CMP SCI 4760 **Operating Systems**
Project # 1 Due Date: September 18, 2022

Important: Please do all projects on `opsys`

Linux System Calls and Library Functions

Purpose

This is your warm up project building on your knowledge from CMPSCI 2750. The goal of this project is to become familiar with the environment in `opsys` while practicing system calls. You will also demonstrate your proficiency in the use of `perror` and `getopt` in this submission. Additionally, you should understand the different steps of compilation and linking.

This project is just reading some integers from an input file and writing them out in reverse order into the output file. All that is required of you is to use `fork()`, parsing the options, and using `perror()`.

Task

Your project should consist of one program, which will `fork` versions of itself to do some file processing. It will start by using some command line arguments. You must implement at least the following command line arguments using `getopt`:

```
-h
-i inputfilename
-o outputfilename
```

The `-h` option will display all legal command line options and how it is expected to run, as well as the default behavior. If input and output filenames are not specified, the defaults should be `input.dat` and `output.dat`.

Once you have parsed the command line arguments and validated them, then you should attempt to open the input file. The input file should start with a number on a line by itself, with that number indicating the amount of times you will be required to do a task with copies of your process using `fork`. Each task list will follow in the file, which will consist of an integer on one line (representing the amount of numbers that will follow) and that many numbers. An example of this input file is below:

```
3
6
3 6 107 8 1 3
4
1 3 50 4
7
5 3 5 7 8 9 1
```

The main process (parent) should read the first line of the file. Once it has read that line, it should then go into a loop based on that number, with each iteration of the loop forking off a copy that will then process the next two lines. Once that child has finished its work (defined below), it will write some data to the output file and then terminate. At that point, as the parent detects its child has terminated, it should do another iteration of the loop until done. After all children have terminated. the parent should write the PIDs of all of its children that it launched, as well as its own PID to the output file.

When a child process starts, it should read the next line of the file, which will tell it how many numbers to read afterwards. We see in our example file that the first forked child would read a `6`. It should then read that number of integers and put them into a stack. After putting all of the numbers into a stack, the child should write its PID to the output file, followed by those numbers in reverse order. For example:

```
13278: 3 1 8 107 6 3
```

After this has been done with the example file given above, we would expect an output file to look as below (with different PIDs, of course):

```
13278: 7 3 1 8 107 6 3
13281: 4 50 3 1
13294: 1 9 8 7 5 3 5
All children were: 13278 13281 13294
Parent PID: 13250
```

I'll like some meaningful error messages. The format for error messages should be:

```
logParse: Error: Detailed error message
```

where `logParse` is actually the name of the executable (`argv[0]`) that you are trying to execute. These error messages should be sent to `stderr`. Any errors generated due to system calls should be reported using `perror` with the same format.

Here is the set of tasks after you have your code compiled.

1. Run your program and observe the results for different number of processes.
2. Report if the number of integer on a line does not match the expected number that was specified on the previous line.

Invoking the solution

Your solution will be invoked using the following command:

```
chain [-h] [-i inputfilename] [-o outputfilename]
```

If a user specifies `-h`, please print the usage message and terminate. If the user does not specify an input file, print the usage message and terminate. If the user does not specify an output file, create the filename by appending `.out` to the input file name.

Please make use of `perror` after each system call to report any error messages. With the use of `perror`, I'll like some meaningful error messages. The format for error messages should be as specified earlier.

It is required for this project that you use version control (git), a `Makefile`, and a `README`. You should check in a version of your code at each step of exercise, at the minimum into your git repository. Your `README` file should consist, at a minimum, of a description of how I should compile and run your project, any outstanding problems that it still has, and any problems you encountered. Your `Makefile` should use suffix-rules or pattern-rules and have an option to clean up object files.

Suggested implementation steps

1. Set up your git repository, if you have not already done so. You must periodically check your code into the git repository (once a day, or whenever you make and test substantial changes). [Day 1]
2. Create your Makefile. Make sure to use suffix rules or pattern rules. Type in the code from the book and have it working. [Day 2]
3. Write code to parse options and receive the command parameters. Study `getopt(3)`, if you do not know how to do it. The man page also has an example to guide you or you may use the tutorial. [Day 3]
4. Write and test code as a function to read one data item and reverse it. [Day 4]
5. Write code to process the entire data file. [Day 5-6]
6. Introduce forking where the function is called. [Day 7-8]
7. Create the `README` file. [Day 9]

Criteria for success

Please follow the guidelines. Make sure that you are making judicious use of `perror` with each system call. The code should be well-documented and appropriately indented.

Grading

1. *Overall submission: 10 pts*. Program compiles and upon reading, seems to be able to solve the assigned problem.
2. *Code readability: 10 pts*. The code must be readable, with appropriate comments. Author and date should be identified.
3. *Command line parsing: 10 pts*. Program is able to parse the command line appropriately, assigning defaults as needed; issues help if needed.
4. *Use of perror: 10 pts*. Program outputs appropriate error messages, making use of `perror(3)`.
5. *Makefile: 10 pts*. Must use suffix rules or pattern rules. You'll receive only 2 points for Makefile without those rules.
6. *README: 10 pts*. Must address any special things you did, or if you missed anything.
7. *Proper processing of each item: 10 pts*. Each item (number to be read and verifying that the correct number of items are present) should be properly processes.
8. *Conformance to specifications: 30 pts*.

Submission

Create your programs in a directory called *username*.1 where *username* is your user name on opsys. Once you are done with developing and debugging, *remove the executables and object files*, and issue the following commands:

```
% make clean
% cd
% chmod 755 ~
% ~sanjiv/bin/handin cs4760 1
% chmod 700 ~
```

Do not copy and paste those commands from the PDF *of the assignment. Type in the commands.* You can resubmit the code any number of times after initial submission (prior to the deadline).

Do not forget `Makefile` (with suffix or pattern rules), your versioning files (`.git` subdirectory), and `README` for the assignment. If you do not use version control, you will lose 10 points. I want to see the log of how the program files are modified. Therefore, you should use some logging mechanism, such as git, and let me know about it in your `README`. You must check in the files at least once a day while you are working on them. I do not like to see any extensions on `Makefile` and `README` files.

Before the final submission, perform a `make clean` and keep the latest source checked out in your directory.

You do not have to hand in a hard copy of the project. Assignment is due by 11:59pm on the due date.