EECS3221 - ASSIGNMENT 2

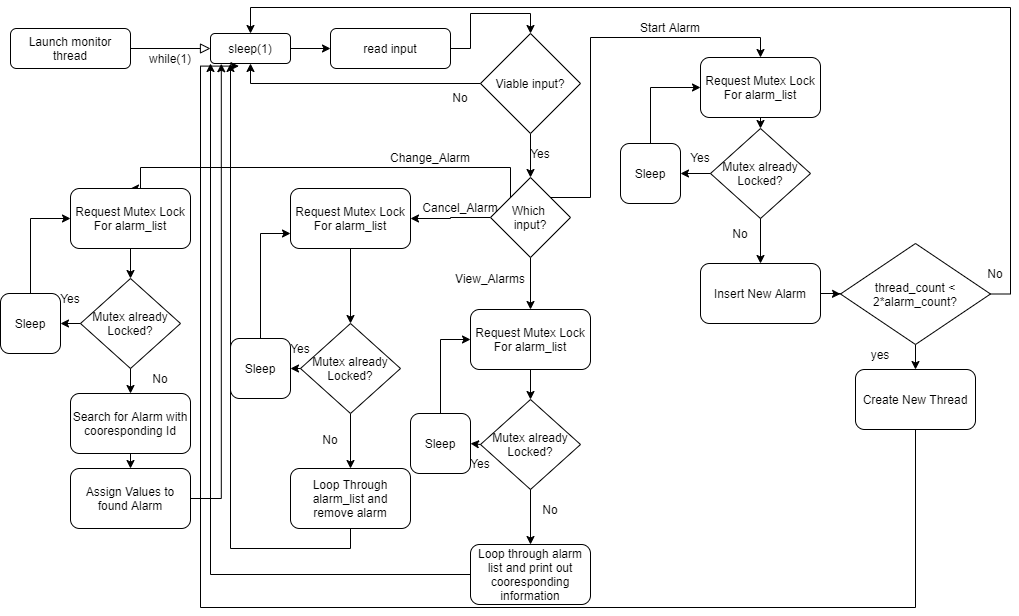
November 16th, 2020

Eli Frungorts - 215659501

Zamir Lalji - 212779997

AmirHossein Razavi - 216715963

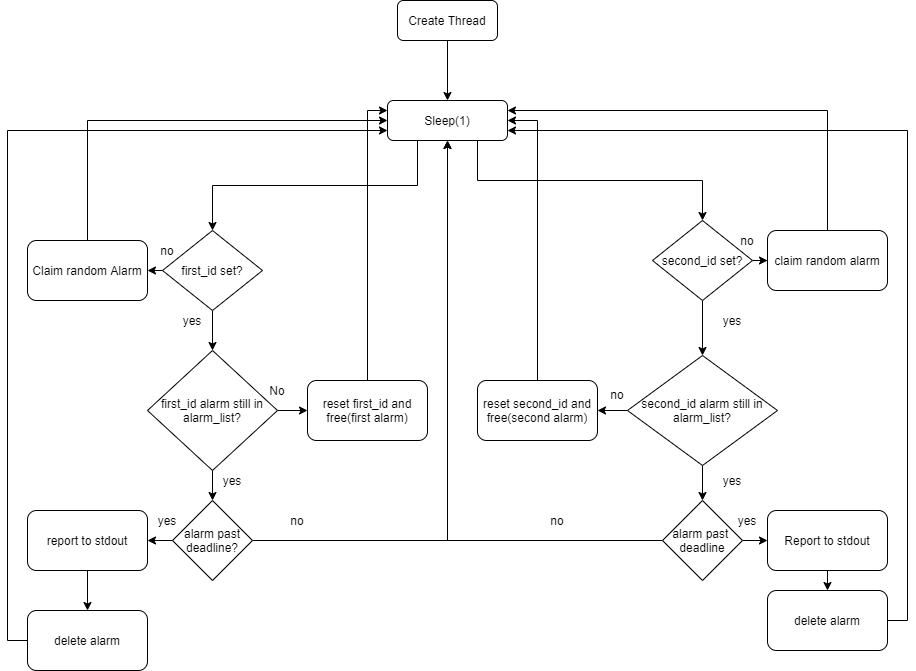
Muhammed mujahid - 213731435

****

**Description:**

In the previous diagram, a decision tree can be seen for the loop of the main method. In the loop 1 of 4 main behaviours will be performed per iteration. These 4 behaviours are as follows:

|  |  |
| --- | --- |
| **Start\_Alarm:**  Given a valid Start Alarm input, the code will perform the following sequence of actions:   1. Mutex lock the **alarm\_list** to ensure no other thread is consuming it 2. Validate that an alarm with the given id does not already exist 3. Insert the new **alarm struct** into the alarm\_list 4. Unlock the mutex for **alarm\_list** 5. Check if there are enough threads such that every alarm is being monitored 6. Create a new thread if needed 7. Sleep for 1 second | **Change\_Alarm:**  Given a valid Change Alarm input, the code will perform the following sequence of events   1. Mutex lock the **alarm\_list** 2. Validate that an alarm with the given id already exists 3. Retrieve the corresponding struct from the **alarm\_list** 4. Update the values with the new ones provided by user 5. Unlock the mutex 6. Sleep for 1 second |
| **Cancel\_Alarm:**  Given a valid Cancel Alarm input, the code will execute the following sequence of actions:   1. Mutex lock the **alarm\_list** 2. Delete alarm with given id 3. Unlock mutex 4. Sleep 1   This might seem incomplete, but the alarm\_thread loop will check if the alarm exists in the list every iteration. If it does not exist, it will remove it from memory. | **View\_Alarms:**  Given a valid View alarms input, the code will perform the following sequence of events   1. Mutex lock the **alarm\_list** 2. Iterate through the alarm\_list, building up the output string 3. Unlock the mutex 4. Sleep 1 |



**Alarm Thread Diagram:**

Since the behaviour is identical for the first and second alarm in each thread, the following will only explain how one of them works.

The following is a step-by-step description of how the alarm thread works:

**Each alarm\_list access call is mutex safe. Mutex logic has been emitted for clarity**.

1. sleep(1)
   1. This is so that the mutex is unlocked for the majority of the time, allowing the main thread to update the alarm\_list
2. Check if first/second has an ID assigned to it.
   1. If the thread does not have an alarm assigned to it, it will go through the alarm list from lowest to highest id, and claim the first unclaimed alarm
   2. If the alarm has an Id, it will go to step 3
3. Check if the claimed alarm is still in the alarm list
   1. This is done so that alarm deletion can be done **asynchronously**. When the main thread receives a cancel alarm request, it will just delete it from the alarm\_list. This is done so that the alarm\_thread and main\_thread do not have to explicitly communicate between themselves. If the claimed alarm is not in the list, then the thread will reset it alarm reference, and will claim a new alarm next iteration
   2. If the claim alarm is still in the list, then it will go to step 4
4. The thread will check if it needs trigger the alarm
   1. If the alarm is passed its due date, it will output to STDOUT that an alarm has been triggered, after which it will expunge the alarm from memory. After which it will return back to step 1
   2. If the alarm is not passed its due date, it will instead just go to step 1.

**KEY DIFFERENCES FROM ORIGINAL CODE AND NEW CODE AND OTHER COMMENTS**

* In the original code only one linked list is used for the alarm list, in this implementation, 2 linked lists are used, one for alarms and another for threads
* In the original code, only one mutex was used to lock and unlock the alarm\_list. In the new code, 2 mutexes are used, one for the alarm list, and another for the thread list. Although counter intuitive, each thread should independently lock the thread\_list when it plans on self-destructing.
* This code allows more user inputs and alarm manipulation. Instead of simply creating alarms, users are able to remove and modify the alarms
* One of the main issues with this code is there are far too many outputs, when attempting to input a value, it will often get split/overridden by debug lines. To overcome this, many print statements have been surrounded by #ifdef debug, this will make the program much more user friendly. If the outputs (as specifically described by the requirements document) are overwhelming, comment out the “#define DEBUG” line.
* The code works mostly as described in the original requirements manifest. A couple minor feature have not been included, these are:
  + Having the thread explicitly state that it has identified a changed alarm