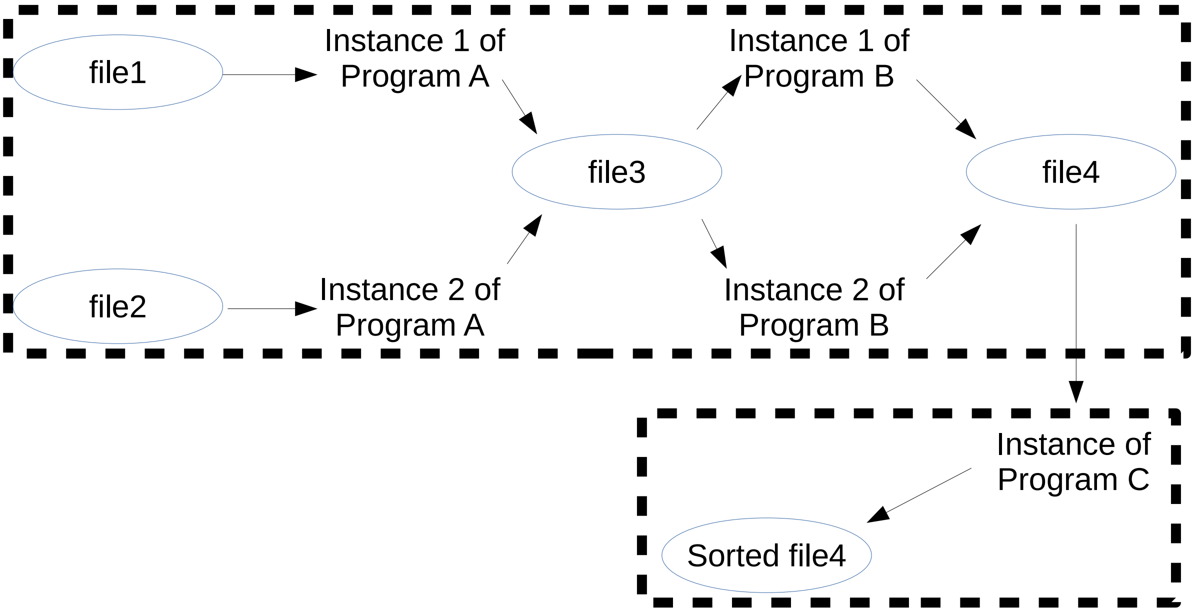
March 5th, 2020

CSE344 – System Programming - Homework #1 - v2
File I/O and file based interprocess communication

The homework consists of 2 parts.



Part I

You'll code 2 programs, program A and program B. Each will be executed twice in arbitrary order, so there will be 4 processes running **in parallel**. Instances of program A will read from distinct input files, and write to a common destination file, while both instances of program B will read from program A's common destination file, modify it, and write to another destination file. All will happen at the same time.

Program A will receive 3 command line arguments:

```
./programA -i inputPathA -o outputPathA -t time
```

`inputPathA` and `outputPathA` are absolute file paths and `time` an integer in [1,50]. `inputPathA` must denote the absolute path of a non-empty ASCII-encoded text file.

Program A will open in readonly mode the file denoted by `inputPathA.`For every 32 bytes that it reads, it'll convert them to 16 complex numbers; e.g. if it reads the characters "a5", since their ASCII equivalent is 9753, it'll convert them to the complex number "97 +i53". Then program A will write those 16 complex numbers to the ASCII file denoted by `outputPathA` as a single line of 16 comma separated complex numbers, at the first available empty row. A row is considered empty if it contains nothing or just the '\n' character. After writing a line, the program will call the `sleep` system call and go to sleep for `time` milliseconds. Once Program A reaches the end of `inputPathA` or if the file doesn't contain 32 more bytes then it will exit and close the file denoted by `inputPathA`.

Two instances of program A will be executed (in arbitrary order), with distinct `inputPathA` arguments, but with the same `outputPathA` parameter. So in practice, we expect two processes reading simultaneously two distinct files, and filling together the same output file with lines containing 16 complex numbers each.

Program B will also receive 3 commandline arguments:

```
./programB -i inputPathB -o outputPathB -t time
```

of the same type as the parameters of program B. The input file of Program B will be the output file of program A.

```
inputPathB = outputPathA
```

Two instances of program B will be executed (in arbitrary order, before, after or between the program A executions), with the same `inputPathB` and `outputpathB` arguments. So in practice, we expect two processes reading simultaneously the same file, removing lines from it, and writing simultaneously to the same output file.

Program B has a simple task. It will open `inputPathB,` and then, it will read a random line from the file. In case of an empty line, it will search linearly for a non-empty line (wrapping around to the beginning of the file if necessary). The same definition of emptiness as before applies.
When a non-empty line is found, it'll read those 16 complex numbers, delete that line by overwriting it with a '\n' and then calculate the discrete Fourier transform (using the FFT algorithm) of those 16 numbers, and write the calculation output to the file denoted by `outputPathB` as comma separated 16 complex numbers at the first available empty line. Then it'll sleep for `time` milliseconds. If there is no filled line by program A yet, then it'll sleep for `time` milliseconds, and try again.

If the instances of program B have finished processing all the content produced by the instances of program A (meaning that both program A instances have finished reading their respective input files) then both instances of program B will exit; and the last instance of program B exiting, will also close the files denoted by `outputPathB` and `inputPathB`.

If everything works out alright, if inputPathA files contain each 32x+y and 32a+b bytes, then the file denoted by outputPathB will contain exactly x+a lines, each with 16 complex numbers.

Tips:
- processes writing/reading at the same time to/from the same file is a disaster waiting to happen unless you use locking.
- you'll need to figure out a way to distinguish between the case where an instance of program B cannot find an empty line because program A instances haven't got a chance to write yet, and the case where program A instances have terminated.
- you'll need to also figure out a way to know when an instance of program B is the last alive of the two, so that it'll close the files denoted by `outputPathB` and `inputPathB` files once it decides to terminate.
- Test your program thoroughly before submission, with large files, small files, small sleep times, large sleep times, different orders of execution for the 4 processes, etc.

Part II
Program C must admit a single commandline argument: the path of the output file filled in by program B's instances: `outputPathB = inputPathC`

```
./programC -i inputPathC
```

If everything went alright before, it'll now contain a certain number of rows of 16 complex numbers in each. Program C will run as a single instance, and sort the lines in ascending order of the file denoted by `inputPathC` using mergesort. The lines will be compared with respect to their cumulative magnitude. The sorting must be done **in-place, in the file, not in memory (except for the lines being compared).**

Rules:

- use the `getopt()` library method for parsing commandline arguments.
- Use system calls for all file I/O purposes, don't use standard C library functions.
- All file operations must be **fully synchronized**. No buffering of standard C library operations, no kernel buffering of system calls. No buffering, period. I want to be able to see LIVE the files change their content while all the processes execute.
- Your programs are not allowed to crash due to any foreseeable reason. In case of an error, exit by printing to stderr a nicely formatted informative errno based message.
- if the command line arguments are missing/invalid your program must print usage information and exit.
- All mathematical operations will be realized with an accuracy down to 3 decimal points.
- Do your best to produce a robust program that will not crash even with the most evil user executing it.
- Don't use any IPC/synchronization methods/techniques/tools, besides file blocking, and certainly none of the IPC or synchronization methods not yet covered in class and/or not explicitly described in this document.
- Compilation must be warning-free.
- You will provide a demonstration of your program to the course assistants. They will provide the details.
- No late submissions will be allowed.
- Close all files and free all resources explicitly.

Submission:

- your source files, your makefile and a report on how you solved the issues in this homework.

Grading:
Part I: 70 points.
Part I+II: 100 points.
Don't submit part II if you aren't submitting also part I.

- Does not compile: -100
- Does not run or crashes with normal input: -100
- Runs with normal input, but the final output is incorrect -100
- The program runs with normal input but can crash for n reasons: -10 * n
- Violating the homework rules: -100
- No report: -30

Good luck.