POSIX Threads, Semaphores, Readers-Writers Problem

Stefan Smolovic, Davyd Zinkiv, Ryan Luk, Viral Patel

EECS3221

Professor Xu

December 5, 2023

Note: Unfortunately, due to a bug in our code a number of the features of the program do not work. The source code contains implementations for all of the features, but because of a bug that appeared near the end of development we are not able to showcase that this code works.

Due to time constraints we were not able to determine where the bug was coming from and how to resolve it, however we believe it has to do with the way we implemented signaling for the alarm display threads.

*We have used the files provided in the /cs/course/3221E/assign3 directory *

Introduction

The purpose of this report is to elaborate on the design and implementation of POSIX Threads and semaphores, and the application of these concepts in solving the Readers-Writers Problem. This program incorporates two types of alarm requests, implements synchronization through POSIX mutexes, and manages data access among multiple threads. The report will cover the program's design and implementation, as well as the challenges we faced.

Design Overview

The alarm management system is designed to handle the scheduling, modification, and display of alarms through a multi-threaded program in C using POSIX Threads.

Program Structure

The alarm system program is structured into several key components:

- Alarm Structure: Defined to store information about individual alarms, including their ID, group ID, duration, message, display thread ID, and timestamp.
- Change Request Structure: Contains data to manage modifications to existing alarms, including details like the alarm ID, new group ID, new duration, new message, and timestamp.
- Alarm Queue Node: Links alarms together in a queue structure
- Alarm Monitor Thread: Responsible for continuously monitoring the alarms, checking their expiration, and handling their removal from the alarm list.
- Display Alarm Threads: Multiple threads that handle the display of alarms, each assigned to a specific group of alarms. These threads manage their respective alarm queues and display the alarms they are responsible for.

Thread Synchronization

- ➔ Mutexes: Used to control access to critical sections of code, such as the alarm list, change request list, and alarm queue.
- ➔ Condition Variables: Enable threads to wait for specific events (e.g., new alarms or change requests) and signal other threads upon changes.

Core Functionality

- Alarm Insertion: New alarms are inserted into a global list sorted by ID to ensure efficient retrieval and monitoring.
- Change Requests: Allow modifications to existing alarms by creating change request structures, which are then processed to update alarm attributes.
- Thread Management: Display alarm threads are created or reused based on group IDs, managing alarms assigned to specific groups.

A3.1 This part of the assignment tells you what changes or what types of new things the program can do with alarms:

(A) you can start an alarm by saying:

Alarm> Start_Alarm(Alarm_ID): Group(Group_ID) Time Message.

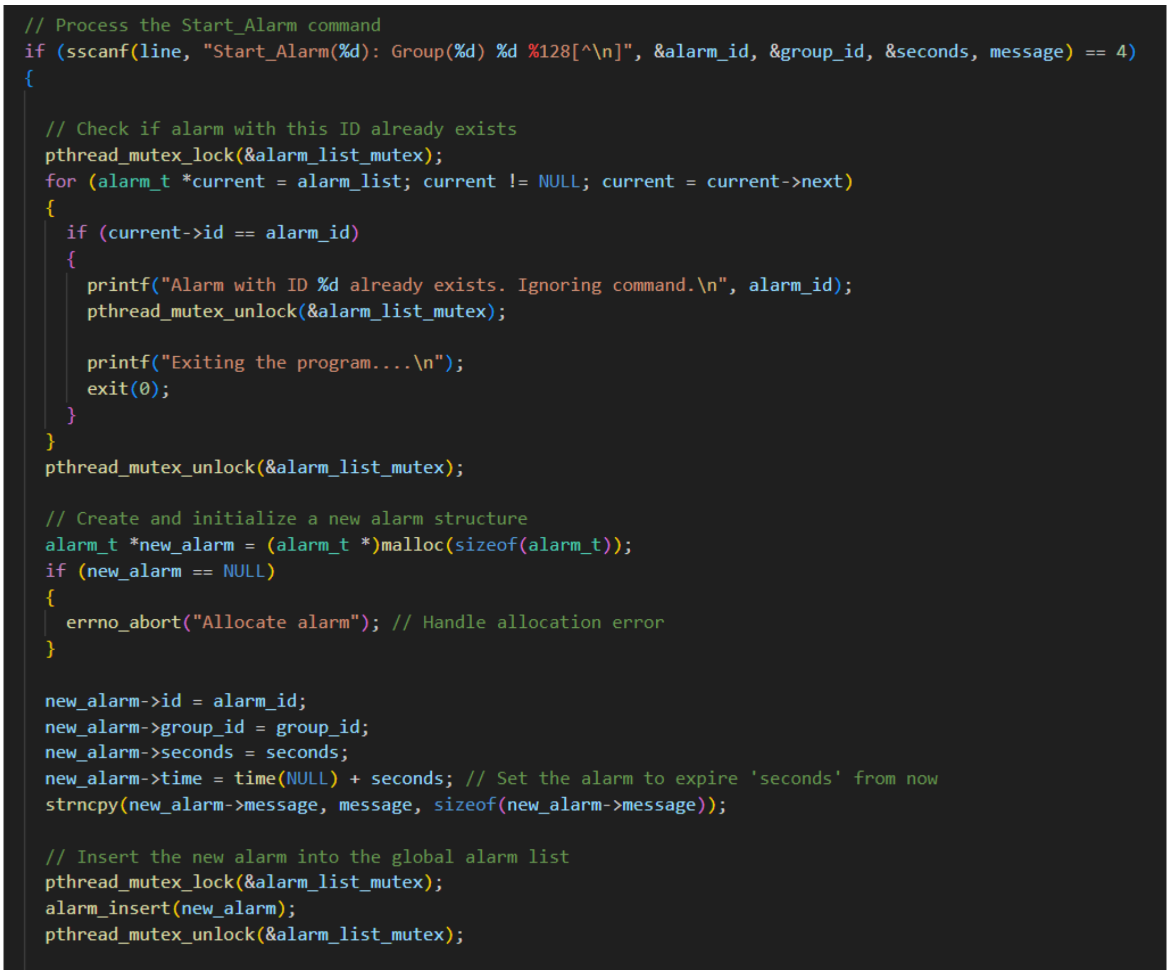
(B) you can replace an alarm by saying:

Alarm> Change_Alarm(Alarm_ID): Group(Group_ID) Time Message

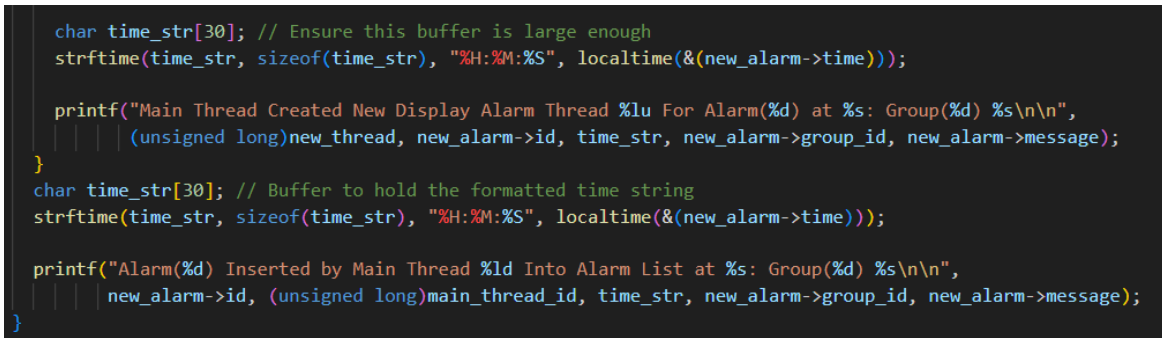
A3.2 The main thread in New_Alarm_Cond.c

A.3.2.1 Start_Alarm Requesting Processing

The main thread reads and parses user input to Identify the request. Firstly it will check the criteria of 128 characters and also check the corresponding alarm is added to the alarm list. Upon receiving such requests, the system must insert the corresponding alarm, identified by its Alarm_ID, into the alarm list. This list will maintain order based on the Alarm_IDs.



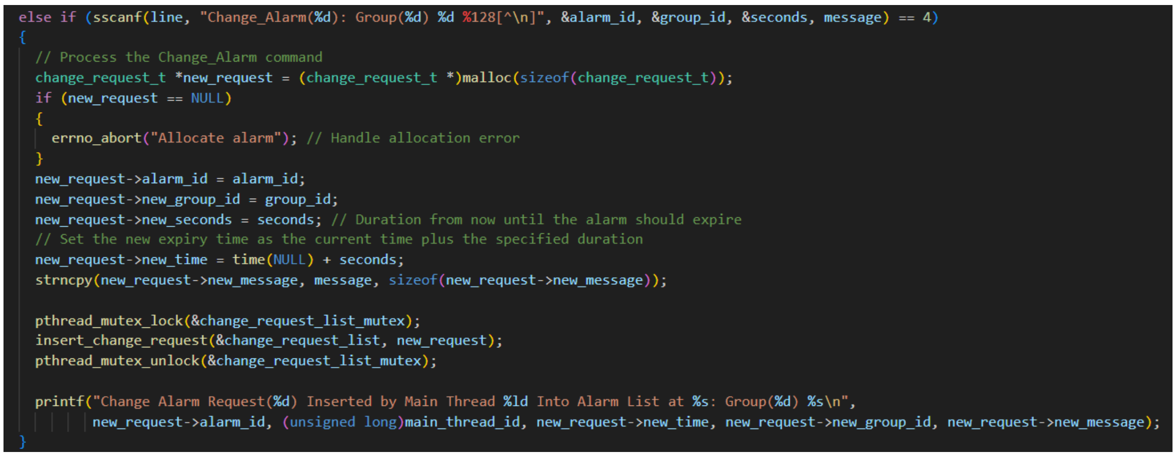
```c
// Process the Start_Alarm command
if (sscanf(line, "Start_Alarm(%d): Group(%d) %d %128[^\n]", &alarm_id, &group_id, &seconds, message) == 4)
{
    // Check if alarm with this ID already exists
    pthread_mutex_lock(&alarm_list_mutex);
    for (alarm_t *current = alarm_list; current != NULL; current = current->next)
    {
        if (current->id == alarm_id)
        {
            printf("Alarm with ID %d already exists. Ignoring command.\n", alarm_id);
            pthread_mutex_unlock(&alarm_list_mutex);

            printf("Exiting the program....\n");
            exit(0);
        }
    }
    pthread_mutex_unlock(&alarm_list_mutex);

    // Create and initialize a new alarm structure
    alarm_t *new_alarm = (alarm_t *)malloc(sizeof(alarm_t));
    if (new_alarm == NULL)
    {
        errno_abort("Allocate alarm"); // Handle allocation error
    }

    new_alarm->id = alarm_id;
    new_alarm->group_id = group_id;
    new_alarm->seconds = seconds;
    new_alarm->time = time(NULL) + seconds; // Set the alarm to expire 'seconds' from now
    strncpy(new_alarm->message, message, sizeof(new_alarm->message));

    // Insert the new alarm into the global alarm list
    pthread_mutex_lock(&alarm_list_mutex);
    alarm_insert(new_alarm);
    pthread_mutex_unlock(&alarm_list_mutex);
```



```c
        char time_str[30]; // Ensure this buffer is large enough
        strftime(time_str, sizeof(time_str), "%H:%M:%S", localtime(&(new_alarm->time)));

        printf("Main Thread Created New Display Alarm Thread %lu For Alarm(%d) at %s: Group(%d) %s\n\n",
                (unsigned long)new_thread, new_alarm->id, time_str, new_alarm->group_id, new_alarm->message);
    }
    char time_str[30]; // Buffer to hold the formatted time string
    strftime(time_str, sizeof(time_str), "%H:%M:%S", localtime(&(new_alarm->time)));

    printf("Alarm(%d) Inserted by Main Thread %ld Into Alarm List at %s: Group(%d) %s\n\n",
            new_alarm->id, (unsigned long)main_thread_id, time_str, new_alarm->group_id, new_alarm->message);
}
```

A.3.2.2 Change_Alarm Requesting Processing

For Change_Alarm request, code parses and verifies the command format to identify Change_Alarm request, then creates a new change_request_t structure and initializes it with the provided details. Ensures thread safety by employing mutex locks. Finally, it outputs a confirmation message, acknowledging the successful insertion of the change alarm request into the list.

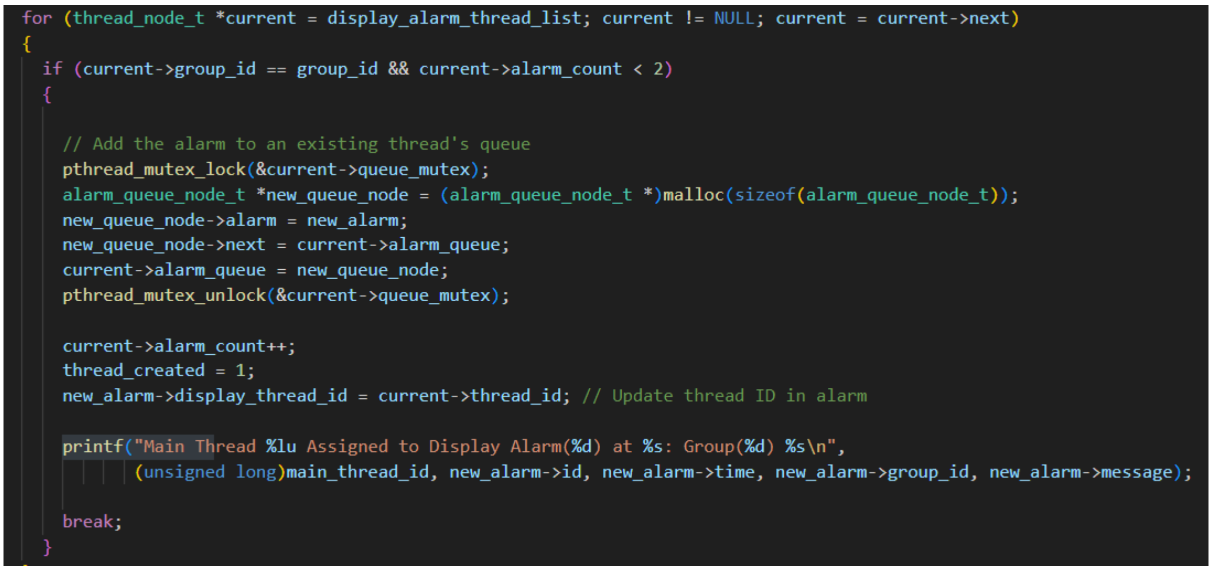


```
else if (sscanf(line, "Change_Alarm(%d): Group(%d) %d %128[^\n]", &alarm_id, &group_id, &seconds, message) == 4)
{
  // Process the Change_Alarm command
  change_request_t *new_request = (change_request_t *)malloc(sizeof(change_request_t));
  if (new_request == NULL)
  {
    errno_abort("Allocate alarm"); // Handle allocation error
  }
  new_request->alarm_id = alarm_id;
  new_request->new_group_id = group_id;
  new_request->new_seconds = seconds; // Duration from now until the alarm should expire
  // Set the new expiry time as the current time plus the specified duration
  new_request->new_time = time(NULL) + seconds;
  strncpy(new_request->new_message, message, sizeof(new_request->new_message));

  pthread_mutex_lock(&change_request_list_mutex);
  insert_change_request(&change_request_list, new_request);
  pthread_mutex_unlock(&change_request_list_mutex);

  printf("Change Alarm Request(%d) Inserted by Main Thread %ld Into Alarm List at %s: Group(%d) %s\n",
         new_request->alarm_id, (unsigned long)main_thread_id, new_request->new_time, new_request->new_group_id, new_request->new_message);
}
```

A.3.2.3 Main Thread

Assignment to Existing Display Alarm Threads:

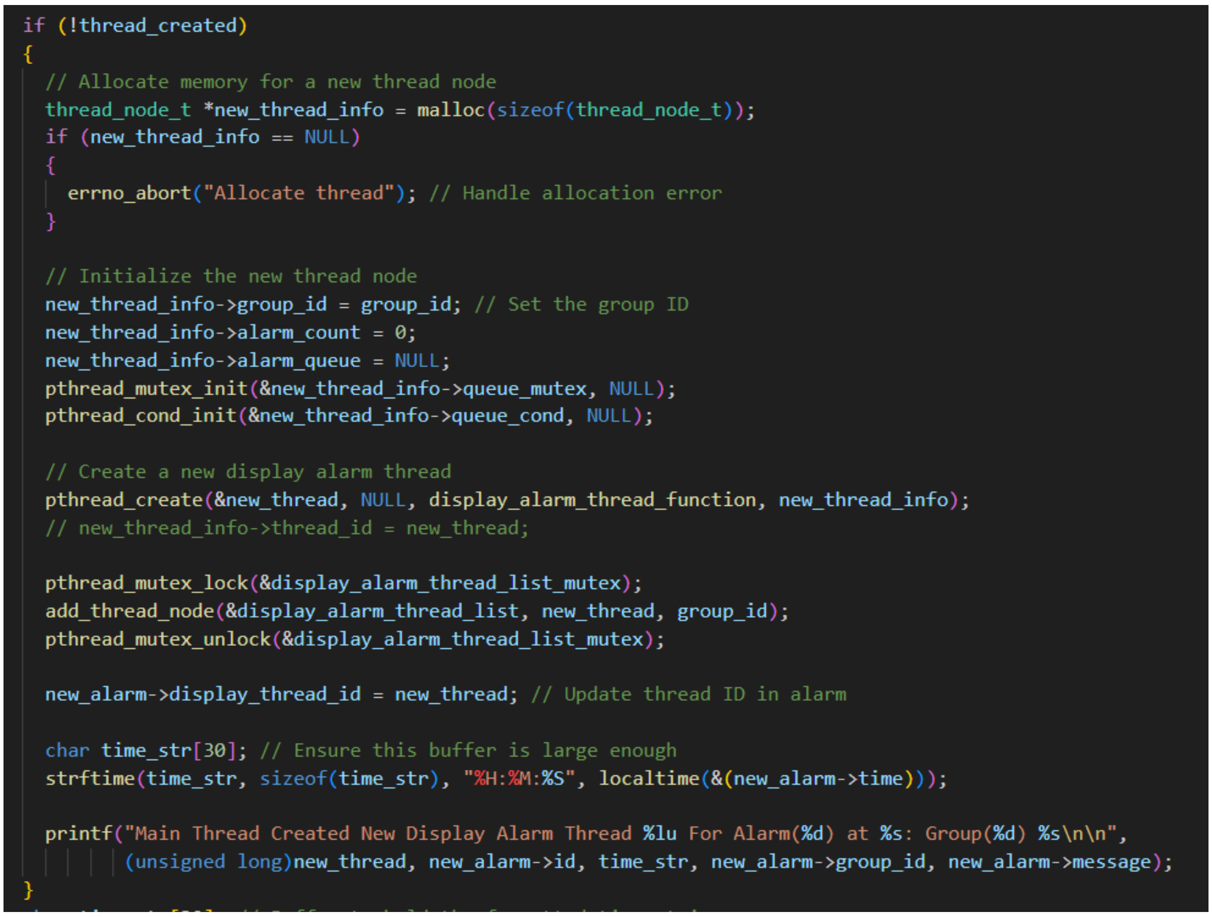
- If a suitable display alarm thread exists and it is responsible for less than two alarms, assigns the current alarm to this existing thread.
- associates the alarm with the respective thread.



```
for (thread_node_t *current = display_alarm_thread_list; current != NULL; current = current->next)
{
  if (current->group_id == group_id && current->alarm_count < 2)
  {

    // Add the alarm to an existing thread's queue
    pthread_mutex_lock(&current->queue_mutex);
    alarm_queue_node_t *new_queue_node = (alarm_queue_node_t *)malloc(sizeof(alarm_queue_node_t));
    new_queue_node->alarm = new_alarm;
    new_queue_node->next = current->alarm_queue;
    current->alarm_queue = new_queue_node;
    pthread_mutex_unlock(&current->queue_mutex);

    current->alarm_count++;
    thread_created = 1;
    new_alarm->display_thread_id = current->thread_id; // Update thread ID in alarm

    printf("Main Thread %lu Assigned to Display Alarm(%d) at %s: Group(%d) %s\n",
           (unsigned long)main_thread_id, new_alarm->id, new_alarm->time, new_alarm->group_id, new_alarm->message);

    break;
  }
```

Creation of Display Alarm Threads:

- checks for existing display alarm threads responsible for a particular Group_ID or determines if they're already responsible for more than one alarm.
- If no suitable display alarm thread exists or the existing threads are handling more than one alarm, a new display alarm thread is created.
- This creation process involves assigning the alarm to this newly created thread.

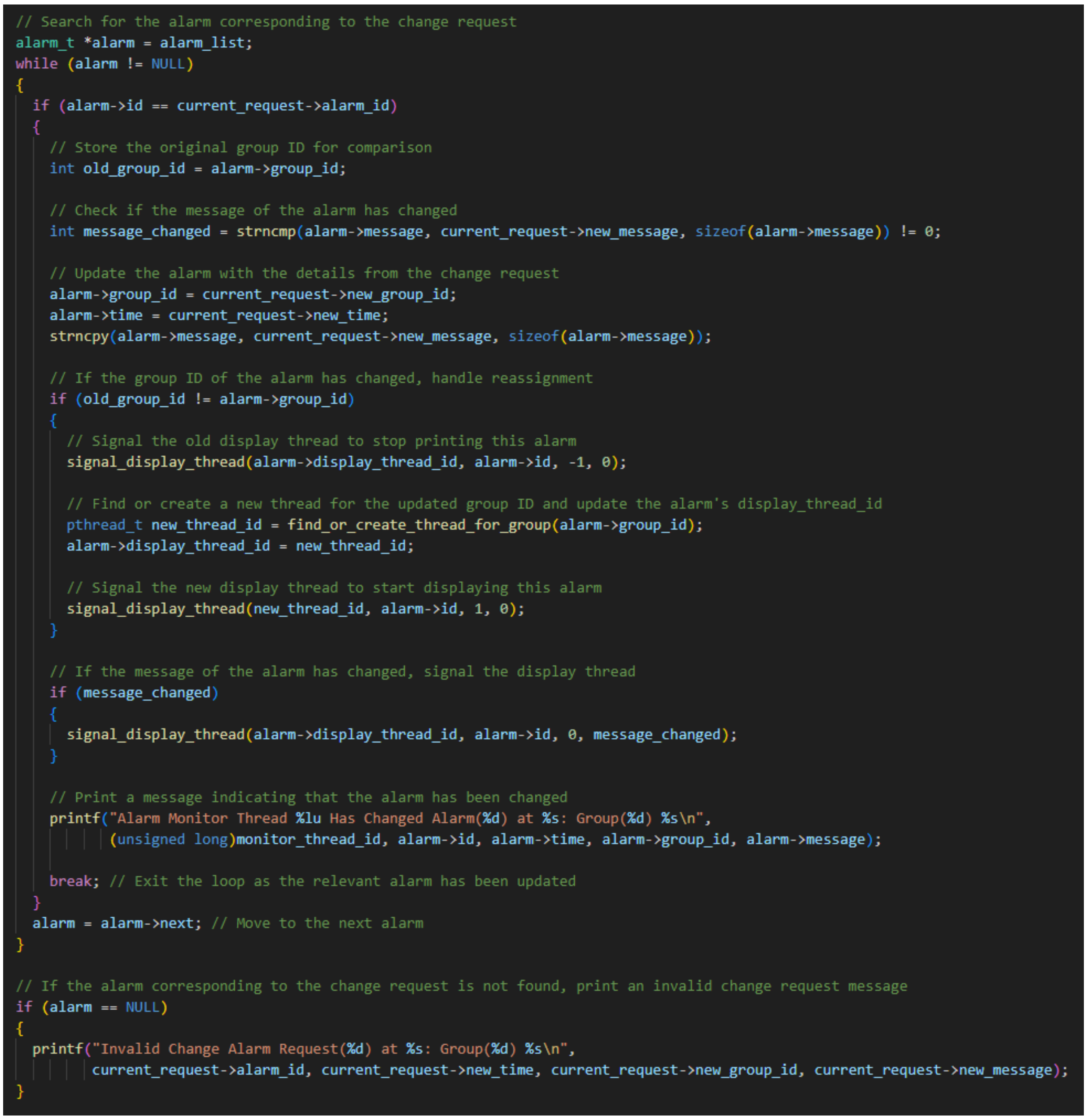


```c
if (!thread_created)
{
    // Allocate memory for a new thread node
    thread_node_t *new_thread_info = malloc(sizeof(thread_node_t));
    if (new_thread_info == NULL)
    {
        errno_abort("Allocate thread"); // Handle allocation error
    }

    // Initialize the new thread node
    new_thread_info->group_id = group_id; // Set the group ID
    new_thread_info->alarm_count = 0;
    new_thread_info->alarm_queue = NULL;
    pthread_mutex_init(&new_thread_info->queue_mutex, NULL);
    pthread_cond_init(&new_thread_info->queue_cond, NULL);

    // Create a new display alarm thread
    pthread_create(&new_thread, NULL, display_alarm_thread_function, new_thread_info);
    // new_thread_info->thread_id = new_thread;

    pthread_mutex_lock(&display_alarm_thread_list_mutex);
    add_thread_node(&display_alarm_thread_list, new_thread, group_id);
    pthread_mutex_unlock(&display_alarm_thread_list_mutex);

    new_alarm->display_thread_id = new_thread; // Update thread ID in alarm

    char time_str[30]; // Ensure this buffer is large enough
    strftime(time_str, sizeof(time_str), "%H:%M:%S", localtime(&(new_alarm->time)));

    printf("Main Thread Created New Display Alarm Thread %lu For Alarm(%d) at %s: Group(%d) %s\n\n",
           (unsigned long)new_thread, new_alarm->id, time_str, new_alarm->group_id, new_alarm->message);
}
```

A3.3 The alarm monitor thread in "New_Alarm_Cond.c"

A.3.3.1 For Each Change_Alarm Request

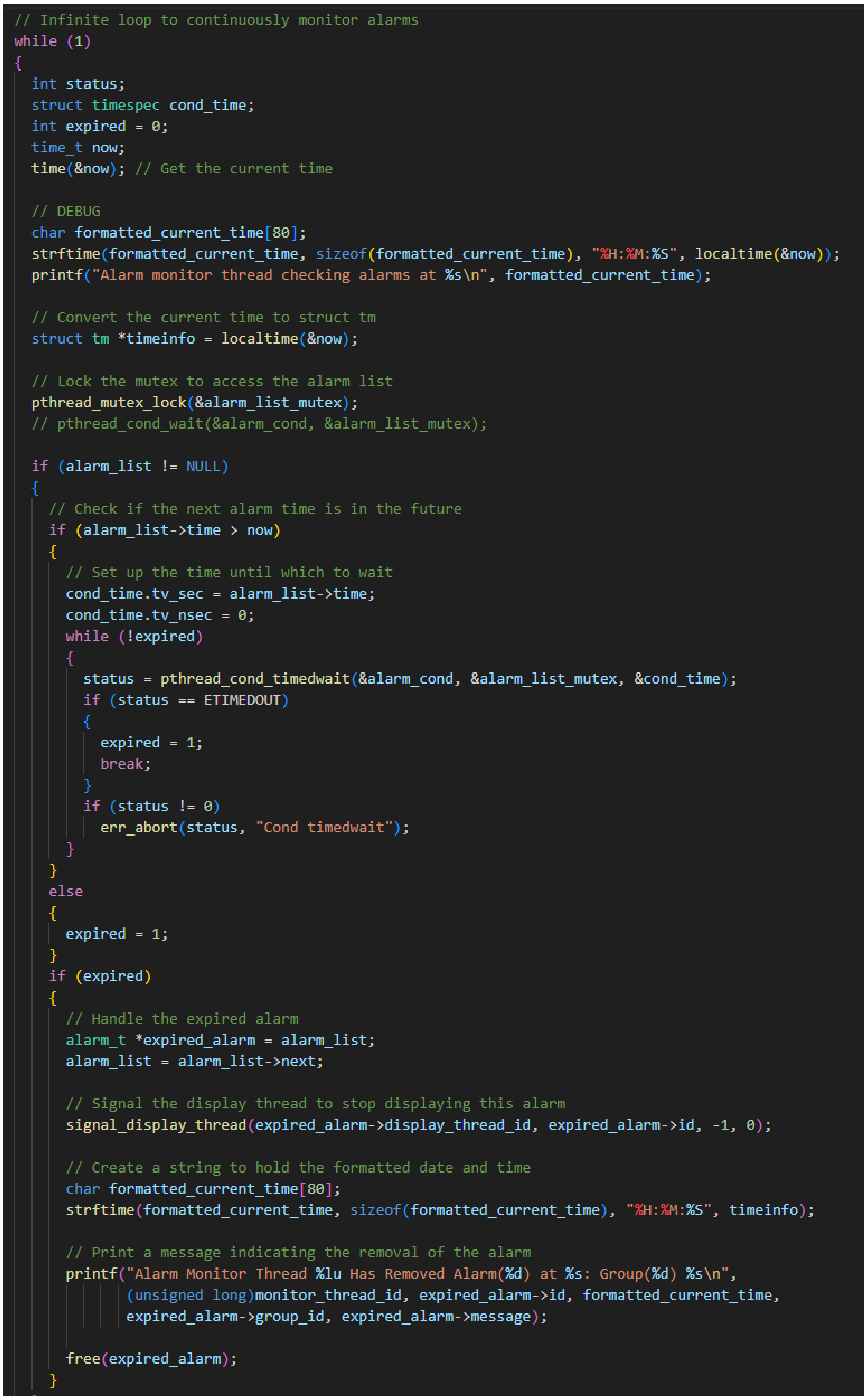
Iterate through the separate change alarm request list to find alarms with corresponding Alarm_IDs in the main alarm list. If a match is found, update alarm details in the main list using Group_ID, Time, and Message values from the Change_Alarm request. Logging actions taken by the alarm monitor thread, such as successful updates or detection of invalid Change_Alarm requests. Finally, remove processed Change_Alarm requests from the separate list.



```c
// Search for the alarm corresponding to the change request
alarm_t *alarm = alarm_list;
while (alarm != NULL)
{
    if (alarm->id == current_request->alarm_id)
    {
        // Store the original group ID for comparison
        int old_group_id = alarm->group_id;

        // Check if the message of the alarm has changed
        int message_changed = strncmp(alarm->message, current_request->new_message, sizeof(alarm->message)) != 0;

        // Update the alarm with the details from the change request
        alarm->group_id = current_request->new_group_id;
        alarm->time = current_request->new_time;
        strncpy(alarm->message, current_request->new_message, sizeof(alarm->message));

        // If the group ID of the alarm has changed, handle reassignment
        if (old_group_id != alarm->group_id)
        {
            // Signal the old display thread to stop printing this alarm
            signal_display_thread(alarm->display_thread_id, alarm->id, -1, 0);

            // Find or create a new thread for the updated group ID and update the alarm's display_thread_id
            pthread_t new_thread_id = find_or_create_thread_for_group(alarm->group_id);
            alarm->display_thread_id = new_thread_id;

            // Signal the new display thread to start displaying this alarm
            signal_display_thread(new_thread_id, alarm->id, 1, 0);
        }

        // If the message of the alarm has changed, signal the display thread
        if (message_changed)
        {
            signal_display_thread(alarm->display_thread_id, alarm->id, 0, message_changed);
        }

        // Print a message indicating that the alarm has been changed
        printf("Alarm Monitor Thread %lu Has Changed Alarm(%d) at %s: Group(%d) %s\n",
               (unsigned long)monitor_thread_id, alarm->id, alarm->time, alarm->group_id, alarm->message);

        break; // Exit the loop as the relevant alarm has been updated
    }
    alarm = alarm->next; // Move to the next alarm
}

// If the alarm corresponding to the change request is not found, print an invalid change request message
if (alarm == NULL)
{
    printf("Invalid Change Alarm Request(%d) at %s: Group(%d) %s\n",
           current_request->alarm_id, current_request->new_time, current_request->new_group_id, current_request->new_message);
}
```

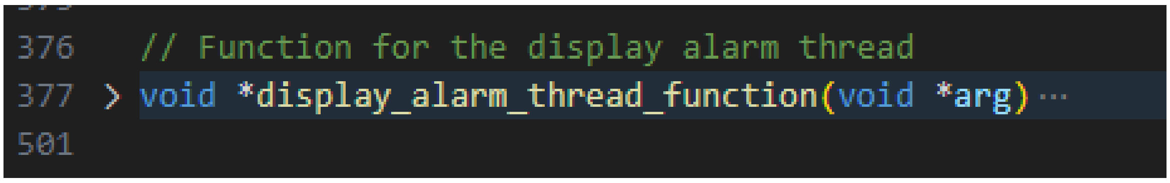
A.3.3.2 (listed as 3.3.3 in assignment description) For Each Alarm in List

The alarm monitor thread inspects each alarm in the alarm list to determine whether its expiry time has been reached. Upon identifying an alarm whose expiry time has passed, it will be removed from the list and the system will print a message confirming the removal of the expired alarm.



```c
// Infinite loop to continuously monitor alarms
while (1)
{
  int status;
  struct timespec cond_time;
  int expired = 0;
  time_t now;
  time(&now); // Get the current time

  // DEBUG
  char formatted_current_time[80];
  strftime(formatted_current_time, sizeof(formatted_current_time), "%H:%M:%S", localtime(&now));
  printf("Alarm monitor thread checking alarms at %s\n", formatted_current_time);

  // Convert the current time to struct tm
  struct tm *timeinfo = localtime(&now);

  // Lock the mutex to access the alarm list
  pthread_mutex_lock(&alarm_list_mutex);
  // pthread_cond_wait(&alarm_cond, &alarm_list_mutex);

  if (alarm_list != NULL)
  {
    // Check if the next alarm time is in the future
    if (alarm_list->time > now)
    {
      // Set up the time until which to wait
      cond_time.tv_sec = alarm_list->time;
      cond_time.tv_nsec = 0;
      while (!expired)
      {
        status = pthread_cond_timedwait(&alarm_cond, &alarm_list_mutex, &cond_time);
        if (status == ETIMEDOUT)
        {
          expired = 1;
          break;
        }
        if (status != 0)
          err_abort(status, "Cond timedwait");
      }
    }
    else
    {
      expired = 1;
    }
    if (expired)
    {
      // Handle the expired alarm
      alarm_t *expired_alarm = alarm_list;
      alarm_list = alarm_list->next;

      // Signal the display thread to stop displaying this alarm
      signal_display_thread(expired_alarm->display_thread_id, expired_alarm->id, -1, 0);

      // Create a string to hold the formatted date and time
      char formatted_current_time[80];
      strftime(formatted_current_time, sizeof(formatted_current_time), "%H:%M:%S", timeinfo);

      // Print a message indicating the removal of the alarm
      printf("Alarm Monitor Thread %lu Has Removed Alarm(%d) at %s: Group(%d) %s\n",
             (unsigned long)monitor_thread_id, expired_alarm->id, formatted_current_time,
             expired_alarm->group_id, expired_alarm->message);

      free(expired_alarm);
    }
```

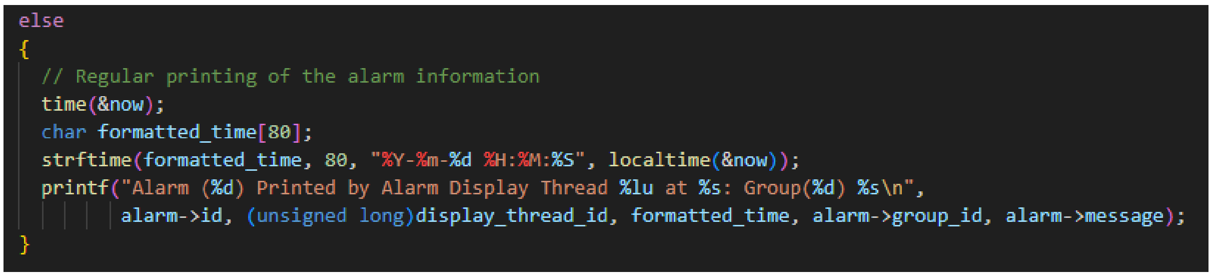
A3.4 The display alarm_threads in “New_Alarm_Mutex.c”



```c
// Function for the display alarm thread
void *display_alarm_thread_function(void *arg)···
```

A3.4.1 Concurrent display of alarm

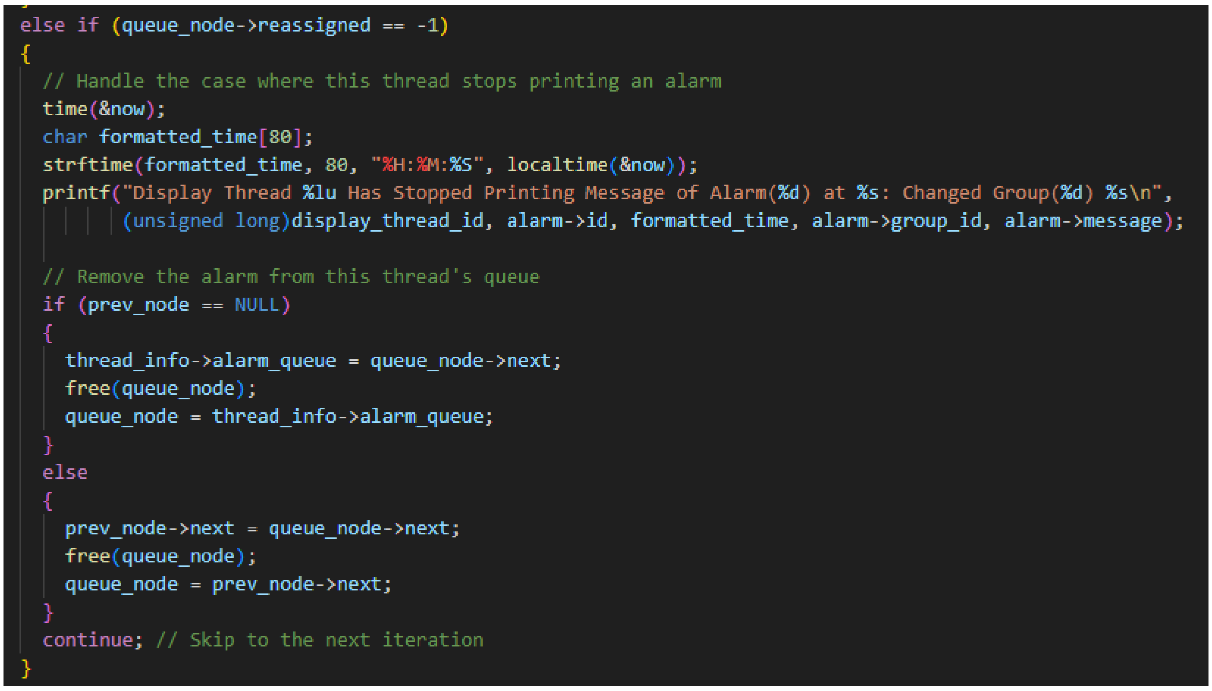
Upon creation by the main thread, each display alarm thread is designated for handling alarms with a specific Group ID. The code uses the display_alarm_thread_function() function that executes every 5 seconds. This function scans and prints alarm details associated with the specific Group ID assigned to that display alarm thread.



```c
else
{
  // Regular printing of the alarm information
  time(&now);
  char formatted_time[80];
  strftime(formatted_time, 80, "%Y-%m-%d %H:%M:%S", localtime(&now));
  printf("Alarm (%d) Printed by Alarm Display Thread %lu at %s: Group(%d) %s\n",
         alarm->id, (unsigned long)display_thread_id, formatted_time, alarm->group_id, alarm->message);
}
```

A.3.4.2 If Alarm has been removed

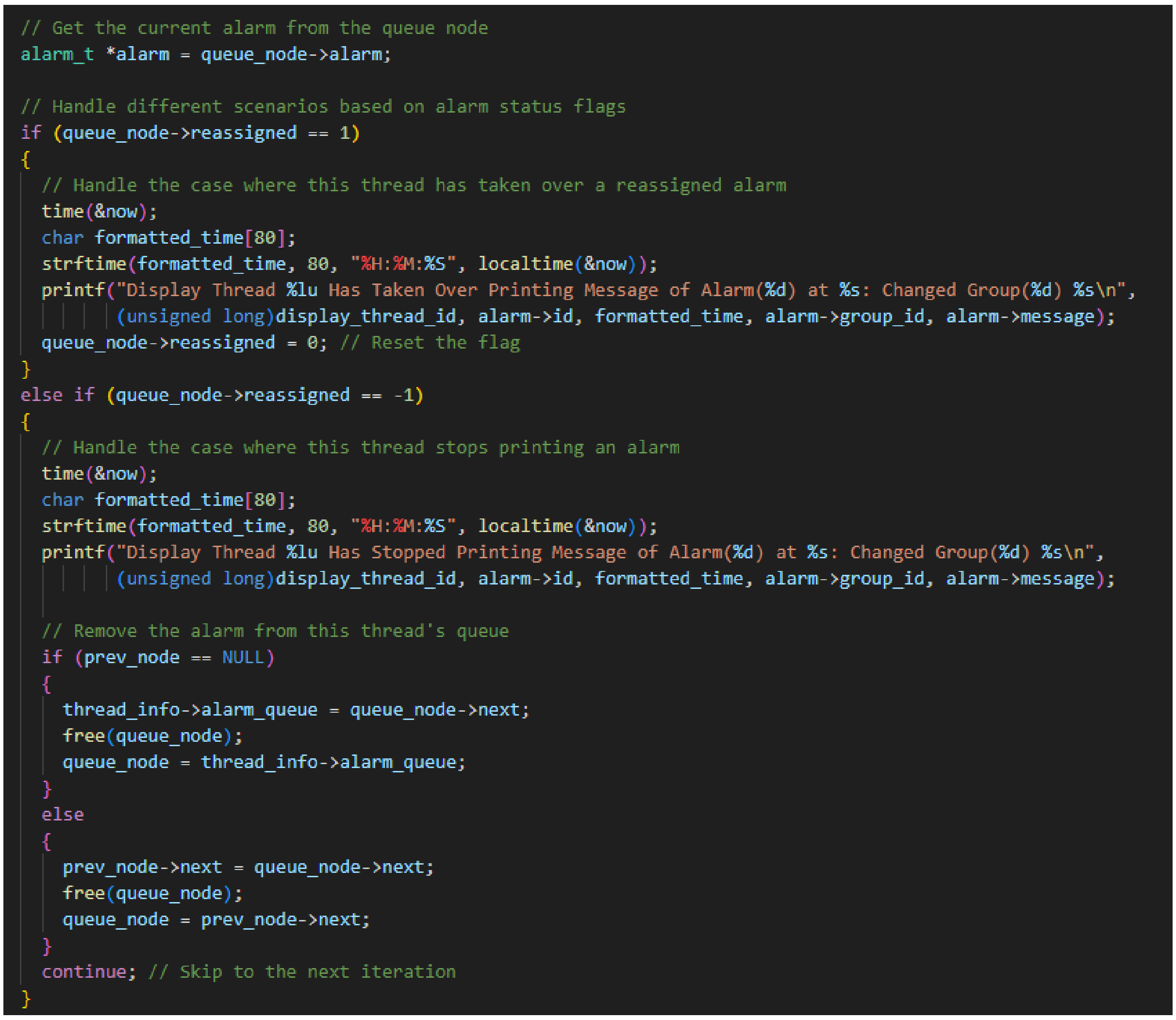
The display alarm thread continuously monitors for any removal of alarms from the alarm list associated with its specific Group ID. Upon detecting the removal of an alarm linked to its Group ID, the display alarm thread stops printing messages related to that specific alarm. The display alarm thread then generates a message to log this action.



```c
else if (queue_node->reassigned == -1)
{
  // Handle the case where this thread stops printing an alarm
  time(&now);
  char formatted_time[80];
  strftime(formatted_time, 80, "%H:%M:%S", localtime(&now));
  printf("Display Thread %lu Has Stopped Printing Message of Alarm(%d) at %s: Changed Group(%d) %s\n",
         (unsigned long)display_thread_id, alarm->id, formatted_time, alarm->group_id, alarm->message);

  // Remove the alarm from this thread's queue
  if (prev_node == NULL)
  {
    thread_info->alarm_queue = queue_node->next;
    free(queue_node);
    queue_node = thread_info->alarm_queue;
  }
  else
  {
    prev_node->next = queue_node->next;
    free(queue_node);
    queue_node = prev_node->next;
  }
  continue; // Skip to the next iteration
}
```

A.3.4.3 If Group ID has been Changed

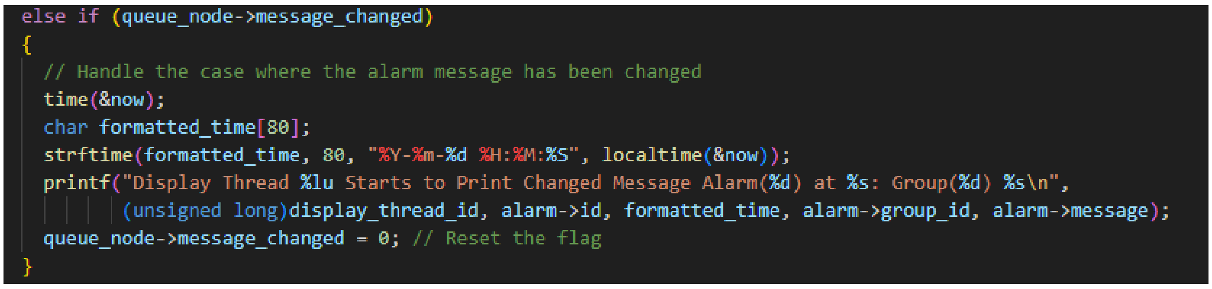
The system constantly monitors changes in Group ID for alarms within the alarm list. Upon detection of a Group ID change for an alarm, the display alarm thread previously responsible for that Group ID stops printing messages for the associated alarm. The system logs this action and outputs this message. Simultaneously, another display alarm thread starts printing messages for the alarm's updated Group ID. The newly responsible display alarm thread logs its takeover message.



```c
// Get the current alarm from the queue node
alarm_t *alarm = queue_node->alarm;

// Handle different scenarios based on alarm status flags
if (queue_node->reassigned == 1)
{
  // Handle the case where this thread has taken over a reassigned alarm
  time(&now);
  char formatted_time[80];
  strftime(formatted_time, 80, "%H:%M:%S", localtime(&now));
  printf("Display Thread %lu Has Taken Over Printing Message of Alarm(%d) at %s: Changed Group(%d) %s\n",
         (unsigned long)display_thread_id, alarm->id, formatted_time, alarm->group_id, alarm->message);
  queue_node->reassigned = 0; // Reset the flag
}
else if (queue_node->reassigned == -1)
{
  // Handle the case where this thread stops printing an alarm
  time(&now);
  char formatted_time[80];
  strftime(formatted_time, 80, "%H:%M:%S", localtime(&now));
  printf("Display Thread %lu Has Stopped Printing Message of Alarm(%d) at %s: Changed Group(%d) %s\n",
         (unsigned long)display_thread_id, alarm->id, formatted_time, alarm->group_id, alarm->message);

  // Remove the alarm from this thread's queue
  if (prev_node == NULL)
  {
    thread_info->alarm_queue = queue_node->next;
    free(queue_node);
    queue_node = thread_info->alarm_queue;
  }
  else
  {
    prev_node->next = queue_node->next;
    free(queue_node);
    queue_node = prev_node->next;
  }
  continue; // Skip to the next iteration
}
```

A.3.4.4 For Newly Changed Alarms in the Alarm List

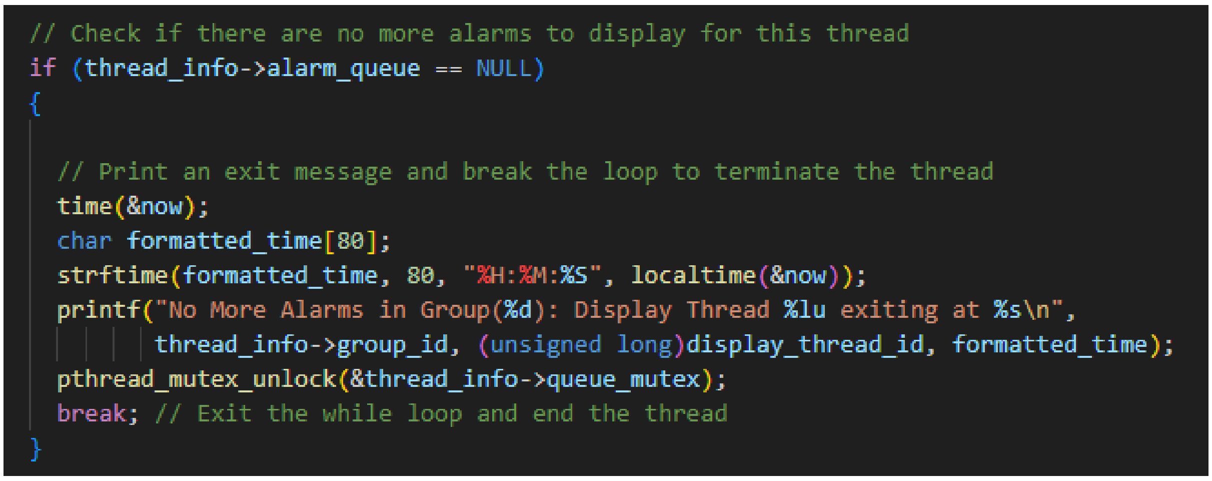
The system continuously monitors changes made exclusively to the message content within alarms without any changes to the Group ID. Upon detecting a message change within an alarm without a Group ID change, the display alarm thread responsible for the associated Group ID begins periodic printing of the updated message. The system generates logs this change and outputs it.



```c
else if (queue_node->message_changed)
{
    // Handle the case where the alarm message has been changed
    time(&now);
    char formatted_time[80];
    strftime(formatted_time, 80, "%Y-%m-%d %H:%M:%S", localtime(&now));
    printf("Display Thread %lu Starts to Print Changed Message Alarm(%d) at %s: Group(%d) %s\n",
           (unsigned long)display_thread_id, alarm->id, formatted_time, alarm->group_id, alarm->message);
    queue_node->message_changed = 0; // Reset the flag
}
```

A.3.4.5 No More Alarms to be Displayed

Each display alarm thread continuously monitors its associated group in the alarm list to determine if no more alarms exist within that specific group. Upon detecting no more alarms within its associated group, the display alarm thread generates a log to output. Then, the thread terminates its execution.



```c
// Check if there are no more alarms to display for this thread
if (thread_info->alarm_queue == NULL)
{

    // Print an exit message and break the loop to terminate the thread
    time(&now);
    char formatted_time[80];
    strftime(formatted_time, 80, "%H:%M:%S", localtime(&now));
    printf("No More Alarms in Group(%d): Display Thread %lu exiting at %s\n",
           thread_info->group_id, (unsigned long)display_thread_id, formatted_time);
    pthread_mutex_unlock(&thread_info->queue_mutex);
    break; // Exit the while loop and end the thread
}
```

A.3.4.6 Display alarm threads should not change any information in any data structure that the display alarm threads share with any other threads.

- Non-modification of Shared Data Structures:
    - Display alarm threads use a read-only policy when accessing shared data structures.
    - Threads are designed only to observe and extract information from these structures without initiating any modifications, which ensures data integrity and prevents potential conflicts or inconsistencies.
- Exclusive Read Access:
    - Display alarm threads are programmed to access shared data structures in a manner that guarantees they do not alter or tamper with the data
    - This approach maintains the stability and reliability of the shared data, where multiple threads can safely access information without compromising the integrity of the data structures.

A3.5 Treating synchronization of the thread accesses to the alarm list and change alarm request list as solving a "Readers-Writers" problem in "New_Alarm_Cond.c"

A.3.5.1 Mutex-based synchronization mechanisms

- By using mutex locks, only one writer process can modify either the alarm list or the change alarm list at a time.
- Reader processes acquire and release mutex locks allowing for simultaneous access to read information from the alarm list and change alarm list.
- This concurrent access makes for an efficient retrieval of data without introducing conflicts among threads accessing data.

Quality of the Design

- ➔ The code is well documented for the reader to understand the functionality easily.
- ➔ The code uses the grouping mechanism which is based on the time.
- ➔ The code is modular which means it is divided into specific tasks which know their functionalities and only do the specific task which will be given by the user.

Testing details

1. Test creating new alarm.

- Sample input:
  - Start_Alarm(2345): Group(13) 50 One
- Intended behaviour:
  - The request is legal according with the requirements specifications and therefore will be inserted into the alarm list. After the specified time alarm time has passed, the alarm will be removed from the alarm list.
- Produced output:
  - Inserting new alarm with ID 2345 and group ID 13
  - Display alarm thread 123145529004032 woke up for processing
  - Main Thread Created New Display Alarm Thread 123145529004032 For Alarm(2345) at 22:28:44: Group(13) One
  - Alarm(2345) Inserted by Main Thread 140704651035840 Into Alarm List at 22:28:44: Group(13) One
  - Alarm> Signaling display thread with ID 123145529004032 for alarm ID 2345
  - Alarm Monitor Thread 123145528467456 Has Removed Alarm(2345) at 22:27:55: Group(13) One

2. Test adding another alarm with the same alarm_id.

- Sample input:
  - Start_Alarm(2345): Group(13) 10 One
  - Start_Alarm(2345): Group(10) 5 Two
- Intended behaviour:
  - The first input is legal and will be inserted into the alarm list.

    - The second input, however, is not allowe, and the program should discard the atempt
    - After the specified time alarm time has passed, the alarm will be removed from the list.
- Produced output:
    - Alarm> Start_Alarm(2345): Group(13) 10 One
    - Inserting new alarm with ID 2345 and group ID 13
    - Display alarm thread 123145457070080 woke up for processing
    - Main Thread Created New Display Alarm Thread 123145457070080 For Alarm(2345) at 22:36:23: Group(13) One
    - Alarm(2345) Inserted by Main Thread 140704651035840 Into Alarm List at 22:36:23: Group(13) One
    - Alarm> Start_Alarm(2345): Group(10) 5 Two
    - Alarm with ID 2345 already exists. Ignoring command.
    - Alarm> Signaling display thread with ID 123145457070080 for alarm ID 2345
    - Alarm Monitor Thread 123145456533504 Has Removed Alarm(2345) at 22:36:14: Group(13) One

3. Test changing the alarm when none exist.

- Sample input:
    - Change_Alarm(2345): Group(10) 5 One
- Intended behaviour:
    - The system should not allow the following request, and notify the user of the illegal request
- Produced output:
    - Due to bugs in the code, the output to the sample input could not be produced.

4. Test changing the alarm.

- Sample input:
    - Start_Alarm(2345): Group(13) 10 One
    - Change_Alarm(2345): Group(10) 5 Two
- Intended behaviour:
    - The alarm from the first input will be inserted into the alarm list.
    - The second input, should change the time of the previous alarm to the new value.
- Produced output:
    - Due to bugs in the code, the output to the sample input could not be produced.

Problems faced during the code implementation:

- ➔ The code was very long which made our debugging work more complicated.
- ➔ When the deadlocks are on during the debugging stage the system freezes itself and has to do manual debugging.
- ➔ The copy and paste function in the terminal is too complicated because for paste we usually press ctrl + V but in the terminal every time we have to use right-click and paste.
- ➔ The code is too long which makes our documentation work more complicated.
- ➔ While coding the logic the use of the linked list is more complicated than we thought.

- ➔ There were instances where pointer access was incorrect, leading to compilation errors and segmentation faults.
- ➔ Memory allocation for structures like alarm_t and display_alarm_thread_t was not handled correctly.
- ➔ The code exhibited segmentation faults during execution.