SEMINARY 2: SHELL PROGRAMMING

With lecture notes

1 UNIX SHELL PROGRAMMING

1.1 STANDARD INPUT/OUTPUT/ERROR AND I/O REDIRECTIONS

1. `0` = standard input – where you read from when you use “scanf” or “gets” in C, “cin” in C++, or “input” in Python.
2. `1` = standard output – where you write when you use “printf” or “puts” in C, “cout” in C++, or “print” in Python.
3. `2` = standard error – similar to the standard output, but conventionally used to display errors, in order to avoid mixing results with errors.
4. I/O redirections
    a. What if I want the output of a command to be stored in a file?
        i. `ls -l --color=never /etc > output.txt`
    b. What if I want to add the output of another command to the same file?
        i. `ps -ef >> output.txt`
    c. What if I want the standard output of a command to be sent to the standard input of another command?
        i. `ls| sort`
    d. What if I want the standard input to be taken from a file?
        i. `sort < a.txt`

1.1.1 LECTURE 2

1.2 COMMAND TRUTH VALUES

1. The truth value of a command execution is determined by its exit code. The rule is the opposite of the C convention, with `0` being `true`, and anything else being `false`. Basically, there is only one way a command can be executed successfully, but many ways in which it can fail. The exit code is <u>not</u> the output of the command.
2. There two standard commands `true` and `false`, that simply return `0` or `1`.
3. Command `test` offers a lot of options for comparing integers, strings and verifying file and directory attributes

1.2.1 LECTURE 2

1. Commands can be chained using logical operators `&&` and `||`. Lazy logical evaluation can be used to nice effects. The negation operator `!` reverses the truth value of a command.
    a. `true || echo This should not be displayed`
    b. `false || echo This should definitely by displayed`
    c. `true && echo This should also be displayed`
    d. `false && echo Should never be displayed as well`
    e. `grep -q “=” /etc/passwd || echo There are no equal signs in file /etc/passwd`
    f. `test -f /etc/abc || echo File /etc/abc does not exist`
    g. `test 1 -eq 2 || echo Not equal`
    h. `test “asdf” == “qwer” || echo Not equal`
    i. `! test -z “abc” || echo Empty string`
2. Test command conditional operators
    a. String: `==`, `!=`, `-n`, `-z`
    b. Integers: `-lt`, `-le`, `-eq`, `-ne`, `-ge`, `-gt`
    c. File system: `-f`, `-d`, `-r`, `-w`, `-x`

1.3 SHELL VARIABLES AND EMBEDDED COMMANDS

1.3.1 LECTURE 2

1. Defined as A=”Tom” or B=5
2. Embedded commands
    a. Delimited by ` (back-quote). Are replaced by the output of the command. Store a command output in a variable: N=`grep “/gr211/” /etc/passwd | wc -l`
3. Referred as $A or ${A}
    a. echo $A is a human
    b. echo $Acat is a feline or an application server – doesn’t work
    c. echo ${A}cat is a feline or an application server
4. When used in strings delimited by “, variables and embedded commands will be replaced by their value. Strings delimited by ‘ do not allow any substitutions in their content.
    a. echo “AA is a GPS navigator”
    b. echo “There are `grep “/gr211/” /etc/passwd | wc -l` students in group 211”

1.4 SHELL SCRIPTS

1.4.1 LECTURE 2

1. Any text file with execution permissions can be a script, as long as it contains commands interpretable by the current shell. Comments start with #
2. Hello World example
    a. Create file a.sh with the content below

```
echo Hello World
```

    b. Give the script execution permissions using chmod 700 a.sh
    c. Execute the script using ./a.sh
3. Permissions
    a. Run ls -l and see the first 10 characters on each line
        i. The first character tells the file type: - is a regular file, d is a directory
        ii. Characters 2-4 show the permissions for the owner of the file (field 3 displayed by ls -l)
        iii. Characters 5-7 show the permissions for the group of the file (field 4 displayed by ls -l)
        iv. Characters 8-10 show the permissions for everybody else
    b. Each permission triplet describes the read, write and execution permissions
    c. Can be described as a number, by considering each of the 3 positions to be a binary digit.
        i. 7 = 111 = rwx
        ii. 6 = 110 = rw-
        iii. 5 = 101 = r-w
    d. Command chmod is used to assign permissions to files
        i. chmod 700 a.sh gives the owner of the file full permissions, and nothing to the group or the others
4. Hello World example with shell specification
    a. Create file hello.sh with the content below

```
#!/bin/bash

echo Hello World
```

    b. Give the script execution permissions using chmod 700 hello.sh
    c. Execute the script using ./hello.sh
5. Special variables
    a. $0 – The name of the command
    b. $1 - $9 – Command line arguments; $n the n-th argument; shift n to shift cmd line arguments by n
    c. $* or $@ - All the arguments together as string or as array
    d. $# - Number of command line arguments
    e. $? – Exit code of the previous command

6. Special variables example, special-vars.sh

```
#!/bin/bash

echo Command: $0
echo First four args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

true
echo Command true exited with code $?

false
echo Command false exited with code $?
```

   a. `chmod 700 special-vars.sh`
   b. `./special-vars.sh a b c d e f g h i j k l m`

7. Accessing arguments using `shift`. Script `using-shift.sh`

```
#!/bin/bash

echo Command: $0
echo First four args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

shift
echo Some args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

shift 3
echo Some args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#
```

   a. `chmod 700 using-shift.sh`
   b. `./using-shift.sh a b c d e f g h i j k l m`

1.5 UNIX SHELL FOR LOOP

1.5.1 LECTURE 2

1. Similar to the Python `foreach.` The variable cycles through a list of space separated values. Basic example `using-for.sh`, showing the do on the same line or on the next line. Semicolon is the command separator.

```
#!/bin/bash

for A in a b c d; do
    echo Here is $A
done

for A in a b c d
do
    echo Here is $A
done
```

   a. `chmod 700 using-for.sh`
   b. `./using-for.sh`

2. Iterating over the command line arguments, `for-args.sh`, showing the short and not very intuitive second possibility

```
#!/bin/bash

for A in $@; do
    echo Arg A: $A
done

for A; do
    echo Arg B: $A
done
```

   a. `chmod 700 for-args.sh`
   b. `./for-args.sh a b c d e f g h i j k l m`

1.5.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will do more complex examples.

Remember hw.sh, chmod +x hw.sh, ./hw.sh:

```
#!/bin/bash
Echo “Hello `whoami`!”
```

1. The list of values through which for iterates can be specified either explicitly as above, or through filename wildcards, or embedded commands
    a. Count all the lines of code in the C files in the directory given as command line argument, excluding lines that are empty or contain only blank spaces

Shell Script

```
#!/bin/bash

S=0
for F in $1/*.c; do
    N=`grep "[^ \t]" $F | wc -l`
    S=`expr $S + $N`
done
echo $S
```

Instead of \t press TAB or you can use option -P with \t where allowed – it uses Perl not Extended regular expressions!

Shell using Awk scr.awk below

```
#!/bin/bash

awk -f scr.awk $1/*.c
```

And scr.awk

```
#!/bin/awk -f
BEGIN { s=0 }
/[^ \t]/  { s+=1 }
END {print s}
```

    b. Filenames that contain spaces will cause problems here

2. Filename wildcards
    a. Similar but much simpler than regular expressions
    b. Rules:
        i. `*` - Matches any sequence of characters, including a void sequence, but not the first dot in a filename
        ii. `?` - Matches any single character, but not the first dot in a filename
        iii. `[abc]` – List of optional characters, support ranges like the regular expressions
        iv. `[!abc]` - Negated list of optional characters (similar to `[^abc]` from regular expressions)
    c. Example: list all the file starting with a letter and having an extension of exactly two characters
        i. `ls [a-zA-Z]*.??`
3. Count all the lines of code in the C files in the directory given as command line argument **and its subdirectories**, excluding lines that are empty or contain only blank spaces

```
#!/bin/bash

S=0
for F in `find $1 -type f -name “*.c”`; do
    N=`grep -E -v -c "^$" $F`
    S=`expr $S + $N`
done
echo $S
```

    a. Filenames that contain spaces will cause problems here as well
    b. Solving this problem with `find ... | while read F`, will avoid the space in file name problems, but incrementing S will not work because while is executed in a sub-shell. Solutions to overcoming this are outside the scope of this course.

1.6 UNIX SHELL IF/ELIF/ELSE/FI STATEMENT

1.6.1 LECTURE 2

1. Every command is a condition. Commands can be grouped with parentheses and logical operators
    a. Check whether a file does not exist or if it exists whether it is not readable
    b. `! test -f a.txt || ( test -f a.txt && ! test -r a.txt )`
2. Present the basic IF syntax, using script `basic-if.sh` which checks each argument and announces whether it is a file, or a directory, or a number, otherwise it states that it does not know what it is. Just like `do`, `then` can be either on the same line or on the next line. Do not introduce the [...] syntax.

```
#!/bin/bash

for A in $@; do
    if test -f $A; then
        echo $A is a file
    elif test -d $A
    then
        echo $A is a dir
    elif echo $A | grep -q "^[0-9]\+$"; then
        echo $A is a number
    else
        echo We do not know what $A is
    fi
done
```

    a. `chmod 700 basic-if.sh`
    b. `./basic-if.sh /etc /etc/passwd . 1234 a2b rr`

1.6.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will introduce the [...] syntax of the conditions and do more complex examples.

1. **To make the condition look a bit more natural, there is a second syntax, in which `[` is an alias of command `test` and `]` marks the end of the command `test`.** Pay attention to leaving spaces around these square brackets or there will be syntax errors. The basic IF example from the lecture, can be re-written as follows

```
#!/bin/bash

for A in $@; do
    if [ -f $A ]; then
        echo $A is a file
    elif [ -d $A ]
    then
        echo $A is a dir
    elif echo $A | grep -q "^[0-9]\+$"; then
        echo $A is a number
    else
        echo We do not know what $A is
    fi
done
```

1.7 UNIX SHELL WHILE STATEMENT

1.7.1 LECTURE 2

1. Read user input until the input is stop
2. The user input is read with command read which stores the input in the variable given as argument
3. Script basic-while.sh

```
#!/bin/bash

while true; do
    read X
    if test "$X" == "stop"; then
        break
    fi
done
```

    a. `chmod 700 basic-while.sh`
    b. `./basic-while.sh`
4. Find all the files in the directory given as first command line argument, larger in size than the second command line argument. Script `large-files.sh`

```
#!/bin/bash

D=$1
S=$2

find $D -type f | while read F; do
    N=`ls -l $F | awk '{print $5}'`
    if test $N -gt $S; then
        echo $F
    fi
done
```

a. `chmod 700 large-files.sh`

b. `./large-files.sh`

c. This example also makes it clear why the AWK program must be provided between apostrophes, not quotes

1.7.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will do more complex examples.

1. The while loop also accepts the [...] condition syntax
2. Read the console input until the user provides a filename the exists and can be read

```
#!/bin/bash

F=””
while [ -z "$F" ] || [ ! -f "$F" ] || [ ! -r "$F" ]; do
    read -p "Provide an existing and readable file path:" F
done
```

or

```
#!/bin/bash

F=””
while test -z "$F" || ! test -f "$F" || ! test -r "$F"; do
    read -p "Provide an existing and readable file path:" F
done
```

1.8 UNIX SHELL PROGRAMMING EXAMPLES

1.8.1 LECTURE 3

1. Find all the students in group 211
   a. `grep "/an1/gr212/" /etc/passwd`
2. Display the most frequent names of the users (first, middle, etc. but not last) in the system. This is similar to a problem solved in lab 3/4 but still a little different as there are more non-last names to a user. Present the thought process for solving the problem, and each command in the pipe chain.

```
awk -F: '{print $5}' /etc/passwd | cut -d ' ' -f2- | \
sed "s/[ -]/\n/g" | tr '[A-Z]' '[a-z]' | grep -v "\.\|^.$" | \
sort | uniq -c | sort -n -r | less
```

   a. The data we need is in the 5th field of `/etc/passwd`
   b. Of the 5th field we need all the words except for the first. We can do this with `cut`, `sed`, or `awk`.
   c. Now we put each word on a line, by replacing space with newline. As some names are linked by dash, we also replace the dash with new line
   d. To avoid upper/lower case issues with `sort` and `uniq`, we convert everything to lower case. We can do it with `sed`’s `y` command, but we would need to type the whole alphabet, so we use command `tr` which also does transliteration and supports a shorter input.
   e. We eliminate initials (as much as we can) by eliminating all lines that contain either dot or a single character
   f. Finally we sort the names and `uniq`-count them and sort them descending, displaying them in a pager.
3. Stop student processes older than the number of seconds given as command line argument.

```
#!/bin/bash

for X in `ps -ef |grep  -v "^root " | tail -n +2 | awk '{print $1 ":" $2}'`; do
    U=`echo $X|cut -d: -f1`
    P=`echo $X|cut -d: -f2`

    echo $U $P
    if grep "^$U:" /etc/passwd | cut -d: -f6 | grep -q "/scs/"; then
        A=`ps -o etime $P | tail -n 1 | awk -F: '{print ($1*60+$2)}'`
        if [ $A -ge $1 ]; then
            echo "Should kill $U $P $A"
        fi
    fi
done
```

a.

   i. Explain the [...] condition syntax for those who didn’t have the seminar yet
   ii. To speed up the script, we skip all processes belonging to `root`.
   iii. As `ps` displays a header, we skip that as well

```
#!/bin/bash

ps -ef | \
grep  -v "^root " | \
tail -n +2 | \
awk '{print $1, $2}' | \
while read U P; do
    echo $U $P
    if grep "^$U:" /etc/passwd | cut -d: -f6 | grep -q "/scs/"; then
        A=`ps -o etime $P | tail -n 1 | awk -F: '{print ($1*60+$2)}'`
        if [ $A -ge $1 ]; then
            echo "Should kill $U $P $A"
        fi
    fi
done
```

b.

   i. Same solution as above, but with while and with command splitting over multiple lines to make it more readable

4. Present the main sources of Shell script syntax errors
   a. Missing spaces around the condition square brackets
   b. Missing quotes in conditions around blank variables

1.8.2 SEMINAR 2

Analyse the code sequences:

echo “expr 1 + 2”
echo `expr 1 + 2`
echo `expr 1+2`
	Foo=sun
	echo $Fooshine
	echo ${Foo}shine
		count=$((count+1))
		count=$((count + 1))
		count=count+1

-1. Verify if a variable is a number

a

b

```
if echo “$var”| grep –q “^-?[0-9]*\.?[0-9]\+$”; then
      echo “$var is a number”
else
      echo “$var is not a number”
fi
```

```
#!/bin/bash
var=a
if [ “$var” –eq “$var” ] 2>/dev/null; then
    echo Is a number
else
    echo Is not a number
fi
```

0. Sort files given as cmd line arguments in ascending order according to file size

```
#!/bin/sh
for i in $* ;do
    if [ -f $i ]
    then
       du -b $i
    fi
done | sort -n
```

1. Write a script that monitors the state of a directory and prints a notification when something changed

```
#!/bin/bash

D=$1
if [ -z "$D" ]; then
  echo "ERROR: No directory provided for monitoring" >&2
  exit 1
fi

if [ ! -d "$D" ]; then
  echo "ERROR: Directory $D does not exist" >&2
  exit 1
fi

STATE=""
while true; do
    S=""
    for P in `find $D`; do
        if [ -f $P ]; then
          LS=`ls -l $P | sha1sum`
          CONTENT=`sha1sum $P`
        elif [ -d $P ]; then
          LS=`ls -l -d $P | sha1sum`
          CONTENT =`ls -l $P | sha1sum`
        fi
        S="$S\n$LS $CONTENT"
    done
    if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then
      echo "Directory state changed"
    fi
    STATE=$S
    sleep 1
done
```

   a. We use sha1sum to get a checksum that is statistically impossible to be identical for different contents
   b. We checksum the details of the file (`ls -l`) as well as its content
   c. For directories, we use the `-d` flag of ls to list the directory details and not its content, and we use the output of `ls -l` for the directory content
   d. We only handle regular files as directories when building the state. We should also address pipes, links, etc but it is outside the scope of this course.

1.8.3 LAB 4, 5

1. Implement and test the script presented in the seminar (see section above)
2. Re-write the same script using conditions without square brackets (ie `[ -f $P ]` becomes `test -f $P`)
3. If there is any time left, solve some of the practice the students find difficult. For instance, here is a solution for problem 10
   a. Display the session count and full names of all the users who logged into the system this month, sorting the output by the session count in descending order. Use the `-t` option of command `last` to get this month's sessions, and the command `date` to generate the required timestamp in the expected format.

```
#!/bin/bash

D=`date +%Y%m`
T=”${D}01000000”
last -t $T | \
sed "s/ .*//" | \
sort | \
uniq -c | \
sort -n -r | \
while read L U; do \
    N=`grep "^$U:" /etc/passwd | cut -d: -f5`
    echo $L $N
done | \
less
```

   b. We could also build the timestamp in a more compact way
      i. `last -t `date +%Y%m01000000``