CMP SCI 4760 **Operating Systems**
Project # 1 Due Date: September 14, 2022

Important: Please do all projects on `hoare`

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 `hoare` 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, and creating your executable. The project is a slight modification of Exercise 2.13 Message Logging (p. 55-56) in your text by Robbins/Robbins. The modification is that each message will also issue a classification as to whether the message is informational (`INFO`), warning (`WARN`), error (`ERROR`), or fatal (`FATAL`).

The logging utility to be developed by you allows the caller to save a message at the end of a list. The logger also records the time that the message was logged. The sample `log.h` file (Program 2-11; p. 55) shows the data structures and prototypes of the functions that will be needed. I'll suggest modifying the data structure as follows:

```
#include <time.h>

typedef struct data_struct
{
    time_t    time;      // Time stamp
    char      type;      // Message type (I/W/E/F)
    char *    string;    // Message string
} data_t;

int    addmsg ( const char type, const char * msg );
void   clearlog();
char * getlog();
int    savelog ( char * filename );
```

The `data_t` structure holds a `time_t` value (`time`) and a pointer to a character string of undetermined length (`string`). The function `addmsg` creates the data structure `data_t` by adding the time stamp to the supplied parameters message type and message string, and inserts a copy of the data structure at the end of the list. It also verifies that the message type is a valid message type and issues an error if it is invalid. The `savelog` function saves the logged message to a disk file. The `clearlog` function releases all the storage that has been allocated for the logged message and empties the list of logged messages. The `getlog` function allocates enough space for a string containing the entire log, copies the log into this string, and returns a pointer to the string. It is the responsibility of the calling program to free this memory when necessary.

If successful, `addmsg` and `savelog` return 0; both of them return -1 if unsuccessful. A successful `getlog` call returns a pointer to the log string; it returns a `NULL` upon unsuccessful invocation. The three functions also set `errno` on failure (part of `perror` function).

Program 2.12 (p. 56) provides the template for the library that you will complete.

Write a program to test your logging utility. This program should give enough options to fully test the various functions in the file. You should include an optional sleep time between different calls to test the time stamp.

Task

Problem: Write source files and header files for your logging utility. Then write a program to use the library and test the library by sending different log messages. You can name the source code for your program as `driver.c` and the executable can be called `driver`.

1. When called with no arguments, `driver` sends messages to the logger and saves those messages in a file called `messages.log`.
2. If `driver` is called with the argument `-h`, print the help (usage) message and terminate. Ignore any other options and arguments.
3. When `driver` is called with option `-t sec`, then your program giving options to create or modify the log will be have a sleep call between them of an average of `sec` seconds. Use a random number to generate this time in the range `[0, 2*sec]`.
4. If `driver` is supplied with a command line argument, interpret the argument as the file name to save the messages.
5. When a fatal message is added, the log should be written to the file (`savelog`) and the program terminated by calling `exit`.

Invoking the solution

Your solution will be invoked using the following command:

```
driver [-h] [-t sec] [logfile]
```

With the use of `perror`, I'll like some meaningful error messages. The format for error messages should be:

```
driver: Error: Detailed error message
```

where `driver` is actually the name of the executable (`argv[0]`) and should be appropriately modified if the name of executable is changed without a need to recompile the source.

It is required for this project that you use version control (git), a `Makefile`, and a `README`. 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. [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. [Day 3]
4. Create the library functions for the logger and test them. Make sure to add the time stamp in a readable way as hh:mm:ss in 24-hour format when you add messages. You can use the functions `time` and `localtime` to get the time. [Day 4-5]
5. Send messages to be logged. You can create the messages in a text file and read them from standard input (or hard-code the file name in `driver`. Specify one message per line. [Day 6-7]
6. Create the `README` file. [Day 8]

Criteria for success

Please follow the guidelines. Pay attention to avoiding memory leaks, as suggested in the text.

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: 5 pts.* Program is able to parse the command line appropriately, assigning defaults as needed; issues help if needed.
4. *Use of perror: 5 pts.* Program outputs appropriate error messages, making use of `perror(3)`.
5. *Makefile: 5 pts.* Must use suffix rules or pattern rules. You'll receive only 2 points for Makefile without those rules. Also ensure your Makefile can do a clean.
6. *README: 5 pts.* Must address any special things you did, or if you missed anything.
7. *Properly implemented functions: 10 pts.* Properly created and used functions.
8. *Conformance to specifications: 50 pts.* Each of the six specified subtasks accounts for 10 points.

Submission

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.1 where *username* is your login name on hoare. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.1

cp -p -r username.1 /home/hauschild/cs4760/assignment1
```

If you have to resubmit, add a .2 to the end of your directory name and copy that over.

Do not forget `Makefile` (with suffix or pattern rules), your versioning files (`.git` subdircetory), 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.