EECS3221

ASSIGNMENT 3 REPORT

MEMBERS:
Pertahp - Singh Tuli
213766910

Nabeel Amod
212175600

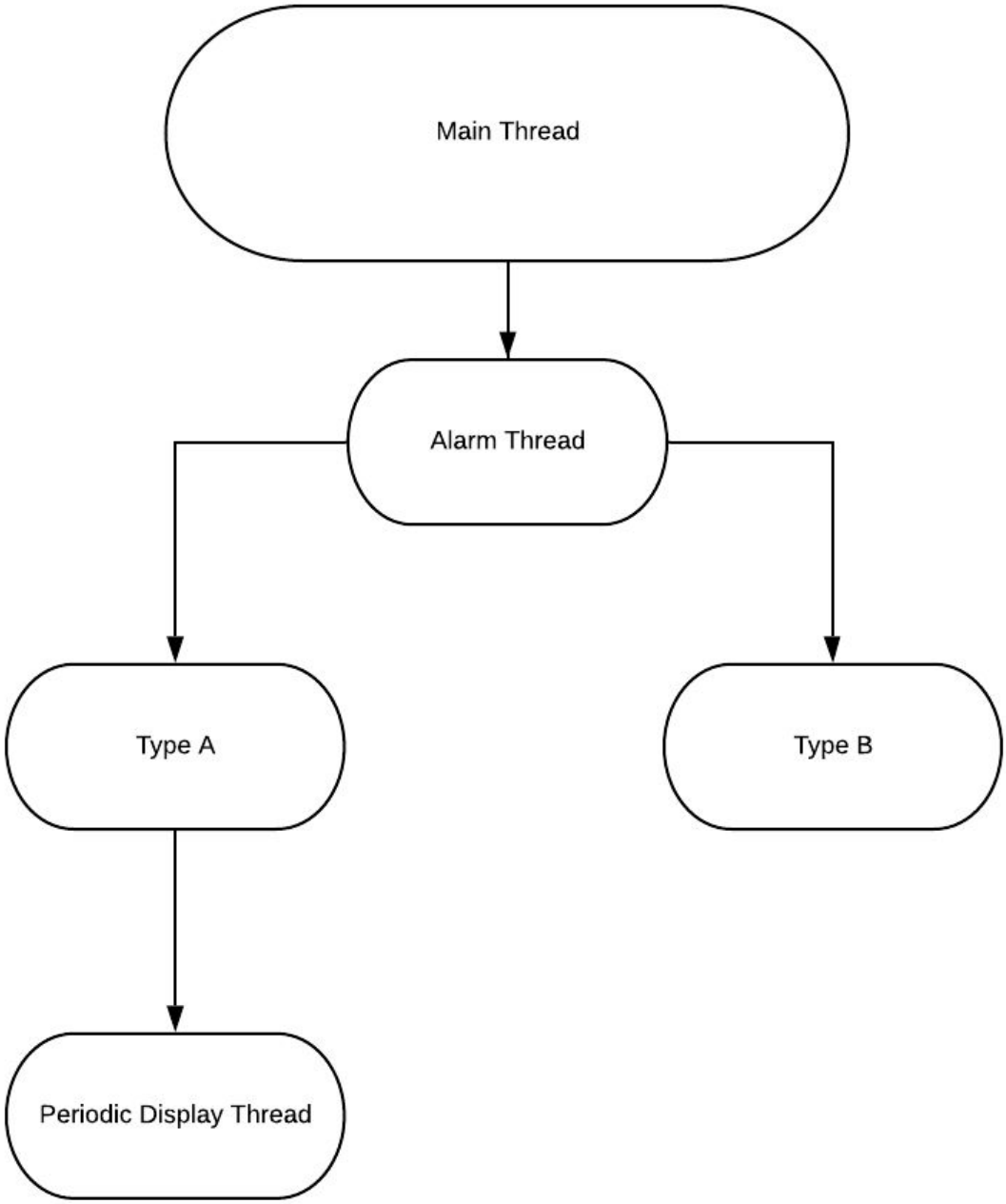
Mark Miguel
214588115

Kristiana Papajani
214466387

Goals and the important design and implementation aspects

The main goal of the assignment is to change or reproduce a better version of the previous program of an alarm that was implemented in the previous assignment. In order to do so, it is required to change a couple of issues. The first tasks would be the two types of Alarm requests that are recognized. As such the alarm requests are required to include as a second parameter, the message number with a specific format: Alarm> Time Message(Message_Number) Message. Time and Message have both the same meaning as in the previous program implemented in assignment 1 while Message_Number will be a positive integer. Another important thing to be added would be the Alarm requests prefixed with the key work Cancel. This will be used in case a user decided to type something rather than the two valid types of alarm requests and if so an error message will be displayed. Moving on, another important task is the main thread. The main thread will be the first to determine if the format of the alarm requests is consistent to the formats given. If the main threat finds out that the format is different then the alarm requests is rejected with an error message. Another important thread to be displayed is also the alarm_thread. This particular thread will check the alarm requests in the alarm lists and if in any case such list has been changed. However if the alarm thread finds out any alarm request with the keyword Cancel then the alarm thread will automatically be able to print an error message and will remove such undesired data.
Moreover, another important goal for us is to create the periodic_display_thread. This one in particular, is responsible for checking the alarm request with a specific message number in the alarm list. Then it is capable to print such message with its information. Lastly, this assignment has an important goal and that is to solve “Readers-Writeers” problem. Any thread that is required to modify the alarm list, is treated as the “writer process” and as such only one particular writer process is capable of modifying the alarm list at a specific time. On the other side, all the threads that only read the information from the alarm list will be treated as the reader process and will be able to have any number of reader processes that are able to read the information at the same time.

Explanation of implementation, Design Decisions and the quality of the design



Periodic_display_thread:

This particular thread has a particular role and that is to be reponsible to periodically look up for any alarm request with a specific message number in the alarm list and then print it. The implementation of the thread starts with holding the status of the mutex locks and unlocks. As such, a particular variable sleeplength will be used to hold the amount of time in seconds that

will be used to pause the loop that we are going to be creating. Furthermore, it is the implementation of the loop. The first step is to acquire the mutex lock and save the status. If the status in that case is not zero, then an error message will be given with the status and a message saying “Lock mutex”. The next step that we follow is iterate through the loop and if by any chance the counter is equal to 1 then a new statement will have to be followed. Such statement requires us to acquire the mutex lock and to save the status and by the end of it, to release the mutex lock and save the status. In any case, if the status will be anything rather than 1, an error message will be displayed giving the status together with a message “Unlock mutex”.The particular sleepLength variable will give us the length of time for sleep from the alarm field in seconds. If by any case the alarm no longer exists then we need to inform the user that it does not exist anymore and we do that by displaying a message as such: printf("DISPLAY THREAD EXITING: Message(%d)\n", alarm->messageNum); After all this, we need to save our status and acquire the mutex lock using this following status = pthread_mutex_lock (&mutex). An important part of the design of the periodic_display_thread is the checks where we check three things. Either if the alarm has been changed, either if the alarm has not been changed or if there has been a change but with an issue in it. Again after such step we save our status and acquire the mutex lock. All of this process continues and the loops waits the specified amount of time before continuing.

Alarm_thread:

Another important part of the design and implementation of the assignment is the alarm thread mentioned previously. The alarm thread itself has a particular role, to check the alarm requests in the alarm list and if in any case the list has been changed. The alarm thread will be disintegrated when the process exits. In the beginning, the mutex should be locked and then it will be unlocked on the condition that it waits so the main thread can meanwhile insert the alarms. During the loop, we acquire the mutex lock and then save the status just like in the previous one and then we iterate through the loop. Differently from the previous thread, this particular one will check some other things. One of them would be if the end of the list has not been reached. Also, it will check if it is of type A and the alarm is not null. Otherwise, it will check if it is type B and that the alarm is not null. If an alarm is found on the list, the alarm has to be removed.

main thread:

Main thread is definitely one of the most important parts of the assignment.
It will be the first to determine if the format of the alarm requests is consistent to the formats given. If the main threat finds out that the format is different then the alarm requests is rejected with an error message. The first step to this, is making sure that the input has been arrived in the

right format using this particular: **if** (sscanf (line, "%d Message(%d) %128[^\n]",
&alarm->seconds,
&alarm->messageNum, alarm->message) < 3)
Since the message has to have a specific output. If in any case, the message is not in the right format, an error message will be printed and there will be updates to the signal. Otherwise, if the format of the input is correct, it will cancel the alarm request by setting the alarm seconds to zero, removing the message and setting the request type to cancel. Meanwhile, the flag will be updated all this time and also we acquire the mutex lock while saving the status.

Thoughts on the quality of design and the problems encountered

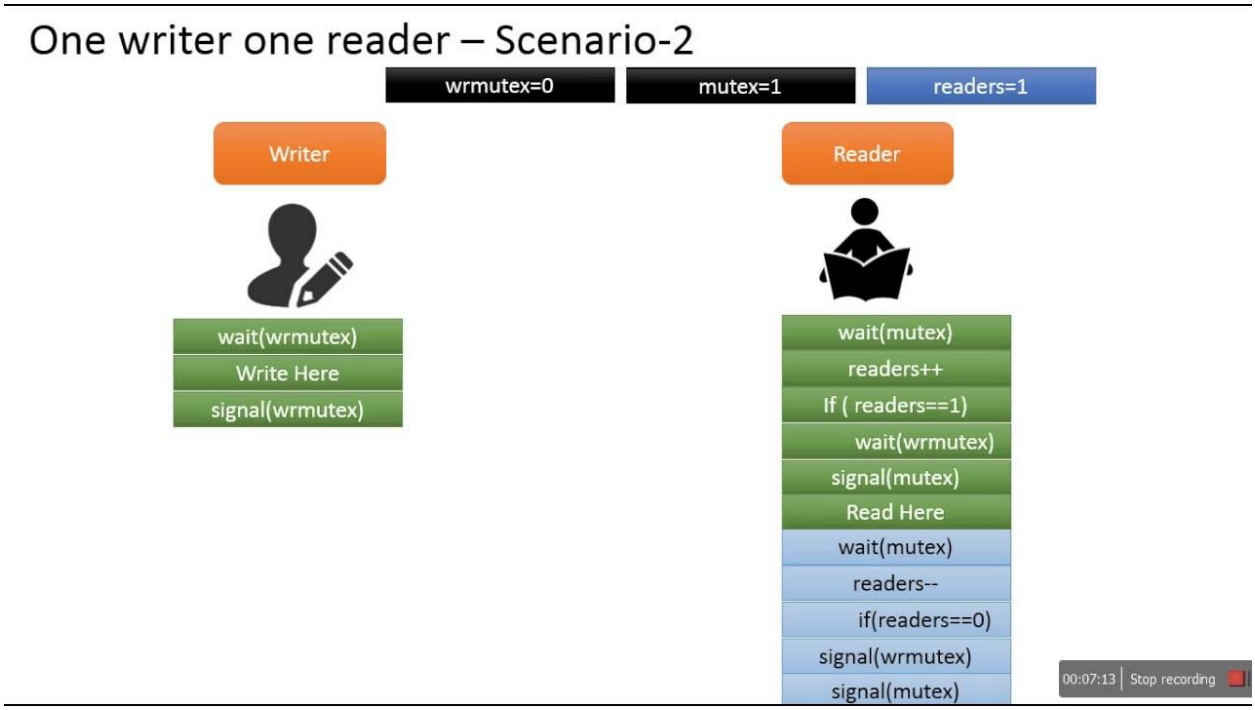
Various problems and difficulties arose during the completion of assignment three. The first difficulty arose when using pointers and properly passing a pointer. In the program we created alarm thread, main thread and also Periodic_display_thread and used the property of a pointer. Without passing a pointer we are passing by value and the function will operate on a copy of the structure pointer, not modifying the original. This issue was also encountered in the previous assignment and hence we found it a bit easier to operate this time.
The second difficulty we came across was using Using sleep(2) instead of wait, to design a loop that loops every 2 seconds. The wait() command blocks the calling process until a child process exits or a signal is received. In this case we are given a block of code that says if an alarm list is empty one second will pass and the main thread will run and read the next command. However, if the alarm is not empty then the number of seconds is computed and will set the sleep time to 0 if the result is less than 0. The resulting while loop has it so that if time is expired it will continue looping if it is not. The loop will print various messages including the thread number, time remaining, alarm request number and the alarm request message and number based on which display thread is printing the message. Using sleep(2) the loop will wait two seconds and then subtract two seconds from the remaining time.
The third issue that came up had to do with the periodic_display_thread and creating the conditions for checking to see if the alarm had been changed, not changed or changed but with no flag. It was difficult to know whether the no flag condition was required but after many attempts it was accomplished.

Testing
To ensure the desired output was correct, our program was tested with different inputs. These input cases range from odd, even, positive, negative, small, and large numbers, improper formats, and extreme cases.
Small and large number inputs resulted in correct and expected results. Particularly after running for long periods of time, inputs with large numbers did not cause our program to crash.

Moreover, inputs with improper formats were also taken into account. In such cases, our program outputted “Error!!! Bad Input” as expected.
We tested our program with many messages with the same message number. Our program handled the replacement alarm request appropriately, displaying “MESSAGE CHANGE: Message(Message_number) Message” and the cancel command executing properly as expected.
We also test our program with various message alarm requests running at the same time. Our program handled the multiple message alarm requests appropriately, displaying each request along with the others. Moreover, upon cancelling individual messages, our program responded appropriately as expected.
Another crucial testing we conducted was a combination of multiple alarm requests displaying simultaneously, while replacing request that were already entered in the program. Our program was able to display multiple alarm requests and responded with the message replacement requests accordingly. Moreover, upon cancelling individual messages, our program cancelled the specific request as expected.
To ensure our program does not generate segmented faults, we made sure we avoided the common causes of it. We made sure the format control string had the same number of conversion specifiers %, and the function printf we used had arguments to be printed along with the specifiers matching the type of variable to be printed. We also made sure our pointer variables were properly initialized and were assigned a valid address that points to a valid memory.

Ending statement

The most important part of this assignment was to understand and learn better about the Readers-Writeers” problem. Also with this comes the learning and understanding of the use of multiple threads and processes. The writer process which allows only one writer process to happen at the specific time the alarm is being modified and then the read process which can happen by many reader processes able to read the information at the same time.



The above photo explains more about the situation.