TCSS 422: Operating Systems   School of Engineering and Technology
Fall 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 October 26th, 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 filename to operate on. The mash shell will **mash** the requests together executing each command separately against the 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.

    *** BONUS – Support unlimited arguments ***

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.

---

1 Image labeled for non-commercial reuse

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

7. To execute the "mash" of commands as fast as possible, mash should execute commands in parallel using separate processes. Consequently the order of completion of commands can vary. (e.g. it's non-deterministic…) Since commands will complete in an unknown order, (we don't know which will finish first), you should capture the output from each command, and display output for each command in the same order the user provided the commands. To support this, use file redirection as shown in the example in class. Capture the output of each command's stdout file stream to a separate temporary file on disk. If storing to disk, it is easy to reread the output file, to then display the command output in the correct order.

   *** BONUS - Store the file stream's temporary output in memory ***

   File redirection example:
   http://faculty.washington.edu/wlloyd/courses/tcss422/examples/exec2.c

   Once printed these temporary files should be deleted.

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 should 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. Using "top -d .1" 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>md5sum
```

```
mash-3>wc -l
file>googlebig.txt
First process finished...
Second process finished...
Third process finished...
-----CMD 1: grep -c the----------------------------------------------------------
15697
Result took:314ms
-----CMD 2: md5sum----------------------------------------------------------------
0362a7bf9035eba363462ea484bb43a6  googlebig.txt
Result took:2664ms
-----CMD 3: wc -l-----------------------------------------------------------------
76860248 googlebig.txt
Result took:803ms
----------------------------------------------------------------------------------
Children process IDs: 12287 12289 12288.
Total elapsed time:2664ms
```

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

```
-----CMD 1: grep -c the----------------------------------------------------------
-----CMD 2: md5sum----------------------------------------------------------------
-----CMD 3: wc -l-----------------------------------------------------------------
```

MASH will print an 80 character delimiter line at the end of the output for all three processes. Commands return **in the same order** the user entered commands, regardless of the order of their completion and how long they require to complete.

When each process returns, a line is printed indicating that one process has returned. We do not specify which, as it could be *ANY* of the three processes:

```
First process finished...
Second process finished...
Third process finished...
```

MASH concludes by echoing back the PID for all of the children processes, and also the total elapsed time of all commands:

```
Children process IDs: 17664 17663 17665.
Total elapsed time:2658ms
```

On a four-core system, the elapsed time is the time of the longest command. Since the other two processes run in parallel, there is a time savings.

The duration of each command is printed after the output:

```
-----CMD 1: grep -c the----------------------------------------------------------
15697
Result took:314ms
-----CMD 2: md5sum----------------------------------------------------------------
0362a7bf9035eba363462ea484bb43a6  googlebig.txt
Result took:2664ms
-----CMD 3: wc -l-----------------------------------------------------------------
76860248 googlebig.txt
Result took:803ms
```

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
First process finished...
Second process finished...
Third process finished...
-----CMD 1: cantfindit---------------------------------------------------------
CMD1:[SHELL 1] STATUS CODE=-1
Result took:1ms
-----CMD 2: missingcommand----------------------------------------------------
CMD2:[SHELL 2] STATUS CODE=-1
Result took:2ms
-----CMD 3: whereisit----------------------------------------------------------
CMD3:[SHELL 3] STATUS CODE=-1
Result took:1ms
-------------------------------------------------------------------------
Children process IDs: 12346 12348 12347.
Total elapsed time:2ms
```

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

```
CMD1:[SHELL 1] STATUS CODE=-1
CMD2:[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, tasks include:
1. Write code that captures 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.

6. Determine how to measure and print out the execution time of individual commands.

7. Capture the output of each command to a file.

8. Print the contents of the file, to have "ordered" output for MASH, instead of random output.

9. Delete the temporary output files of each command. (recommended)

10. Print the elapsed time of all commands at the end of the program.

While writing your MASH program using processes, consider why it is non-trival to simply redirect the output stream of each fork command to a stream and use the C sscanf() function to consolidate/aggregate the output at the end...

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%
115 points are possible.

Toal:	90 points
5 points	Run 1 command with at least 1 argument against the file - (arg no mash)
5 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)
5 points	Run 3 commands with up to 5 arguments against the file - (arg chop mash 3)
5 points	* Support an unlimited number of command line arguments
10 points	Run 3 commands in parallel to reduce overall execution time

10 points	End gracefully. Parent process prints last line reporting PIDs of finished children. The program returns cleanly to the calling shell. (It is not necessary to press <ENTER> to get to the command prompt.
5 points	STATUS CODE message shown for a failed command.
5 points	If one command fails, others can work.
5 points	Display 80-character command header lines in order of when each process is started
5 points	Display 80-character delimiter line separating output at end of program
5 points	Display wall clock time in milliseconds for processing each MASH command
5 points	Display total elapsed time in milliseconds for running all of the MASH commands.
15 points	Display MASH command output in the order the user entered each command.
5 points	* Store stdout of individual commands in memory, as opposed to a temp file.

* - must indicate in comments or documentation that these features have been implemented to receive credit.

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)
O*ptionally*, 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. Effort reports are not used to directly numerically weight assignment grades.

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 **effort report** 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.