Assignment 2 Report

Course: EECS3221 Section E
Term: Fall 2024
Assignment: Assignment 2 Report
Group Members:

Mohammad Zarif
Rami Omer
Shaylin Ziaei
Rayhaan Yaser Mohammed

1. Introduction

1.1 Overview

This project involves designing and implementing a multi-threaded alarm management system as part of the EECS3221 course assignment on POSIX Threads. The system is built to manage multiple alarms simultaneously, allowing users to create, modify, cancel, and view active alarms in real time. By leveraging POSIX threading, the program enables concurrent processing of alarm requests, where each alarm can be handled independently without interfering with others.

The program operates with three types of threads—Main Thread, Alarm Thread, and Display Threads—that work together to manage alarms. The Main Thread processes user commands, the Alarm Thread assigns alarms to Display Threads, and each Display Thread is responsible for periodically displaying messages for assigned alarms. The result is a system that efficiently handles multiple alarms, with each alarm displaying its message until it expires or is canceled, providing a robust, thread-safe alarm management solution.

1.2 Requirements

Requirement	Description
Multi-threaded architecture	Implement three thread types: 1. Main Thread 2. Alarm Thread 3. Display Threads
Alarm requests management	Updates an existing alarm's type, seconds, and message fields.
Concurrency control and synchronization	Use mutexes and condition variables to manage concurrent access to shared resources, like alarm_list.
Error handling and input validation	Implement error handling for invalid input and unexpected errors, including thread synchronization failures.
Display thread assignment and reassignment	Assign alarms to display threads based on alarm type; reassign if the alarm type changes.
Memory management	Ensure proper allocation and deallocation of memory for alarms to prevent memory leaks.
Testability and platform compatibility	Ensure the program compiles and runs on the designated platform (Red server at York) with POSIX compliance.

1.3 References and Glossary

https://eclass.yorku.ca/pluginfile.php/6497497/mod_resource/content/28/3221E_F24_a2.pdf

2. Architecture

2.1 Architecture Overview

The architecture of this alarm management system, as shown in the use case and thread structure diagrams, is designed to handle multiple alarms concurrently in real time through three main threads: the Main Thread, Alarm Thread, and Display Thread.

In the use case diagram, the user interacts with the system by performing actions such as starting, changing, canceling, and viewing alarms. Each of these actions initiates different functions within the alarm system, supported by logging, checking alarm status, and error handling to ensure the system operates smoothly. These supporting functions ensure that each user action is tracked and processed accurately.

The thread structure diagram further explains how these actions are managed within the code. The Main Thread handles user commands, updating the shared alarm list when alarms are added, modified, canceled, or viewed. When the alarm list is updated, the Alarm Thread steps in to either assign alarms to existing Display Threads or create new ones as needed, distributing alarms for efficient concurrent display.

Each Display Thread independently manages the display of its assigned alarms, printing messages periodically and handling alarm expirations, cancellations, or updates. When a Display Thread has no alarms left to display, it automatically terminates to free up system resources.

Together, these diagrams illustrate a structured approach where each user action is handled in a controlled, synchronized manner, allowing for efficient and concurrent alarm processing. The system uses mutexes and condition variables to ensure that threads interact with the shared alarm list without conflicts, providing a real-time, reliable alarm display experience.

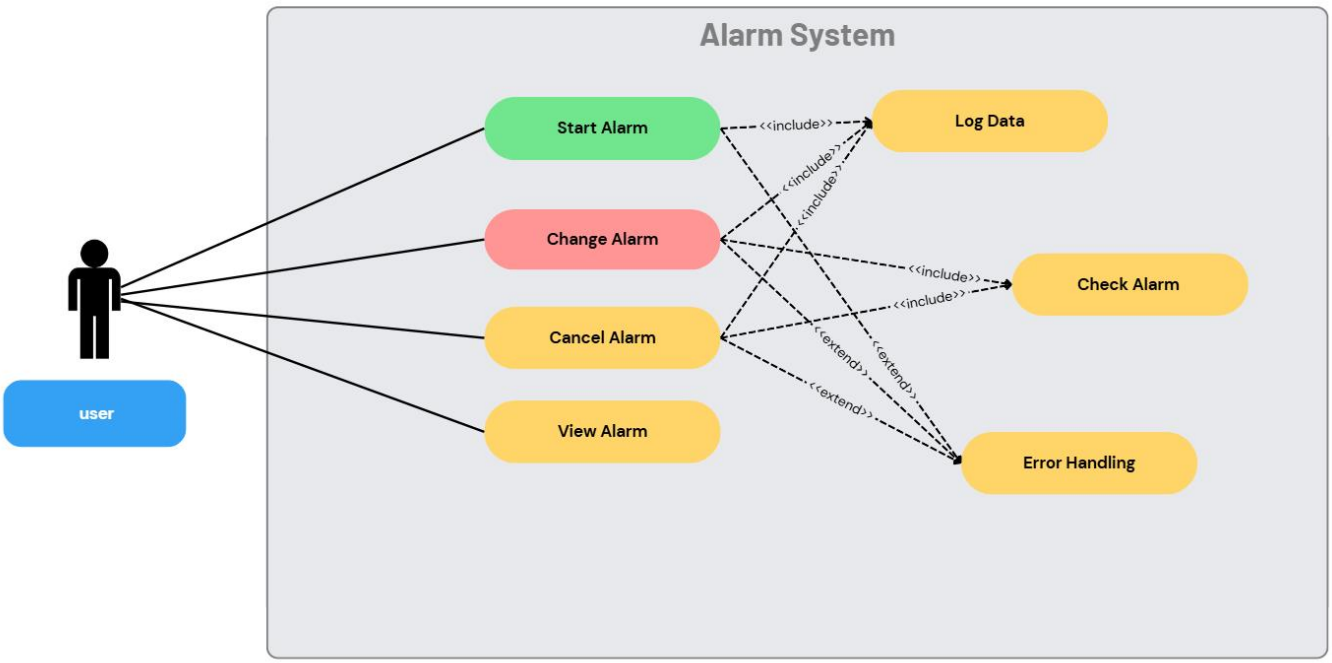

Figure1: Use Case Diagram

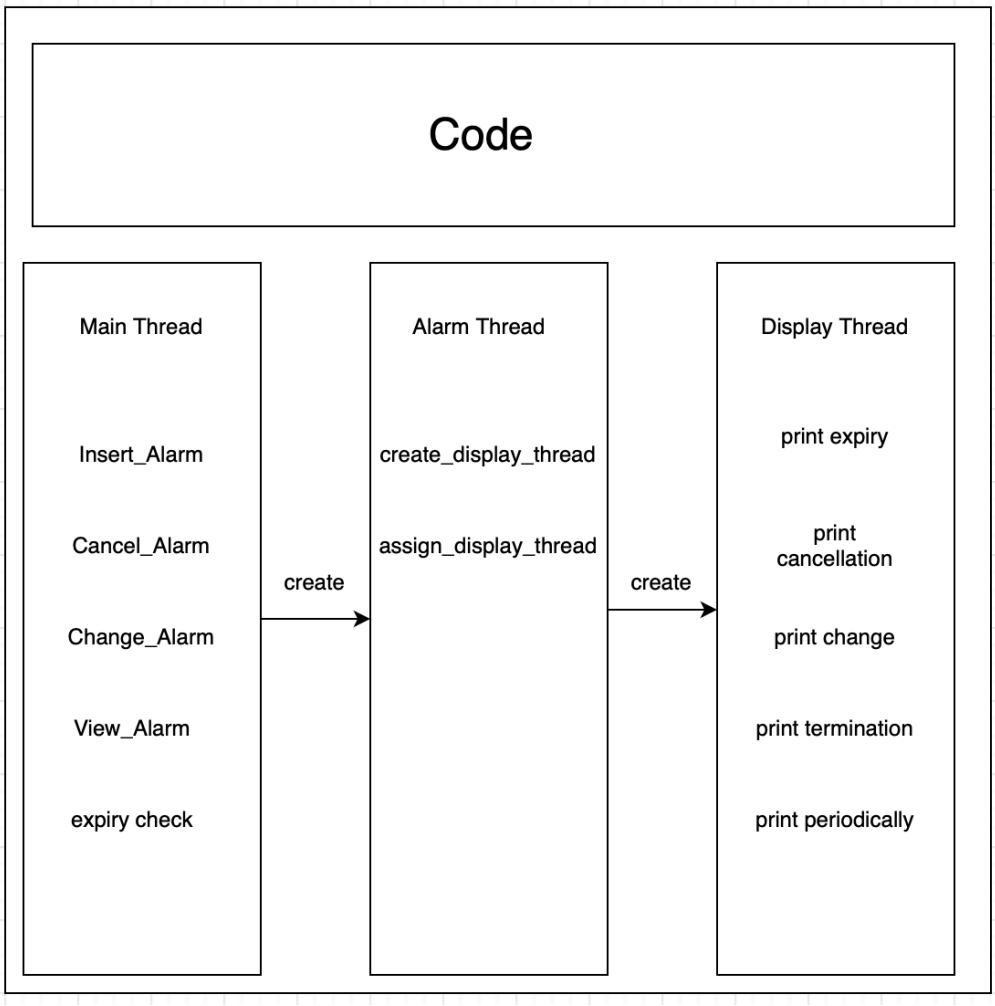

Figure2: Thread Structure Diagram

2.2 Component Definitions

Data Structure		
Field	Variable	Description
Alarm Structure (alarm_t)	alarm_id	A unique identifier for each alarm.
	type	A type string (e.g., "T001") used to categorize alarms.
	seconds	The duration in seconds until the alarm triggers.
	message	A custom message that will be displayed by the Display Thread.
	insert_time	The time the alarm was added, stored in UNIX epoch time.
Flags	is_type_changed	Indicates if the alarm's type has changed.
	is_canceled	Indicates if the alarm has been canceled.
	is_expired	Indicates if the alarm has expired.
	display_thread_id	Stores the ID of the Display Thread responsible for displaying the alarm.
	link	A pointer to the next alarm in the linked list, allowing alarm_list to be organized as a linked list.
Global Variables	alarm_list	A pointer to the head of the linked list containing active alarms.
	alarm_mutex	A mutex used to control access to alarm_list, ensuring thread-safe operations when alarms are added or modified.

Function	
Name	Purpose
insert_alarm	Adds a new alarm to the alarm_list in sorted order by alarm_id
change_alarm	Updates an existing alarm's type, seconds, and message fields.
cancel_alarm	Removes an alarm from the alarm_list by alarm_id.
view_alarms	Displays all active alarms along with their assigned Display Threads.
main	The entry point for the program that initializes threads and processes user commands.
alarm_thread_func	The function executed by the Alarm Thread, responsible for monitoring and assigning alarms to display threads.
display_thread_func	Handles displaying alarms of a specific type, periodically printing assigned alarm messages.

Thread		
Name	Role	Responsibilities
Main Thread	Manages user input and coordinates alarm operations.	- Processes commands: Start_Alarm, Change_Alarm, Cancel_Alarm, and View_Alarms. - Modifies alarm_list and creates the Alarm Thread.
Alarm Thread	Monitors alarm_list and manages Display Thread assignments.	- Listens for signals when alarms are added or modified. - Ensures each alarm type has a Display Thread. - Distributes alarms to Display Threads with a max of two alarms per thread.
Display Threads	Displays alarms of a specific type, periodically printing assigned alarm messages.	- Periodically prints message for each assigned alarm. - Monitors alarms for expiration, cancellation, or type changes. - Terminates if no alarms are left to display.

2.3 Data Flow and Synchronization

2.3.1. Data Flow

Data in the program primarily flows through the `alarm_list`, a central structure that holds active alarms and their attributes. Multiple threads access and modify this list as they process alarms based on user commands.

- **Main Thread to Alarm List:**
  The Main Thread directly interacts with `alarm_list` to add, modify, and cancel alarms. Each user command (e.g., starting, changing, or canceling an alarm) triggers a specific function—`insert_alarm`, `change_alarm`, or `cancel_alarm`—that modifies `alarm_list`. When a new alarm is created, a structure is initialized with attributes like `alarm_id`, `type`, `seconds`, and `message`, and then inserted into the list in a sorted order. Similarly, updates and cancellations modify the relevant fields in the list.
- **Alarm Thread to Alarm List:**
  The Alarm Thread continuously monitors `alarm_list` to assign or reassign alarms to Display Threads. If it detects an unassigned alarm or one that has changed type, it assigns it to a suitable Display Thread by updating the `display_thread_id` within the alarm structure. This field keeps track of which Display Thread is responsible for displaying each alarm message.
- **Display Threads to Alarm List:**
  Each Display Thread periodically checks `alarm_list` for alarms it is responsible for displaying. It reads the alarm's data (such as `type`, `seconds`, and `message`) to print the message at regular intervals. When an alarm expires, the Display Thread marks it as expired by updating the `is_expired` field and releases its `display_thread_id`, indicating that the alarm is no longer managed by a thread. This allows the Main Thread to remove expired alarms safely.

2.3.2 Synchronization

Synchronization is essential in this program because multiple threads need to read and modify the shared `alarm_list` at the same time. The program uses mutexes and condition variables to coordinate actions across threads, ensuring they access shared resources in a controlled and consistent way.

The mutex `alarm_mutex` controls access to `alarm_list`. Before any thread—whether the Main Thread, Alarm Thread, or a Display Thread—can access or change `alarm_list`, it must lock `alarm_mutex`. This lock makes sure that only one thread can work on the list at a time, preventing conflicting changes. When the thread finishes its task, it unlocks `alarm_mutex`, allowing other threads to access the list. This locking and unlocking process keeps `alarm_list` stable and prevents data issues.

The program also uses condition variables to help threads communicate when certain events happen. The `alarm_cond` condition variable lets the Alarm Thread wait until there's a relevant change in `alarm_list`, like a new alarm, a type change, or a cancellation. When such changes occur, the Main Thread signals `alarm_cond`, waking up the Alarm Thread to handle any necessary updates. This way, the Alarm Thread only activates when it's needed, which helps save CPU resources.

Another condition variable, `display_cond`, is used by Display Threads to stay in sync with the Main Thread, especially when alarms are canceled or types change. When a Display Thread detects a cancellation or type change, it signals `display_cond`, allowing the Main Thread to safely remove or update the alarm in `alarm_list`. This keeps the Display Threads and the Main Thread working together smoothly without conflicts.

These synchronization techniques allow threads to interact with `alarm_list` safely, supporting reliable data flow and consistent alarm management across the system.

3. Design and Implementation

3.1 Thread Design

The program's thread design is built around three main threads: the **Main Thread**, the **Alarm Thread**, and multiple **Display Threads**. Each thread has a specific role in managing alarms, displaying messages, and ensuring safe access to shared resources. To coordinate their actions and prevent conflicts, the program uses **mutexes** and **condition variables** for synchronization.

1. Main Thread

The Main Thread acts as the control center, handling user commands to add, modify, cancel, and view alarms. It interacts with the Alarm Thread and Display Threads as follows:

- **Creating the Alarm Thread**: At program startup, the Main Thread creates a single Alarm Thread, which remains active throughout the program's execution.
- **Processing User Commands**:
    - For Start_Alarm commands, the Main Thread calls insert_alarm to add a new alarm to the list and signals the Alarm Thread to process the addition.
    - For Change_Alarm commands, it modifies an existing alarm and signals the Alarm Thread to reassign it to the appropriate Display Thread if needed.
    - For Cancel_Alarm commands, it flags an alarm as cancelled and waits for confirmation from the Display Thread assigned to that alarm.
- **Periodic Expiration Check**: The Main Thread periodically checks for expired alarms and removes them from the list as needed, signalling the Alarm Thread if there are changes.

2. Alarm Thread

The Alarm Thread manages the alarm_list and ensures that each alarm is assigned to a Display Thread for handling. Its primary tasks include creating and managing Display Threads as needed and reassigning alarms when changes occur.

- **Monitoring Changes**: The Alarm Thread waits on the alarm_cond condition variable and remains idle until signalled by the Main Thread. Signals from the Main Thread indicate that an alarm has been added, modified, or cancelled.

- **Reassigning Alarms**: When activated, the Alarm Thread scans the alarm_list to identify any alarms that need assignment or reassignment. If an alarm's type has changed or it is unassigned, the Alarm Thread checks for an available Display Thread managing the same alarm type and can handle up to two alarms. If no such thread is available, it creates a new Display Thread.
- **Creating New Display Threads**: When a new Display Thread is required, the Alarm Thread initializes it using pthread_create(). Each new Display Thread is tasked with displaying alarms of a specific type, printing messages periodically until the alarms expire, are cancelled, or change type.

3. Display Threads

Display Threads are dedicated to handling alarms of specific types and are dynamically created based on active alarms and their types.

- **Alarm Type Assignment**: Each Display Thread monitors alarms of a particular type and periodically outputs their messages to the console.
- **Displaying Messages**: Every five seconds, each Display Thread checks if it has active alarms. If an alarm is still valid (not expired or cancelled), it displays the alarm's message.
- **Handling Expirations, Cancellations, and Type Changes**:
    - **Expiration**: When an alarm expires, the Display Thread marks it as inactive by setting its display_thread_id to zero, indicating it is no longer assigned to any thread.
    - **Cancellation**: Upon cancellation, the Display Thread sets display_thread_id to zero and signals the Main Thread to safely remove the alarm from the list.
    - **Type Change**: If an alarm's type changes, the Display Thread marks it as unassigned, signalling the Alarm Thread to reassign it.
- **Thread Termination**: A Display Thread terminates when it has no active alarms left. Before exiting, it releases any held locks and logs its termination, freeing up resources.

4. Synchronisation Mechanisms

The program uses mutexes and condition variables to coordinate threads and ensure safe access to the alarm_list:

- **alarm_mutex**: This mutex protects the shared alarm_list from concurrent access. Only one thread can lock and modify the list at a time, preventing data inconsistencies.
- **alarm_cond**: This condition variable signals the Alarm Thread whenever a new alarm is added, an alarm's type is changed, or an alarm is cancelled. It allows the Alarm Thread to activate only when needed, improving efficiency.

3.2 Error Handling and Resource Management

This program employs several strategies to manage errors and resources efficiently, ensuring stability, preventing memory leaks, and handling unexpected issues smoothly. Key techniques include custom error-handling functions, careful memory management, and effective use of threads and synchronization tools.

3.2.1 Error Handling

Error handling in this program focuses on thread operations, memory management, and input validation:

- **Thread Operations:** For each thread created with `pthread_create()` and each mutex or condition variable operation, the program checks for errors. If an error occurs, the custom function `err_abort` is triggered, printing an error message and stopping the program to prevent unsafe operation.

- **Memory Allocation:** When memory is allocated for new alarms, the program checks if `malloc` was successful. If `malloc` fails, `errno_abort` is called to report the issue and exit, avoiding problems caused by null pointers.

- **Input Validation:** The program carefully checks user input for each command (e.g., `Start_Alarm`, `Change_Alarm`). Invalid commands trigger an error message and are ignored, keeping the system stable.

3.2.2 Resource Management

Efficient resource management is essential due to the dynamic creation and termination of threads and frequent memory operations.

- **Memory Management for Alarms:** Memory for each alarm is allocated when added and freed when the alarm expires or is cancelled, preventing memory leaks.
- **Thread Management:**
    - Display Threads: Created based on active alarms, each Display Thread automatically terminates when it has no alarms left, releasing resources.
    - Alarm Thread: Runs throughout the program's lifetime, managing alarms continuously and only terminating when the program ends, reducing the overhead of frequent thread creation.
- **Synchronisation:**
    - Mutexes: `alarm_mutex` ensures that only one thread can access `alarm_list` at a time, preventing data conflicts.
    - Condition Variables: `alarm_cond` signals the Alarm Thread when alarms are added, modified, or cancelled, allowing it to respond only when needed.

These strategies help the program remain stable, efficient, and responsive, ensuring safe multithreaded operations and effective resource use.

3.3 Challenges

Developing the alarm management system involved several key challenges, especially around managing concurrent tasks and reassigning modified alarms.

The first challenge was having the Main Thread handle both user inputs and periodic checks for expired alarms in the `alarm_list`. The Main Thread needed to process user commands promptly while also removing expired alarms to keep the system up to date. To solve this, we used the `select` method, which allowed the Main Thread to monitor multiple sources of input at once. This approach enabled the thread to handle both tasks efficiently without delays or added complexity.

Another challenge was reassigning alarms that had their types changed to the appropriate Display Threads. When an alarm's type is modified, it may no longer belong in its original Display Thread, requiring it to be reassigned to a new thread that handles the updated type. To address this, we introduced an `is_type_changed` flag within each alarm's structure. This flag helped us track when an alarm's type was modified, prompting the Alarm Thread to reassign it to the correct Display Thread based on its new type. This solution ensured that each alarm was consistently displayed in the right place without disrupting other alarms.

These solutions allowed us to handle both concurrent tasks and dynamic alarm updates effectively, resulting in a reliable and responsive alarm management system.

4. Evaluation

For the evaluation of the alarm system and code, we conducted tests through a series of test that each designed to verify different functionalities of the system and validate the responses by the code under various conditions. The results of each test were focused on input handling, logic, and output processing. Also with proper thread management, alarm expiration and message consistency. The evaluation can be divided into part below:

1. **Basic Functionality Testing:** Initial tests were more focused on basic alarm operation, creation and expiration. Test 1 and Test 2 were built to evaluate whether the alarm was correctly created, printed timely on the display and removed upon expiration.

2. **Handling Multiple Alarms and Types:** Test3 evaluated multiple alarms of the same type and examine the systems' ability to manage multiple alarms, It verifies whether the alarms exits without any conflict and that the messages are displayed correctly all according to their own alarms, this also test the systems ability to manage concurrent threads that exist at the same time. Additionally, it also explored the ability to store alarms of different types and checked its ability to distinguish alarms by its unique identifiers.

3. **Edge Cases and Error Handling:** Edges cases are very important for the evaluation of this system as it determines the capacity of the system. For this we tested overlapping of alarms, alarms scheduled with large intervals and alarms that were created and cancelled at the same time. These tests highlight the system's ability to respond to command sequences, ensuring all alarms are updated and terminated precisely based on the latest instructions. It also tests

how the system responds to commands which are invalid or like cancellation of nonexistent alarms which result in the according error messages; this is to evaluate the systems to communicate issues without messing up ongoing processes and commands.

4. **Dynamic Property changes to alarms:** Test 7 evaluates how well the system changes to alarm duration. We tested that the alarm initially set with a specified duration time was successfully updated mid operation and the new duration was correctly carried out by the system. This test evaluates the system's flexibility in adjusting the alarm dynamically.

5. **Stress Testing and Rapid Inputs:** Test 11 provided a scenario with alarm creation in quick successions and deletions, putting stress on the system's threading and alarm management abilities. The system evaluated stability under high-frequency commands, effective management of quick successions of alarm insertions and deletions. Also, its robustness in handling intensive operations without memory leakage or processing delays.

6. **Viewing Alarms and System State:** Several tests (e.g., Test 4, Test 9, and Test 13) incorporated commands to view the current state of alarms, allowing for observation of real-time updates and validating that the alarm list reflected the accurate status of active alarms. This functionality was essential for confirming that all created alarms were tracked, managed, and displayed in accordance with their properties and that expired alarms were accurately removed from the view.

The evaluation of this alarm system demonstrated reliable performance across a wide range of functionalities. The system met expectations for standard and complex operations, showing robust threading capabilities, resilience to invalid inputs, and flexibility in handling dynamic adjustments. The thorough testing ensures that the alarm system can manage real-world scenarios with a high degree of accuracy and stability, to fulfil its intended design requirements.

5. Conclusion

This report explained the design, implementation, and testing of a multi-threaded alarm management system for the EECS3221 course. The program, new_alarm_mutex.c, extends a basic POSIX alarm program to support more advanced features, allowing multiple alarm types to be processed and displayed at the same time. By using three types of threads—the Main Thread, Alarm Thread, and Display Threads—the system can handle various alarm operations, including starting, modifying, canceling, and viewing alarms, in real time.

Several challenges were faced during development, such as managing user inputs while checking for expired alarms and reassigning alarms when their types changed. These challenges were addressed by carefully managing threads and using synchronization tools like mutexes and condition variables to make sure that shared resources were accessed safely.

The program handles concurrency, memory, and errors well, making it stable for multi-threaded operations. Testing showed that the system meets the requirements, running smoothly on the Red server at York University and handling different alarm requests as expected.

The table below shows how the design satisfies the requirements.

Requirement	How Code Meets Requirement
Multi-threaded architecture	Implements Main, Alarm, and Display Threads. Each thread has a defined role, with Display Threads dynamically created for different alarms.
Alarm requests management	The program processes user inputs to execute Start_Alarm, Change_Alarm, Cancel_Alarm, and View_Alarms commands.
Concurrency control	Uses alarm_mutex for locking alarm_list and alarm_cond and display_cond condition variables for signaling between threads.
Error handling and validation	Includes error handling for input validation and thread operations, using err_abort and errno_abort functions for robust error control.
Display thread assignment	The Alarm Thread assigns each alarm to a Display Thread based on type, reassigning if the alarm type changes.
Memory management	Allocates memory for each new alarm with malloc and releases it with free upon expiration or cancellation.
Testability and compatibility	The code is structured to be compatible with the POSIX standard, allowing it to compile and execute on the Red server at York University.