# COMP3511 Project 1 Spring 2019

## Introduction

This project covers entry-level Linux system programming. Students are required to finish a multi-process application using Linux system calls, such as `fork()` and `pipe()`. The aim of this project is to help students understand the basic idea of process and inter-process communication methods and gain first-hand experience in related programming works.

## Project Requirement

### Product

You are required to finish a multi-process batch calculator. The calculator reads a text file containing arithmetic expressions and calculates the expressions with floating point accuracy using arbitrary number of child processes. The program should be executed with the following command: `multical.out input.txt output.txt 5`, where `input.txt` is the input file name and `output.txt` is the output file name, and `5` means 5 child process will be created to calculate the expressions in `infile.txt`. The program can only read the input file after the `5` child processes are forked.

### Input Format

Your program should read a text file. Each line is an expression the program needs to calculate. An empty line denotes the end of processing. The text file is formated as following:

```
[expression]\n

[expression]     := [expression][expression]
[expression]     := [id] [number] [operation] [number]\n
[id]             := #[integer]
[number]         := [integer]
[number]         := [floating point number]
[operation]      := +
[operation]      := -
[operation]      := *
[operation]      := /
```

### Sample Input

Note: `'\n'` is converted to line break in this sample.

```
#0 1 + 1
#1 2 * 5
#2 385 / 134
#3 153.23 - 54324
```

```
#4 5.64 - 0.873
```

## Output Format

Your program should write the output to a new file. The output starts with a line stating the pid of parent process. Follows the result of the calculations, line-by-line, with the corresponding `[id]` and the pid of the processing child, `[by]`. Every answer should be formatted to 3 decimal points, regardless the answer has decimals or not. Ordering of `[id]` is not important. An empty line denotes the end of answers.

```
[parent pid]\n[result]\n

[result] := [result][result]
[result] := [id] [by] [floating point number]\n
[id]     := #[integer]
[by]     := [integer]
```

## Sample Output

Note: `'\n'` is converted to line break in this sample. Suppose `1020` is the pid of parent, `1021 ~ 1024` are pids of children.

```
1020
#1 1021 10.000
#3 1022 -54170.770
#0 1024 2.000
#2 1023 2.873
#4 1025 4.767
```

## Even distribution of work

Every child should get the same amount of work. Each child should get at most 1 more calculation to work on comparing to other children. It is recommended to use round robin task distributing. Failing to distribute the calculations evenly will result in a 10 point deduction in the completion points.

## Tools

You are only allowed to use C language, and specific functions for specific tasks:

1. `fork()` for spawning child process, `getpid()` to get current process pid. You are not allowed to use other means of concurrent/ parallel processing. 10 points will be deducted from completion points if other multi-processing functions is used.
2. `pipe()` or `mkfifo()` for inter-process communication. `shm_open()` related functions and `socket()` are also allowed, but not recommended. Use of other means of IPC will result in 10 points deducted

from completion points.

3. `printf()`, `dprintf()`, `fprintf()` and `stdio.h` related functions are thread-safe functions, as required in POSIX standard. You don't need to worry about race conditions when using them. Use other output functions carefully. Do check the manual `man 7 attributes` and man pages of the functions if you are not sure about thread-safety issues.

## Project Completion

There are two ways you can complete the project.

### From scratch

Create the calculator from scratch. You get full mark if you comply with the rules and complete the project. Partial marks is given according to the completeness comparing to the skeleton if the submission is not complete.

### From skeleton

Skeleton is provided. You get the marks for completing different missing parts of the calculator. Marks for different parts are noted in the skeleton source files.

### Code explanation

Either ways, you might need to explain the code. Marks will be deducted if you fail to explain your own code. So don't try to copy from others or the Internet, or be smart when copying.

## Marks

1. Completion: 70% - See the `//TODO`s in the skeleton code files for skeleton partial marks.
2. Explanation: 30%
3. Plagirism: Instant 0

## Submission

Tar gzip your source code folder and submit the `.tar.gz` file to CASS. The deadline for submission is 29 March, 23:59.

## Materials

### Skeleton code

The skeleton has the following folder structure:

```
.
bin
    multical.c
include
    expression.h
    fd.h
src
    expression.h
    fd.h
test
    t_expression.h
Makefile
README.md
infile
```

#### Makefile

You can use `make` in the project directory to compile everything. The makefile is ready to use as long as you follow the folder structure below.

#### `bin`

`./bin/` is the folder for executable sources. An executable `.out` file is generated for each `.c` file in the `./bin/` folder. For example, the `./bin/multical.c` file in will be compiled into `./bin/multical.out`.

The file `./bin/multical.c` is the skeleton file you need to modify. Add code to the missing parts of the file.

#### `include` and `src`

`./include/` and `./src/` stores the source files for auxillary functions and data structures. `./src/` stores the `.c` files and `./include` stores the `.h` files.

`./include/expression.h` and `./src/expression.c` helps you with handling the expression strings. You can use these in your project code.

#### `test`

`./test/` is the folder for testing of auxillary functions. `./test/t_expression.c` provided a nice example of how you can write a test for your own function. Testing is not required in project submission.

## Required knowledge

#### Unix API

1. `fork()`
2. `wait()`
3. `getpid()`
4. `pipe()` or `mkfifo()`

You can always use `man 2 [function]` or `man 3 [function]` to see the documents. Manual section 2 is Liunx specific and section 3 is Unix in general.

## C Library

1. `fdopen()` converts an `int` fd into a `FILE *` file pointer that can be used with `stdio.h` standard I/O functions.

2. `fopen()`, `fclose()`, `fscanf()`, `fprintf()` and other `stdio.h` functions that deals with file and I/O. Beware that these I/O functions have their internal buffer which has to be `fflush()`ed to take effect.

## Reference Book

Stevens, W. R., & Rago, S. A. (2013). Advanced programming in the UNIX environment. Addison-Wesley.