

## Assignment 1



### Mash Shell

Due Date: Thursday May 7<sup>th</sup>, 2020 @ 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, called “mash” will **\*\*MASH\*\*** three Linux command requests together and run them against the same input file in **parallel** (i.e. at the same time). 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 the fork, exec, and wait commands.

The following limitations will apply:

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. All commands provided to “mash” will run using the user’s original path.  
Type “echo \$PATH” to see the current path variable setting.  
*The mash shell is not responsible for finding the command file to run.*
4. For each command, the maximum number of arguments including the command itself will not exceed 5. This equates to 4 command arguments, plus the command, for a total of 5 inputs.

**\*\* Extra Credit-1: SUPPORT UNLIMITED ARGUMENTS \*\***

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

---

<sup>1</sup> Image labeled for non-commercial reuse

6. Mash does not accept command line arguments. Running mash requests 3 commands (plus arguments) and a file name interactively from the user.
7. To execute the “mash” of commands as fast as possible, mash should execute commands in parallel using separate processes. Consequently, the commands may complete out of order.

**\*\* Extra credit-2: DISPLAY OUTPUT SORTED BY PROCESS COMPLETION:** capture the output from each command to a temporary file, and display the output for each command in the same order as requested by the user. 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. Read the temporary files to then regenerate the command output in the correct order. Once results are display to the screen temporary files should be deleted. \*\*

File redirection example:

<http://faculty.washington.edu/wlloyd/courses/tcss422/examples/exec2.c>

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

```

**\*\* Extra Credit 3: DISPLAY 80-CHAR PROCESS DELIMITER LINE** - When each process is forked, an 80-character wide line is printed for each command. These lines should indicate the order in which processes are forked from the parent. Examples are below: \*\*

```

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

```

Mash follows the output of all commands with an 80-character FIXED delimiter line at the end of the output.

```

-----

```

As each process finishes, a line is printed indicating that a 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.

**\*\* Extra Credit 4: DISPLAY RUNTIME OF EACH COMMAND** - The duration of each command can be printed after each command's 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.

## Summary of Tasks

1. Write code to obtain three user provided commands and a filename (all strings).
2. Split individual words from the user provided commands to extract the command arguments so they can be provided to `exec()`. For example, a user may type `"grep -ci the"`. This string will be split into three strings: `"grep"`, `"-ci"`, and `"the"`.

Here is an example of hard coding these strings to call `execvp()`:

```
execvp("grep","-ci","the",(char *) NULL);
```

This approach is not dynamic.

A dynamic number of arguments can be accepted using `execvp()` which accepts a pointer to a NULL terminated array of char pointers (`char **`). Each char pointer points to a null terminated word.

3. Fork the original parent process by nesting 3 levels. Use `wait()` to wait for processes to exit. 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".

```

// NESTING: Each fork is nested within the previous fork
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 (**EC**) (80 characters) and the 80-character delimiter lines.
6. (**EC**) Determine how to measure and print out the execution time of individual commands.
7. (**EC**) Store the output of each command to a temporary file.
8. (**EC**) Read the temporary output files and generate “ordered” output for mash, instead of random output. Need to determine order commands finish.
9. (**EC**) Delete the temporary output files of each command. (recommended)
10. Print the elapsed (total) time of all commands at the end of the program.

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%  
120 points are possible.

**EC** – indicates EXTRA CREDIT

<b>Total:</b>	<b>100 points</b>
10 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)
10 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 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 delimiter line separating output at end of program
5 points	Display total elapsed time in milliseconds for running all of the MASH commands.
5 points	* <b>EC-1:</b> Support an unlimited number of command line arguments
5 points	* <b>EC-2:</b> Display MASH command output in the order the user entered each command.
5 points	* <b>EC-3:</b> Display 80-char command header lines in order of when each process is started
5 points	* <b>EC-4:</b> Display wall clock time in milliseconds for processing each MASH command
<b>Miscellaneous:</b>	<b>20 points</b>
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!)
<b>WARNING!</b>	
10 points	Automatic deduction if main program file is not named “mash.c”

**\* - EXTRA CREDIT- COMMENTS ARE REQUIRED:**

*Comments must be included at the top of the mash.c file to indicate which extra credit features (EC1, EC2, EC3, and EC4) have been implemented to receive credit. If there is no indication that extra credit features are implemented, no extra credit will be awarded.*

*Example of **required** comment:*

```
// EXTRA CREDIT FEATURES: EC2, EC3 implemented
```

**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. A makefile with “all” and “clean” targets should be included.

Tar archive files can be created by going back one directory from the project source directory with “cd . .”, then issue the command “tar czf A1.tar.gz my\_dir”. “my\_dir” is the directory of where your program is stored. Canvas automatically appends your name to the file upon upload. Here “my\_dir” is the directory that contains source code and the makefile.  
**No other files should be submitted.**

**Pair Programming (optional)**

*Optionally*, it is encouraged to complete this programming assignment 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.

**Please do not submit 50-50 scores for all categories.**

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

**NEW SPRING 2020:** TCCS 422 effort reports should include a **short description** of how pair programming was conducted virtually. The description should LIST tools that were used and how they facilitated pair programming. Some recommended tools include: Google Hangouts, Zoom, and Slack.

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)*

Disclaimer regarding pair programming:

The purpose of TCCS 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 that only passively participates! Tasks and challenges should be shared as equally as possible to maximize learning opportunities.