A: SE 2XA3 (2018/19, Term I) Major Lab 2 -- lab section L01

Back To Lab Menu | Back To Main Menu | Submissions | Log Out

sample solutions: `script1` and `script2`

If you are in the correct lab room and during the SCHEDULED time slot for your lab section for Major Lab 2 and working on one of the lab's workstations, go into Marks+Comments to see if your attendance was registered. If you signed on the course website too early (the most common problem), your attendance would not be recorded; in such case, please log out and sign on again. If your attendance is still not recorded, please, ask your TA to record your attendance and email Prof. Franek right away. If you are not using one of the lab's workstations, please, have the TA record your attendance.

The Major Labs are open book labs, you can bring and use any notes etc. You can access any website, Google, Wikipedia etc. The only thing that is not allowed is cooperation with other people, inside or outside the lab. You must submit your own work. The TA can only help with administrative and technical aspects and not with the solutions of the problems of the Major Lab.

In this lab, there are two tasks and two deliverables: text files `script1` and `script2`. Each can be submitted either via the course website, or using `2xa3submit`. For `2xa3submit` submission please use `2xa3submit AAA proj2 BBB` where `AAA` is your student number and `BBB` the name of the file you want to submit. You can submit every file as many times as you wish, the latest submission will be used for marking. The submission is opened from 8:30-11:20 or 14:30-17:20 depending on the lab section; after the closing time no submission is be possible. If submission is not possible for whatever reason (typically right after the submission closes), email the file or files as an attachment immediately (needed for the time verification) to Prof. Franek (franek@mcmaster.ca) with an explanation of the problem; you must use your official McMaster email account for that. Include the course code, your lab section, full name, and your student number. Note that if there is no problem with submission (typically the student using a wrong name for the file), you might be assessed a penalty for email submission, depending on the reason you used email.

Task 1. bash script named `script1`

The name of your bash script must be `script1` and below is a description of what it should do when executed. *Note that the name of the script, the names of the files and directories, and the messages must be used exactly as described here, including the lower and upper cases.*

1. `script1` creates in the current directory a subdirectory `PROJ2` and then displays `PROJ2 directory created`
2. Then in the directory `PROJ2` creates a subdirectory `proj2` and then displays `PROJ2/proj2 directory created`
3. In `proj2` it creates 10 ASCII text files named `proj2_1`, ..., `proj2_10` . The content of `proj2_1` consists of two lines: the first saying `File proj2_1` and the second saying `Followed by proj2_2` . The content of `proj2_2` consists of two lines `File proj2_2` and `Followed by proj2_3` etc. Note that the last file `proj2_10` contains two lines, the first saying `File proj2_10` and the second saying `Not followed` . (*the first 9 files ought to be created in a single loop*)
4. Then the script displays `All regular files of PROJ2/proj2` and then displays just the names of all regular files in `PROJ2/proj2` (*note that non-regular files are not to be displayed, and since the* `ls` *command displays all entities in the directory, you cannot use* `ls` *for this purpose*).
5. Then the script displays `Contents` and then it displays contents of all regular files in `PROJ2/proj2` separated by a line consisting of dashes (character `-`).
6. Then it moves all files among `proj2_1`, ..., `proj2_10` containing a character `4` or a character or `5` or a character `8` from `PROJ2/proj2` to `PROJ2` .

7. Then the script displays `All regular files of PROJ2` and then displays just the names of all regular files in `PROJ2` .
8. Then it creates a single file in `PROJ2` named `BIGFILE` that contains all the lines from all the regular files in `PROJ2` .
9. Then it removes all other regular files from `PROJ2` except `BIGFILE` .
10. Then the script displays `All regular files of PROJ2` and then displays just the names of all regular files in `PROJ2` .
11. Then the script displays `BIGFILE:` and then displays the contents of `BIGFILE` .
12. Then the script displays `All regular files of PROJ2/proj2` and then displays just the names of all regular files in `PROJ2/proj2` .
13. Then it creates a single file in `PROJ2/proj2` named `bigfile` that contains all the lines from all the regular files in `PROJ2/proj2` .
14. Then it removes all other regular files from `PROJ2/proj2` except `bigfile` .
15. Then the script displays `All regular files of PROJ2/proj2` and then displays just the names of all regular files in `PROJ2/proj2` .
16. Then the script displays `bigfile:` and then displays the contents of `bigfile` .
17. Then it removes the directory `PROJ2/proj2` with the file `bigfile` still in it.
18. Then it removes the directory `PROJ2` with the file `BIGFILE` still in it.
19. Now the current directory should contain exactly the same files and subdirectories as just before this script was executed.

What commands and concepts you might need: `cd mkdir echo ls grep mv cat` and ***for loop***. *Note, that after the script has been executed, the current directory is in the same state as it was when the script started its execution. Note that the message must be used exactly as in the above description, including the lower and upper cases.*

A few useful hints:

1. *Current directory is referred to as* `.` *, the parent directory as* `..`
   For instance, `ls .` *will show all files/subdirectories in the current directory, while* `ls ..` *will show all files/subdirectories in the parent directory*
2. *The number of command line arguments is stored in* `$#` *variable.*
3. *Since* `$0` *is a pathname, we can extract the name of the file using* `basename`, *e.g.* `basename $0`
4. *a range from 1 to 15 can be expressed as* `{1..15}`, *for instance*
   `for i in {1..15}`
5. *to concatenate a string with a number* (`x` *contains a string,* `i` *contains a number*), *use* `$x$i`
6. *to increment a variable* `i` *containing a number, use* `i = $(($i+1))`
7. *to test if a name stored in a variable* `file` *is a name of a regular file, use* `[ -f $file ]`
8. *Another way to deal with regular files is to use* `find . -type f`
9. *to figure out if a file whose name is stored in a variable* `file` *contains a symbol* `4`,`5`, *or* `8`, *use*
   ``x = `grep '4\|5\|8' $file` `` *and then test whether* `x` *is empty* (*note the single quotes inside and the back quotes around the whole expression*).
10. *In bash you quite often need to check to see if a variable has been set or has a value other than an empty string. This can be done using the* `-n` *or* `-z` *string comparison operators.*
    The `-n` *operator checks whether the string is not null. Effectively, this will return true for every case except where the string contains no characters. ie*:

    ```
    if [ -n "$VAR" ]
    then
    echo "VAR is not empty"
    fi
    ```

    Similarly, the `-z` *operator checks whether the string is null. ie*:

    ```
    if [ -z "$VAR" ]
    then
    "VAR is empty"
    fi
    ```

    Note the spaces around the square brackets. Bash will complain if the spaces are not there.

A sample run:

```
PROJ2 directory created
PROJ2/proj2 directory created
All regular files of PROJ2/proj2
proj2_1
proj2_10
proj2_2
proj2_3
proj2_4
proj2_5
proj2_6
proj2_7
proj2_8
proj2_9
Contents
proj2_1:
File proj2_1
Followed by proj2_2
-------------------------------------------
proj2_10:
File proj2_10
Not followed
-------------------------------------------
proj2_2:
File proj2_2
Followed by proj2_3
-------------------------------------------
proj2_3:
File proj2_3
Followed by proj2_4
-------------------------------------------
proj2_4:
File proj2_4
Followed by proj2_5
-------------------------------------------
proj2_5:
File proj2_5
Followed by proj2_6
-------------------------------------------
proj2_6:
File proj2_6
Followed by proj2_7
-------------------------------------------
proj2_7:
File proj2_7
Followed by proj2_8
-------------------------------------------
proj2_8:
File proj2_8
Followed by proj2_9
-------------------------------------------
proj2_9:
File proj2_9
Followed by proj2_10
-------------------------------------------
All regular files of PROJ2
proj2_3
proj2_4
proj2_5
proj2_7
proj2_8
All regular files of PROJ2
BIGFILE
BIGFILE:
File proj2_3
Followed by proj2_4
File proj2_4
Followed by proj2_5
File proj2_5
Followed by proj2_6
File proj2_7
Followed by proj2_8
File proj2_8
Followed by proj2_9
All regular files of PROJ2/proj2
proj2_1
proj2_10
proj2_2
proj2_6
proj2_9
All regular files of PROJ2/proj2
bigfile
bigfile:
File proj2_1
Followed by proj2_2
File proj2_10
```

```
Not followed
File proj2_2
Followed by proj2_3
File proj2_6
Followed by proj2_7
File proj2_9
Followed by proj2_10
```

Task 2. bash script named `script2`

The name of your bash script must be **script2** and below is a description of what it should do when executed.

1. First the script **script2** checks the command line arguments. It should have 1 or 2 or 3 command line arguments (we are not counting the name of the script, thus `script2 a` is considered to have one command line argument).
   If a single command line argument is used, it must be `-X`
   If two command line arguments are used, it must be `-X <file1>`
   If three command line arguments are used, it must be `-X <file1> <file2>`
2. If the number of arguments is wrong, the script displays what we call ***usage***

   ```
   wrong number of command line arguments
   Usage: script2 -X
       or
          script2 -X <file1>
       or
          script2 -X <file1> <file2>
   ```

   and terminates.
3. If the number of command line arguments is correct, but the first argument is not `-X`, the script displays an error message, then ***usage*** and terminates.
4. If 1 correct command line argument is used, the script displays `Well done` and then the script terminates.
5. If 2 command line arguments are used, the script displays `Well done, file1 is A` and creates in the current directory a file named A . The created file contains 10 lines
   `1,A`
   `2,A`
   ...
   `10,A`
   where A is the second command line argument (referred to as `<file1>` in the description above). The content of the file must be created by a loop
6. The created file is displayed and then the script terminates. Note that the created file stays in the directory after the script finishes its execution.
7. If 3 command line arguments are used, the script displays `Well done, file1 is A and file2 is B` and creates in the current directory a file named A ,where A is the second command line argument (referred to as `<file1>` in the description above) and a file B , where B is the third command line argument (referred to as `<file2>` in the description above).
8. The first file contains the 10 lines as in 5., the second file contains 10 lines
   `1,B`
   `2,B`
   ...
   `10,B`
9. The created files are displayed and the script terminates. Note that the created files stay in the directory after the script finishes its execution.

A few useful hints:

- *The number of command line arguments is stored in* `$#` *variable.*
- *Since* `$0` *is a pathname, we can extract the name of the file using* `basename`*, e.g.* `basename $0`
- *If you create by mistake a file called* `-d`*, you must remove it using command* `rm -- -X`*, as* `rm -X` *will not work as* `rm` *will think* `-X` *is a switch.*
- *To terminate execution of a script, you can use the* `exit` *command.*
- *It may be useful to learn how to do general counting loop in bash -- an example:* `COUNTER` *will go through values* 0, 2, 4, ... `$i`

```
for (( COUNTER = 0;COUNTER <= $i;COUNTER += 2 ))
        do
            echo $COUNTER
        done
```

For more on the meaning of `((   ))`*, i.e. arithmetic integer evaluation, please see bash manual in the help section,* *section 3.5.5*

Sample runs: executing `script2`

```
wrong number of command line arguments
Usage:
script2 -X
  or
script2 -X <file1>
  or
script2 -X <file1> <file2>
```

Sample runs: executing `script2 hello`

```
incorrect command line argument: hello
Usage:
script2 -X
  or
script2 -X <file1>
  or
script2 -X <file1> <file2>
```

Sample runs: executing `script2 -X`

```
Well done
```

Sample runs: executing `script2 -X McMaster\ rules`

```
Well done, file1 is McMaster rules
```

and a file named `McMaster rules` is in the current directory and it contains

```
1,McMaster rules
2,McMaster rules
3,McMaster rules
4,McMaster rules
5,McMaster rules
6,McMaster rules
7,McMaster rules
8,McMaster rules
9,McMaster rules
10,McMaster rules
```

Sample runs: executing `script2 -X McMaster\ rules McMaster\ sucks`

```
Well done, file1 is McMaster rules and file2 is McMaster sucks
```

and two files named `McMaster rules` and `McMaster sucks` are in the current directory. The first file contains

```
1,McMaster rules
2,McMaster rules
3,McMaster rules
4,McMaster rules
5,McMaster rules
6,McMaster rules
7,McMaster rules
8,McMaster rules
9,McMaster rules
10,McMaster rules
```

and the second file contains

```
1,McMaster sucks
2,McMaster sucks
3,McMaster sucks
4,McMaster sucks
5,McMaster sucks
6,McMaster sucks
7,McMaster sucks
```

```
8,McMaster sucks
9,McMaster sucks
10,McMaster sucks
```

Sample runs: executing `script2 -X Hello`

```
Well done, file1 is Hello
```

and a file named `Hello` is in the current directory and it contains

```
1,Hello
2,Hello
3,Hello
4,Hello
5,Hello
6,Hello
7,Hello
8,Hello
9,Hello
10,Hello
```