TCSS 422: Operating Systems　　Institute of Technology
Winter 2018　　University of Washington – Tacoma
http://faculty.washington.edu/wlloyd/courses/tcss422
Instructor: Wes Lloyd

Assignment 1

M*A*S*H [1]

Mash Shell

Due Date: Friday February 9th, 2018 @ 11:59 pm, tentative
Version: 0.10

Objective

The purpose of this assignment is to use the fork, wait, and exec commands to write a simple Linux shell. This shell is called "mash", and the goal of mash is to **mash** three Linux command requests together and run them against the same input file. The user will provide three distinct Linux commands with arguments, and a single file name. The mash shell will **mash** the requests together executing each command separately against the backend file.

For this program, implement your mash shell using fork, exec, and wait commands.

The following limitations and/or requirements define how mash should operate:

1. User commands plus arguments will not exceed 255 characters

2. The filename will not exceed 255 characters. The file will either be in the local directory, or the user will provide a fully qualified path name which is 255 characters or less. The mash shell is not responsible for finding the input file.

3. Commands run in "mash" will assume the user's original path:
   Type "echo $PATH" to see the current path variable setting.

4. For each command, the maximum number of arguments including the command itself will not exceed 5. So this implies 4 arguments, plus the command.

5. If the user makes a mistake typing a command and/or its arguments, mash should simply fail to run the command. A simple error should be shown, but only if the exec fails.

6. Mash does not accept any command line arguments. Running mash simply starts the shell which requests 3 commands and a file name.

1 Image labeled for non-commercial reuse

7. In an effort to execute the mash of commands as fast as possible, mash should not wait for each command to complete before executing the next one. Consequently the order of execution of commands can vary. (e.g. it’s non-deterministic…) The only expectation is that every command should run, and output should be shown. The output will be ordered based on which commands finish first.

To test mash, a number of commands may be used. Here are some possible commands to test your mash shell:

“wc”	Reports the line count, word count, and character count
“md5sum”	Generates a unique 128-bit md5 (checksum) hash message digest
“grep –c the”	Counts the number of occurrences of a given word, here “the”
“grep –ci the”	Counts the number of occurrences of a given word ignoring case, here “the”
“tail –n 10”	outputs 10 lines from the end of a file
“head –n 10 ”	outputs 10 lines from the start of a file
“ls –l”	provides a long directory listing

By forking to run these commands at the same time (in parallel) on multi-core machines the tasks can collectively finish in less time achieving a performance speedup versus performing the tasks separately. Using fork to run multiple processes in parallel helps to exercise multiple available CPU cores for unrelated tasks (*embarrassingly parallel*). Using “top” it is possible to watch mash run multiple processes at the same time when working on large files.

Input

There are no command line arguments for mash. The mash shell should be invoked as follows:

$./mash

Output

Here is a sample output sequence for running MASH.

```
$ ./mash
mash-1>grep -c the
mash-2>wc -l
mash-3>wc
file>cc.log
-----LAUNCH CMD 1: grep -c the---------------------------------------------------
-----LAUNCH CMD 2: wc -l--------------------------------------------------------
-----LAUNCH CMD 3: wc------------------------------------------------------------
1193713 cc.log
Result took:35ms
--------------------------------------------------------------------------------
4
Result took:133ms
--------------------------------------------------------------------------------
  1193713  13318354 104857721 cc.log
Result took:937ms
--------------------------------------------------------------------------------
Done waiting on children: 17664 17663 17665.
```

When each process is forked an 80 character line should be printed for each command. These lines should indicate the order in which processes are forked from the parent:

```
-----LAUNCH CMD 1: grep -c the------------------------------------------------
-----LAUNCH CMD 2: wc -l------------------------------------------------------
-----LAUNCH CMD 3: wc---------------------------------------------------------
```

Then using the wait() call, MASH should print 80 character delimiter lines between each return result. Note, commands will return in random order based on how long they require to execute:

```
1193713 cc.log
--------------------------------------------------------------------------------
4
--------------------------------------------------------------------------------
  1193713  13318354 104857721 cc.log
--------------------------------------------------------------------------------
```

MASH concludes by echoing back the PID for all of the children processes:

```
Done waiting on children: 17664 17663 17665.
```

As extra credit, the duration of each command can be printed:

```
1193713 cc.log
Result took:35ms
--------------------------------------------------------------------------------
4
Result took:133ms
--------------------------------------------------------------------------------
  1193713  13318354 104857721 cc.log
Result took:937ms
--------------------------------------------------------------------------------
```

Here it is possible to see how long the individual output lines took to be generated. What we don't know, is what command they are associated with.
It is not required to indicate the commands the output lines are associated with. However, we can see that the first command returned is "wc -l", and then "grep -c the", followed by the full "wc" command.

Full wc prints lines words and characters of the entire file and takes the longest, in the case nearly 1 second against the ~ 100MB cc.log file.

Test your program with a variety of commands on large text file(s) to confirm parallel execution.

If the fork command fails, then print out the status code as below:

```
$ ./mash
mash-1>cantfindit
mash-2>missingcommand
mash-3>whereisit
file>cc.log
-----LAUNCH CMD 1: cantfindit-------------------------------------------------
```

```
-----LAUNCH CMD 2: missingcommand--------------------------------------------
-----LAUNCH CMD 3: whereisit-------------------------------------------------
CMD1:[SHELL 1] STATUS CODE=-1
CMD2:[SHELL 2] STATUS CODE=-1
CMD3:[SHELL 3] STATUS CODE=-1
Result took:0ms
----------------------------------------------------------------------------
Result took:0ms
----------------------------------------------------------------------------
Result took:0ms
----------------------------------------------------------------------------
Done waiting on children: 19245 19246 19247.
```

When mash can’t run an external command, a message indicating failure should be displayed:

```
[SHELL 1] STATUS CODE=-1
[SHELL 2] STATUS CODE=-1
```

The message identifies which mash command failed (1, 2, or 3), with a status code.

To implement this assignment successfully, you will need to:

1. Write code that captures a user provided strings from the console to collect 3 individual commands and a filename.

2. Chop individual words from the user provided commands to extract the command arguments so they can be provided to exec(). For example, a user may provide “grep –ci the”. This string will be chopped into three strings: “grep”, “-ci”, and “the”. These strings can be hard coded in an execlp call as follows:

   ```
   execlp(“grep”,”-ci”,”the”,(char *) NULL);
   ```

   However, this approach is not dynamic.
   It should be possible to support a dynamic number of arguments.
   A recommended alternative to execlp() is execvp() which accepts a pointer to a NULL terminated array of char pointers (char **). Each char pointer points to a null terminated word.

3. Implement fork() and wait() successfully with 3 levels of nesting. Without nesting, only one fork() would execute at any given time causing all three commands to run sequentially. This would result in a slower “mash”.

   ```
   p1 = fork();
   if (p1 == 0) // child
   if (p1 > 0) // parent
      p2 = fork();
      if (p2 == 0) // child
      If (p2 > 0)
         p3 = fork();
         if (p3 == 0) // child
         if (p3 > 0)
            wait(..)
   ```

4. Wait for children to finish to allow the parent to gracefully exit.
5. Print out command header lines (80 characters) and the 80 character delimiter lines.

It is recommended to tackle key design challenges individually (one at a time) to simplify the testing/debugging of the implementation.

Grading Rubric

This assignment will be scored out of 100* points. (100/100)=100%
110 points are possible.

Toal:	90 points
5 points	Run 1 command with at least 1 argument against the file - (arg no mash)
10 points	Run 1 command with up to 5 arguments against the file - (arg chop no mash)
5 points	Run 3 command with no arguments against the file – (mash 3)
10 points	Run 3 commands with up to 5 arguments against the file - (arg chop mash 3)
10 points	Run 3 commands with nested forks in parallel
10 points	End gracefully. Parent process prints last line reporting PIDs of finished children. The program returns cleanly to the calling shell.
5 points	STATUS CODE message shown for a failed command.
5 points	If one command fails, others can work.
10 points	Display 80-character command header lines in order of when each process is started
10 points	Display 80-character delimiter lines separating output from each command
10 points	Display wall clock time in milliseconds for processing of each output result

Miscellaneous:	20 points
5 points	Program compiles, and does not crash upon testing
5 points	Coding style, formatting, and comments
5 points	Makefile with valid “all” and “clean” targets
5 points	Output format matches the provided example (even if a portion doesn’t work!)

WARNING!	
10 points	Automatic deduction if program is not named “mash”

What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas.

Package up all of the files into the single tar gzip archive.
This should include a makefile with “all” and “clean” targets.

Tar archive files can be created by going back one directory from the project source directory with “`cd ..`”, then issue the command “`tar czf <lastname_firstname>_A1.tar.gz my_dir`”. Name the tar gzip file with your last name underscore firstname underscore A1 for assignment 1. “my_dir” would be the directory that contains the source code and makefile. **No other files should be submitted.**

Pair Programming (optional)

Optionally, this programming assignment can be completed with two person teams.

If choosing to work in pairs, ***only one*** person should submit the team’s tar gzip archive to Canvas.

Additionally, ***EACH*** member of a pair programming team must provide an **effort report** of team members to quantify team contributions for the overall project. **Effort reports** must be submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member to capture each person's overall view of the teamwork and outcome of the programming assignment. <u>Effort reports are not used to directly numerically weight assignment grades.</u>

Effort reports should be submitted in confidence to Canvas as a PDF file named: "effort_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format.

Distribute 100 points for category to reflect each teammate's contribution for: research, design, coding, testing. Effort scores should add up to 100 for each category. Even effort 50%-50% is reported as 50 and 50.

<u>Please do not submit 50-50 scores for all categories.</u>

It is highly unlikely that effort is truly equal for everything. Ratings must reflect an honest confidential assessment of team member contributions. ***50-50 ratings and non-confidential scorings run the risk of an honor code violation.***

Here is an <u>**effort report**</u> for a pair programming team (written from the point of view of Jane Smith):

1. John Doe

Research	24
Design	33
Coding	71
Testing	29

2. Jane Smith

Research	76
Design	67
Coding	29
Testing	71

Team members may not share their **effort reports**, and should submit them independently in Canvas as a PDF file. Failure of one or both members to submit the **effort report** will result in both members receiving NO GRADE on the assignment… (*considered late until both are submitted*)

<u>Disclaimer regarding pair programming:</u>
The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer simply watching the champion! The tasks and challenges should be shared as equally as possible.