EECS3221 Section E

Operating System Fundamentals

Fall 2023

Assignment 2 Posix Threads

Team Members:

John Yacoub (217988908)

Yvan Semana (217099425)

Sathira Williams (218131938)

Mikhail Malinovskiy(218759977)

November 12 2023

Professor Jia Xu

Table of Contents

Abstract

This assignment requires the implementation of an alarm program to satisfy the requirements, using POSIX Threads and data synchronization to preserve data consistency.

1. - Implementation Requirements

Create a separate C File called New_Alarm_Mutex.C.

1.1 - Alarm Requests

The program will feature three commands:

1. Start_Alarm: Creates a new alarm.
   Format: "*Alarm> Start_Alarm(Alarm_ID): Time Alarm_Category Message*"
   Alarm_ID - Unique identifier for the alarm.
   Time - Duration of how long to be alarmed.
   Message - Message displayed on alarm.

2. Replace_Alarm: Replaces pre existing alarm.
   Format: "*Replace_Alarm(Alarm_ID): Time Alarm_Category Message*"
   Alarm_ID - Unique identifier for the alarm.
   Time - Duration of how long to be alarmed.
   Message - Message displayed on alarm.

3. Cancel_Alarm: Removes Alarm.
   Format: "*Cancel_Alarm(Alarm_ID)*"
   Alarm_ID - Unique identifier for the alarm.
   Time - Duration of how long to be alarmed.

1.2 - Alarm Time Groups

Alarm requests will be sorted into categories based on their time values. Each category will be divided into multiples of five.

Example:

- Alarms with a "Time" value from 1 to 5 belong to group 1.
- Alarms with a "Time" value from 6 to 10 belong to group 2.
- Alarms with a "Time" value from 11 to 15 belong to group 3.
- And so on…

1.3 - The Main Thread

The main responsibilities of the main thread are two fold:

1. The main thread checks if the format of the incoming alarm requests match the specified format, if not prompt an error.
2. If the message of the alarm is over 128 characters, truncate it.

1.3.1. Handling Start_Alarm Requests

1. **Determine Group Number:** For each Start_Alarm request, calculate the corresponding "*Alarm_Time_Group_Number"* based on the *Time* value provided.
2. **Insert Alarm:** Insert the alarm into an ordered list based on the *Alarm_ID.*
3. **Print: "***Alarm(<alarm_id>) Inserted by Main Thread <thread-id> Into Alarm List at <insert_time>: <time message>"* on completion.

1.3.2. Managing Display Threads for New Alarms

1. **Check for Existing Threads:** Check for pre existing display threads that match the "*Alarm_Time_Group_Number".*
2. **Create Thread if Needed:** If one does not exist for the "*Alarm_Time_Group_Number"* then create one for it.
3. **Print:** *"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>"* on the confirmation on creating a new display thread.

1.3.3. Handling Replace_Alarm Requests

1. **Update Alarm:** It updates the "*Time*" and message for the alarm specified by "*Alarm_ID*" and recalculates the "*Alarm_Time_Group_Number*" if needed.
2. **Print 1:** *"Alarm(<alarm_id>) Replaced at <replace_time>: <time message>"* is printed after successfully replacing the alarm.
3. **Print 2:** *"Created New Display Alarm Thread <thread-id> for Alarm_Time_Group_Number <Alarm_Time_Group_Number> to Display Alarm(<alarm_id>) at <replace_time>: <time message>"* is printed if there is no corresponding *Alarm_Time_Group_Number* in the display threads.
4. **Print 3: "***Display Alarm Thread <thread-id> for Alarm_Time_Group_Number <Alarm_Time_Group_Number> Terminated at <replace_time>"* is printed if after **Update Alarm,** there is no more alarms in the alarm list for the associated *Alarm_Time_Group_Number,* resulting in the termination of that group numbers display thread.

1.3.4. Handling Cancel_Alarm Requests

1. **Remove Alarm:** Removes the alarm from the alarm list.
2. **Remove Displaying Message:** The display thread stops showing the message for the canceled alarm.
3. **Print 1:** *"Alarm(<alarm_id>) Canceled at <cancel_time>: <time message>"* is printed at **Remove Alarm**.
4. **Print 2:** *"Display Alarm Thread <thread-id> for Alarm_Time_Group_Number <Alarm_Time_Group_Number> Terminated at <cancel_time>"* is printed if no other alarms exist for the associated *Alarm_Time_Group_Number,* it will then terminate the display thread.

1.4 - The Display Thread

After being created by the main thread; the display thread is responsible for the following:

1. Periodically checking the alarms inside the alarm list for the *Alarm_Time_Group_Number* that the alarm thread is responsible for.
2. For every alarm in the alarm list that is relevant to that display thread; it will print every *Time* seconds.
   a. **Print:** *"Alarm (<alarm_id>) Printed by Alarm Thread <thread-id> > for Alarm_Time_Group_Number <Alarm_Time_Group_Number> at <time>: <time message>"*

2. System Interface

2.1 - Constraints

In the program, we ensure users can input commands continuously, choosing from three options: Start, Replace, and Cancel. To ensure data consistency and prevent race conditions, the program employs a combination of flags and mutex locks. These locks are essential for safely managing concurrent access to shared resources, thereby maintaining the integrity of data throughout the execution of these commands.

2.2 - New_Alarm_Mulex

2.2.1. Variable and Struct Declarations

Struct 1 - *display_alarm_info:* Defines a struct, that will be used to
Struct 2 - *alarm_tag:* Defines a struct, that will be used to
pthread_mutex_t alarm_mutex: A data type used in POSIX threads to create a mutual exclusion, which prevents multiple threads from simultaneous execution in the critical section.
pthread_cond_t alarm_cond: Another data type used in POSIX threads, which is used to create a condition variable. It will block the thread from accessing shared data until the condition is met.
Variable 1 - *alarm_t:* Linked list of alarms, each time a command is entered the structure dynamically allocates details to store/free information of the command like AlarmID, Time, and Message.
Variable 2 - *display_alarm_info_t:* Pointer to the display threads.
Variable 3 - *num_display_reading:* Number of threads that are currently reading.

2.2.2. Function Declarations

Function 1 - *display_alarm:* A thread function that displays the alarms in the specific Alarm_Time_Group_Number.
Function 2 - *create_or_check_display_alarm_thread:* Creates or checks display alarm thread for a specific Alarm_Time_Group_Number.
Function 3 - *insert_alarm:* Inserts the alarm into the list, while maintaining the order of AlarmID's. If the same AlarmID exists already, it is not inserted and a message is printed.
Function 4 - *check_or_remove_display_thread:* Check and possibly remove a display alarm if it is no longer responsible for any alarms in that Alarm_Time_Group_Number.
Function 5 - *replace_alarm:* Replaces a pre existing alarm with a new one, as long as its AlarmID exists in the alarm list. If none is found a message is printed.

2.2.3. - New_Alarm_Mulex Interface

```
/*
 * New_Alarm_Mutex.h
 *
 * Header file containing function and data structure declarations
 * for the New_Alarm_Mutex.c program.
 */
#ifndef ALARM_MUTEX_H
#define ALARM_MUTEX_H

#include "errors.h"
#include <pthread.h>
#include <time.h>

// Define a structure to pass the condition variable and alarm group
to each
// display thread
typedef struct {
  int alarm_group;
  pthread_cond_t condition;
} ThreadArguments;

// Define a structure to store information about display alarm threads
typedef struct display_alarm_info {
  pthread_t thread;
  int alarm_time_group;
  int alarms_in_group;
  pthread_cond_t condition;        // Condition variable for signaling
  struct display_alarm_info *next; // Pointer to the next thread in
the list
} display_alarm_info_t;

// Define a data structure to store information about each alarm
typedef struct alarm_tag {
  struct alarm_tag *link;
  int alarm_id;
  int seconds;
  time_t time; /* seconds from EPOCH */
  char message[128];
} alarm_t;

// Mutex for managing the alarm list
pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;

// Alarm list and current_alarm
alarm_t *alarm_list;

// List of display threads and how many threads are currently reading
display_alarm_info_t *display_alarm_threads;
int num_display_reading = 0;

// Mutex for num_display_reading, to keep track of reading threads
pthread_mutex_t display_alarm_mutex = PTHREAD_MUTEX_INITIALIZER;

// Function declarations

/**
 * @brief Cleanup handler function for display alarm threads.
 *
 * This function is responsible for unlocking the alarm mutex in case
the
 * display alarm thread exits while its locked. It decrements the
count of
 * display threads, and if it is the last display thread, it unlocks
the alarm
 * mutex to allow other threads to proceed. The cleanup handler is
registered
 * with the pthread_cleanup_push function in the display_alarm
function.
 *
 * @param arg Unused parameter (required by pthread_cleanup_push).
 */
void cleanup_handler(void *arg);

/**
 * @brief The display alarm thread function that displays alarms in
its group.
 *
 * This function is responsible for displaying alarms associated with
a specific
 * Alarm_Time_Group_Number. It continuously checks for alarms in the
specified
 * group and displays them when it's their time.
 *
 * @param arg A pointer to a structure containing the group number and
condition
 * variable for the display thread.
 */
void *display_alarm(void *arg);

/**
 * @brief Creates or checks a display alarm thread for a specific
 * Alarm_Time_Group_Number.
 *
 * This function checks if a display alarm thread for a particular
 * Alarm_Time_Group_Number already exists. If it exists, it
increments the
 * count of alarms in that group and signals that thread for an
update. If it
 * doesn't exist, it creates a new display alarm thread and updates
the display
 * alarm threads list.
 *
 * @param alarm_group The group number to create or find a thread
 * for.
 * @param alarm The alarm to be associated with the display thread.
 */
void create_or_check_display_alarm_thread(int alarm_group, alarm_t
*alarm);

/**
 * @brief Check and potentially remove a display alarm thread if
it's no longer
 * responsible for any alarms.
 *
 * This function checks whether a display alarm thread associated
with a
 * specific group number is responsible for any remaining alarms in
 * the group. If there are no alarms left in the group, it
terminates the thread
 * and removes it from the display alarm threads data structure.
 *
 * @param alarm_group The group number the display thread is
responsible for.
 */
void check_or_remove_display_thread(int alarm_group);

/**
 * @brief Insert an alarm into the list, maintaining order by alarm
ID.
 *
 * This function inserts a new alarm into the list while maintaining
the order
 * of alarms based on their alarm IDs. If an alarm with the same ID
already
 * exists, the new alarm is not inserted, and a message is printed.
 *
 * @param alarm A pointer to the new alarm to insert.
 */
void insert_alarm(alarm_t *alarm);

/**
 * @brief Replace an existing alarm with a new one based on the same
alarm ID.
 *
 * This function replaces an existing alarm with a new alarm based
on the same
 * alarm ID. If an alarm with the same ID exists in the alarm list,
it updates
 * the existing alarm's information. If no such alarm is found, it
prints a
 * message indicating that the alarm with the specified ID was not
found.
 *
 * @param alarm A pointer to the new alarm that will replace the
existing one.
 */
void replace_alarm(alarm_t *alarm);

/**
 * @brief Cancel an alarm with the specified ID.
 *
 * This function cancels an alarm with the specified alarm ID. It
searches the
 * alarm list for the alarm with the given ID, removes it from the
list, and
 * terminates any associated display alarm thread if it's no longer
responsible
 * for any alarms in the same group.
 *
 * @param alarm_id The ID of the alarm to be canceled.
 */
void cancel_alarm(int alarm_id);

/**
 * @brief Main function for the New_Alarm_Mutex program.
 *
 * This function continuously reads user input and performs actions
based on the
 * input commands. It implements a command-line interface for
interacting with
 * the alarm management system to create, replace or cancel alarms.
 *
 */
int main(int argc, char *argv[]);

#endif // NEW_ALARM_MUTEX_H
```

2.3 - The Main Thread

2.3.1. Main Thread Responsibilities

The main thread will be responsible for the following:

- Variable initialization:
  - int status, char line[128], alarm_t *alarm, pthread_t thread
- Starts infinite loop, takes continuous input from the stdin.
- Allows for three possible commands:
  - *Start_Alarm(<u>Alarm_ID</u>): <u>Time</u> Alarm_Category <u>Message</u>*
  - *Replace_Alarm(<u>Alarm_ID</u>): <u>Time</u> Alarm_Category <u>Message</u>*
  - *Cancel_Alarm(<u>Alarm_ID</u>)*
- Once either of the commands are triggered, it calls its corresponding functions:
  - insert_alarm, cancel_alarm, and replace_alarm.

2.3.2. Main Thread Interface

Below contains the code for the main thread of the function, where the commands are inputted from the stdin, and associated with the correct commands.

```
int main(int argc, char *argv[]) {
  int status;
  char line[128];
  alarm_t *alarm;
  pthread_t thread;

  while (1) {
    printf("Alarm>");
    if (fgets(line, sizeof(line), stdin) == NULL)
      exit(0);
    if (strlen(line) <= 1) {
      continue;
    }
    // Allocate new alarm
    alarm = (alarm_t *)malloc(sizeof(alarm_t));
    if (alarm == NULL)
      errno_abort("Allocate alarm");

    /*
     * Parse input line into alarm_id (%d), seconds (%d),
and a message
     * (%128[^\n]), consisting of up to 128 characters
separated by
     * whitespace.
     */
    // COMMAND 1: Start_Alarm
    if (sscanf(line, "Start_Alarm(%d) %d %128[^\n]",
&alarm->alarm_id,
               &alarm->seconds, alarm->message) == 3) {
      if (alarm->alarm_id > 0 && alarm->seconds > 0) {
        // Valid alarm_id and seconds, proceed with adding
the alarm

        // Insert the new alarm into the list of alarms,
sorted by alarm id
        insert_alarm(alarm);
      } else {
        // Invalid alarm_id or seconds
        if (alarm->alarm_id <= 0) {
          fprintf(stderr, "Alarm ID must be greater than
0\n");
        }
        if (alarm->seconds <= 0) {
          fprintf(stderr, "Alarm time must be greater than
0\n");
        }
        // Free the invalid alarm
        free(alarm);
      }
    }
// COMMAND 2: Replace_Alarm
    else if (sscanf(line, "Replace_Alarm(%d) %d
%128[^\n]", &alarm->alarm_id,
                    &alarm->seconds, alarm->message) == 3)
{
      if (alarm->alarm_id > 0 && alarm->seconds > 0) {
        // Valid alarm_id and seconds, proceed with
replacing the alarm
        replace_alarm(alarm);
        free(alarm); // Free dataholder alarm
      } else {
        // Invalid alarm_id or seconds
        if (alarm->alarm_id <= 0) {
          fprintf(stderr, "Alarm ID must be greater than
0\n");
        }
        if (alarm->seconds <= 0) {
          fprintf(stderr, "Alarm time must be greater than
0\n");
        }
      }
    }
    // COMMAND 3: Cancel_Alarm
    else if (sscanf(line, "Cancel_Alarm(%d)",
&alarm->alarm_id) == 1) {
      if (alarm->alarm_id > 0) {
        cancel_alarm(alarm->alarm_id);
      } else {
        fprintf(stderr, "Invalid alarm ID. Please enter a
non-negative ID.\n");
      }
    } else {
      fprintf(stderr, "Bad command\n");
      free(alarm); // Free the invalid alarm
    }
    fflush(stdout); // Manually flush the stdout buffer
  }
}
```

2.4 - The Display Alarm Thread

2.4.1. Display Alarm Responsibilities

The display thread will be responsible for the following:

- Variable initialization:
    - ***ThreadArguments *thread_args***: Used to access thread-specific arguments like the alarm group and condition variable.
    - ***int alarm_group***: Determines the group of alarms the thread is responsible for.
    - ***pthread_cond_t condition***: The condition variable used for timed waiting.
    - ***alarm_t *curr_alarm; *closest_alarm***: Pointers to navigate the closest alarm to display.
    - ***time_t closest_expiration_time***: Stores the expiration time of the alarm closest to triggering.
- Starts an infinite loop, continuously looking for alarms to display.
- Mutex Locking:
    - Locks the display_alarm_mutex, to safely increment the num_display_reading, which is responsible for tracking the number of threads reading the alarm list.
- Alarm Identification:
    - Iterates through the alarm_list to find the alarm that belongs to its alarm_group.
- Timed Waiting:
    - Calculates the time to sleep, until the closest alarm is due.
    - Uses the pthread_cond_timedwait to sleep until the calculated time, or until it's signaled by another thread (*ex. When a new alarm is created*).
- Alarm Display:
    - Once woken from the sleep function, it checks if the closest alarm's time has come.
        - If yes, display the alarm information and update the alarms time to the current time.
        - If not, continue monitoring; entering another iteration of the loop.
- Mutex Unlocking:
    - Locks display_alarm_mutex again to safely decrement num_display_reading.
    - If it is the last display thread reading, it unlocks alarm_mutex.

2.4.2. Display Alarm Interface

Below contains the code for the main thread of the function, where the commands are inputted from the stdin, and associated with the correct commands.

```
void *display_alarm(void *arg) {
  // Extract thread arguments
  ThreadArguments *thread_args = (ThreadArguments *)arg;
  int alarm_group = thread_args->alarm_group;
  pthread_cond_t condition = thread_args->condition;
  free(arg);

  // Initialize condition variable
  pthread_cond_init(&condition, NULL);
  sleep(1); // Allow time for initialization, since alarm
time >=1

  while (1) {
    // Register a cleanup handler to unlock the mutex
    pthread_cleanup_push(cleanup_handler, NULL);

    // Increment the count of display threads
    pthread_mutex_lock(&display_alarm_mutex);
    num_display_reading++;

    // If this is the first display thread, lock the alarm
(writer) mutex
    if (num_display_reading == 1) {
      pthread_mutex_lock(&alarm_mutex);
    }
    pthread_mutex_unlock(&display_alarm_mutex);

    // Variables to find the closest alarm for display
    alarm_t *curr_alarm = alarm_list;
    alarm_t *closest_alarm = NULL;
    time_t closest_expiration_time = 0;

    // Iterate through the alarm list to find the closest
alarm
    while (curr_alarm != NULL) {
      if ((curr_alarm->seconds + 4) / 5 == alarm_group) {
        time_t expiration_time = curr_alarm->time +
curr_alarm->seconds;
        if (closest_alarm == NULL ||
            expiration_time < closest_expiration_time) {
          closest_alarm = curr_alarm;
          closest_expiration_time = expiration_time;
        }
      }
      curr_alarm = curr_alarm->link;
    }

    // Calculate the time to sleep until the closest alarm
    int sleep_seconds = closest_expiration_time -
time(NULL);

    // Continue the loop to recheck the list if the
condition was not met
    while (sleep_seconds > 0) {
      // Use the passed condition variable for timed_wait
      struct timespec sleep_time = {sleep_seconds, 0};

      // Perform timed wait, or unless signaled
      int wait_result =
          pthread_cond_timedwait(&condition, &alarm_mutex,
&sleep_time);

// Recheck the list after waking up from the wait
      curr_alarm = alarm_list;

      if (wait_result == ETIMEDOUT) {
        // Break out of the loop when the condition is met
and time is expired
        break;
      }

      // Recalculate sleep time
      sleep_seconds = closest_expiration_time -
time(NULL);
    }

    // If the closest alarm's time has expired, display it
    if (closest_expiration_time <= time(NULL) &&
closest_alarm != NULL) {
      // Update the added time to the current time
      closest_alarm->time = time(NULL);

      // Display the alarm information
      printf("Alarm(%d) Displayed by Display Thread %lu
for "
             "Alarm_Time_Group_Number %d at %ld: %s\n",
             closest_alarm->alarm_id, pthread_self(),
alarm_group, time(NULL),
             closest_alarm->message);
    }

    // Reading is done, pop the cleanup handler, unlocking
the mutex
    pthread_cleanup_pop(1);
  }

  // Cleanup, destroy the condition variable
  pthread_cond_destroy(&condition);

  return NULL;
}
```

3. Challenges

The design of display threads involves addressing the concurrency challenge of managing shared resources, particularly the alarm list, in a multi-threaded environment. This can be modeled as a Readers-Writers problem, where multiple display threads act as readers accessing the shared alarm list concurrently, while the main thread functions as the writer, modifying the list.

3.1 Issues Arisen

Race Conditions: A race condition is when multithreaded code would try to access a shared resource, causing the output to create unintended results.

- When the Alarm List is being changed from the Main Thread, while the Display Thread is trying to read it, inconsistencies can happen.
    - Example: If the main thread is trying to input an alarm into the alarm list, and at the same time the display thread tries to read the alarm list, it may use outdated information or even cause the program to crash.

Starvation: Starvation occurs when the two or more threads that are allocated to the CPU take excessive time in execution, causing other threads not able to be processed.

- If the main thread has to wait for a long time to get the lock from the display thread, if the display thread is doing constant reading. Then this will lead to the case where alarms aren't replaced, canceled, or inserted into the alarm list, and could potentially display outdated or incorrect information to the user.

Timeliness: Traditionally when an alarm is inserted the display thread is usually sleeping. Leading in a delay to updating the alarm list till the display thread is on its next cycle.

- Example: The display thread is currently asleep. And the user inputs a request to Start a new alarm. The main thread inserts the alarm into the alarm list. However since the display thread is still asleep no changes occur till the next cycle of the display thread.

4. System Design

The challenges encountered in designing display threads, particularly in managing the shared alarm list in a multi-threaded environment, closely resemble the Readers-Writers problem. The simultaneous access by multiple display threads as readers and the main thread as the writer corresponds to this classic concurrency problem. The issues identified, such as race conditions during alarm list modifications, potential starvation of the main thread, and delays in updating due to display thread sleeping, underscore the need for a structured synchronization approach. A solution to this problem also incorporates a condition variable for each thread alongside using a reader-writers approach. This mechanism allows the main thread to signal the completion of its write operation, enabling display threads to safely proceed with their reads and preventing unintended inconsistencies. Additionally, it addresses concerns of thread starvation by providing a means to prioritize the main thread when necessary. The use of condition variables ensures a timely response to changes, minimizing delays in updating the alarm list during the display thread's sleep cycles. In essence, the challenges at hand align with the characteristics of a Readers-Writers problem, and employing condition variables for synchronization emerges as a suitable solution.

4.1 - Readers-Writers Model

Shared Resource:
The shared resource in this context is the alarm list, which contains information about the alarms scheduled for display. Display threads act as readers when they periodically check and display alarms from the list. The main thread serves as the writer, updating the list when new alarms are created, existing alarms are replaced, or alarms are canceled.

Readers:
- Display threads act as readers, concurrently accessing the alarm list to display relevant alarm information.
- Multiple display threads can read from the list simultaneously, ensuring efficient parallel processing.

Writers:
- The main thread acts as the writer, acquiring exclusive access to the alarm list when modifying its contents.
- Modifications include inserting new alarms, replacing existing alarms, and canceling alarms.
- The use of mutex locks ensures that only one thread (writer) can modify the list at a time, preventing data inconsistencies.

4.2 - Synchronization Mechanism

4.2.1 - Mutex Locks:

- To address the Readers-Writers problem, mutex locks are employed to synchronize access to the alarm list.

- Display threads lock and unlock a display mutex (display_alarm_mutex) to keep track of the number of threads currently reading the alarm list.
- When at least one alarm is reading from the list, the writing mutex (alarm_mutex) is locked ensuring mutual exclusion among writers and readers.
- When there are no display threads currently reading/going through the alarm list (the display threads are sleeping) the writing mutex is unlocked allowing the main thread access to the alarm list.
- The main thread locks the mutex when modifying the list, ensuring exclusive access during write operations.

4.2.2 - Condition Variables:

- Condition variables are used for efficient signaling between the main thread and display threads.
- Display threads perform timed waits on their own conditional variable, allowing them to sleep until new alarms are added to the list or an alarm the thread is responsible for expired and must have its message printed.
- At a given point in time a display thread is either sleeping for a time period until the nearest alarm message needs to be printed or is actively checking the list for an updated nearest time.
- The main thread signals the display thread's conditional variable when new alarms are inserted, prompting that particular display thread to wake up and process the alarms.
- Using different condition variables allows the main thread to signal a particular display thread (to wake up from sleep) without waking up all threads to save reading time and access to the alarm list.

4.3 - Benefits

Using this approach not only minimizes reading time but also ensures that users experience minimal wait times when adding a new command. By utilizing a mutex and an efficient update mechanism, the system allows the main thread to actively update the list without delaying user commands. Importantly, only the relevant display thread is alerted and updated, optimizing responsiveness. This design ensures that users can seamlessly interact with the system, adding commands without noticeable delays while maintaining effective synchronization of the shared alarm list. Timely updates using the condition variables are crucial for maintaining the accuracy of information within the alarm list. When the main thread modifies alarms, relevant display threads are promptly alerted to update their sleep times, ensuring users receive up-to-date information without unnecessary delays. Using condition variables to signal the completion of write operations and to update the display thread minimizes the risk of race conditions, ensuring consistent access to the alarm list and preventing unintended consequences from simultaneous reads and writes. Additionally, the incorporation of timed waits using condition variables enables prioritization of the main thread, preventing potential delays and ensuring execution of critical write operations, thus avoiding potential starvation scenarios.

5. Test Cases

5.1 - Scenarios

In output.txt, the scenarios were tested, and the exact outputs were placed there.

Scenario 1: Testing Start Alarm Insert
Inputs:

- **Time 0 Seconds:** Start_Alarm(1) 10 Message_1

Expected Output:

- **Time 0 Seconds:** "*Alarm(1) Inserted by Main Thread (Thread_ID) Into Alarm List at (Time from Epoch): 10 Message_1*"
- **Time 0 Seconds:** "*Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 2 to Display Alarm(1) at (Epoch Time): 10 Message_1*"
- **Time Every 10 Seconds:** "*Alarm(1) Displayed by Display Thread (Thread_ID) for Alarm_Time_Group_Number 2 at (Epoch Time): Message_1*"

Actual Output:

```
Alarm>Start_Alarm(1) 10 Message_1
Alarm(1) Inserted by Main Thread 140703149411200 Into Alarm List at 1699907185: 10 Message_1
Created New Display Alarm Thread 140703149405760 for Alarm_Time_Group_Number 2 to Display
Alarm(1) at 1699907185: 10 Message_1
Alarm>Alarm(1) Displayed by Display Thread 140703149405760 for Alarm_Time_Group_Number 2
at 1699907195: Message_1
```

Scenario 2: Testing Cancel Alarm Deletion
Inputs:

- **Time 0 Seconds:** Start_Alarm(1) 30 Message_1
- **Time 5 Seconds:** Cancel_Alarm(1)

Expected Output:

- **Time 0 Seconds:** "*Alarm(1) Inserted by Main Thread (Thread_ID) Into Alarm List at (Time from Epoch): 30 Message_1*"
- **Time 5 Seconds:** "*Alarm(1) Canceled at (Epoch Time): 30 Message_1 Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 6 Terminated at (Epoch Time)*"

Actual Output:

```
Alarm>Start_Alarm(1) 30 Message_1
Alarm(1) Inserted by Main Thread 139707421580160 Into Alarm List at 1699907226: 30 Message_1
Created New Display Alarm Thread 139707421574720 for Alarm_Time_Group_Number 6 to Display
Alarm(1) at 1699907226: 30 Message_1
Alarm>Cancel_Alarm(1)
Alarm(1) Canceled at 1699907232: 30 Message_1
Display Alarm Thread 139707421574720 for Alarm_Time_Group_Number 6 Terminated at
1699907232
```

Scenario 3: Testing Replace Alarm
Inputs:

- **Time 0 Seconds:** Start_Alarm(1) 30 Message_1
- **Time 5 Seconds:** Replace_Alarm(1) 10 Message_1

Expected Output:

- **Time 0 Seconds:** "*Alarm(1) Inserted by Main Thread (Thread_ID) Into Alarm List at (Epoch Time): 30 Message_1*"
- **Time 0 Seconds:** "*Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 6 to Display Alarm(1) at (Epoch Time): 30 Message_1*"
- **Time 5 Seconds:** *"Alarm(1) Replaced at (Epoch Time): 10 Message_1 Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 2 to Display Alarm(1) at (Epoch Time): 10 Message_1"*
- **Time 15 Seconds:** *"Alarm(1) Displayed by Display Thread (Thread_ID for Alarm_Time_Group_Number 2 at (Epoch Time): Message_1"*

Actual Output:

```
Alarm>Start_Alarm(1) 30 Message_1
Alarm(1) Inserted by Main Thread 140190907804544 Into Alarm List at 1699907280: 30 Message_1
Created New Display Alarm Thread 140190907799104 for Alarm_Time_Group_Number 6 to Display
Alarm(1) at 1699907280: 30 Message_1
Alarm>Replace_Alarm(1) 10 Message_1p
Alarm(1) Replaced at 1699907284: 10 Message_1
Display Alarm Thread 140190907799104 for Alarm_Time_Group_Number 6 Terminated at
1699907284
Created New Display Alarm Thread 140190899299904 for Alarm_Time_Group_Number 2 to Display
Alarm(1) at 1699907284: 10 Message_1
Alarm>Alarm(1) Displayed by Display Thread 140190899299904 for Alarm_Time_Group_Number 2
at 1699907294: Message_1
```

Scenario 4: Invalid Input to Start_Alarm()
Inputs:

- **Time 0 Seconds:** Start_Alarm(10) -1 Hello World
- **Time 5 Seconds:** Start_Alarm(-1) 2 Hello World
- **Time 10 Seconds:** Start_Alarm(x) 49 id test

Expected Output:

- **Time 0 Seconds:** *"Alarm time must be greater than 0"*
- **Time 5 Seconds:** *"Alarm ID must be greater than 0"*
- **Time 10 Seconds:** *"Bad command"*

Actual Output:

```
Alarm>Start_Alarm(10) -1 Hello World
Alarm time must be greater than 0
Alarm>Start_Alarm(-1) 2 Hello World
Alarm ID must be greater than 0
Alarm>Start_Alarm(x) 49 id test
Bad command
```

Scenario 5: Starting alarm with existing ID
Inputs:

- **Time 0 Seconds:** Start_Alarm(20) 20 Hello World
- **Time 5 Seconds:** Start_Alarm(20) 20 Hello World

Expected Output:

- **Time 0 Seconds:** *"Alarm(20) Inserted by Main Thread (Thread_ID) Into Alarm List at (Epoch Time): 20 Hello World"*
- **Time 0 Seconds:***"Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 4 to Display Alarm(20) at (Epoch Time): 20 Hello World"*
- **Time 5 Seconds:** *"An alarm with ID 20 already exists"*

Actual Output:

```
Alarm>Start_Alarm(20) 20 Hello World
Alarm(20) Inserted by Main Thread 140611675908992 Into Alarm List at 1699907395: 20 Hello World
Created New Display Alarm Thread 140611675903552 for Alarm_Time_Group_Number 4 to Display
Alarm(20) at 1699907395: 20 Hello World
Alarm>Start_Alarm(20) 20 Hello World
An alarm with ID 20 already exists.
```

Scenario 6: Starting an alarm -> Replacing the alarm -> Canceling the alarm
Inputs:

- **Time 0 Seconds:** Start_Alarm(20) 20 Message_1
- **Time 5 Seconds:** Replace_Alarm(20) 10 Message_2
- **Time 10 Seconds:** Cancel_Alarm(20)

Expected Output:

- **Time 0 Seconds:** *"Alarm(20) Inserted by Main Thread (Thread_ID)Into Alarm List at (Epoch Time): 20 Message_1 Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 4 to Display Alarm(20) at (Epoch Time): 20 Message_1"*
- **Time 5 Seconds:** *"Alarm(20) Replaced at (Epoch Time): 10 Message_2 Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 4 Terminated at (Epoch Time) Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 2 to Display Alarm(20) at (Epoch Time): 10 Message_2"*
- **Time 10 Seconds:** *"Alarm(20) Canceled at (Epoch Time): 10 Message_2 Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 2 Terminated at (Epoch Time)"*

Actual Output:

```
Alarm>Start_Alarm(20) 20 Message_1
Alarm(20) Inserted by Main Thread 140256732875648 Into Alarm List at 1699910099: 20 Message_1
Created New Display Alarm Thread 140256732870208 for Alarm_Time_Group_Number 4 to Display
Alarm(20) at 1699910099: 20 Message_1
Alarm>Replace_Alarm(20) 10 Message_2
Alarm(20) Replaced at 1699910107: 10 Message_2
Display Alarm Thread 140256732870208 for Alarm_Time_Group_Number 4 Terminated at
1699910107
Created New Display Alarm Thread 140256653342272 for Alarm_Time_Group_Number 2 to Display
Alarm(20) at 1699910107: 10 Message_2
Alarm>Cancel_Alarm(20)
```

```
Alarm(20) Canceled at 1699910114: 10 Message_2
Display Alarm Thread 140256653342272 for Alarm_Time_Group_Number 2 Terminated at
1699910114
```

Scenario 7: Replacing an alarm that does not exist. Canceling an alarm that does not exist.
Inputs:

- **Time 0 Seconds:** Replace_Alarm(20) 10 Message_1
- **Time 5 Seconds:** Cancel_Alarm(20) 10 Message_1

Expected Output:

- **Time 0 Seconds:** *"Alarm with ID 20 not found and cannot be canceled."*

Actual Output:

```
Alarm>Replace_Alarm(20) 10 Message_1
Alarm with ID 20 not found and cannot be replaced.
Alarm>Cancel_Alarm(20) 10 Message_1
Alarm with ID 20 not found and cannot be canceled.
```

Scenario 8: Starting two alarms with the same Alarm_Time_Group_Number.
Inputs:

- **Time 0 Seconds:** Start_Alarm(20) 20 Message_1
- **Time 5 Seconds:** Start_Alarm(21) 20 Message_2

Expected Output:

- **Time 0 Seconds:** "*Alarm(20) Inserted by Main Thread (Thread_ID) Into Alarm List at (Epoch Time): 20 Message_1*"
- **Time 0 Seconds:** "*Created New Display Alarm Thread (Thread_ID) for Alarm_Time_Group_Number 6 to Display Alarm(1) at (Epoch Time): 20 Message_1*"
- **Time 5 Seconds:** "*Alarm(21) Inserted by Main Thread (Thread_ID)Into Alarm List at (Epoch Time): 20 Message_2"*
- **Time 20 Seconds:** *"Alarm(20) Displayed by Display Thread (Thread_ID)for Alarm_Time_Group_Number 4 at (Epoch Time): Message_1"*
- **Time 25 Seconds:** *"Alarm(21) Displayed by Display Thread (Thread_ID)for Alarm_Time_Group_Number 4 at (Epoch Time): Message_2"*

Actual Output:

```
Alarm>Start_Alarm(20) 20 Message_1
Alarm(20) Inserted by Main Thread 140629504666496 Into Alarm List at 1699910795: 20 Message_1
Created New Display Alarm Thread 140629504661056 for Alarm_Time_Group_Number 4 to Display
Alarm(20) at 1699910795: 20 Message_1
Alarm>Start_Alarm(21) 20 Message_2
Alarm(21) Inserted by Main Thread 140629504666496 Into Alarm List at 1699910797: 20 Message_2
Alarm>Alarm(20) Displayed by Display Thread 140629504661056 for Alarm_Time_Group_Number 4
at 1699910815: Message_1
Alarm(21) Displayed by Display Thread 140629504661056 for Alarm_Time_Group_Number 4 at
1699910817: Message_2
```

Scenario 9: Starting five alarms with the same Alarm_Time_Group_Number.
Inputs:

- **Time 0 seconds:** Start_Alarm(1) 51 M1
- **Time 5 seconds:** Start_Alarm(2) 52 M2
- **Time 10 seconds:** Start_Alarm(3) 53 M3
- **Time 15 seconds:** Start_Alarm(4) 54 M4
- **Time 20 seconds:** Start_Alarm(5) 55 M5

Expected Output:

- **Time 0 seconds:** *“Alarm(1) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912030: 51 M1”*
- **Time 0 seconds:** *”Created New Display Alarm Thread 140683756873280 for Alarm_Time_Group_Number 11 to Display Alarm(1) at 1699912030: 51 M1”*
- **Time 5 seconds:** *“Alarm(2) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912044: 52 M2”*
- **Time 10 seconds:** *”Alarm(3) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912055: 53 M3”*
- **Time 15 seconds:** *”Alarm(4) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912066: 54 M4”*
- **Time 20 seconds:** *”Alarm(5) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912075: 55 M5”*
- **Time 51 seconds:** *“Alarm(1) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at 1699912081: M1”*
- **Time 57 seconds:** *”Alarm(2) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at 1699912096: M2”*
- **Time 63 seconds:** *”Alarm(3) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at 1699912108: M3”*
- **Time 69 seconds:** *”Alarm(4) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at 1699912120: M4”*
- **Time 75 seconds:** *”Alarm(5) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at 1699912130: M5”*

Actual Output:

```
Alarm>Start_Alarm(1) 51 M1
Alarm(1) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912030: 51 M1
Created New Display Alarm Thread 140683756873280 for Alarm_Time_Group_Number 11 to Display
Alarm(1) at 1699912030: 51 M1
Alarm>Start_Alarm(2) 52 M2
Alarm(2) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912044: 52 M2
Alarm>Start_Alarm(3) 53 M3
Alarm(3) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912055: 53 M3
Alarm>Start_Alarm(4) 54 M4
Alarm(4) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912066: 54 M4
Alarm>Start_Alarm(5) 55 M5
Alarm(5) Inserted by Main Thread 140683756878720 Into Alarm List at 1699912075: 55 M5
Alarm(1) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at
```

```
1699912081: M1
Alarm(2) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at
1699912096: M2
Alarm(3) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at
1699912108: M3
Alarm(4) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at
1699912120: M4
Alarm(5) Displayed by Display Thread 140683756873280 for Alarm_Time_Group_Number 11 at
1699912130: M5
```

6. Conclusion

In conclusion, we have built a multi-threaded alarm application using POSIX threads, with a focus on preventing data inconsistencies through the use of mutex locks and conditional variables, to prevent the readers-writers problem. This approach has been crucial in maintaining smooth operation across multiple threads. By applying these concepts, we've managed to meet the project's requirements effectively. This development process has been a practical demonstration of how threading and synchronization can be used to create a stable and functional application. The Readers-Writers model provides a structured approach to managing concurrent access to shared resources. It ensures that multiple display threads can efficiently read from the alarm list without conflicts, while the main thread maintains exclusive control during write operations. This design effectively balances the need for parallelism in displaying alarms with the requirement for data consistency when modifying the shared alarm list.