PRGM 2000 - Bash Study Guide

Setup

Hashspling

`#!/bin/bash` - needs to be first thing in script to make it work

Permissions:

`chmod u+x <script>` - this will make it so you can execute a script

Running Scripts:

`./<script>` - implicit execution - script can be found and run
`bash <script>` - explicit execution - script told what to use to run it

Comment block example

```
#!/bin/bash
#Authors Name:
#Description:
Date:
```

Script Debugging

`-x` - prints the command before running it
`-v` - writes the verbose version of the script
`bash <script> -<v/x>` - this is the usage
or `set -x` / `set -v` in the script

Special Characters / Metacharacters

`""` - double quotes - hides most special characters (**except $**)
`''` - single quotes - hides **all** meta characters
`\` - backslash - same as single quoting a **single character**
`~` - tilde - shortcut for current user's home dir

Exit Codes

- 0-255

- 0 is a success
- above 0 is a failure, the number is the error code
  to get the code:
  `echo $?` - this will print the exit code
- **it only shows the LAST command's exit code**

Test command

- has two forms:
  `test <something>` or
  `[ <something> ]` - note the spaces around the thing
- if the test is true = exit code is 0
- if the test is false = exit code not 0

`if [ "$USERVALUE" = "1" ]` - checks for the value of `$USERVALUE` to be `1`
`if test "$USERVALUE" = "1"` - the same as above with `test` instead of `[]`
^ syntax of **test command**

Variables

- don't put spaces between things
- variables are default set to string values

`MYVAR="Test1"` - this would make a variable called **MYVAR** with a **string value of Test1**
`echo $MYVAR` - would have this output: `Test1`

`echo -n "Enter Value: "; read MYVAR` - the output of this would be `Enter Value:`
then the user input would become the **value** of `MYVAR`

Numerical Variables

`MYVAR=123` is still a string
`let RESULT=$MYVAR+5`
`echo $RESULT` output: `128`

Arrays

`MYARRAY=(thing1 thing2 thing3)`
`echo ${MYARRAY[@]}` - this puts the full output: `thing1 thing2 thing3`
`echo ${MYARRAY[1]}` - this would put the **second** item of the array: `thing2`
`echo ${MYARRAY[-1]}` - this would show you the **last** item in the array: `thing3`

`echo ${#MYARRAY[@]}` - this shows you the **length** of the array: `3`
`echo ${#MYARRAY[1]}` - this shows you the **length** of the **second item**: `6`

Functions

command	description
`function test(){`	creating the function
`echo -n "enter value: "; read USERVALUE`	code inside of function
`echo $USERVALUE`	code inside of function
`}`	closing the function
`test`	calling the function

^ this is the syntax of a **function**

Loops and Conditions

- to break out of a single layer of loop: `break`
- to go back to the top of the loop: `continue`
- `exit` will kill the code

if

- often use a test command in conjunction

command	description
`if [ MYVAR = what ]`	beginning the **if** - `[]` represents a test command
`then echo "what detected"`	what happens if condition is met
`else echo "no what detected"`	what happens if condition isn't met
`fi`	closes the **if**

^ syntax of **if** statement

elif

- also used as a part of **if** statement
  `elif <something>` - syntax
- has the function of being a secondary "if" in the if statement

else

- used as a part of **if** statement
  `else <something>` - syntax
- is the last thing in an if statement - if everything else is passed, this is what activates

```
if [ MYVAR = what ]
        then echo "what detected"
elif [ MYVAR = no ]
        then echo "no detected"
else echo "something else detected"
fi
```

^ syntax of it all put together

Indentation - this is a recommendation

Case

- acts like a switch in PowerShell
  `echo "enter value: "; read USERVALUE`
  `case $USERVALUE in`
  `1) echo "user entered value 1" ;;`
  `2) echo "user entered value 2" ;;`
  `*) echo "user entered something else" ;;`
  `esac`

For

- used for incrementally doing things to an array
  `for <declared variable> in <array>`
  `do <something>`
  `done` - used to close for
  ^ syntax

While

- activates while the condition is TRUE
  `while [ USERVALUE = "1" ]`
  `do echo "still 1"`
  `done` - used to close while
  ^ syntax

Until

- the opposite of a while loop
- activates while the condition is FALSE
  `until [ USERVALUE = "1" ]`
  `do echo "not 1 yet"`
  `done` - used to close until
  ^ syntax

Comparators / Conditionals

- you can use PowerShell style comparators (-eq, -ne, -gt, -lt, etc.)
  - **this only works if you are using numbers**
  `USERVALUE=-1`
  `if test $USERVALUE -lt "0"; then echo "pass"; else echo "fail"; fi`
  output: `pass`
- you can also use comparators like (!, -a, -o)
  `test ! $USERVALUE = "1"` - checking for `$USERVALUE` to be **not** `1`
  `test $USERVALUE -a $USERVALUE2 = "1"` - looking for both `$USERVALUE` and `$USERVALUE2` to be `1`
  `test $USERVALUE -o $USERVALUE2 = "1"` - looking for either `$USERVALUE` or `$USERVALUE2` to be `1` (both also works)
- `&&` and `||` can also be used
  `&&` acts like an "and"
  `||` acts like an "or"
  `test $USERVALUE = "1" && test $USERVALUE2 = "2"` - this checks if **both** $USERVALUE = 1 **and** $USERVALUE = 2 to be true
  `test $USERVALUE = "1" || test $USERVALUE2 = "2"` - this only needs one of the statements to be true

I/O Redirection

`for NUMBER in 1 2 3 4`
`do echo $NUMBER`
`done > myfile` - this will put the **output** into "myfile"
^ output syntax

`while read LINE` - used to go line by line - **input from end line**
`do echo $LINE`
`done < myfile` - this get its input **FROM** "myfile"
^ input syntax

Automation

Scheduling Tasks

cron

- used to schedule tasks
- global config file is `/etc/crontab`
- individual config is `/var/spool/cron`
  `minute hour day month dayofWeek command` - this is the syntax in the file

to add a scheduled task: `crontab -e` > select nano (1)
to view: `crontab -l`

cron control

`/etc/cron.allow` and `/etc/cron.deny` are for restricting access

- format is just to put a username on a new line
- if `cron.allow` exists - only these users can use cron
- if `cron.allow` does **NOT** exist - only `cron.deny` users cannot use cron

at

- used to run a one time task at a specific time
  `-l` : view list of scheduled jobs (reg users can see their own)
  `-c` : view system environment at scheduling time
  `-d` : delete a job
  `-f` : run scheduled job from shell script1
  `-r <ID-Number>` : deletes the job
  To exit you need to press `[CTRL+D]`
  `at 5:00pm FRI`
  `at> <command>`
  `[CTRL+D]`
  ^ syntax

batch

- used to run a one time task based on system load drops below 0.8 (basically nothing happening)
- same controls as at

Resource Monitoring

iostat

- used for CPU monitoring

vmstat

- used for network monitoring

free

- used for RAM monitoring

df

- used for file system monitoring
- use `df -h` for human readable

service monitoring

- `ps ax` or
- `systemctl show -p ActiveState`

Email

`mail -s "Subject Line" username < datafile.txt` - datafile.txt has the contents of the email
`command | mail -s "Subject Line" username` - command has contents of email

Backups

purposes:

- to restore individual files
- to restore entire file systems

Components

- **Scheduler** - the object (root, cron, etc.) that decides when a backup should be done - and what should be backed up
- **Transport** - utility/program that puts the backed-up data on the media
- **Media** - where the actual backup is saved
- two fundamental approaches

- filesystem image - where you move the whole disk to another device
    - ex. `dd` and `dump`/`restore`
- file by file - where you copy files to another device
    - ex. `tar` and `cpio`

- types of incremental backups
    - level 0 / full backup - backup of everything
    - level 1 - backup of all changes since last level 0
    - level 2 - backup of all changes since last level 0 or 1
    - level 3 - backup of all changes since last level 0, 1, or 2
    - partial/unscheduled backup - only selected files

tar

archive:

```
tar -cf /tmp/home-backup.tar -C $HOME
```

- creates an archive file called `home-backup.tar` in `/tmp/`
- copied the contents of `$HOME`
- add `-z` to the command to compress it (will change name to `/tmp/home-backup.tar.gz`)
  restore:
  `tar -Xf /tmp/home-backup.tar -C $HOME` - **CAPITAL X**
- this will extract the entire archive to `$HOME`
  `tar -xf /tmp/home-backup.tar -C $HOME ./payroll.txt` - **lowercase x**
- this will extract just `payroll.txt` to `$HOME`

cpio

- reads the names of files it needs to process then puts it into a destination of your choice
- good idea to use a find command to pipe into cpio
  `find /home/ | cpio -o > /mnt/backup/home-backup.cpio`
- this will archive `/home/` into `home-backup.cpio` in the `mnt/backup/` dir

```
find /home/ -atime +365 | cpio -o > /mnt/backup/home-backup.cpio
```

- this will backup files that haven't been accessed within the past year

dump / restore

- only useful for unmounted file systems

dd

- disk duplication - used for making bootable images and duplications of filesystems
  `dd if=/dev/fd0 of=$HOME/image`
- this command will copy `/dev/fd0` (a bootable disk) to the `$HOME/image`
- the same works in reverse