POSIX Threads

Report

(Assignment 2)

Github link for source code:

https://github.com/goIsaiah/EECS3221-a2

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 enhance the provided C program, alarm_mutex.c" for managing alarm requests to incorporate advanced functionalities using POSIX threads understood from the book "Programming with POSIX Threads". These advanced functionalities are done using Main and Display alarm threads. Using these threads, allows handling of various alarm requests starting and modifying alarms to canceling, suspending, and reactivating them. This document provides an overview of the design and implementation aspects of the program from parsing commands, creating display threads, creating alarms to events, printing alarms and life cycle of threads and alarms. We also look at how the testing of the program was conducted and any issues that we encountered.

Design and Implementation Aspects

To fulfill the requirements of this assignment, our program had to have one main thread and multiple display threads. The main thread is responsible for receiving and parsing user input, creating display threads, and communicating with display threads.

Parsing Commands

The main thread consists of an event loop that waits on user input. When the user inputs a command, the main thread parses the command and performs actions to handle it. If the command is invalid (incorrect format, references an alarm that does not exist, etc.), then the command is ignored.

Commands are parsed using regular expressions. The standard Linux C header file regular expressions, **regex.h**, is used to compile and test regular expressions (regexes). There are six regular expressions defined, one for each command. In addition to ensuring a valid format for commands, regexes are also useful for parsing information out of commands. Using the REG_EXTENDED option when compiling the regex, we are able to capture groups within the regex. This allows us to easily extract the alarm ID, time, and message from commands.

The logic for this regular expression testing and parsing into commands is found in the function **command_t *parse_command(char input[])**.

Creating Display Threads

One of the responsibilities of the main thread is creating display threads. A linked list is maintained containing information about all the display threads currently in use. The header of the list is a global variable called **thread_header** that is accessible to both the main thread and the display threads. To provide synchronization, a mutex called **thread_list_mutex** must be locked when accessing the list of threads. Synchronization will be discussed in more detail later.

When a new alarm is created the main thread checks the list of threads for any display threads that have space for the alarm (because each display thread can hold a maximum of two alarms). If no display thread has space for the alarm, or if no display threads exist, the main thread will create the display thread, add it to the list, and give the alarm to the newly created thread.

Creating Alarms

Alarms are created by the main thread when a user enters the **Start_Alarm** command. Alarms are added to a linked list. The header of the list is a global variable called **alarm_header**. Since multiple threads can access the list of alarms, a mutex called **alarm_list_mutex** must be locked when accessing the list. Synchronization will be discussed in more detail later.

The main thread will create a display thread if there are no display threads able to take the alarm. If there are display threads that can take the alarm, the main thread will create an "event" that can be sent to display threads. The first display thread that responds to the event that has space for the alarm will take it. Events will be described in more detail below.

Synchronization

Since this is a multithreaded program, we must have synchronization algorithms in place to protect data integrity and avoid race conditions and errors. To address this, we protect critical sections (areas in the code where only one thread should be accessing shared data at a time) with mutexes.

There are three areas of shared data in our program:

- **List of threads:** The thread list header is a global variable called **thread_header**. It is protected by a mutex called **thread_list_mutex**. Whenever the thread list is accessed or updated, the thread list mutex must be locked. Note that pointers to the elements in the thread list are also kept in their corresponding display threads. As a result, a display thread must lock the mutex when modifying its entry in the list.

- **List of alarms:** The alarm list header is a global variable called **alarm_header**. It is protected by a mutex called **alarm_list_mutex**. The program centers around the list of alarms. The main thread will lock the alarm list mutex when it receives a command, and unlocks it when it sends the event to the display threads. Inside the display thread event loop, the alarm list mutex is always locked. This is handled by a condition variable **alarm_list_cond**. Display threads wait on this condition variable. Whenever the alarm list is updated, the condition variable is broadcasted, waking up display threads with the alarm list mutex locked so that they can read and manipulate alarms.

- **Event:** Along with waking up the display threads through the condition variable broadcast, the main thread can give additional data to the display threads through an event. There can only be one event at a time and it is stored in a global variable called **event**. The event variable is protected by a mutex called **event_mutex**.

Events

The list of alarms is crucial to our program. All alarms are stored in the list and display threads store pointers to entries in the list for the two alarms that they have. Display threads respond to changes to the alarm list through condition variable broadcasts. However, the alarm list alone does not provide enough information for the display threads to take appropriate action to the changes in the list. To address this, we introduced the concept of "events" in our design.

The alarm list condition variable broadcast wakes up all the display threads and the event tells the display threads why they woke up and what to do. Before sending the broadcast the main thread sets the event. The event contains information about the command that was entered, the ID of the alarm that is being referenced (if applicable), and the alarm itself (if applicable).

Upon being woken up by a broadcast, the display threads will check the event and determine if it applies to them. (For example, a display thread receiving an event to suspend an alarm that it does not have will ignore the event). When a display thread decides it can handle an event (for example, assigning a new alarm to itself or updating an alarm it has), it will take appropriate action, then free the event. All subsequent threads responding to the event will see that the event has been freed (already handled) and will ignore it.

Printing Alarms

A key requirement is that threads should print their alarms every five seconds. To implement this, display threads use a timed wait (**pthread_cond_timedwait**) on the condition variable. The timeout is set as five seconds in the future, unless one of the thread's alarms is about to expire

sooner, in which case the timeout is set to the expiry of that alarm. An additional 10 milliseconds is added to the timeout to ensure that the alarm expiry is reached by the timeout.

Upon being woken up from the timed wait, the display threads will check if the wakeup was due to a timeout. If so, they print their alarms. Otherwise, the wakeup was due to an event, which is handled separately.

Life Cycle of Threads and Alarms

Display threads and alarms are created by the main thread. After that, the life cycle of display threads and alarms are handled by the display threads.

When alarms are updated (**Change_Alarm**, **Suspend_Alarm**, **Reactivate_Alarm**), canceled (**Cancel_Alarm**), or expired, they are modified by the display thread that has the corresponding alarm. With the alarm list mutex locked, the display thread will update an alarm (by changing its time, message, expiry time, and/or active status), or it will free an alarm upon cancellation/expiration.

When a display thread's alarms are canceled or expired it will check if it has any alarms left. If not, it will remove its entry from the thread list, free the entry, and exit (allowing the thread to be recycled by the operating system).

Testing Design

When testing the program, we would try to test each request independently of the others to see if they would work. Initially, the tests would not be meant to break the program, but to see if the correct output would be displayed with very basic inputs. Since we tested each request independently, it would be easier to debug the program because the modular design ensured that no requests would conflict with each other. If any bugs were found, specifically segmentation faults, we added print statements around parts of the code that we thought the error might occur. For example, when we got a segmentation fault error in the “Cancel_Alarm” request, we placed print lines for “1”, “2”, “3”, and so-on throughout the code for “Cancel_Alarm”. When we ran the program again, we could see which numbers were printed, making it easy to determine how far the program got before the error occurred.

For more heavy-duty debugging, we used GDB, a debugger for C programs. Like a normal debugger, we inserted breakpoints in the code and stepped through one line at a time. There was difficulty when debugging multiple threads, so we had to isolate places where we thought the

errors were and enter input that would hit those lines. Once those breakpoints were hit, we could examine values of variables and see which line specifically produced a segmentation fault.

To test for correctness and robustness, we first tested each request with small, basic inputs to see if they functioned properly. Then, we created multiple alarms, and tested the requests with them. If they all worked, then the program successfully handled multiple threads. We also tested the program using invalid inputs to see if it could handle them.

Our **debug.h** file also added print messages throughout the program that could only be seen when the program was run in debug mode, giving us more specific messages on why or where an error occurred. Some ways we used this feature was printing the alarm list after adding an alarm, and printing when a thread responded to an event but did not handle it. The benefit of this file was that we could compile the code with the debug option (**gcc -DDEBUG**) to have the print statements, and for production we could compile without the debug flag to have a clean version. We defined macros that evaluated to print statements with the debug flag, and evaluated to empty statements without the debug flag. Since this happened in the compilation step, we incurred no performance cost from the debug statements.

Issues Encountered and Solutions

One difficulty we had was making the alarms expire. In our while loop that would always be active when the program is running, we created a series of if statements to calculate the timeout of the alarm, so it would know when to remove the alarm from the list and thread. The first issue that occurred was that all alarms would expire at the same time as the one that expires the earliest. We knew that this had to be a problem caused by the if-statements that check when the alarms expire, so we reviewed the logic of them and made the necessary changes. After we fixed this problem, another one occurred where the alarms would be printed in quick succession before they expire. To fix this issue, we added 10 milliseconds to the timeout so that the expiration is reached.

Another difficulty we had was finding a way to iterate through the alarms we had, which would be hard due to the amount of threads that may be needed. We originally thought of using an array to keep track of the alarms, but this idea was discarded since it would be difficult to remove alarms that are not at the end of an array. After researching possible data structures, we decided to use linked lists due to their simplicity and easy implementation.