EECS 3221 Assignment 3 Report

Working with POSIX Threads

Francis Okoyo	213811336
Tyler Nobel	212270369
Adham El-Shafie	212951018
Lindan Thillanayagam	213742176

Table of Contents

Introduction

This program served the purpose of learning about the use of POSIX threads. By implementing an alarm system, we were able to cover a use case for POSIX threads that taught us about multithreading in C as well as proper process handling. This was done by having us change the implementation of the New_Alarm_Cond.c file. Our goal was to have it handle multiple alarm types as well as implement a main method that would check to see if alarm inputs were valid. We would also create and handle alarm threads that worked on the alarms themselves. This document will further discuss the implementation and thoughts that went behind it.

Approach to Requirements

A3.1 and A3.2

The main method is responsible for this portion of the requirements. All inputs are parsed through here. If the user input does not meet the specifications in the readme file, it prints out “Bad command”. Because the main method inserts alarms into the list, we treated it as a “writer process” this meant that we had to implement an efficient way for it to write to the alarm list without causing catastrophic problems for “readers”. Hence the use of semaphores.

A3.3

The alarm thread is a “writer” process like the main thread but is also doubles as a “reader” process. When the thread loops through the alarm list in search of Type A or B alarm requests, this is treaded as a “reader process” because it does not alter the configuration of the alarm list. However, when the thread sees a Type C request, it requires control of the read write semaphore and semaphore variables to take and keep control of the alarm list until it is finished altering it.

A3.4

The periodic display thread is solely a “reader”. This makes semaphore implementation easy because we only need to worry about when a writer is ready to insert or delete from the list. Because semaphores don't have timed condition waits like pthreads do, we implemented the delay function by adding the time given to the alarm to the current system time. When the system time exceeded this time, the message for that alarm is printed. A3.2 - 3.4 use combinations of while loops, variables and semaphores to allow fluid access of the shared data between readers and writers. It is important to note that the writers **must** wait for the readers to finish reading before they can write. This is accomplished by using a global variable to notify readers that a writer is ready to do something. This means that reads cannot be interrupted in the middle of theirs process.

A3.5

A3.5.1

The only threads that needed to write to the alarm list were the alarm_thread and main thread. To ensure mutual exclusion between these two writers, a semaphore is used to control access to the list. Only one of them can use the list at a time. They also use variables that act like semaphores to let the writer know if readers are using the list. This prevents reader starvation.

A3.5.2

Because there can be multiple periodic_display_threads, using a semaphore to protect its critical section would hinder their runtime because unnamed semaphores don't have condition waits like pthreads do. To get around this, we used busy waiting and a read_count variable that acts like a semaphore. This variable tells the writers if there are readers (as well as how many are) currently using the data structure. When a writer is ready to write to the list, it notifies all the readers. They then wait until the writer has written into the alarm list before they continue.

Design

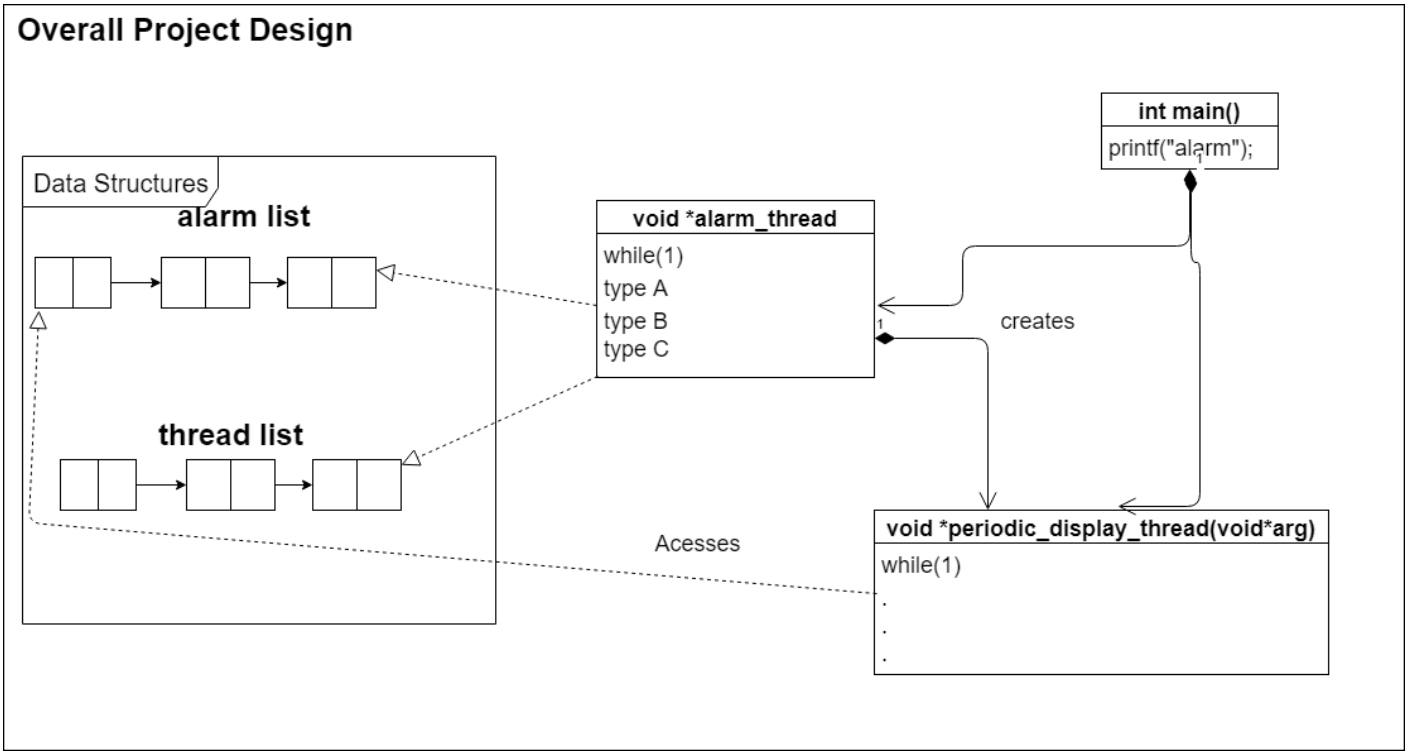

Figure 1. Overall system design.

Design Challenges

- There were many small bugs that weren't conceptually hard but proved challenging to debug given the development environment and amount of code developed.
- Strategic mutex and semaphore placement was crucial to allowing fluid data access by threads
- Waiting methods needed to be modified from assignment 2, sadly they would not work well in this assignment due to the varied requirements
- There was an issue with making Type B alarms able to print without summing up the time till the next print call. This would have led to print calls in increasing increments that would become unusable if too many alarms of the same type assigned.

Design Decisions

Our design decisions include many of the challenges we faced while writing this program. The following are the implementations we used to solve them. Starting with thread handling, our group realized that deleting threads would pose a challenge if we had no way of storing thread ID's. When going over potential data types to store them we realized that a linked list would be best suited for both performance and management reasons. However, since C does not have built-in support for linked lists we had to create one ourselves.

The last large decision we had to make was with dealing with thread termination. Termination a thread has the potential to perma-lock the system as a mutex can be locked with no way of unlocking it. This was solved by using deferred cancellation types. The two cancellation types, asynchronous and deferred were what we had planned to use to handle thread deletion. The asynchronous, however, was vulnerable to the perma-lock effect mentioned earlier. With deferred cancellation, we can say when the thread can be cancelled. This allows us to only cancel when the thread has unlocked the mutex.

In the previous assignment, we used a sparse matrix to store thread ids. Although the thread ids could be accessed in constant time, it proved to be extremely inefficient because it allocated 1 million spots in memory. Realistically, less than 1% of that space would be used. In this assignment, we opted for a linked list which is more efficient

Implementation

This program uses a plethora of helper methods to help encapsulate operations that would be used repeatedly or are just too much to read and need abstraction from the important areas of the code. Prime examples would be insertion and deletion of alarms as well as checking the contents of the alarm list.

Helper methods

display_lists

The method, mainly used for debugging purposes, prints out all the contents of the thread list and the contents of the alarm list to allow the user / programmer see the contents of the lists. This method can still be accessed by typing “debug” when the program requests for an alarm. This input will toggle “DEBUG MODE” which calls display_lists and prints the values of the semaphores used to control access to the alarm list.

insert_thread

Insert a new thread by iterating through each thread in the thread list either until the current thread’s succeeding thread is null, or until the current thread’s succeeding thread is strictly less than the type of thread to be inserted.

terminate_thread

Search through the thread list and remove and cancel any Type B thread of a specified type. It is important to note that when a thread is terminated, it is allowed to complete its current iteration. This is to ensure that the thread isn’t canceled in its critical section. This may result in a message being printed one extra time.

check_useless_thread

Iterate through the thread list and remove threads that are deemed to be useless. A thread is deemed useless if there is no type A alarm of its message type. Return 1 after removal if such case occurs, 0 otherwise.

check_prev

Check if a Type A alarm request’s type has changed. Returns 1 if true, 0 otherwise. This method can be considered a “reader”.

check_type_a_exists

Check the alarm list to see if a type A alarm of a specified type number exists. Return 1 if it does exist, 0 otherwise. This method can be considered a “reader”.

check_number_a_exists

Check the alarm list to see if a type A alarm of a specified message number exists by iterating through the alarm list. Return 1 if it does exist, 0 otherwise. This method can be considered a “reader”.

check_dup

Check the alarm list to see if an alarm of a specified message **type** already exists by iterating through the alarm list. The method takes a message type and a request type as inputs. Return 1 if it exists, 0 otherwise. This method can be considered a “reader”.

check_dup_2

Check the alarm list to see if an alarm of a specified message **number** already exists. The method takes a message number and an alarm request type (A, B or C) as inputs. Return 1 if it exists, 0 otherwise. This method can be considered a “reader”.

remove_alarm

Removes an alarm of the specified message number from the alarm list. The method requires that the caller has locked the alarm mutex. Returns the message type of the alarm that was just removed from alarm list. If the list is empty, then return 0. This method is only called by threads that write to the alarm list.

remove_alarm_B

Removes a type B alarm request responsible for type A alarms with a specified type by iterating through the alarm list. If the type B alarm request is found, then remove it, otherwise there is nothing to be removed. This method is only called by threads that write to the alarm list.

remove_alarm_C

Removes a type C alarm request responsible for cancelling alarms with the specified alarm number type. If a type C alarm of the specified alarm number type is found, then remove it. If not, then there is nothing to be removed. This method is only called by threads that write to the alarm list.

alarm_insert

Inserts alarms into the alarm list. Can only be called by writer threads which implies mutual exclusion.

Essential methods

periodic_display_thread

A type B created alarm (thread) responsible for periodically looking up a type A alarm request with a message type in the alarm list, then printing every Time seconds. If the alarm list is empty, that would imply that the thread is useless. So it waits to be terminated. Uses busy waiting to prevent itself from reading when a writer has control of the alarm list. Disabled and enables cancellation around its critical section to prevent it from being cancelled when it changes, is changing or changed the semaphore.

alarm_thread

An initial thread which is responsible for looping through the alarm list and checking the status of each type A alarm, as well as performing type B or C requests as they are inserted. Uses semaphores to control access to the alarm list. This method is one of the two writers that write to the alarm list. The second being the main thread as it is responsible for inserting alarms.