# Assignment 3 Report

**Course:** EECS3221 Section E
**Term:** Fall 2024
**Assignment:** Assignment 3 Report
**Group Members:**

Mohammad Zarif
Rami Omer
Shaylin Ziaei
Rayhaan Yaser Mohammed

# 1. Introduction

## 1.1 Overview

This report documents the design, implementation, and testing of new_alarm_cond.c, a POSIX thread-based program for managing alarms in a multithreaded environment. The program builds on the original alarm_cond.c by implementing additional features such as group-based alarm management, advanced thread handling, and support for six types of alarm requests. The program uses unnamed semaphores and condition variables to synchronize access to shared resources while solving the Readers-Writers problem. The objective is to ensure that the alarm management system operates efficiently in a multithreaded environment, providing robust synchronization, clear thread roles, and accurate processing of user commands.

The program operates with four types of threads—Main Thread, alarmGroup_displayCreation_thread, alarmGroup_displayRemoval_thread, and display_alarm_threads—that work together to manage alarms. The Main Thread first creates threads for creating and removing display_alarm_threads then it processes user commands, checks if they comply with the format expected and then prints a message about the respective operation. The alarmGroup_displayCreation_thread thread checks if there has been a change to the list of alarms, if there is a change such as insertion etc. it will check if there already exists a display thread for the respective Group_ID if it does, it will assign the alarm to that display_alarm_thread if not it will create a new display_alarm_thread and assign the alarm to that thread. The 'alarmGroup_displayRemoval_thread' on the other hand checks if display_alarm_threads have assigned alarms, if empty it will terminate the thread and print the message. Finally, the 'display_alarm_thread' is responsible for printing each alarm every 'time' seconds apart. It is also responsible for printing a message if an alarm has been removed or changed. The thread will stop printing alarms if they have been suspended and will reprint an alarm if the status has been set to active again. The architecture and design ensure that the program can run efficiently and handles multiple alarms commands, with each alarm printed until it is suspended or cancelled, this results in a robust system that maintains the 'Readers-Writers' property and ensures threads run safely and achieve all the requirements specified below.

## 1.2 Requirements

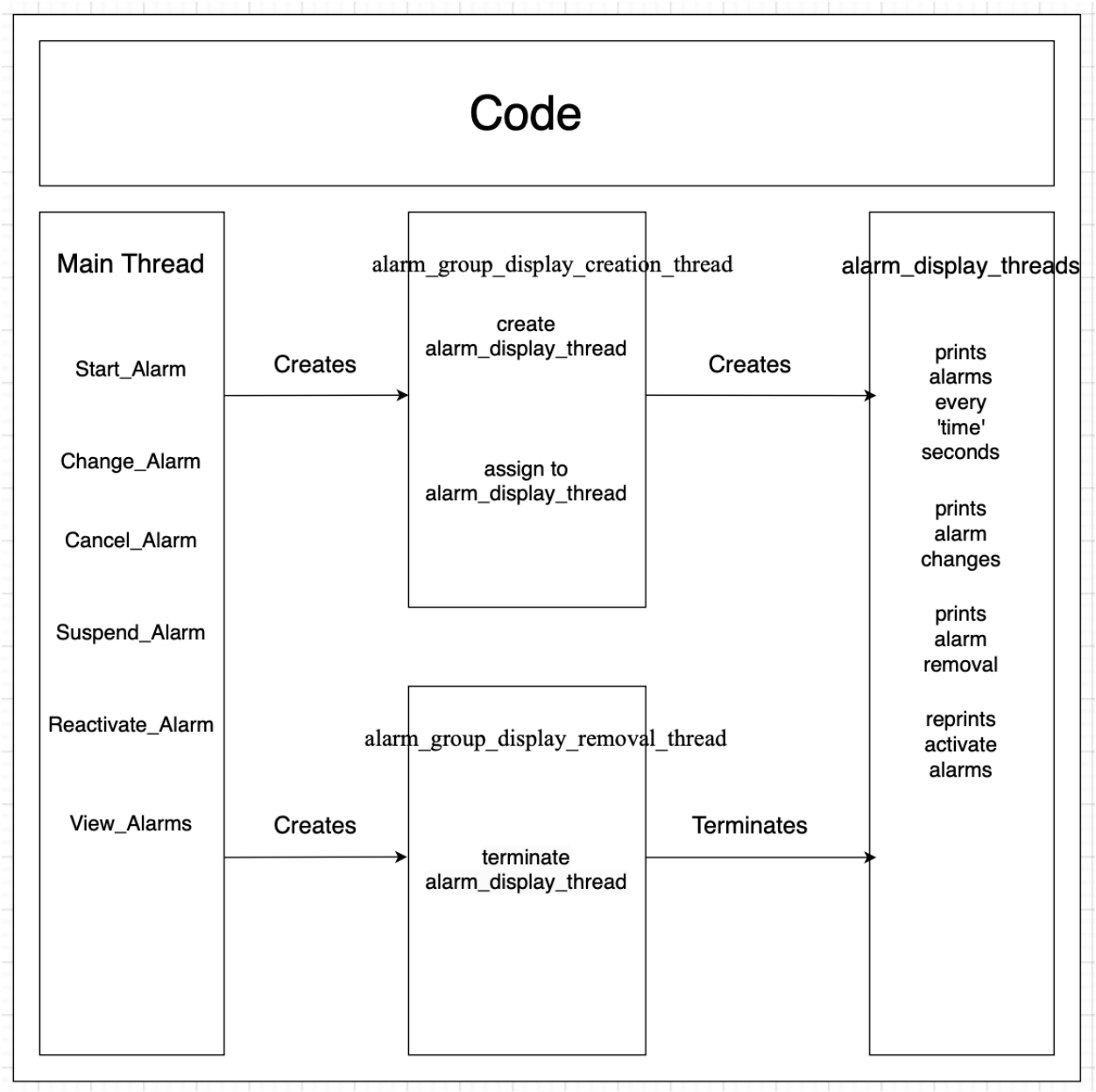
| Requirement | Description |
|---|---|
| Multi-threaded architecture | Implement four thread types:<br>1. Main Thread<br>2. alarmGroup_displayCreation_thread<br>3. alarmGroup_displayRemoval thread<br>4.Display alarm threads |

| | |
|---|---|
| Alarm requests management | Six types of Alarm Requests: Start_Alarm, Change_Alarm, Cancel_Alarm, Suspend_Alarm, Reactivate_Alarm, View_Alarms |
| Concurrency control and synchronization | Synchronization to shared data is handled by a 'Readers-Writers' approach where at a given time only one process should be able to write and at any given time all read processes should be able to read from the shared data structure list. |
| Error handling and input validation | Implement error handling for invalid input (mostly from the Main Thread) and unexpected errors, including thread synchronization failures. |
| Display threads assignment, removal and reassignment | The Main thread in "new_alarm_cond.c" creates an "alarm group_display creation_thread" and an "alarm group_display removal thread".<br>The alarm group display creation thread creates display threads, displaying alarms with the same Group_ID. |
| Memory management | Ensure proper allocation and deallocation of memory for alarms to prevent memory leaks and errors. Additionally, ensure shared data is accessed through the 'Readers-Writers' approach explained above. |
| Testability and platform compatibility | Test with different scenarios and test cases. Also ensure that 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/6497502/mod_resource/content/19/3221E_F24_a3.pdf

# 2. Architecture




## 2.1 Architecture Overview

Central to the architecture of this program are the alarm_list and the four different types of threads namely, ‘main thread’, ‘ alarmGroup_displayCreation_thread’, ‘ alarmGroup_displayRemoval_thread’ and finally the ‘display_alarm_threads’.

Initially when the program starts the main thread starts operating from the main function. Here, the main thread will first create the two threads ‘alarmGroup_displayCreation_thread’ which will assign or create display_alarm_threads. Additionally, the main thread will also create the ‘alarmGroup_displayRemoval_thread’ which will check and remove any display_alarm_thread that does not have any alarms assigned to it. Finally, the main thread will accept input from the user for actions on Starting, Modifying, Canceling, Suspending or Reactivating an alarm.

The ‘alarmGroup_displayCreation_thread’ will create or assign alarms to an alarm_display_thread corresponding to its unique group_id. The display thread is then responsible for printing periodically every ‘times’ interval for each alarm assigned to it.

Critical to this architecture are the Reader-Writer locks, which ensure that no more than thread can write to the list of alarms at a time. This is managed by the functions ‘writer_lock()’ and ‘writer_unlock()’.

## 2.2 Component Definitions

| Data Structure | | | |
|---|---|---|---|
| struct | Field Type | Field Name | Field Description |
| alarm_t | int | alarm_id | Unique alarm id |
| | int | group_id | Group id of the alarm |
| | int | seconds | Duration after which the alarm prints |
| | char[] | message | Message related to the alarm |
| | int | isActive | Status indicates if alarm is active |
| | int | isChanged | Indicates if alarm has been changed |
| | int | isCanceled | Indicates if alarm is canceled |
| | int | isAssigned | Indicates if alarm is assigned to a display thread |

| | | | |
|---|---|---|---|
| | time_t | created_at | Time when the alarm was created |
| | time_t | assigned_at | Time when the alarm was assigned to a display thread |
| | time_t | next_print_time | Time when the alarm should be printed again |
| | struct alarm_t | *link | Pointer to the next alarm in the list |

| Variables | | |
|---|---|---|
| Name | Type | Description |
| alarm_cond | pthread_cond_t | Condition variable for signaling changes |
| timed_wait | pthread_cond_t | Condition variable for triggering the timed wait |
| wait_mutx | pthread_mutex_t | Mutex for condition variable protection |
| request_mutex | pthread_mutex_t | Protects the `current_request_type` variable |
| thread_mutex | pthread_mutex_t | Mutex for thread-level synchronization |
| rw_mutex | sem_t | Semaphore for writer locks |
| mutex | sem_t | Semaphore for reader count management |
| group_thread | sem_t | Semaphore for synchronizing access to group_threads array |
| read_count | int | Number of active readers (for reader/writer locks) |
| *alarm_list | alarm_t | Head pointer for the linked list of alarms |
| group_threads[] | static pthread_t | Thread IDs for group handling threads |
| current_request_type[] | char | Tracks the current request type (e.g., "Reactivate_Alarm") |

| Functions | | |
|---|---|---|
| Function Name | Function Return Type | Function Description |
| view_alarms() | void | Displays all active and suspended alarms grouped by their group_id |
| reactivate_alarm(int alarm_id) | void | Reactivates a suspended alarm by setting its isActive flag to 1 |
| suspend_alarm(int alarm_id) | void | Temporarily deactivates an alarm by setting its isActive flag to 0 |
| cancel_alarm(int alarm_id) | void | Cancels an existing alarm by removing it from the linked list. |
| change_alarm(int alarm_id, int group_id, int seconds, char *message) | void | Modifies the attributes of an existing alarm (e.g., group_id, duration, or message) |
| start_alarm(int alarm_id, int group_id, int seconds, char *message) | void | Adds a new alarm to the linked list. |
| reader_unlock() | void | Unlocks the alarm list after reading |
| reader_lock() | void | Locks the alarm list for reading |
| current_time() | void | Function to retrieve the current system time in UNIX epoch format |
| writer_lock() | void | Locks the alarm list for writing |
| writer_unlock() | void | Unlocks the alarm list after writing |

| Thread | | |
|---|---|---|
| Name | Role | Responsibilities |
| Main Thread | Creates threads, manages user input, checks the format, prints messages. | - Creates an “alarm group_display creation_thread” and an “alarm |

| | | |
|---|---|---|
| | | group_display removal thread.<br>- Manages user input for Change_Alarm, Cancel_Alarm, Suspend_Alarm, Reactivate_Alarm, View_Alarms<br>- Prints a message about the action performed. |
| alarm group_display_creation_thread | Waits until the alarm_list has been modified. Responsible for creating 'display_alarm_thread'. Each will only print alarms with the same Group_ID | - Create new 'display_alarm_threads' if none exists for a newly inserted alarm with a given Group_ID<br>- Assign an alarm to a display_thread if a display_thread already exists for that Group_ID |
| alarm group_display removal thread | Waits until the alarm_list has been modified. Responsible for removing a 'display_alarm_thread' if it contains no alarm. | - Checks if an alarm exists in a 'display_alarm_thread' if not it will terminate the thread |
| display_alarms_thread | Prints alarms periodically given 'time' intervals. Also prints about status if the alarm has been modified, cancelled, suspended. | - Prints Alarms every 'time' seconds<br>- Prints if alarm has been removed<br>- Prints if alarm has been changed<br>- Prints a stop printing message if alarm has been suspended<br>- Reprints alarms that have been reactivated |

## 2.3 Data Flow and Synchronization

### 2.3.1. Data Flow

The alarm management system outlined in the code is centered around handling, processing, and displaying alarms grouped by their associated group IDs. The data flow involves the following key components:

1. **Input Commands**: The system allows user input through commands such as Start_Alarm, Change_Alarm, Cancel_Alarm, Suspend_Alarm, and Reactivate_Alarm. These commands

initiate specific operations on the alarm list.

2. **Alarm Creation and Initialization**:
    - When an alarm is created using the Start_Alarm command, it is initialized with attributes like alarm_id, group_id, seconds, and message.
    - The system timestamps the alarm (created_at) and calculates the next trigger time (next_print_time).

3. **Alarm List**:
    - Alarms are stored in a linked list (alarm_list), sorted by alarm_id.
    - Each node in the list is an alarm_t struct, which contains metadata about the alarm (e.g., activation status, assignment details, timestamps).
4. **Group Threads**:
    - Alarms are grouped by their group_id, and each group is assigned a dedicated thread from the group_threads array.
    - A thread processes all active alarms for a group, handling their display and managing their states (e.g., canceled, suspended).
5. **Thread Operations**:
    - **Display Threads**: Continuously monitor and process alarms in their respective groups, printing messages or updating their statuses as needed.
    - **Thread Creation and Removal**:
        - ➔ The alarmGroup__displayCreation_thread dynamically creates threads for new groups when an unassigned alarm is detected.
        - ➔ The alarmGroup_displayRemoval_thread removes threads for groups that no longer have active alarms.
6. **Output**:
    - The system outputs detailed logs for each operation, such as creating, changing, canceling, or suspending alarms.
    - Active alarms are displayed at their specified intervals, grouped by their group_id.

This architecture ensures that alarms are efficiently managed and displayed while maintaining clarity in data flow between components.

### 2.3.2 Synchronization

The system relies heavily on multithreading and employs robust synchronization mechanisms to ensure thread-safe operations. These include:

1. **Mutex Locks**:
    - **pthread_mutex_t:**
        - The request_mutex protects access to the current_request_type variable, ensuring that only one thread can update or read it at a time.

        - The thread_mutex is used for managing thread-level synchronization during timed waits or group-specific processing.
    - **Reader-Writer Locks**:
        - The mutex semaphore manages the read_count variable, ensuring consistent access to the alarm_list by multiple reader threads.
        - The rw_mutex semaphore prevents simultaneous writes or a mix of read and write operations, maintaining data integrity.
2. **Condition Variables**:
    - The alarm_cond variable signals threads waiting for changes in the alarm_list, such as new alarms being added or modified.
    - The timed_wait condition variable allows threads to wait for specific alarm trigger times.

3. **Semaphores**:
    - rw_mutex and mutex: Implement reader-writer synchronization for the shared alarm_list.
    - group_thread: Synchronizes access to the group_threads array, preventing race conditions during thread creation and deletion.
4. **Thread-Safe Operations**:
    - **Alarm Management**:
        - Writer locks (writer_lock and writer_unlock) ensure exclusive access to the alarm_list during modifications (e.g., adding, changing, or canceling alarms).
        - Reader locks (reader_lock and reader_unlock) allow multiple threads to safely view alarms without interference from writers.
    - **Thread Creation and Removal**:
        - The alarmGroup__displayCreation_thread ensures that only one thread is created per group, avoiding redundant threads.
        - The alarmGroup_displayRemoval_thread dynamically removes threads for inactive groups, optimizing resource usage.
5. **Priority Handling**: Conditional waits support priority-based scheduling, where processes with smaller priority numbers are resumed first. This mechanism prevents starvation and ensures fairness.

6. **Synchronization Challenges**: Potential issues such as race conditions, deadlocks, and missed signals are addressed by locking protocols and proper use of condition variables and semaphores.

By combining these synchronization techniques, the system achieves robust, thread-safe management of alarms, ensuring data consistency and operational efficiency in a multithreaded environment.

# 3. Design and Implementation

## 3.1 Thread Design

**1. Main Thread**

The main thread acts as the central controller of the alarm management system. Its responsibilities include initializing resources, processing user commands, and coordinating the creation and management of worker threads.

**Key Responsibilities:**

**Initialization**:

- Initializes semaphores like mutex, rw_mutex, and group_thread to ensure proper synchronization.
- Creates two monitoring threads: alarmGroup__displayCreation_thread and alarmGroup_displayRemoval_thread for dynamically managing group-based display threads.

**Command Processing**:

- Reads user input commands (e.g., Start_Alarm, Change_Alarm, Cancel_Alarm, etc.) and invokes corresponding functions to manage alarms.
- Uses condition variables like alarm_cond to notify worker threads of changes in the alarm list.

**System Termination**: Joins the created threads upon termination (though unreachable in the infinite loop) and destroys semaphores to release resources.

The main thread ensures that user commands are translated into specific operations on the shared alarm list while maintaining system synchronization and stability.

**2. AlarmGroup__displayCreation_thread**

This thread is responsible for dynamically creating display threads for alarm groups as needed. It continuously monitors the alarm list for unassigned alarms and ensures that every group with active alarms has a dedicated thread.

**Key Responsibilities**:

**Monitoring the Alarm List**:

- Waits for signals on alarm_cond indicating changes to the alarm list.
- Traverses the alarm list to identify alarms that are unassigned (isAssigned = 0).

**Thread Creation**:

- Creates a new thread (display_alarm_threads) for groups without an existing thread in group_threads.
- Dynamically allocates memory for the group ID and assigns the created thread to the corresponding group.
- Logs thread creation events for traceability.

**Thread Assignment**: Marks alarms as assigned (isAssigned = 1) and updates their assigned_at timestamp.

This thread ensures efficient resource utilization by dynamically creating threads only for active groups, avoiding redundant thread creation.

**3. alarmGroup_displayRemoval_thread**

This thread continuously monitors alarm groups and removes display threads for groups that no longer have active alarms. Its goal is to optimize resource usage by cleaning up idle threads.

**Key Responsibilities**:

**Monitoring the Alarm List**:

- Waits for signals on alarm_cond to detect changes in the alarm list.
- Traverses the alarm list to check for groups without any active alarms.

**Thread Removal**:

- Cancels threads for groups with no alarms and clears their entry in the group_threads array.
- Logs thread removal events for debugging and system monitoring.

**Synchronization**: Uses semaphores like group_thread to ensure thread-safe modifications to the group_threads array.

This thread ensures system efficiency by dynamically removing idle threads, preventing unnecessary resource consumption.

**4**. **display_alarm_threads**

Each group is assigned a dedicated display_alarm_threads thread to handle and display its active alarms. These threads continuously process alarms in their respective groups, ensuring that they are triggered at the correct times.

**Key Responsibilities:**

**Alarm Processing:**

- Collects all active alarms (isActive = 1 and isCanceled = 0) for the assigned group.

- Waits until the next_print_time of each alarm and then processes it (e.g., prints the alarm message, checks for modifications, cancellations, or reassignments).

**Handling Alarm States:**

- Skips cancels alarms and logs their cancellation.
- Updates the next_print_time of alarms after processing them.
- Handles modified alarms (isChanged = 1) by resetting their change flag and updating their state.

**Synchronization:**

- Uses reader-writer locks (reader_lock, reader_unlock) to safely traverse the shared alarm list.
- Relies on condition variables like timed_wait for waiting until alarms are due for processing.

This thread ensures accurate and timely processing of alarms for its assigned group, maintaining the integrity and reliability of the alarm management system.

## 3.2 Error Handling and Resource Management

### 1. Error Handling

Error handling mechanisms are implemented at multiple levels of the system to ensure predictable behaviour even in edge cases. These include:

**Command Validation**

Before executing any user command, the main thread performs rigorous validation:

- Syntax Checking: Commands are parsed and checked against the expected syntax. If a command is missing parameters or using incorrect formatting, it is flagged as an invalid command. For example:
  - Input: Start_Alarm (0): Group(2) 10 Invalid ID
  - Output: Bad Command
- Logical Validation: Commands are validated for logical correctness:
  - Alarm IDs and group IDs must be positive integers.
  - Durations must be non-negative.

**Error Logging**

Errors are not only reported to the user but also logged for debugging and monitoring.

- Attempting to start an alarm with a duplicate ID produces a clear error message:
  - Output: Error Alarm with ID 1 already Exists

**Detection of Anomalous States**

The system actively detects and resolves unexpected states in the alarm list or threads:

- Non-existent alarms: commands targeting alarms that do not exist result in an appropriate error message:
  - Input: Cancel_Alarm (999)
  - Output: Error: Alarm ID 999 not found.

**Fault Isolation and Recovery**

- Errors in one part of the system are isolated to prevent cascading failures. Also, for other errors like thread-specific error, modular operations and log-based Diagnostics. One such example is:
  - if a thread encounters an error while processing an alarm, it logs the issue and continues operating, ensuring that other threads and alarms are unaffected.
- The systems include mechanisms to recover from recoverable errors without requiring a complete restart:
  - If an alarm fails to initialize due to memory constraints, the error is logged, and the system continues to process other commands.

## 2. Resource Management

In the implementation of the alarm management system, resource management is crucial for ensuring system stability, efficiency, and proper cleanup during execution. The primary focus is on the efficient utilization and cleanup of dynamic memory, semaphores, threads, and shared data structures. Below is a detailed explanation of resource management strategies applied in the provided code:

**1. Memory Allocation and Deallocation**

**Dynamic Memory Allocation**:

- The system dynamically allocates memory for each new alarm using malloc. This is necessary because the number of alarms is not fixed, and the linked list data structure used to store alarms requires dynamic memory allocation.
- For example, in the start_alarm function, memory for a new alarm_t structure is allocated:
  `alarm_t *alarm = (alarm_t *) malloc(sizeof(alarm_t));`
- Memory is also dynamically allocated for the group ID when creating new display threads in alarmGroup__displayCreation_thread:
  `group_id_ptr = (int *) malloc(sizeof(int));`

**Memory Deallocation**:

- When alarms are canceled, the memory allocated to the respective alarm_t structure is released using free. This is done in the cancel_alarm function:
  `free(current);`

- Similarly, memory allocated for group IDs in alarmGroup__displayCreation_thread is freed after the ID is passed to the new thread:
  free(group_ID);

**Error Prevention**:

Memory allocation failures are checked, and appropriate error-handling macros (errno_abort) are invoked to prevent the program from proceeding in an inconsistent state:

```
if (alarm == NULL)
        errno_abort ("Allocate alarm");
```

**2. Semaphore Management**

Semaphores are a core synchronization primitive used to manage access to shared resources, such as the alarm list and group threads.

**Initialization**:

Semaphores are initialized at the start of the program to ensure proper synchronization:

```
sem_init(&mutex, 0, 1);
 sem_init(&rw_mutex, 0, 1);
 sem_init(&group_thread, 0, 1);
```

Each semaphore has a specific purpose:

- mutex: Protects the read_count variable, ensuring mutual exclusion during updates.
- rw_mutex: Ensures exclusive access for writers, blocking both readers and other writers.
- group_thread: Synchronizes access to the group_threads array, preventing race conditions during thread assignments.

**Usage**:

Semaphores are used to implement reader-writer locks for safe access to the shared alarm list:

```
sem_wait(&mutex);
read_count++;
if (read_count == 1)
        sem_wait(&rw_mutex);
sem_post(&mutex);
```

**Cleanup**:

At the end of the program, semaphores are destroyed to release system resources:

```
sem_destroy(&rw_mutex);
```

```
sem_destroy(&mutex);
sem_destroy(&group_thread);
```

### 3. Thread Management

**Thread Creation**:

Threads are created dynamically to handle specific tasks, such as monitoring alarm groups (alarmGroup__displayCreation_thread and alarmGroup_displayRemoval_thread) and processing alarms in a group (display_alarm_threads).

For example, the creation of a display thread is logged in alarmGroup__displayCreation_thread:

```
pthread_create(&new_thread, NULL, display_alarm_threads, group_id_ptr);
```

**Thread Termination**:

Threads responsible for alarm groups are canceled when the group no longer has active alarms. This is done in the alarmGroup_displayRemoval_thread:

```
pthread_cancel(group_threads[group_id]);
```

**Joining Threads**:

In the main thread, monitoring threads are joined upon program termination to ensure proper cleanup:

```
pthread_join(display_thread_creation, NULL);
pthread_join(display_thread_removal, NULL);
```

### 4. Shared Resource Management

**Alarm List**:

- The alarm list is a shared resource accessed by multiple threads. To ensure safe access, reader-writer locks are implemented using semaphores.
- Readers use reader_lock and reader_unlock to safely traverse the list, while writers use writer_lock and writer_unlock to make modifications:

```
reader_lock();
// Traverse alarm list
reader_unlock();
```

**Group Threads**:

The group_threads array stores thread IDs for managing alarms by group. Access to this array is synchronized using the group_thread semaphore to prevent race conditions:

```
sem_wait(&group_thread);
```

```
group_threads[group_id] = new_thread;
sem_post(&group_thread);
```

**5. Condition Variable Management**

Condition variables are used to signal changes in the alarm list to other threads (alarm_cond and timed_wait):

- alarm_cond signals when a new alarm is added or an existing alarm is modified or canceled.
- timed_wait allows display threads to wait until an alarm is due to trigger:
  `pthread_cond_timedwait(&timed_wait, &thread_mutex, &timeout);`

**6. Error Handling in Resource Management**

All resource initialization, usage, and cleanup operations are accompanied by error handling to ensure robustness. For example:

- Checking semaphore operations:
  ```
  if (sem_wait(&rw_mutex) != 0)
          err_abort(status, "sem wait");
  ```
- Logging errors and program state to assist in debugging and recovery:
  `fprintf(stderr, "Error: Alarm ID %d not found\n", alarm_id);`

The alarm management system employs a structured approach to resource management, ensuring efficient allocation, usage, and cleanup of resources. By combining dynamic memory management, synchronization primitives, and robust error handling, the system maintains stability and performance, even under concurrent operations. These strategies prevent resource leaks, race conditions, and deadlocks, ensuring the system operates reliably.

## 3.3 Challenges

During the development and implementation of the alarm management system, we encountered several challenges that required thoughtful analysis and careful adjustments to overcome. Below, we discuss two major challenges and the solutions we implemented to address them effectively.

**Challenge 1: Reassigning Display Threads for Existing Alarms**
One of the significant issues we faced was the repeated reassignment of display threads to alarms that had already been correctly assigned. Whenever a new alarm was inserted or an existing alarm was modified, all alarms in the list were being reassigned to new display threads. This behavior was unnecessary and inefficient, as it disrupted the established thread assignments and led to redundant resource usage.

**Solution:**
To resolve this, we refined the logic in the alarmGroup__displayCreation_thread. Instead of iterating through all alarms, the thread was updated to specifically target alarms that were either newly inserted or modified (i.e., alarms with isAssigned = 0). By ensuring that only these specific alarms were reassigned, we prevented the unnecessary reassignment of already assigned alarms. This optimization improved resource utilization and maintained stable thread assignments for alarms that did not require changes.

**Challenge 2: Inconsistent Timing for Displaying Alarms**
Another challenge occurred when a display thread was responsible for handling multiple alarms within the same group ID. The thread was printing periodic statements for these alarms at inconsistent intervals, failing to respect the assigned time for each alarm. This inconsistency undermined the accuracy of the alarm management system.

**Solution:**
We addressed this issue by introducing a mechanism to manage the timing of alarms individually within each display thread. Each alarm's next_print_time was tracked independently, and the display thread processed each alarm only when its specific print time was due. Additionally, we incorporated condition variables and precise timing logic to ensure that alarms within the same group were printed at their respective scheduled intervals. This approach resolved the inconsistency and ensured that the periodic statements for all alarms were displayed at the correct times.

# 4. Evaluation

The system was rigorously tested with a variety of scenarios to ensure functionality, robustness, and performance. Below is a detailed summary of the test cases:

| TEST CASE | DESCRIPTION | INPUT | EXPECTED OUTPUT | RESULT |
|---|---|---|---|---|
| TEST 1 | Create a new alarm with valid inputs | Start_Alarm(1): Group(10) 10 Test alarm Message | Alarm created successfully and displayed by thread. | PASSED |
| TEST 2 | Attempt to create a duplicate alarm. | Start_Alarm(1): Group(10) 10 Duplicate Alarm | Error: Alarm with ID 1 already exists. | PASSED |
| TEST 3 | Test Invalid inputs for Alarm Creation | Start_Alarm(0): Group(2) 10 Invalid ID | Bad Command | PASSED |
| TEST 4 | Modify an existing alarm's properties. | Change_Alarm(1): Group(3) 15 Modified Message | Alarm updated and reflected in thread output. | PASSED |
| TEST 5 | Attempt to modify a non-existent alarm | Change_Alarm(999): Group(4) 20 Non-Existent Alarm | Error: Alarm ID 999 not found | |
| TEST 6 | Cancel an existing alarm. | Cancel_Alarm(1) | Alarms removed and Thread terminated | PASSED |
| TEST 7 | Attempt to cancel a non-existent alarm | Cancel_Alarm(999) | Error: Alarm ID 999 not found | PASSED |
| TEST 8 | Suspend an active alarm | Suspend_Alarm(1) | Alarm status changed to "Suspended." | PASSED |
| TEST 9 | Reactivate a suspended alarm | Reactivate_Alarm(1) | Alarm status changed to "Active." | PASSED |
| TEST 10 | View all alarms. | View_Alarms | Displays current alarm details grouped by threads. | PASSED |
| TEST 11 | Start alarms in different groups | Start_Alarm(2): Group(3) 20 Alarm 2 | Threads created for respective groups | PASSED |
| TEST 12 | Remove threads for empty groups. | Cancel_Alarm(2) | Thread for group removed after alarm cancellation | PASSED |
| TEST 13 | Handle Simultaneous | Multiple commands in sequence | System maintains consistency and correctness. | PASSED |

# 5. Conclusion

This report outlines the overall design, requirements, explains the respective implementation, and illustrates and evaluates testing of a multi-threaded alarm management system for the EECS3221 course. The program, new_alarm_cond.c, improves the basic POSIX alarm program by adding additional features that include allowing multiple alarm types to be processed. Utilizing the four types of threads—the Main Thread, 'alarmgroup_displaycreation_thread', 'alarmgroup_displayremoval_thread' and 'alarm_display_thread'—the system can handle various alarm operations, including starting, modifying, canceling, suspending, reactivating and finally viewing alarms.

Several challenges were faced during development, mainly Reassigning Display Threads for Existing Alarms and Inconsistent Timing for Displaying Alarms. These challenges were resolved after several redesigns and tests and by carefully managing threads and using synchronization tools like mutexes and flags to make sure that shared resources were accessed in a safe and synchronized manner.

After performing several tests and modifications the program runs in an efficient manner that satisfies all the criteria outlined in the corresponding requirements. Testing confirmed that the program meets the requirements, running smoothly on the red server at York University and handling different user inputs.

Below is a table that shows the requirements and how they are satisfied by the program.

| Requirement | How Code Meets Requirement |
|---|---|
| Multi-threaded architecture | This requirement is met by the creation of the four threads. Main Thread, which creates threads 'alarm group_display creation_thread', and 'alarm group_display removal_thread'. And 'alarm group_display creation_thread' creates 'alarm_display_thread' |
| Alarm requests management | This requirement is satisfied through the main thread, where the thread validates and accepts user inputs for Start, Change, Cancel, Suspend, Reactivate requests for the alarms. |
| Concurrency control and synchronization | This is ensured by synchronization to shared data is handled by a 'Readers-Writers' approach where at a given time only one process should be able to write and at any given time all read processes should be able to read from the shared data structure list. |
| Error handling and input validation | The program Includes error handling for the user input validation and thread operations, using err_abort and errno_abort functions for robust error control. If user inputs invalid commands such errors are handled from the main thread, additional issues with memory access and synchronization, errors while creating, |

| | |
|---|---|
| | aborting threads are handled from the other threads. |
| Display threads assignment, removal and reassignment | The Main thread in “new_alarm_cond.c” creates an “alarm group_display creation_thread” and an “alarm group_display removal thread”.<br>The ‘alarmGroup_displayCreation_thread’ thread checks and ensures that each alarm has a corresponding ‘alarm_display_thread’. Finally, ‘alarmGroup_displayCreation_thread’ checks and ensure that ‘alarm_display_threads’ that have no assigned threads and safely terminated and the messaged is printed. |
| Memory management | Ensure proper allocation and deallocation of memory for alarms to prevent memory leaks and errors. Additionally, ensure shared data is accessed through the ‘Readers-Writers’ approach explained above. |
| Testability and platform compatibility | This is ensured by testing with different scenarios and test cases. These test cases can be observed in the ‘Evaluation’ section above. Additionally, the program compiles and runs on the designated platform (Red server at York) with POSIX compliance. |