CMPT 300: Operating Systems
School of Computing Science
Fall 2015, Section D1

Assignment #3: Message Passing & Scheduling
Due Date: Thursday, November 12, 2015

Assignments in this Course:

All four assignments in CMPT 300 this term will be related to each other. Although it may not be obvious now, you will be using the knowledge (and some code) from this assignment in your future assignments. Therefore, please take the time to understand the concepts and to write clean, readable code.

If you were unable to complete assignment #2 and would like a solution to build upon for this assignment, you may purchase an assignment for a penalty of -20% (i.e., 20/100) off your mark for this assignment. You cannot arbitrarily adopt another student's (or anyone else's) code; that is considered plagiarism. This purchased assignment is not guaranteed to be bug free, but we will provide a solution of reasonable quality. Please contact your instructor if you would like to do this. Of course, read this assignment description first.

Cleaning Up Your Processes

When using `fork()` (and related functions) for the first time, it is easy have bugs that leave processes on the system, even when you logout of the workstation. It is **your** responsibility to clean up (i.e., kill) extraneous processes from your workstation before you logout. Learn how to use the **ps** and **kill** (and related) commands.

Marks will be deducted if you leave processes on a workstation after you logout.

Overview:

In this assignment, you will be extending and improving the `lyrebird` program from Assignment #2. The format of the input is changing in order to make `lyrebird` more flexible and to allow for some new behavior. You will also have to make changes to the internal structure of `lyrebird`.

The `lyrebird` program from assignment #2 had the ability to use multiple cores through the use of child processes. However, it still had various limitations. First, a large number of encrypted files would lead to a large number of child processes being created, more than the number of CPU cores available. This process creation step has an overhead cost, so creating more children than CPU cores may not be the most efficient use of resources. An additional issue with the `lyrebird` program from assignment #2 is that there was little ability to control how much CPU time or memory `lyrebird` could end up using. If we could limit the number of `lyrebird` processes to a more reasonable amount, we could make sure that there are CPU and memory resources available on that machine to still do other tasks.

In this assignment, you will be creating one `lyrebird` process for each CPU core. In this situation, you will run into two new problems:

1. There is no longer a 1-to-1 mapping between the child processes and the encrypted files. As a result, we have the question of which of the encrypted files should be assigned to which child process. This is a scheduling problem similar to that of assigning processes to CPUs. You will implement and examine two possible scheduling algorithms in this assignment: *round robin* and *first come first served (FCFS)*.

2. The parent `lyrebird` process needs a way of communicating to its children what files it should decrypt, and the child processes need a way of communicating to their parent process when they are ready to decrypt a new file. This task will be done using pipes to pass messages between the parent and its children.

Standard Comment About Design Decisions

Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In those cases, you should make reasonable design decisions (e.g., that do not contradict what we have said and do not significantly change the purpose of the assignment), document them in your source code, and discuss them in your report or README file. Of course, you may ask questions about this assignment (for example, on the mailing list) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

Input/Output and Behavior Specification:

Your version of `lyrebird` for assignment #3 will maintain the similar parent-child relationship from assignment #2. The user will pass a list of encrypted files to the parent process and the parent process will use its child processes to do the decryption. The main difference in this assignment from the last is that the number of child processes will not necessarily match the number of encrypted files.

Like the previous assignments, your program should have the name `lyrebird`. Like assignment #2, `lyrebird` should take exactly one command-line argument. That one command-line argument is the name of a configuration file containing (a) a choice of scheduling algorithm, (b) a list of files to decrypt and (c) a list of locations in which to save the decrypted output. An example of how the program is started from the command line is:

```
$ ./lyrebird config_file.txt
```

The first line of the configuration file will be the choice of scheduling algorithm. Valid options for the scheduling algorithm are `round robin` or `fcfs` (both will be discussed further below). If a different string is given as a scheduling option, `lyrebird` should recognize that as an error.

The remaining contents of the configuration file are pairs of file locations (i.e. file names with, optionally, its full path specified), with two file locations per line. The first file location in each line should correspond to a file containing encrypted tweets. The second file location should correspond to the file where the decrypted tweets will be saved. For example, the contents of `config_file.txt` may look like:

```
round robin
./encrypted_tweets.txt ./decrypted_tweets.txt
/home/userid/more_tweets.txt /home/userid/output.txt
```

where `round robin` is the choice of scheduling algorithm, `./encrypted_tweets.txt` and `/home/userid/more_tweets.txt` are files containing the encrypted tweets, while `./decrypted_tweets.txt` and `/home/userid/output.txt` are the files where the corresponding decrypted tweets will be saved. As with assignment #2, you can safely assume that each string in the configuration file (i.e. every file location) is a maximum of 1024 characters long. There is no limit on the number of lines (i.e. encrypted files) in the configuration file. Note that the files containing the encrypted tweets will have the same format as in the previous assignments.

Creating Child Processes

When `lyrebird` first starts running, it should create its child processes as well as *read and write* pipes to communicate with each child process. As mentioned in the overview section, there should be the same number of `lyrebird` processes as there are CPU cores on the machine. Therefore, number of children that should be created on a machine with `N` CPU cores is `N-1` (as the parent process should get its own core). Note that the number of cores should *not* be hard coded; your program should discover how many cores the machine has, then create the appropriate number of child processes.

Scheduling

Once the child processes have been created, the parent `lyrebird` process should determine which encrypted files should be sent to which child process. This task is a scheduling problem and in this assignment, your program should implement two scheduling algorithms:

1. <u>round robin</u>: The round robin scheduler will assign encrypted files to child processes so that each child process gets, roughly, an equal number of files to decrypt. Furthermore, the files will be distributed in order in the same way you deal out cards in a card game. For example, if there are three (3) child processes (let's call them `childA`, `childB`, and `childC`) and there are seven (7) files to decrypt (let's call them `file1.txt` to `file7.txt`), then the round robin scheduler will assign files to the child processes as follows:

Child Processes	**Files Decrypted**		
`childA`	`file1.txt`	`file4.txt`	`file7.txt`
`childB`	`file2.txt`	`file5.txt`	
`childC`	`file3.txt`	`file6.txt`	

2. First come first served (FCFS): The FCFS scheduler will assign an encrypted file to a child whenever a child is free to do work. In other words, the first child process to come and ask for work will be the first to be served with the name of an encrypted file. For example if we have three child processes and seven files (named like in the round robin example), and let's say we (magically) know that the encrypted files take the following lengths of time to be decrypted:

File Name	Time Needed to Decrypt File
`file1.txt`	10 seconds
`file2.txt`	5 seconds
`file3.txt`	3 seconds
`file4.txt`	1 second
`file5.txt`	2 seconds
`file6.txt`	3 seconds
`file7.txt`	6 seconds

then the FCFS scheduler would assign files to the child processes as follows:

Child Processes	Files Decrypted			
`childA`	`file1.txt`			
`childB`	`file2.txt`	`file6.txt`		
`childC`	`file3.txt`	`file4.txt`	`file5.txt`	`file7.txt`

Note that you should *not* try to estimate the length of time a file will take to decrypt (i.e. do not schedule files to child processes based on the number of tweets in the file). Instead, the parent process should simply send a child process the name of an encrypted file whenever that child process is ready to decrypt a new file.

Message Passing

When the parent `lyrebird` process has decided, using its scheduling algorithm, to assign an encrypted file to a child process, it should send the names of the encrypted file and the output file to the child process through a pipe. When the parent process has sent that message, it should output the following to the terminal (note the following text has been updated from assignment #2):

```
[Sat Sep 19 20:43:14 2015] Child process ID #5137 will decrypt
./encrypted_tweets.txt.
```

As in assignment #2, the first item in each line is the current time in **`date`** command format (for information regarding this format, check the `ctime()` function).

Meanwhile, the child process should read the names of the encrypted file and the output file from the pipe and perform the decryption. Once the child process has completed the decryption, it should output the following to the terminal (note the following text has been updated from assignment #2):

```
[Sat Sep 19 20:43:14 2015] Process ID #5137 decrypted
./encrypted_tweets.txt successfully.
```

The child should also send a message, over the pipe, to its parent process saying that the child is ready to decrypt another file. The format of that message is entirely up to you.

Exiting Protocol

When all the encrypted files have been assigned to child processes, the parent process should close the pipes it uses to write to its children. The parent process should then read any remaining messages from its child processes until the child processes close their end of the pipes that the parent reads from.

When a child process identifies that the pipe they're reading file names from has been closed, it should close both of the pipes it uses to communicate with the parent and exit with a zero exit status.

Once a child process exits, the parent should close the pipe it uses to read messages from the child. It then *must* check the exit status of that child process. If the child process exited as a result of an error (more on this below), the parent should output the following message:

```
[Sat Sep 19 20:43:14 2015] Child process ID #5138 did not terminate
successfully.
```

Once all child processes have terminated, the parent process can then proceed to exit.

Error Handling

If the child process encounters an error, that process should display an error message containing its process ID, then exit with a non-zero exit status. Any error message that is outputted should follow the same format as the messages described above: the error message should start with the date & time, followed by the message, and the process ID should appear somewhere in the message.

Once the child process has outputted an error message, it should then determine whether it can recover from the error, or whether it should exit. If the integrity of the child process is compromised, then the process should exit with a non-zero exit status. Otherwise, the child should continue to run. For example, if the encrypted file does not exist, an error message should be printed out, but the child process is still in good shape and can continue running. On the other hand, if a `malloc()` call fails in the child, then there's clearly something wrong with the heap of that child process and it should exit with a non-zero exit status.

If the parent process encounters an error, it should display an error message that follows the same format as the messages described above: the error message should start with the date & time, followed by the message, and the process ID should appear somewhere in the message. Once the parent process has outputted that error message, it is still responsible for managing its child processes. So, when the parent wants to exit due to an error, it should start the exiting protocol described above and attempt to exit in an organized fashion.

Required Design:

Your program *must* use `fork()` to create child processes. If your program does not use `fork()`, you will receive a mark of zero for correctness. You may also require functions like `getpid()`, `waitpid()`, and `sysconf()`.

Also, your program must pass information between parent and child processes using pipes. Two pipes should be created for each child process: one to pass messages from parent to child, and one to pass messages from child to parent. To take care of this task, you'll likely need to use the `pipe()`, `read()`, `write()` and `close()` functions. If your program does not use `pipe()`, you will receive a mark of zero for correctness.

For the scheduling algorithms, particularly FCFS, you'll need the ability to monitor multiple pipes to see if any information has arrived from the children. For this, you'll need the `select()` function.

As appropriate, you must use C memory allocation (e.g., `malloc()`, `free()`) and C file I/O functions (e.g., `fopen()`, `fscanf()`, `fclose()`). Because of the use of MEMWATCH (see below), you cannot use C++ streams, the Standard Template Library (STL), or the C++ standard library extensions (e.g., cannot use type/class `string`). Also, your TA may not have any expertise in C++ and therefore we cannot guarantee support for languages other than C.

You must write a Makefile for your program. When someone types `make`, your Makefile should build the executable program `lyrebird`. When someone types `make clean`, your Makefile should remove the executable `lyrebird` (if any), all `.o` files (if any), `memwatch.log`, and all `core` files (if any).

It is IMPERATIVE that your program properly deallocates ALL dynamic memory in a correct fashion (i.e., using `free()`) before your program terminates, or else your assignment will LOSE marks. To check that your program properly allocates and deallocates ALL dynamic memory it uses, you must use the MEMWATCH package, as described in assignment #1. If your assignment is not properly compiled with MEMWATCH enabled, or if MEMWATCH reports that your memory allocation/deallocation was incorrect, then you will lose marks.

When developing and testing your program, make sure you clean up all of your processes (including `lyrebird`) before you logout of a workstation. Marks will be deducted for processes left on workstations.

What to Hand In:

You will submit your assignment through http://courses.cs.sfu.ca. Your submission should be a `zip` archive file with the name `submit.zip`. The zip file should contain the following:

1. A **README** file (ASCII text is fine) for your assignment with: (1) your name, (2) student number, (3) SFU user name, (4) lecture section, (5) instructor's name, and (6) TA's name clearly labeled. All these items of information should also be part of **each file** that you submit (e.g., as a comment in your code files). The **README** file must also include a short description of your program, as well as a description of the relevant commands to build (e.g. `make all`) and how

to execute your programs including command line parameters. As per the academic honesty guidelines, you should list your sources and the people you have consulted within this **README** file.
2. A report in HTML file format, in a file called **report.html**, describing the design, implementation, and testing of your assignment. The report should contain *no more than 750 words.* You do not need to repeat any information contained in this assignment description. I recommend you spend 25% of your report on an overview of your assignment, 50% on your design and implementation, and 25% on how you tested your program, and some concluding remarks. Note the emphasis on testing your program.
3. Your source code file(s) for `lyrebird`, including all header files. Do NOT submit any MEMWATCH files, as the TA will use his own fresh copy of that code, but the use of MEMWATCH should be enabled in your code and `Makefile`.
4. Your `Makefile`.

NOTE: Do **not** submit files or test data **not** described above. Only submit what is requested and what is required to compile your program (except, of course, the MEMWATCH files). When you submit your assignment, please make sure that we can unzip, make, and run your assignment without having to switch directories.

Marking:

This assignment is worth 10% of your final mark in this course. **This is an individual assignment. Do not work in groups.** Review the course outline on this matter.

The assignment itself will be marked as follows: 20% for your report (clarity, technical accuracy, completeness, thoroughness of the testing, etc.), 50% for the correctness of the program when we test it using the CSIL Linux machines, using `gcc`, and 30% for the quality of the implementation (design, modularity, good software engineering, coding style, useful and appropriate comments, etc.).

Note that the correctness mark will be computed solely on how your program runs and not on what the code looks like (with the exception of the use of `fork()` and `pipe()`). If your source code, **as submitted**, does not compile and run (using the submitted Makefile) on the CSIL Linux workstations using `gcc`, you will receive a mark of zero for correctness. Review the Course Outline on this matter.

When it comes to your quality of implementation mark, all that you have learned about good programming style and comments in your code will apply. Having correct code is important, but good style, design, and documentation are also important. We cannot provide an exhaustive list of what we will look for, but an incomplete list includes: a comment for each source code file, a comment for each procedure/function, a comment for each significant (global or local) variable, good choice of names/identifiers, proper modularity (e.g., do NOT put all/most of the code in `main()`), checking function return values for errors, etc.

NOTE: There are a number of programs that you can download off the Internet that provide similar functionality to what you are asked to implement for this assignment. We are familiar with them. Therefore, do **not** download these programs; write your own solution to this problem. Modifying someone else's program (including programs that you can download) is against the requirements of this assignment and is an Academic Offense. If you have any doubts about whether your actions are permissible or not, you should ask the instructor **before** proceeding.

Hints:

- Keep in mind that pipes are identified using file *descriptors*, which are different from the file *pointers* you're probably more familiar with. It might be helpful to know the difference between the two. Here's a decent description of the difference between file descriptors and file pointers: http://www.go4expert.com/articles/understanding-file-descriptor-file-t28936 .
- You may also want to learn about the following Unix programs: **`ps`**, **`grep`**, **`kill`**.
- Before you submit, make sure your `submit.zip` file works from within a fresh directory. That way, you make sure that `submit.zip` contains all the needed files for testing (with the exception of MEMWATCH, of course).
- Remember, make sure that your program does **not** produce any debugging or extraneous output during **normal** execution. Only the requested output should be generated. Marks will be deducted for incorrect and other unrequested output. That said, it is acceptable to have output to report an actual error.
- Remember to list whatever sources you use in your README file.

Further hints and clarifications may be given later on the course mailing list, if warranted. Be sure to read the emails on the mailing list on a regular basis.