EECS3221 Assignment 3 Report

Michael Podimov (210816692)
Dimitry Mochkin (213253256)
Ramona Sartipi (213080619)
Matthew MacEachern (213960216)

Table of Contents

1. Design and Implementation of Assignment Requirements

This section will go over the specific requirements detailed in the assignment specification document to show that each of the requirements were fulfilled and justification for the design decisions made to fulfill these requirements.

A3.1 specifies that two types of alarm requests are to be recognized by the application. This is satisfied in the main thread of our program, where in our main loop (that loops continuously until the user decides to exit) we check to see if the user either entered an alarm request of "Type A" form or "Type B" form using sscanf. The appropriate error message is shown to the user if the alarm request entered does not match either of these specific formats. At the end of the while loop in the main thread, it is checked if the flag variable has been set, in which case a valid alarm request has been entered by the user and we continue accordingly. The alarm request that was requested by the user is then entered into the alarm list using the alarm_insert method. This alarm_insert method takes the alarm request that was created in the main thread (either type A or B) and enters it into the alarm list. This method checks to see the type of the alarm and performs error checking accordingly.

A3.2 specifies that the message of type "Type A" should be a maximum of 128 characters, so we store the message of the alarm thread in a character array of size 128 as specified in the structure of the alarm at the beginning of the New_Alarm_Cond.c file. In our alarm_insert method, we checking to see if there is currently an alarm with the same message number as the one about to be inserted into the alarm list. This is for A3.2.1 and is achieved through the searchAlarmA method. This method takes as an argument an alarm and searches through the alarm list to see if there is an alarm with the same message number already in the alarm list. If there is, the method returns 1, otherwise it returns 0. When it returns to the alarm_insert method, checks are performed to see if there is an alarm with that number already in the list. If there is, a replacement message is displayed to the user and replaceAlarmA is called, otherwise it is indicated that the alarm that the user created is the first with that number, satisfying requirement A3.2.1 and A3.2.2. the replaceAlarmA method replaces the alarm that is currently in the alarm list with the specified message number instead of simply inserting it into the list. In alarm_insert, there is also functionality for alarms of type B. The searchAlarmA method is used again to see if there is an alarm to cancel with the specified message number. If there isn't, the error message specified in A3.2.3 is displayed to the user and no further action is taken. Otherwise, a new method searchAlarmB is used which is similar to searchAlarmA. This method is used to fulfil the error checking of A3.2.4 to see if there is more than one cancel request for that specific message number. If there are no errors, the cancel message specified in A3.2.5 is shown to the user and the method loops through the alarm list in order to remove the alarm with the alarm number received.

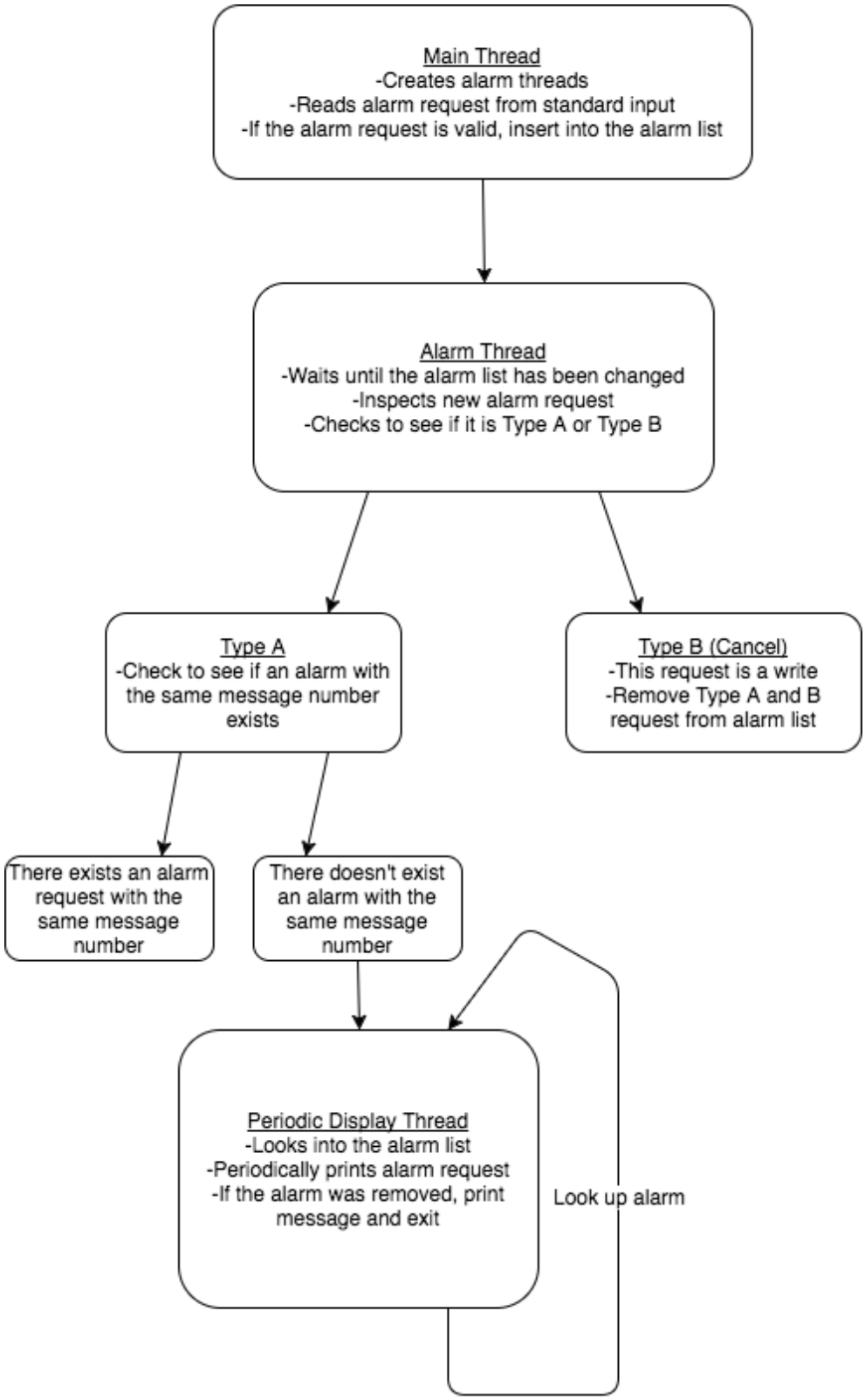
A3.3 specifies the requirements for the alarm thread for the New_Alarm_Cond.c file. This new alarm thread checks to see if the alarm list has changed and checks alarm requests in the alarm list whenever this happens. This thread loops until the user decides to close the program similarly to the main thread of our program. While looping, the alarm is checked to see if it has changed (it is not NULL) and if it is type 1 (which corresponds to type A in our program). A3.3.1 if a type A alarm request is found in the alarm list, the alarm_thread creates a new POSIX thread (specifically a periodic_display_thread) which is indicated by the periodic_display_thread method in our program. For A3.3.2, if the alarm type is 0 (indicating a type B alarm), the alarm and the data corresponding to both the Type A and Type B alarm

request is removed from the alarm list. In both the above-mentioned cases, the message specified in A3.3.3 is displayed to the user.

A3.4 specifies the requirements for the periodic_display_thread thread that we had to create for this application. This thread has the routine periodic_display_thread corresponding to it. The alarm that is passed in as an argument is checked in the main loop of the thread periodically, to see if it has changed at all. In A3.4.1 it is specified that the periodic display thread should look up type A alarm requests and periodically display information pertaining to that alarm request. This is done by first checking to see if the current alarm is modified. This is accomplished simply by having a variable "modified" associated with each alarm in the alarm structure. If modified is set to 0, the alarm has not been modified and the periodic display thread displays the alarm information to the terminal. For A3.4.2, if the modified variable is set to 1, then we know that this alarm has been modified in some way. If it is detected as being replaced for the first time, the message "Alarm With Message Number (Message_Number) Replaced at <time>: <alarm_request>" is displayed once, otherwise the periodic display message for replacement alarms is displayed every number of seconds specified in the alarm. Before all of this, however, the periodic display thread needs to make sure that alarm request is still in the list by checking the linked variable of the alarm structure to satisfy A3.4.3. If the alarm linked variable is set to 1, the alarm is still in the list, if it is 0 then the alarm request has been removed from the list and an appropriate message should be displayed to the user. This means that the thread should terminate, which is accomplished in the if statement "if(alarm->linked == 0)" by returning NULL to the calling routine. The periodic display thread ensures that the message required is displayed only every number of seconds specified in the alarm by sleeping the display thread for that number of seconds.

A3.5 specified that the alarm list access must be synchronized to ensure that threads are locked appropriately by treating threads as "readers" and "writers". This is a requirement for the entire system and must be explained in each thread in our implementation. In the alarm thread, the pthread_mutex_lock is used to set up the thread as a reader before checking the type of the alarm that is being processed. If the alarm type is 1 and therefore a type A alarm request, we only need to read, after which, the reader/writer mutex is unlocked. If the alarm is type B, then the alarm thread is a writer since it is changing the alarm list by attempting to remove an alarm. Therefore, we lock the rw_mutex and continue our processing of the alarm. In the main thread, if our flag variable is set to 1, the input was parsed correctly as a type A or type B alarm request, and we must process the soon to be alarm request as a writer since it will be modifying the alarm list. The pthread_mutex_lock is called to lock the rw_mutex and the alarm_insert routine is called to insert the newly formed alarm request into the alarm list, then it is unlocked to allow more alarm requests to access the alarm list. In the periodic display thread, we only need to read, so this thread is treated as a reader, and the appropriate mutex is locked. The mutex is locked at the beginning of the loop and unlocked at the end of the loop and upon termination, to ensure proper synchronization of accesses to the alarm list.

2. Design Diagram



3. Design Problems

The main problem (more of a new kind of implementation that we weren't familiar with) was treating certain threads as readers and writers depending on what the thread is doing at a certain point in its execution. Previously, we only had one type of mutex we had to worry about locking before entering the critical sections in our program. In this assignment, we had to worry about both semaphores: mutex and rw_mutex, and which one to use in certain critical sections. This was alleviated by using reasoning about what each thread does. For example, the periodic display thread only displays information about each alarm request in the alarm request, so it is a reader thread. The alarm thread, however, can be a reader or a writer, depending on if it is removing an item from the alarm list (cancel command) or not. Finally, the main thread has a section of code that requires us to lock rw_mutex briefly, making it a writer thread.

Other than the readers and writers problem, the program was fairly straightforward, since we are familiar with working with POSIX threads. The alarm request structure in our implementation has variables to check the status of an alarm request for a variety of situations (eg type, new, modified, linked, etc). This ensures that the different threads can easily deduce the state of an alarm, to reduce the likelihood of bugs in the program.

4. Testing

File Name	Description of the goals of this test
t1.txt	In this test, we wanted to showcase some of our program's error checking, as well as the basic functionality of the commands. A 30 second alarm request is created after a Cancel command is entered (which returns an error message). After some time, a replacement alarm request with the same number as the 30 second request is entered, replacing the 30 second request. This alarm is then cancelled. Two more alarm requests are entered afterwards, one with a 10 second time and one with a 2 second time. This is used to showcase our periodic display thread and how it outputs the alarm based on the time entered for that alarm request.
t2.txt	This test is a simple two command test in which we create an alarm request that has a 1 second time, which means it is output every second thanks to the periodic display thread. The cancel command is used at the end to remove the alarm request from the alarm list and thus the periodic display thread exits as well.
t3.txt	This test shows what happens when many messages with the same message number are entered the program. It shows how the program handles replacement alarm requests compared to new alarm requests (see diagram section for details). The replacement alarm request is handled appropriately and the cancel command executes properly in the end and stops the one thread that is running.
t4.txt	This test again shows many replacement alarms with the same message number as each other. The cancel command is then used at the end to terminate the thread.
t5.txt	In this test, a total of ten alarm requests are entered, each with a time of 2 seconds to show that the program can handle many threads running quickly at the same time. The cancel command is used on each of the ten alarm requests and each of them terminates accordingly.
t6.txt	This test shows many alarm requests displaying at the same time, while replacement requests are entered to the program. The cancel command is again used to terminate each of these alarm requests before the program exits.
t7.txt	This test creates eleven unique alarm requests with longer times and lets them run for a period of time before calling the cancel command for each of them. They all terminate properly and display the appropriate message to the user.