OS 2014 - EX3
A Multi-threaded Output Device

Supervisor - Noga H. Rotman

He who holds me by a thread is not strong; the thread is strong.
Antonio Porchia, *Voices*

Due Date: 1/5/2013

Before you start, don't forget to read the course guidelines!
This exercise is **complicated** and the debugging process may take a while. Considering Moadey Bet, we've given you plenty of time to finish it. Please start early.
No extensions will be given without a good reason (Miluim, sickness or childbirth).

Assignment

Consider an output device in multi-threaded system that accepts output tasks from several client threads concurrently. Each client thread task comprises of writing a block of data to the output device as a single uninterrupted atomic operation. A trivial implementation is to have a single lock that protects the device, where each client thread gets the lock, performs the operation and then releases the lock. Your task is to implement an efficient non-blocking package, which is delivered in the form of a static library. In other words, you will implement a daemon thread that queues client threads' output tasks requests and write these tasks asynchronously to the device, which is simulated, in this exercise, as an output disk file.

The public interface of your library is given by outputdevice.h and explained below. Obviously, there are few internal functions (and data structures) that you may find necessary to implement. These functions are not visible outside the library; therefore, you are not restricted in their number, signatures, and content as they are part of your private implementation. Your code should not have any memory leaks, but you are allowed to use one (and only one!) statically initialized mutex that will not be destroyed by your library. Also, you may assume that no more than `int` tasks will be written concurrently.

`int initdevice(char *filename)`

The function creates the output file `filename` if it does not exist and opens the output file for writing (appending if the file exists). This function should be called prior to any other functions as a necessary precondition for their success. However, you cannot assume it is done so, and should deal with cases in which this does not occur.

Return values are 0 on success, -2 for a filesystem error (inability to create the file, etc.), and -1 otherwise.

`int write2device(char *buffer, int length)`

The function writes the input `buffer` (which is of `length` size) to the output file. The buffer may be freed once this call returns (You should deal with any memory management issues). Note this is non-blocking package you are required to implement, therefore, you should return ASAP, even if the buffer has not yet been written to the disk.

On success, the function returns a `task_id` (>= 0), which identifies this write operation. Note that you should reuse `task_id`s when they become available (striving for low numbers). On failure, -1 will be returned.

Please refer to the exercise guidelines below for more information on how to implement this function.

`int flush2device(int task_id)`

The function blocks until the specified `task_id` has been written to the file. The `task_id` is a value that was previously returned by `write2device` function. If the `task_id` has already been written to disk, you should return immediately. In case all went well, the function should return 1. If the `task_id` doesn't exist the function should return -2, and if there is some other error due to the function (not in potential background threads), it should return -1.

`int wasItWritten(int task_id)`

The function returns (without blocking) whether the specified `task_id` has been written to the file. The `task_id` is a value that was previously returned by `write2device` function. If the `task_id` was written the function should return 0, if it wasn't, the function should return 1. If the `task_id` doesn't exist the function should return -2, and if there is some other error it should return -1.

`int howManyWritten()`

The function returns (without blocking) how many tasks were written to file so far (since last `initdevice`. If the number is higher than MAX_INT, then MIN_INT is returned. If there is some error, the function should return -1.

`void closedevice()`

The function closes the output file and resets the system so that it is possible to call `initdevice` again. All pending `task_id`s should be written to the output disk file. The function is non-blocking, and any attempt to write new buffers (or initializing) while the system is shutting down should cause an error in their respective functions (`flush2device` and `howManyWritten` should still work as long as tasks are still being written to file). Calling `closedevice` several times should not cause any ill-effects (in case `initdevice` was called sometime in the past). If the `closdevice()` function has an error, it should cause the process to exit.

`int wait4close()`

The function blocks until `closedevice` has finished, and `initdevice` can be used again. If closing finished successfuly, it returns 1. If `closedevice` has not been called at all, it returns -2, in case of other errors, it returns -1.

Error Messages STDERR

Message Error	**Error Description**
"system error\n"	When system call fails
"Output device library error\n"	The rest

Errors that cause the writing to fail (in the background threads - not while the above functions are running) should cause the process to exit (with an appropriate error message, and with a status which is not 0).

Background reading and Resources

1. Carefully read the pthread man-pages, which grouped into 3 major classes:
    1. ***Thread management:*** Working directly on threads.
    2. ***Mutexes:*** Dealing with synchronization ("mutual exclusion").
    3. ***Condition variables:*** Communication between threads that share a mutex.
2. POSIX Threads Programming a good tutorial

Submission

Submit a tar file on-line containing the following:

- A README file. The README should be structured according to the course guidelines, and contain an explanation on how and why your library functions are built the way they are.
- The source files for your implementation of the library.
- Your Makefile. Running *make* with no arguments should generate the *liboutputdevice.a* library.

Do not change the header file. Your exercise should work with our version of outputdevice.h.

Guidelines

- This exercise objective is Multi-threaded programming; therefore, make sure you protect all shared resources. You may want to use a multi-processor like `river` to test your code. You should see a nice interleaving of tasks in the output file without any corruption.
- Note that in the writing of the data, **you must use** the fwrite() function (seen here) **-** this is the only one we can guarantee will pass our tests successfully.
- Make sure to check the exit status of all the system calls and pthread calls 1you use.
- Use the Q&A forum, and be sure to check the existing topics, as some of your questions might have already been answered.
- Make your code readable (indentation, function names, etc.).

- Knock knock

- Race Conditions

- Who's there?