POSIX Threads Assignment Report

Stefan Smolovic, Davyd Zinkiv, Ryan Luk, Viral Patel

EECS3221

Professor Xu

November 13, 2023

Note: We have used the files provided in the “Assignment 2 additional material Folder” folder on eClass. This includes the alarm_mutex.c and error.h file.

Introduction

The purpose of this report is to elaborate on the design and implementation of an alarm-setting program using POSIX threads. This alarm program can handle multiple alarms concurrently, utilizing threads to manage alarm execution and display. The report will cover the program's design and implementation, as well as the challenges we faced.

Design Overview

1. Program Structure

The alarm system program is structured into several key components:

- Alarm Structure (alarm_t): Represents individual alarms, storing details like alarm ID, time, message, and group number.
- Display Thread Structure (display_thread_t): Represents threads responsible for displaying alarms, storing thread IDs and associated group numbers.
- Main Thread: Handles user input, parses commands, and manages alarm list and display threads.
- Display Alarm Threads: Responsible for displaying alarms at specified intervals.

2. Thread Synchronization

To ensure thread-safe operations, two mutexes were used:

- Alarm Mutex (alarm_mutex): Protects access to the alarm list.
- Display Thread Mutex (display_thread_mutex): Protects access to the display thread list.

3. Alarm Time Grouping

- Alarms are grouped based on their time values, facilitating efficient thread management and minimizing the number of threads.

Implementation Details

This section will discuss changes we have made and the new functionalities we have added.

A3.1

(A) To create an alarm, use the command format: Alarm> Start_Alarm(Alarm_ID): Time Message.

This command will set an alarm to trigger after the specified Time and repeat at intervals of Time seconds indefinitely. For instance, in the example provided, an alarm is set to go off in 10 seconds, and it will continue to do so every 10 seconds, repeating this cycle without end.

```
ptlb25 316 % make
gcc -o a.out new_alarm_mutex.c -D_POSIX_PTHREAD_SEMANTICS -lpthread -lm
./a.out
alarm> Start_Alarm(2): 10 ten second timer
Created New Display Alarm Thread 139747058632448 for Alarm_Time_Group_Number 2 to Display Alarm(2) at 1699908637: ten second timer
Alarm(2) Inserted by Main Thread 139747070650176 Into Alarm List at 1699908637: ten second timer
alarm> Alarm (2) Printed by Alarm Thread 139747058632448 for Alarm_Time_Group_Number 2 at 1699908637: ten second timer
Alarm (2) Printed by Alarm Thread 139747058632448 for Alarm_Time_Group_Number 2 at 1699908647: ten second timer
Alarm (2) Printed by Alarm Thread 139747058632448 for Alarm_Time_Group_Number 2 at 1699908657: ten second timer
ptlb25 317 %
```

(B) To modify an existing alarm, use the command: Alarm> Replace_Alarm(Alarm_ID): Time Message. This allows you to update the settings of a previously set alarm. For example, in the provided instance, an alarm originally set to trigger every 10 seconds has been updated to go off every 15 seconds instead.

```
ptlb25 319 % make
gcc -o a.out new_alarm_mutex.c -D_POSIX_PTHREAD_SEMANTICS -lpthread -lm
./a.out
alarm> Start_Alarm(2): 10 ten second timer
Created New Display Alarm Thread 139952895371008 for Alarm_Time_Group_Number 2 to Display Alarm(2) at 1699908818: ten second timer
Alarm(2) Inserted by Main Thread 139952907388736 Into Alarm List at 1699908818: ten second timer
alarm> Alarm (2) Printed by Alarm Thread 139952895371008 for Alarm_Time_Group_Number 2 at 1699908818: ten second timer
Alarm (2) Printed by Alarm Thread 139952895371008 for Alarm_Time_Group_Number 2 at 1699908828: ten second timer
Alarm (2) Printed by Alarm Thread 139952895371008 for Alarm_Time_Group_Number 2 at 1699908838: ten second timer
Alarm (2) Printed by Alarm Thread 139952895371008 for Alarm_Time_Group_Number 2 at 1699908848: ten second timer
Replace_Alarm(2): 15 changed alarm two
Alarm(2) Replaced at 1699908855: 15 changed alarm two
Created New Display Alarm Thread 139952886978304 for Alarm_Time_Group_Number 3 to Display Alarm(2) at 1699908855: changed alarm two
Display Alarm Thread for Alarm_Time_Group_Number 2 Terminated at 1699908855
alarm> Alarm (2) Printed by Alarm Thread 139952886978304 for Alarm_Time_Group_Number 3 at 1699908870: changed alarm two
Alarm (2) Printed by Alarm Thread 139952886978304 for Alarm_Time_Group_Number 3 at 1699908885: changed alarm two
Alarm (2) Printed by Alarm Thread 139952886978304 for Alarm_Time_Group_Number 3 at 1699908900: changed alarm two
Alarm (2) Printed by Alarm Thread 139952886978304 for Alarm_Time_Group_Number 3 at 1699908915: changed alarm two
ptlb25 320 %
```

(C) To cancel an existing alarm, use the command: Alarm> Cancel_Alarm(Alarm_ID). This command effectively deactivates the specified alarm based on its Alarm_ID.

```
ptlb25 321 % make
gcc -o a.out new_alarm_mutex.c -D_POSIX_PTHREAD_SEMANTICS -lpthread -lm
./a.out
alarm> Start_Alarm(2): 10 ten second timer
Created New Display Alarm Thread 139935174608640 for Alarm_Time_Group_Number 2 to Display Alarm(2) at 1699909065: ten second timer
Alarm(2) Inserted by Main Thread 139935186626368 Into Alarm List at 1699909065: ten second timer
alarm> Alarm (2) Printed by Alarm Thread 139935174608640 for Alarm_Time_Group_Number 2 at 1699909065: ten second timer
Alarm (2) Printed by Alarm Thread 139935174608640 for Alarm_Time_Group_Number 2 at 1699909075: ten second timer

alarm> Alarm (2) Printed by Alarm Thread 139935174608640 for Alarm_Time_Group_Number 2 at 1699909085: ten second timer

alarm> Alarm (2) Printed by Alarm Thread 139935174608640 for Alarm_Time_Group_Number 2 at 1699909095: ten second timer

alarm> Cancel_Alarm(2)
Alarm(2) Canceled at 1699909100: ten second timer
Display Alarm Thread for Alarm_Time_Group_Number 2 Terminated at 1699909100
```

A3.2 This part of the assignment talks about how we can calculate the **Alarm_Time_Group_Number**. We have created a function **get_group_number** in our design to return an integer value of time. The function uses "ceil" to group them in 5 seconds of buckets. The main function uses the get_group_number to find out the group value whenever the Start, Replace, or Cancel function is called.

```c
// Function to calculate the group number of an alarm based on its time.
int get_group_number(int time)
{
    return (int)ceil((double)time / 5.0); // Calculate and return the ceiling of time divided by 5.
}
```

A.3.3.1 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.

```c
// Parse input line into command format for Start_Alarm.
if (sscanf(line, "Start_Alarm(%d): %d %63[^\n]", &alarm->alarm_id, &alarm->seconds, alarm->message) == 3)
{

    // Lock the mutex to ensure thread-safe access to the shared alarm list.
    // This is important to prevent concurrent access issues.
    status = pthread_mutex_lock(&alarm_mutex);
    if (status != 0)
    {
        err_abort(status, "Lock mutex"); // Abort if mutex lock fails.
    }

    // Calculate the alarm time (current time plus the specified seconds).
    // This determines when the alarm should trigger.
    alarm->time = time(NULL) + alarm->seconds;
    alarm->next_display_time = alarm->time; // Set the next display time to the alarm time initially.

    // Calculate the alarm's group number based on its time,
    // grouping alarms into buckets of 5 seconds each.
    alarm->alarm_time_group_number = (int)ceil((double)alarm->seconds / 5.0);

    // Insert the new alarm into the alarm list in a sorted order by time.
    last = &alarm_list; // Start at the head of the list.
    next = *last;       // Initialize the next pointer for traversal.

    // Traverse the list to find the correct insertion point.
    while (next != NULL)
    {
        // If we find an alarm with a time greater or equal to the new alarm,
        // insert the new alarm before it to maintain sorted order.
        if (next->alarm_id >= alarm->alarm_id)
        {
            alarm->link = next; // Point new alarm to the next one in the list.
            *last = alarm;      // Insert the new alarm into the list.
            break;              // Break as we have inserted the alarm.
        }
        // Move to the next alarm in the list.
        last = &next->link;
        next = next->link;
    }
```

```
    // If we reached the end of the list, append the new alarm at the end.
    if (next == NULL)
    {
        *last = alarm;      // Set the last alarm's link to the new alarm.
        alarm->link = NULL; // The new alarm is now the last one, so its link is NULL.
    }

    // Unlock the alarm list mutex so that manage_display_threads can access the alarm list.
    pthread_mutex_unlock( & alarm_mutex); // Unlock the alarm list mutex.
    if (status != 0) {
        err_abort(status, "Unlock mutex");
    }

    // Print a confirmation message indicating successful insertion.
    printf("Alarm(%d) Inserted by Main Thread %lu Into Alarm List at %ld: %d %s\n",
        alarm->alarm_id, (unsigned long)main_thread_id, alarm->time, alarm->seconds, alarm->message);

    // After inserting the alarm into the list, manage display threads for this group number
    manage_display_threads(alarm->alarm_time_group_number, alarm->alarm_id, alarm->time);
}
```

A.3.3.2 :

After the start request is given, it checks whether the alarm group number exists or not if not then it creates a new display thread and uses the mange_display_thread function.

```
if (!exists) {
    pthread_t new_thread;
    int *time_group = malloc(sizeof(int));
    *time_group = group_number;
    int status = pthread_create(&new_thread, NULL, display_alarm_thread, time_group);
    if (status != 0) {
        err_abort(status, "Create display alarm thread");
    }

    display_thread_t *new_display_thread = (display_thread_t *)malloc(sizeof(display_thread_t));
    if (new_display_thread == NULL) {
        errno_abort("Allocate display thread");
    }
    new_display_thread->thread_id = new_thread;
    new_display_thread->time_group_number = group_number;
    new_display_thread->next = display_thread_list;
    display_thread_list = new_display_thread;

    // Print the confirmation along with the alarm message.
    printf("Created New Display Alarm Thread %lu for Alarm_Time_Group_Number %d to Display Alarm(%d) at %ld: %d %s\n",
        (unsigned long)new_thread, group_number, alarm_id, now, current_alarm->seconds, alarm_message);
}
```

A.3.3.3 Replacing the Alarm Request Process

This part of the code identifies the user replace request with a parameter of Alarm group number and Alarm_ID. It replaces the value of the alarm with the specified Alarm_ID provided by the user. If the display alarm group does not exist then it creates a new thread and if no other alarm in the group exists it terminates the corresponding display threads.

```c
else if (sscanf(line, "Replace_Alarm(%d): %d %63[^\n]", &alarm->alarm_id, &alarm->seconds, alarm->message) == 3)
{
    int found = 0;           // Flag to check if the alarm is found in the list.
    time_t now = time(NULL); // Get the current time.

    // Lock the mutex to ensure exclusive access to the alarm list.
    status = pthread_mutex_lock(&alarm_mutex);
    if (status != 0)
    {
        err_abort(status, "Lock mutex");
    }

    // Iterate through the alarm list to find the alarm to be replaced.
    for (last = &alarm_list;
        (next = *last) != NULL; last = &next->link)
    {
        if (next->alarm_id == alarm->alarm_id)
        {                                                          // Check if the alarm IDs match.
            int old_group_number = next->alarm_time_group_number;              // Store old group number for later use.
            next->alarm_time_group_number = get_group_number(alarm->seconds);  // Recalculate the group number.
            next->seconds = alarm->seconds;                                    // Update the seconds.
            next->time = now + alarm->seconds;                                 // Update the alarm time.
            next->next_display_time = next->time;                              // Update the next display time.
            strncpy(next->message, alarm->message, sizeof(next->message) - 1); // Copy the new message.
            next->message[sizeof(next->message) - 1] = '\0';                   // Ensure null termination.
            found = 1;                                                         // Set the found flag.

            // Print confirmation that the alarm has been replaced.
            printf("Alarm(%d) Replaced at %ld: %d %s\n",
                alarm->alarm_id, now, alarm->seconds, alarm->message);

            // Unlock the alarm list mutex so that terminate_display_thread_if_empty can access the alarm list.
            pthread_mutex_unlock(&alarm_mutex); // Unlock the alarm list mutex.
            if (status != 0)
            {
                err_abort(status, "Unlock mutex");
            }

            // Manage display threads for the new group number and check if old threads need to be terminated.
            manage_display_threads(next->alarm_time_group_number, alarm->alarm_id, now);
            terminate_display_thread_if_empty(old_group_number, now);
            break; // Break the loop as the alarm has been found and replaced.
        }
    }

    // Unlock the mutex after modifications.
    // status = pthread_mutex_unlock( & alarm_mutex);
    /*
    if (status != 0){
        err_abort(status, "Unlock mutex");
    }
    */

    // If the alarm ID was not found in the list, print an error message.
    if (!found)
    {
        fprintf(stderr, "Replace_Alarm: No alarm found with ID %d.\n", alarm->alarm_id);
    }
    free(alarm); // Free the memory allocated for the alarm.
}
```

A.3.3.4 Cancel Alarm Request Process.

It part of the code identifies the user request and if the request says “cancel alarm” then it stops the corresponding thread. Moreover, if no alarm in the group exists after cancellation, it terminates the tread.

```
else if (sscanf(line, "Cancel_Alarm(%d)", &user_alarm_id) == 1)
{
    alarm_t *current, *prev = NULL; // Pointers to traverse and keep track of previous alarm.
    int found = 0;                  // Flag to check if the alarm is found in the list.
    int group_number = -1;          // To store the group number of the cancelled alarm.

    // Lock the mutex to ensure exclusive access to the alarm list.
    status = pthread_mutex_lock(&alarm_mutex);
    if (status != 0)
    {
        err_abort(status, "Lock mutex");
    }

    // Iterate through the alarm list to find and remove the specified alarm.
    current = alarm_list;
    while (current != NULL)
    {
        if (current->alarm_id == user_alarm_id)
        {                                               // Check if the alarm IDs match.
            found = 1;                                  // Set the found flag.
            group_number = current->alarm_time_group_number; // Store the group number.
            if (prev == NULL)
            {
                alarm_list = current->link; // Remove the alarm from the list.
            }
            else
            {
                prev->link = current->link; // Remove the alarm from the middle or end of the list.
            }
            time_t cancel_time = time(NULL); // Get the current time for the cancellation message.
            printf("Alarm(%d) Canceled at %ld: %d %s\n", user_alarm_id, cancel_time, current->seconds, current->message);
            free(current); // Free the memory allocated for the alarm.
            break;         // Break the loop as the alarm has been found and cancelled.
        }
        prev = current;          // Update the previous pointer.
        current = current->link; // Move to the next alarm in the list.
    }

    // Unlock the mutex after modifications.
    pthread_mutex_unlock(&alarm_mutex);
    if (status != 0)
    {
        err_abort(status, "Unlock mutex");
    }

    // If the alarm ID was not found in the list, print an error message.
    if (found)
    {
        // Check if any other alarm exists in the same group and manage display threads accordingly.
        terminate_display_thread_if_empty(group_number, time(NULL));
    }
    else
    {
        fprintf(stderr, "Cancel_Alarm: No alarm found with ID %d.\n", user_alarm_id);
    }
    free(alarm); // Free the memory allocated for the alarm structure, even if not used.
}
```

A.3.4 Display Alarm Tread Process.

The display_alarm_thread function is used as the entry point for display alarm threads, and these threads are created and managed in other parts of the code, such as the manage_display_threads function. The threads continuously check for alarms in their assigned time groups and display them as needed.

```c
// Thread function for display alarm threads.
void *display_alarm_thread(void *arg)
{
    int time_group_number = *((int *)arg);
    free(arg);

    while (1)
    {
        pthread_mutex_lock(&alarm_mutex);

        time_t now = time(NULL);
        alarm_t *current = alarm_list;
        while (current != NULL)
        {
            if (current->alarm_time_group_number == time_group_number)
            {
                if (now >= current->next_display_time)
                {
                    printf("Alarm (%d) Printed by Alarm Thread %lu for Alarm_Time_Group_Number %d at %ld: %d %s\n",
                        current->alarm_id, (unsigned long)pthread_self(), time_group_number, now, current->seconds, current->message);
                    current->next_display_time = now + current->seconds; // Set the next display time.
                }
            }
            current = current->link;
        }

        pthread_mutex_unlock(&alarm_mutex);
        sleep(1); // Sleep for a second before checking again.
    }
    return NULL;
}
```

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.Start_Alarm:

Testing creating a new alarm:

The request is legal and therefore will be inserted into the alarm list. Then the message will be printed in the following format “Alarm(<alarm_id>) Inserted by Main Thread <thread-id> Into Alarm List at <insert_time>: <time message>". Currently, there are no display alarm threads for the Alarm_Time_Group_Number associated with the alarm existing in the system. The main thread will create a new display alarm thread for the specified Alarm_Time_Group_Number (in this case the group number is ceil(n/5). It will output a message in the following format “Created New Display Alarm Thread <thread-id> for

Alarm_Time_Group_Number<Alarm_Time_Group_Number> to Display Alarm(<alarm_id>) at <create_time>: <time message>". Every 10 seconds, the thread will print the message in the following format: "Alarm (<alarm_id>) Printed by Alarm Thread <thread-id> > for Alarm_Time_Group_Number <Alarm_Time_Group_Number> at <time>: <time message> ".

2.Cancel_Alarm:

Canceling existing alarm:
There exists the alarm with input id, therefore the program is expected to print a message in the following format: "Alarm(<alarm_id>) Cancelled at <cancel_time>: <time message>". Since no other alarms are having same Alarm_Time_Group_Number as the cancelled alarm (it was the only alarm in the alarm list), the main thread will terminate the display alarm thread associated with that Alarm_Time_Group_Number and will output in the following format "Display Alarm Thread <thread-id> for Alarm_Time_Group_Number <Alarm_Time_Group_Number> Terminated at <cancel_time>", and will no longer display updated about the alarm.

Canceling non-existent alarm:
Because there is no active alarm with input id, some meaningful message is printed to the console.

3. Replace_Alarm:

Replacing existing alarm
There exists the alarm with input id, therefore the program is expected to loop through the linked list to find this alarm and change the time and message values to the corresponding inputs, then finally print a message in the following format "Alarm(<alarm_id>) Replaced at <replace_time>: <time message>".

Replacing non-existent alarm
Because there is no active alarm with input id, some meaningful message is printed to the console.

4. Invalid Input:
Expected output - Bad command or format. Discarded: <input>

5. 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.
- The most challenging part was ensuring the correct synchronization between threads, especially when accessing and modifying shared data structures.

Conclusion

The design and implementation of the POSIX threads-based alarm system successfully demonstrate the handling of concurrent alarms using multithreading. The challenges encountered, particularly in synchronization and thread management, provided valuable insights into the complexities of multithreaded programming. The testing process was thorough, ensuring the reliability and robustness of the program.