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

Assignment #2: Process Management
Due Date: Thursday, October 15, 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 #1 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.

Overview:

In this assignment, you will be extending and improving the `lyrebird` program from Assignment #1. The format of the input is changing in order to make `lyrebird` more flexible and some of the simplifications made in the previous assignment will no longer be allowed. You will also have to make changes to the internal structure of `lyrebird`.

Specifically, project lyrebird needs to be sped up. The number of encrypted tweets that your group are finding is more than your current version of `lyrebird` can keep up with. However, you know that most computers these days come with multiple CPU cores. You decide to speed up the decryption process by taking advantage of all cores on the system, and plan to spread the decryption task across multiple processes. You will use `fork()` to create these multiple processes.

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.

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 #2 will be setup much like a parent-child relationship. The user will pass a list of encrypted files to the parent process and the parent process will create child processes to do the decryption. One child process will be created to decrypt each encrypted file.

Like assignment #1, your program should have the name `lyrebird`. Unlike assignment #1, `lyrebird` should take exactly one command-line argument. That one command-line argument is the name of a configuration file containing (a) a list of files to decrypt and (b) 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 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:

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

where `./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. 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 assignment #1.

When `lyrebird` first starts running, it should read the configuration file. For each encrypted file listed in the configuration file, `lyrebird` should use the `fork()` system call to create a child process to decrypt that file. When a child process is created, the parent process should output the following message:

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

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

Once each child process has been created, the parent process should wait until all its child processes have terminated before exiting. Once a child process exits, the parent *must* check the exit status of that child process. If the child process exited as a result of an error, the parent should output the following message:

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

The child processes should behave in a similar way as assignment #1 with the exception that the parent process should provide the encrypted and output file locations; the user should not interact directly with the child processes at all. When the child process successfully completes the decryption of its file, the child process should output the following message:

```
[Sat Sep 19 20:43:14 2015] Decryption of ./encrypted_tweets.txt complete.
Process ID #5137 Exiting.
```

If the child or the parent process encounters an error, the 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.

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()` and `waitpid()`.

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++ `stdlib` (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()`). 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:

- 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.
- Note that the CSIL lab will be without power October 10th and 11th as the university replaces a high voltage transformer for the Advanced Sciences Building. Keep this in mind as you schedule your time for this assignment.

Further hints 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.