EECS3221 Assignment 2 Report

Michael Podimov

Ramona Sar

Dimitry Mochkin

Matthew MacEachern

Implementation of assignment requirements:

This portion of the report will explain in detail how each of the requirements in the assignment document were properly addressed and implemented in our program.

A3.1: In addition to the alarm thread created in the default program provided, we also created the two additional threads d1_thread and d2_thread. These two threads are initialized exactly the same as the alarm thread in our main function, and our status variable is checked after each initialization in order to ensure that the threads were created without any errors.

A3.2: On line 240 of our program we first check that the input is correct else we send an error message to stdout. If the command entered is valid, we enter the code block starting on line 244, where the command is processed and the required message is printed on line 250.

A3.3: At this point in execution our alarm thread is running and checking if there is an alarm to be processed. If there is no such alarm the thread's while loop sleeps for 1 second. Otherwise, the alarm is received and we check to see if the alarm expiry time is (a) closer to an odd integer number (on line 72) or (b) closer to an even integer number (on line 85).

A3.3 (c): In both cases, the set of instructions are practically the same, but we pass the alarm to display thread 1 in the case of (a) and display thread 2 in the case of (b), where we first display the required message for the respective display thread.

A3.3 (d): Because of the while loop in the alarm thread, the alarm thread is still free to process additional incoming alarms that are sent to it.

A3.4: Similar to the alarm thread, both display thread functions loop continuously to ensure that any subsequent alarm requests sent from the alarm thread are processed properly, and on line 132 of display thread 1 and line 179 of display thread 2 we call pthread_cond_wait which waits for an alarm request from the alarm thread.

A3.4 (a): Both display threads perform essentially the exact same instructions. The proper receive message is displayed upon the display thread receiving the alarm request.

A3.4 (b): The display threads repeatedly display the status of the thread and sleep(2) is used to ensure that the message is only displayed every 2 seconds. The condition of the while loop checks that the alarm time has not expired.

A3.4 (c): Immediately after the while loop terminates meaning the alarm request has finished processing in the display thread, the appropriate display thread expiry message is displayed indicating that the display thread is now free to process more alarm requests. The alarm_mutex is unlocked at this point using pthread_mutex_unlock.

Testing:

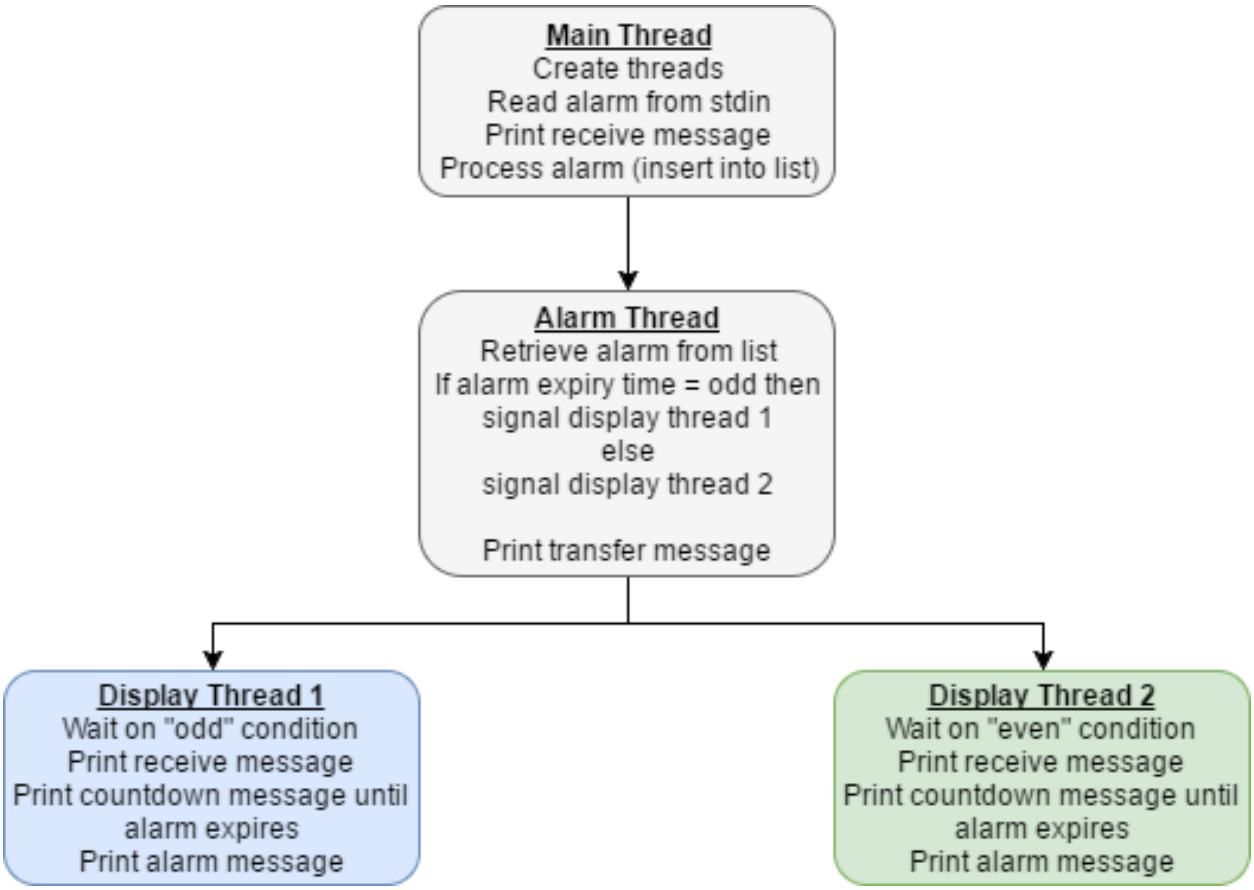
For testing we ran the program with a wide variety of inputs to ensure that the correct output was given. This included cases where the inputted time was very short, and cases where the inputted time was much larger. This was to ensure that the program would not crash after running for a long period of time or not produce the correct result for a short period of time. During this testing both display thread 1 and display thread 2 were showcased, even though their functionality is practically identical to one another.

In addition to regular, expected input, we also included cases where the input did not follow the format required. In these cases, the program behaves as it should, outputting “Bad command” and prompting the user to enter another command.

Also, we tested what happens when multiple inputs are entered into the program while execution is still happening. This is shown in test11.txt, test12.txt and test13.txt. Our program handles the multiple input properly and outputs the appropriate messages.

Design Diagram:

The following diagram represents the main idea behind our implementation of the specified assignment requirements. It gives a visual representation of what each thread is responsible for during execution of the program.



Design Problems:

The original design of the alarm system included a predicate that corresponds to whether an alarm expiry time is odd or even. The purpose of this predicate was so that the alarm thread can signal the appropriate display thread when the mutex is unlocked. This was done to avoid awakening a display thread when the mutex is locked, since the thread would become blocked, waiting on the mutex.

One problem with this approach was the need to properly reset the predicate at the beginning of each loop iteration. Without resetting the predicate, the alarm thread would spam stdout with the message that it has passed the alarm to a display thread. This was easily fixed, by choosing an appropriate default value for the predicate.

The second problem was that signaling a waiting thread when the mutex is unlocked, did not guarantee that a different thread would not lock the mutex prior to the waiting thread receiving the signal. This creates a race condition that is better be avoided.

Considering how this system is simple, these problems were solved by signaling the display thread while the mutex is locked. For larger, more complex systems, a “wait morphing” optimization can be used, that upon sending a signal, moves a thread directly from the condition variable wait queue, to the mutex wait queue, without a context switch.