$ ./myvar2.sh`
    ```
    MYVAR is:
    MYVAR is: hi there
    ```
  - `$ MYVAR=hello`
    `$ ./myvar2.sh`
    ```
    MYVAR is:
    MYVAR is: hi there
    ```

Variable Scopes

```
#!/bin/bash
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

- `export` variables that need to be inherited (scope = within the interactive shell)
  - `export` is used to mark variables and functions to be passed to child processes
  -
```
$ MYVAR=hello
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

Variable Scopes

```
#!/bin/bash
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I am creating a file after your name"
touch $USER_NAME_file
touch "${USER_NAME}_file”
```

- Differences?
  - `touch "$USER_NAME_file”:` Looks for a variable named “`USER_NAME_file`” which does not exist – causing an error
  - `touch "${USER_NAME}_file”:` This creates a file using “`USER_NAME`” followed by the ”_file” string

Escape Characters

```
#!/bin/sh
echo ”Hello World"
echo ”Hello \"World\""
echo ”Hello \\ World"
```

- Differences?
  - `\"`: becomes double quotes (")
  - `\\`: becomes a single backslash (\)

Loops

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo ”Iterating ... # $i"
done
```

- Results
  - Iterating ... # 1
    Iterating ... # 2
    Iterating ... # 3
    Iterating ... # 4
    Iterating ... # 5

Loops

```
#!/bin/bash
for  i in hello 1 * 2 goodbye
do
  echo "Iterating ... i=$i"
done
```

- Results
  - Iterating ... i=hello
    Iterating ... i=1
    Iterating ... i=(filename)
    ... ...
    Iterating ... i=(filename)
    Iterating ... i=2
    Iterating ... i=goodbye

Loops

```
#!/bin/bash
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)”
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

- Results
  - Please type something in (bye to quit)
    hi
    You typed: hi
    ... ...
    bye
    You typed: bye

Branches

```
#!/bin/bash
read INPUT_STRING
if ["$INPUT_STRING" = "hi" ]; then
  echo ”hi there!"
elif [ “$INPUT_STRING" = ”bye" ]; then
  echo "good bye-"
else
  echo ”I didn’t get that"
fi
```

Branches

```
#!/bin/bash
read X
if [ "$X" -lt "0" ]; then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
```

- -lt: less than (<)
- -le: less than or equal to (<=)
- -gt: greater than (>)
- -ge: greater than or equal to (>=)
- -eq: equal to (==)
- -ne: not equal to (!=)

Arrays

- Arrays can store a virtually unlimited number of data elements
  - *C.f.*, Variables store single data elements

- Two types of arrays in shell scripts
  - Indexed arrays: Store elements with an index starting from 0
  - Associative arrays: Store elements in key-value pairs (hash tables)

- Declaration
  - Indexed arrays
    - `declare -a idx_array`
      `idx_array[0]=value`
  - Associative arrays
    - `declare -A assoc_array`
      `assoc_array[key]=value`

Arrays

- Examples

```
#!/bin/bash
declare -a arr=(1 2 3 4 5 6 7 8 9 0)
for i in #{arr[@]}
do
    echo $i
Done
echo ”array size is ${#arr[@]}”
```

```
#!/bin/bash
declare -A hashtable=( ["moo"]="cow" ["woof"]="dog")
for key in ${!hashtable[@]}
do
    echo ${key} ${hashtable[${key}]}
done
echo ”array size is ${#hashtable[@]}”
```

Functions

- Examples

```
#!/bin/bash
# A simple script with a function...

add_a_user()
{
  USER=$1
  PASSWORD=$2
  COMMENTS=$3
  echo "Adding user $USER ..."
  echo $ useradd -c "$COMMENTS" $USER
  echo $ passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein "Bob Holness the presenter"
add_a_user fred badpassword "Fred Durst the singer"
add_a_user bilko worsepassword "Sgt. Bilko the role model"
echo "End of script..."
```

Command-Line Argument

- Example

```
#!/bin/bash
echo "Script was called with $@"
```

```
$ ex15_cmdlinearg.sh args from cmd
Script was called with args from cmd
$ ex15_cmdlinearg.sh 1   2    3
Script was called with 1 2 3
```

Recursion

- Example: factorial.sh

```
#!/bin/bash

factorial()
{
    if [ "$1" -gt "1" ]; then
        i=`expr $1 - 1`
        j=`factorial $i`
        k=`expr $1 \* $j`
        echo $k
    else
        echo 1
    fi
}

while :
do
    echo "Enter a number:"
    read x
    factorial $x
done
```