Institute of Technology
University of Washington – Tacoma
Instructor: Wes Lloyd

Assignment 1

[1]

Mash Shell

Due Date: Friday February 10th, 2016 @ 11:59 pm, tentative

Objective

The purpose of this assignment is to use the fork, wait, and exec commands to write a very simple Linux shell. This shell is called "mash", and the goal of mash is to **mash** three Linux command requests together to run 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, you are to implement the mash shell using fork, exec, and wait commands.

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

1. User commands 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 fullly qualified path name. 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.

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.

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

[1] Image labeled for non-commercial reuse

commands can vary. (e.g. it's non-deterministic…) The only expectation is that every command should run, and output should be shown.

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 when working on large files.

Input

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

```
$./mash
```

Output

Here are a number of possible sample input output sequences.
Key status output provided by MASH is shown in BOLD.

```
$ ./mash
mash-1>grep -ci the
mash-2>grep -c the
mash-3>wc -l
file>/var/log/syslog
57
1540 /var/log/syslog
46
Done waiting on children: 14618 14620 14619.

$ ./mash
mash-1>grep -ci the
mash-2>grep -c the
mash-3>wc -l
file>/var/log/nofile
grep: /var/log/nofile: No such file or directory
wc: /var/log/nofile: No such file or directory
grep: /var/log/nofile: No such file or directory
Done waiting on children: 14631 14633 14632.

$ ./mash
mash-1>grep -ashfdsahfkjshfasjfkashfdkj the
mash-2>grep -asjfhksahfskjfhdskjfashfksaj the
```

```
mash-3>wc -lsahfakdsjhfaksjhfsakjfdhas
file>/var/log/syslog
grep: dsahfkjshfasjfkashfdkjgrep: invalid option -- 'j'
: No such file or directoryUsage: grep [OPTION]... PATTERN [FILE]...

Try 'grep --help' for more information.
wc: invalid option -- 's'
Try 'wc --help' for more information.
Done waiting on children: 14635 14636 14637.

$ ./mash
mash-1>grep -ci -e the
mash-2>grep -c -e the
mash-3>wc -l -e
file>/var/log/syslog
[SHELL 1] STATUS CODE=-1
[SHELL 2] STATUS CODE=-1
wc: invalid option -- 'e'
Try 'wc --help' for more information.
Done waiting on children: 14641 14642 14643.
```

When mash exits, it echoes back the PIDs used to execute the individual commands. Since mash simply executes other programs, other programs already do a nice job of handling errors. If mash can't run an external command, then a message indicating failure of one of the mash shell attempts 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.
Often commands will fail if too many arguments are provided.
Try "grep –c –e the" as an example. The "-e" is an extra argument which causes the exec() to fail.

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);
   ```

   You will need to parameterize execlp() with variables not hard code its use. 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".

4. Wait for children to finish to allow the parent to gracefully exit.

These represent key design challenges. It is recommended to tackle each challenge individually (one at a time) to simplify the implementation.

Grading

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

* *If necessary the total points scored from may be lowered, while the total available points will remain 100.*

Rubric:

110 possible points: (Currently 10 extra credit points are available)

Toal:	90 points
5 points	Run 1 command with no arguments against the file - (no mash)
5 points	Run 2 commands with no arguments against the file – (mash 2)
10 points	Run 3 command with no arguments against the file – (mash 3)
10 points	Run 1 command with at least 1 argument against the file - (arg no mash)
10 points	Run 1 command with at least up to 5 arguments against the file - (arg chop no mash)
5 points	Run 2 commands with at least up to 5 arguments against the file - (arg chop mash 2)
5 points	Run 3 commands with at least up to 5 arguments against the file - (arg chop mash 3)
10 points	Run 3 commands with variable number of arguments against the file (arg chop mash *) pointers used, no hard coding (EXTRA CREDIT)
10 points	Run 3 commands with nested forks (in parallel)
10 points	End gracefully. Parent process prints last line reporting IDs of finished children. The program returns cleanly to the calling shell.
5 points	An error message is shown for a failed command.
5 points	Even if one command fails, others can work.

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”

***** THE PROGRAM SHOULD BE CALLED “mash.c” *****

FAILURE TO USE THIS NAME WILL RESULT IN A 10 point deduction.

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 kernel module code with "`cd ..`", then issue the command "`tar czf <lastname_firstname>_A1.tar.gz my_dir`". Name the 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.

Provide 1 (low effort) to 10 (high effort) scores for each: research, design, coding, testing
Effort scores should add up to 10 for each category. Even effort 50%-50% is reported as 5 and 5.

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

1. John Doe

Research	2
Design	3
Coding	8
Testing	3

2. Jane Smith

Research	8
Design	7
Coding	2
Testing	7

Team members may not share their **effort reports**, but 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*)

Disclaimer regarding pair programming:
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.