Homework 3 due tomorrow
Any questions?
Last time we wrote a simple script to backup a directory:

```
backupwithdate.sh

#!/bin/bash

tar -czf ~/b kp/cs2042_$(date +%d_%m_%y).tar.gz ~/cs2042/
```

What if we wanted to run this script automatically, say nightly?
**cron**

**cron** is a program that enables Unix users to execute commands or scripts automatically at a specified date/time

- **cron** is a daemon, which means it only needs to be started once and will lay dormant until it is required
- On most Linux distributions is automatically installed and entered into the start up scripts so you don’t have to start it manually:
  - Check by tying `ps -e | grep cron`
  - Depending on your system, it may show up as `cron` or `crond`
- We can control the **cron** daemon in a few different ways...
If you have a look in your /etc directory you will find sub directories called

- cron.hourly
- cron.daily
- cron.weekly
- cron.monthly

If you place a script in any of these directories, it will be run either hourly, daily, weekly or monthly depending on the name of the directory.

Note: If we did this with our backup script, we would need to replace ∼ with /home/hussam since the script would be run as root.
If you want more flexibility in scheduling you can edit a `crontab` file.

**crontab**

- The main config file is normally `/etc/crontab`
- You can create your own crontab files without root access!

Type `cat /etc/crontab` to have a look at the file:
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don’t have to run the ‘crontab’
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user command
17 * * * * root cd / && run-parts --report /etc/cron.hourly
25 6 * * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6 * * 7 root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6 1 * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
crontab

Syntax:

a. min (0-59)
b. hour (0-23)
c. day of month (1-31)
d. month (1-12)
e. day of week (0-6) (Sunday = 0)

Values can be * (all legal values), a range separated by a hyphen, a single value, a set of values separated by commas or a step value (i.e. */2 could be every two hours).
multiuser crontab

- To edit your crontab file type `crontab -e`
- To view your crontab file type `crontab -l`
- To delete your crontab file type `crontab -r`

A sample line:

```
30 18 * * * ./home/hussam/backup.sh
```

This runs the backup script everyday at 6:30PM.
Now back to scripting!
When we pass arguments to a bash script, we can access them in a very simple way:

- `$1, $2, \ldots, $10, $11`: are the values of the first, second etc arguments
- `$0`: The name of the script
- `#$`: The number of arguments
- `*$`: All the arguments, `"$*"` expands to `"$1 $2 \ldots $n"`
- `@$`: All the arguments, `"@$"` expands to `"$1" "$2" \ldots "$n"
- You almost always want to use `@$`
- `$?`: Exit code of the last program executed
- `$$`: current process id.
Simple Examples

**multi.sh**

```bash
#!/bin/bash

echo $(( $1 * $2 ))
```

- Usage: `./multi.sh 5 10`
- Returns first argument multiplied by second argument
- To do arithmetic in bash use `$(( math ))`

**uptolow.sh**

```bash
#!/bin/bash

tr '[A-Z]' '[a-z]' < $1 > $2
```

- Usage: `./uptolow.sh file1 file1low`
- transates all upper case letters to lowercase and writes to file1low
If statements are structured just as you would expect:

```bash
if cmd1
    then
        cmd2
        cmd3
    elif cmd4
    then
        cmd5
    else
        cmd6
fi
```

- Each conditional statement evaluates as true if the `cmd` executes successfully (returns an exit code of 0)
A simple script

```
#!/bin/bash
# This script searches a file for some text then
# tells the user if it is found or not.
# If it is not found, the text is appended
if grep "$1" $2 > /dev/null
then
    echo "$1 found in file $2"
else
    echo "$1 not found in file $2, appending."
    echo $1 >> $2
fi
```
We would not get very far if all we could do was test with exit codes. Fortunately bash has a special set of commands of the form `[ testexp ]` that perform the test `testexp`. First to compare two numbers:

- `n1 -eq n2`: tests if \( n_1 = n_2 \)
- `n1 -ne n2`: tests if \( n_1 \neq n_2 \)
- `n1 -lt n2`: tests if \( n_1 < n_2 \)
- `n1 -le n2`: tests if \( n_1 \leq n_2 \)
- `n1 -gt n2`: tests if \( n_1 > n_2 \)
- `n1 -ge n2`: tests if \( n_1 \geq n_2 \)

If either \( n_1 \) or \( n_2 \) is not a number, the test fails.
We can use test expressions in two ways:

- `test EXPRESSION`
- `[] EXPRESSION []`

Either of these commands returns an exit status of 0 if the condition is true, or 1 if it is false.

Use `man test` to learn more about testing expressions.

Note: Remember you can check the exit status of the last program using the `$?` variable.
#!/bin/bash
# Created on [2/20/2009] by David Slater
# Purpose of Script: Searches a file for two strings and prints which
# is more frequent
# Usage: ./ifeq.sh <file> string1 string2

arg='grep $2 $1 | wc -l'
arg2='grep $3 $1 | wc -l'
if [ $arg -lt $arg2 ]
then
    echo "$3 is more frequent"
elif [ $arg -eq $arg2 ]
then
    echo "Equally frequent"
else
    echo "$2 is more frequent"
fi
string comparison

To perform tests on strings use

- `s1 == s2`: `s1` and `s2` are identical
- `s1 != s2`: `s1` and `s2` are different
- `s1`: `s1` is not the null string

Make sure you leave spaces! `s1==s2` will fail!
When using testexp variable substitution is performed, but no matching is perform.

If \( x \) is the null string, what will \([ \$x \neq \text{monster} ]\) return?
When using testexp variable substitution is performed, but no matching is perform.

If $x$ is the null string, what will \[ $x \neq \text{monster} \] return?

It will return an error, because $x$ is expanded to the null string and the test becomes \[ != \text{monster} \] . To make sure there are no errors, place your variables inside double quotes. Then \[ $x \neq \text{monster} \] is expanded to \[ "\" \neq \text{monster} \] which returns true.
If *path* is a string indicating a path, we can test if it is a valid path, the type of file it represents and the type of permissions associated with it:

- `-e path`: tests if *path* exists
- `-f path`: tests if *path* is a file
- `-d path`: tests if *path* is a directory
- `-r path`: tests if you have permission to read the file
- `-w path`: tests if you have write permission
- `-x path`: tests if you have execute permission
You can combine tests:

```bash
if [ testexp1 -a testexp2 ]
then
    cmd
fi
```

- `-a`: and
- `-o`: or
- `! testexp1`: not
A note about debugging

To debug your code, invoke the script with the \(-x\) option. You will then see all the commands successfully executed:

$ bash -x ifeq.sh Frankenstein.txt monster the
++ grep monster Frankenstein.txt
++ wc -l
+ arg=33
++ grep the Frankenstein.txt
++ wc -l
+ arg2=3850
+’[’ 33 -lt 3850 ’]’
+ echo ’the is more frequent’
We can now begin to ensure our scripts get the input we want:

```bash
if [ -f $1 ]
then
    Perform the action you want
else
    echo "This script needs a file as its input dummy!"
fi
```
Example:

In homework 1, you had to download and run a script, which we should now understand.
Sometimes to make our code cleaner we would like to pipe between lines. To do this we just need to escape the invisible newline character

```
cat myfile | grep 'someregularexpression' | tr ' ' '\n' |\sort | head
```
Putting it on one line

Sometimes we might want to type a multiline command into the shell, we can do this by hitting enter for each line, or by using semicolons to tell the shell to start new lines:

**Example:**

```bash
if [ testexpr ] ; then command1 ; command2 ; fi
```

**Real Example:**

```bash
if [ $? -eq 0 ] ; then echo "Last Command Successful!" ; fi
```
Remember that **gawk** and **sed** are complete scripting languages so we can write **gawk** and **sed** scripts:

**Example: iouscript.gwk**

```gawk
#!/bin/gawk -f

BEGIN {FS = " " }
NR > 1 { Names[$1]+=$2 }
END {for(i in Names) print i " owes me " Names[i] " Dollars."}
```

- Note: You must tell **gawk** to read from a file by using the `-f` flag.
sed scripts work similarly

```bash
trim.sed

#! /bin/sed -f

s/^$//
s/^#[^!]+//
s/[^!]+$//
```

Hussam Abu-Libdeh based on slides by David Slater