POSIX Threads, Semaphores, Producer-Consumer-Shared-Circular-Buffer and Readers-Writers Problem

Report

(Assignment 3)

Github link for source code:

https://github.com/goIsaiah/EECS3221-a3

Name	Student Number	Email
Isaiah Ashton-Kenny	217260263	iak605@my.yorku.ca
Sanchit Duggal	217554668	san249@my.yorku.ca
Isaiah Gocool	218918052	goisaiah@my.yorku.ca
Yuri Roopnarine	214955744	yuriroop@my.yorku.ca
Daniel Di Giovanni	218204818	dand02@my.yorku.ca

Table of Contents

Introduction

The aim of this assignment was to help us master the utilization of POSIX threads, semaphores, and shared data structures. The task at hand was to develop a multi-threaded program, '**New_Alarm_Cond.c'**, which builds upon the provided '**alarm_cond.c'** program. We were to implement a program for managing alarm requests, manipulating various synchronization techniques and shared data structures. In this report, we will take a look at the implementation of all the features such as the types of alam requests, the design and management of the shared data structures and the utilization of synchronization mechanisms used for concurrent access.

Design and Implementation Aspects

Shared Data Structures

In the program the threads have to share data structures to be able to handle their processes. There are three data structures being used:

- **Alarm List:** The alarm list is stored in **New_Alarm_Cond.c** as the variable **alarm_list_header** of type **alarm_request_t** and is initialized to empty on program start. This list is implemented as a linked list of alarms which inserts new alarms using the **insert_to_alarm_list** method and is shared between the main thread and the alarm thread. The **insert_to_alarm_list** method simply adds the specified alarm_request to the alarm list based on their time value. This is the only data structure shared between the main thread and alarm thread.
- **Circular Buffer:** The circular buffer is stored in **New_Alarm_Cond.c** as the variable **circularBuffer** of type array which contains alarm_request_t and is initialized to empty on program start. This list is shared between the consumer thread and the alarm thread. When an alarm thread is handling a change to the alarm list it passes the most recent alarm request to the circular buffer using the **write_to_circular_buffer** method. From there the consumer thread checks if the **circularBuffer** is not empty, and if so it consumes the alarm request the circular buffer provides.
- **Alarm Display List:** The alarm display list is stored in **New_Alarm_Cond.c** as the variable **alarm_display_list_header** of type **alarm_request_t** and is initialized to empty on program start. This list is implemented as a linked list of alarms which inserts new alarms using the **insert_to_alarm_list** method and is shared between the consumer thread and the periodic display thread. When the consumer thread handles a alarm_request, it either inserts it into the **alarm_display_list** based off of its time value (**Start_Alarm** and **Change_Alarm**) or it removes it from the **alarm_display_list** (**Change_Alarm** and **Cancel_Alarm**).

Alarm Requests

In order to parse the request a user makes, we had a file called Command_Parser.c, which used regexes to break up the user input into different parts, such as the command used, the alarm ID, the time, and message. For all alarm requests made, the main thread will check if it is valid, and then add it to the list of all alarms called alarm_list_header, which is a global variable that is accessed by the main thread and the alarm thread. The alarm thread then gets the alarm request most recently added to alarm_list_header by the main thread, and adds it to the correct periodic display thread or creates a new periodic display thread if there isn't one to handle the request. It also adds the alarm request to the circular buffer for the consumer thread to use. The periodic display thread has its own list called periodic_display_list_header which is a local variable, since each thread will have its own alarms based on their time value. The periodic display thread uses the helper function search_alarm_list, which gets the request type by using the current alarm's ID in periodic_display_list_header, and comparing that alarm's parameters in the alarm_list_header to see how it has been changed. Based on the changes made, it returns an integer that tells the periodic display function which request was made, and what message to print.

- **Start_Alarm:** The main thread checks if an alarm with the same ID already exists, and if it does not it inserts it to alarm_list_header. The alarm thread adds the request to the circular buffer, alarm_list_header, and creates a new periodic display thread if the request's time does not already have a periodic display thread that handles it. The periodic display thread will then start to print the message of the alarm every Time seconds.
- **Cancel_Alarm:** The main thread checks if an alarm with the same ID exists so it can be cancelled. If it does, the request will not have the time and message so it must get it from a copy of the original alarm request. After the data has been added, the main thread inserts the request to alarm_list_header. The alarm thread removes the alarm request from alarm_list_header based on its ID. Since the alarm has been removed from the global list, the periodic display thread will check if the alarm in the local list is in the global one, and if it is not then it has been cancelled. The periodic display thread will print the message for cancelling an alarm and then remove it from its local list.
- **Change_Alarm:** One important parameter included in the alarm_request_t data structure is change_status, which is only used for the Change_Alarm request. If the time is changed when Change_Alarm is invoked, the value of change_status is set to true, to tell the periodic display thread that the alarm has recently had its time changed. This is important because the alarm needs to be added to a different periodic display thread for its new time value. After the initial message for an alarm with a changed time is printed by the periodic display thread, the value of change_status is set back to false so the thread knows to print the default message.

Thread Responsibilities

Four essential threads used in this program which were: Main thread, Alarm thread, Consumer thread and Periodic Display Threads which were created by Alarm thread.

The main thread interacts with the user directly, the alarm thread handles the alarm list and makes periodic display threads. The consumer thread then takes that request from the circular buffer and

processes it whilst the periodic display threads ensure that the alarm requests are displayed at specific times. Each thread plays a vital role in order to have the alarm management system running smoothly.

Main Thread:

- Ensures user input is received and parses that input into alarm requests.
- Inserts alarm requests into the alarm list in a safe manner to avoid any conflicts.
- When the user input is received, it handles the request by adding it to the alarm list and prompts the alarm thread to wake up.
- Constantly monitors user input and responds to each request accordingly.

Alarm Thread:

- Monitor changes in the alarm list.
- Whenever a signal is received from the main thread, it wakes up and handles any updates to the alarm list.
- In charge of managing the alarm requests and making the appropriate adjustments toe the alarm list.
- Responsible for alarm requests that are added, removed or modified in the alarm list based on the user commands.
- Maintains the list of periodic display threads and makes new threads if needed.
- Ensures that accessing and modifying the alarm list are done in a thread safe manner.

Consumer Thread:

- Processes the alarm requests received from the circular buffer.
- Constantly monitors changes in the circular buffer, whenever an alarm request becomes available, retrieve it.
- Responsible for handling any alarm requests retrieved from the circular buffer.
- Manages different types of alarm requests such as start, change, cancel by inserting, removing or updating respectively in order to update the alarm requests in the alarm display list.
- Upon accessing and modifying the circular buffer, guarantees mutual exclusion.
- Outputs the contents of the circular buffer for debugging purposes.

Periodic Display Threads:

- Display alarm requests at regular intervals based on their time values.
- Originally made by the alarm thread to handle alarm requests that are scheduled for specific time values.
- Ensures that alarm requests are showcased at the times specified.
- Responsible for accurate timing for each alarm request displayed.

Synchronization

Since this program requires multiple threads to work together, different synchronization techniques are used to allow for safe memory access. Within the program there are two different synchronization methods used.

- **Mutexes:** During the execution of the program, mutexes are used when the various threads are trying to access the different shared data structures. This is done to ensure mutual exclusion and avoid threads reading invalid data from the data structures. Each of the data structures contains its own mutex.
- **Semaphores:** During the execution of the program, semaphores are used to give control to a process to ensure multiple processes are not accessing the same data structures concurrently. In the program this is done by calling the wait method which decrements the value of the semaphore, which when negative, blocks processes from executing. When the post method is invoked it increments the semaphore value, which activates the first blocked process which called wait previously. Resulting in threads taking turns reading and manipulating shared data structures.

In the program there are three uses of mutexes/semaphores:

- **Alarm List:** The alarm list is synchronized with a standard mutex, ensuring that the main thread and the alarm thread cannot both write to the list at the same time.
- **Circular Buffer:** The semaphores associated with the circular buffer ensures keeps track of the number of empty and taken spaces in the buffer. When the alarm thread adds a request to the buffer, it will signal to the consumer thread, which will then consume the request. The consumer thread will signal to the alarm thread when spaces in the buffer are available.
- **Alarm Display List:** The semaphore associated with the alarm display list ensures that as the consumer thread is handling the alarm request the alarm display list cannot be accessed by other threads to avoid concurrent memory access. This solves the reader-writer problem on the writer side, allowing no other thread to access the list when it is being written to.
- **Reader Counter:** The semaphore associated with the read counter controls the number of readers that are currently reading the list. This semaphore allows many reader threads to access the list simultaneously, but will restrict writer threads (consumer thread) from writing to the list until the readers are finished.
-

Testing Design

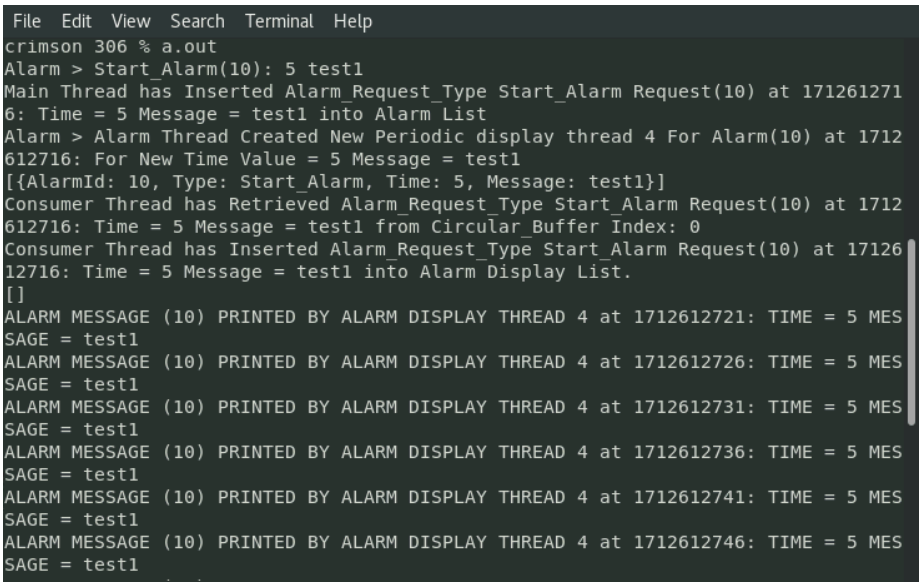
Overview of testing methodologies and results.

Unit Testing: To check if the commands for each individual function works correctly.

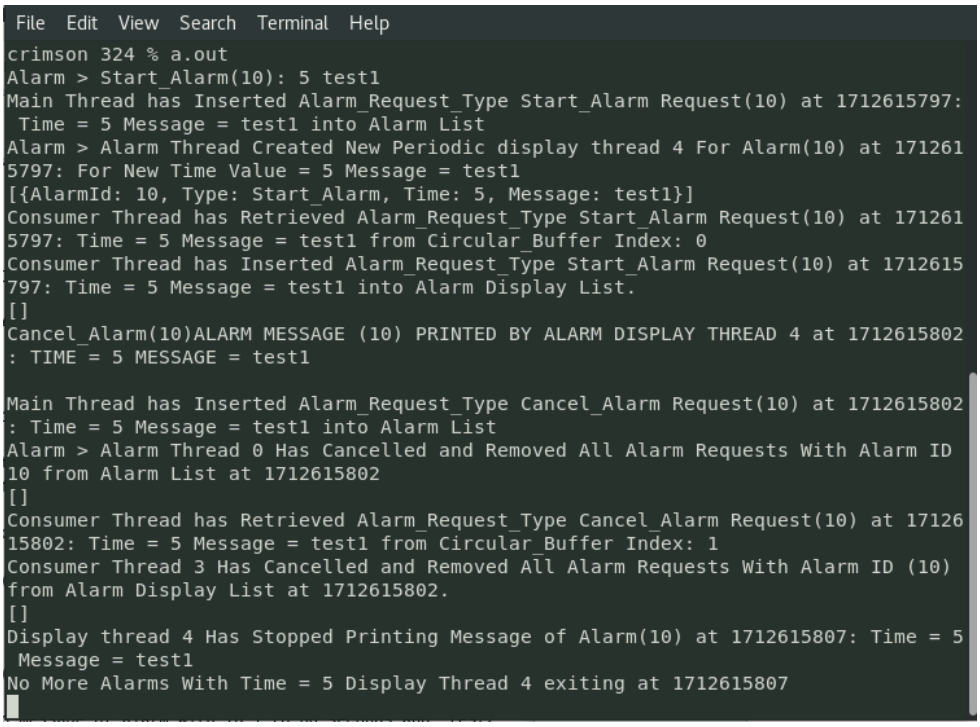
This can be done by calling each thread such as Start_Alarm, Change_Alarm, Cancel_Alarm and verifying the expected outputs without any errors occurring.

Result of Unit Testing:

Each of the above stated commands were tested, below shows the output for the Start_Alarm call which creates an alarm with ID 10, that has the message “test1” which will expire after 5 seconds.



Tested the functionality of Cancel_Alarm with the same ID to make sure it worked correctly and there would be no alarm with the same time upon completion.



Error Handling: To check how the program works when an invalid command is entered.

Result of Error Handling: Tried to change alarm as first command in console, however there were no previous alarms entered, this was the output from the program.

```
File  Edit  View  Search  Terminal  Help
crimson 308 % a.out
Alarm > Change_Alarm(1): 20 test2
Alarm with ID 1 does not exist, so request type Change_Alarm cannot be performed
 on alarm ID 1
Alarm > Alarm thread found error: alarm list is empty!
```

Error Handling (2): To check how the program works if the command is formatted incorrectly.

Result of Error Handling(2): Did not follow the correct regex and did not include a space between the colon and 30, thus resulting in an output of "Bad Command" and the ability to retry a correct command.

```
File  Edit  View  Search  Terminal  Help
crimson 313 % a.out
Alarm > Start_Alarm(1):30 test1
Bad command
Alarm >
```

Concurrency Testing: To check if the alarm thread synchronization is functional which would ensure that alarms are started, changed and canceled without any conflicts or race condition issues.

Created two Start_Alarm requests simultaneously, which attempted to start at the same time, however the program correctly accesses the situation and there are no race conditions or deadlocks occurring. The results show since the Start_Alarm ID already exists, the request type could not be performed and the circular index buffer is incremented.

```
Alarm > Start_Alarm(1): 10 test1
Main Thread has Inserted Alarm_Request_Type Start_Alarm Request(1) at 1712614388
: Time = 10 Message = test1 into Alarm List
Alarm > Alarm Thread Created New Periodic display thread 4 For Alarm(1) at 17126
14388: For New Time Value = 10 Message = test1
[{AlarmId: 1, Type: Start_Alarm, Time: 10, Message: test1}]
Consumer Thread has Retrieved Alarm_Request_Type Start_Alarm Request(1) at 17126
14388: Time = 10 Message = test1 from Circular_Buffer Index: 0
Consumer Thread has Inserted Alarm_Request_Type Start_Alarm Request(1) at 171261
4388: Time = 10 Message = test1 into Alarm Display List.
[]
Start_Alarm(1): 10 test1ALARM MESSAGE (1) PRINTED BY ALARM DISPLAY THREAD 4 at 1
712614398: TIME = 10 MESSAGE = test1

Alarm with ID 1 already exists, so request type Start_Alarm cannot be performed
Alarm > [{AlarmId: 1, Type: Start_Alarm, Time: 10, Message: test1}]
Consumer Thread has Retrieved Alarm_Request_Type Start_Alarm Request(1) at 17126
14399: Time = 10 Message = test1 from Circular_Buffer Index: 1
Consumer Thread has Inserted Alarm_Request_Type Start_Alarm Request(1) at 171261
4399: Time = 10 Message = test1 into Alarm Display List.
[]
```

Issues Encountered and Solutions

Throughout the implementation of the program, there were various issues that were faced, each requiring careful consideration, experimentation and adaptation to overcome in order for us to deliver a functional assignment.

One of the challenges was managing the different lists for periodic display threads with other lists. Managing these lists required careful coordination and synchronization between different threads to ensure data integrity. Additionally, when attempting to break down code using helper functions lead to some complications requiring modularity and simplicity in order to maintain code readability. Additionally, debugging also proved challenging due to the presence of multiple threads, requiring additional time spent to track down errors using proper error handling and logging mechanisms. In the initial phase of the project, opting for multiple semaphores for synchronization introduced some complexities which required careful consideration that resulted in refactoring the synchronization mechanism to use mutexes where appropriate to help ease thread synchronization. Additionally, in the early phase of the project, calculating the correct index of the circular buffer for both reader and writer threads posed a challenge that required careful consideration of thread synchronization and data structures in order to help us calculate an accurate index. Lastly, we had to implement dynamic thread creation logic to ensure that the alarm thread wasnt creating too little or too many periodic displays.