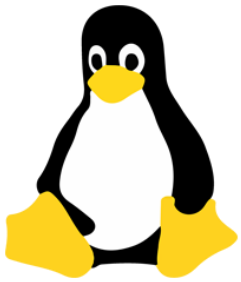

Systems programming

Shell scripting

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

While Statements

- The while structure is a looping structure. Used to execute a set of commands while a specified condition is true. The loop terminates as soon as the condition becomes false. If condition never becomes false, loop will never exit.

```
while expression
    do
            statements
    done
```

```
$ cat while.sh
#!/bin/bash
echo –n "Enter a number: "
read x
let sum=0
let i=1
while [ $i -le $x ]; do
sum=$(($sum+$i))
i=$(($i+1))
done
echo "the sum of the first $x numbers is: $sum"
```

–le: less or equal

Continue Statements

- The continue command causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

```
$ cat continue.sh
#!/bin/bash
let LIMIT=19
echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)"
a=0
while [ $a -le $LIMIT ]; do
        a=$(($a+1))
        if [ $a -eq 3 ] || [ $a -eq 11 ]
        then
                continue
        fi
        echo -n "$a "
done
```

–eq: equal

Break Statements

- The break command terminates the loop (breaks out of it).

```
$ cat break.sh
#!/bin/bash
LIMIT=19
echo
echo "Printing Numbers 1 through 20, but something happens after 2 … "
a=0
while [ $a -le $LIMIT ]
do
        a=$(($a+1))
        if [ $a -gt 2 ]
        then
                break
        fi
        echo "$a "
done
```

–gt: greater than

Until Statements

- The until structure is very similar to the while structure. The until structure loops until the condition is true. So basically, it is “until this condition is true, do this”.

```
until expression
    do
            statements
    done
```

```
$ cat countdown.sh
#!/bin/bash
echo "Enter a number: "
read x
echo "Count Down"
until [ $x -le 0 ]
do
            echo $x
            x=$(($x - 1))
            sleep 1
done
echo "GO !"
```

Manipulating Strings

- Bash supports a number of string manipulation operations.

${#string} gives the string length
${string:position} extracts sub-string from $string at $position (skip before position)
${string:position:length} extracts $length characters of sub-string from $string at $position

- Example

```
$ st=0123456789
$ echo ${#st}
        10
$ echo ${st:6}
        6789
$ echo ${st:6:2}
        67
```

Parameter Substitution (learn how useful ? Can be)

${parameter?msg}, if a parameter is set, use it, else print error msg and exit.

main.sh

```
1   echo "Hello World"
2   value=${total?'total is not set'}
3   echo $value
4   echo "Hello World"
```

```
> bash main.sh
Hello World
main.sh: line 2: total: total is not set
exit status 1
```

main.sh

```
1   echo "Hello World"
2   total=10
3   value=${total?'total is not set'}
4   echo $value
5   echo "Hello World"
```

```
> bash main.sh
Hello World
10
Hello World
```

Functions

- Functions make scripts easier to maintain. Basically, it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish.

- Function Declaration:

```
function_name(){
    List of commands
}
```

or

```
function function_name(){
    List of commands
}
```

- The function name must be followed by parentheses, followed by a list of commands enclosed by curly brackets.

Functions - Example

```
#!/bin/bash


# Define you function here
hello()
{
      echo “You are in function hello()”
}


# Main script starts here
echo “Calling function hello()…”
hello
echo “You are now out of function hello()”
```

```
Calling function hello()…
You are in function hello()
You are now out of function hello()
```

- In the above, we called the hello() function by name by using the line: hello. When this line is executed, bash searches the script for the line hello(). It finds it right at the top, and executes its contents.

Passing parameters to a function

- You can define a function that will accept parameters while calling the function. These parameters would be represented by $1, $2 and so on.
- Following is an example where we pass two parameters Zara and Ali and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here
Hello () {
echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

```
Hello World Zara Ali
```

Returning values from a function

- you can return any value from your function using the return command.

```
#!/bin/sh

# Define your function here
Hello () {
echo "Hello World $1 $2"
return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returnd by last command
ret=$?

echo "Return value is $ret"
```

```
Hello World Zara Ali
Return value is 10
```

Functions - Example

$ cat function.sh

```
#!/bin/sh
function check() {
if [ -e "/home/$1" ]
then
return 1
else
return 0
fi
}

echo -n "Enter the name of the file: "; read x
check $x
ret=$?
if [ $ret -eq 1 ]
then
echo "$x exists!"
else
echo "$x does not exist!"
fi
```

Note….
How to pass variables to functions?
How to receive variables in functions?

Functions - Scope of variables

- Programmers used to other languages may be surprised at the scope rules for shell functions.

```
#!/bin/sh
myfunc()
{
x=2
}
x=1
echo "x is $x"
myfunc
echo "x is $x"
```

```
x is 1
x is 2
```

```
#!/bin/sh
myfunc()
{
x=2
}
myfunc
echo "x is $x"
```

```
x is 2
```

A faction calls another function

- One of the more interesting features of functions is that they can call themselves and also other functions.

```
#!/bin/sh

# Calling one function from another
number_one () {
echo "This is the first function speaking..."
number_two
}

number_two () {
echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

```
This is the first function speaking...
This is now the second function speaking...
```

Permissions and access modes

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

ls-l permission indicators

$ls -l /home/amrood

-rwxr-xr-- 1 amrood users 1024 Nov 2 00:10 myfile

drwxr-xr-- 1 amrood users 1024 Nov 2 00:10 mydir

Permission: read (r), write (w), execute (x) –

Who has full permissions here??

Only the owner

- First column file/directory.
- The first three characters (2-4) represent the permissions for the file's owner.
- The second group of three characters (5-7) consists of the permissions for the group
- The last group of three characters (8-10) represents the permissions for everyone else

Changing Permissions

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod

- **Using chmod in Symbolic Mode**
- **Using chmod with Absolute Permissions**

Using chmod in Symbolic Mode

- Easy.
- You can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Example

$ls -l testfile

-rwxrwxr-- 1 amrood   users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes −

$chmod o+wx testfile #Other

$ls -l testfile

-rwxrwxrwx 1 amrood   users 1024 Nov 2 00:10 testfile

$chmod u-x testfile #User

$ls -l testfile

-rw-rwxrwx 1 amrood   users 1024 Nov 2 00:10 testfile

$chmod g=rx testfile #Group

$ls -l testfile

-rw-r-xrwx 1 amrood   users 1024 Nov 2 00:10 testfile

.

One line merge is possible

$chmod o+wx,u-x,g=rx testfile

$ls -l testfile

-rw-r-xrwx 1 amrood   users 1024 Nov 2 00:10 testfile

Using chmod with Absolute Permissions

Number	Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Example

$ls -l testfile

-rwxrwxr-- 1 amrood   users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes –

$ chmod 755 testfile

$ls -l testfile

-rwxr-xr-x 1 amrood   users 1024 Nov 2 00:10 testfile

$chmod 743 testfile

$ls -l testfile

-rwxr---wx 1 amrood   users 1024 Nov 2 00:10 testfile

$chmod 043 testfile

$ls -l testfile

----r---wx 1 amrood   users 1024 Nov 2 00:10 testfile

Changing Owners and Groups

Two commands are available to change the owner and the group of files –

- **chown** – The chown command stands for "change owner" and is used to change the owner of a file.
- **chgrp** – The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership

The chown command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.

Following example −

```
$ chown amrood testfile
```

Changes the owner of the given file to the user amrood.

Changing Group Ownership

The chrgp command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

Following example −

```
$ chgrp special testfile
```

Changes the group of the given file to special group.