EECS 3221 Assignment 2 Report



Working with POSIX Threads

Tyler Noble, Lindan Thillanayagam, Francis Okoyo, Adham El-Shafie

Table of Contents

[Table of Contents 2](#_Toc507435915)

[Introduction 2](#_Toc507435916)

[Design 3](#_Toc507435917)

[Design Challenges 3](#_Toc507435923)

[Design Decisions 3](#_Toc507435924)

[Implementation 4](#_Toc507435925)

[New\_Alarm\_Mutex.c 4](#_Toc507435926)

[insert\_type\_A 8](#_Toc507435927)

[test 8](#_Toc507435928)

[terminate 8](#_Toc507435929)

[remove\_alarms 8](#_Toc507435930)

[alarm\_thread 9](#_Toc507435931)

[main 9](#_Toc507435932)

[README 9](#_Toc507435933)

[makefile 9](#_Toc507435934)

# Introduction

This program served the purpose of learning about the use of POSIX threads. By implementing an alarm system, we were able to cover a use case for POSIX threads that taught us about multithreading in C as well as proper process handling. This was done by having us change the implementation of the New\_Alarm\_Cond.c file. Our goal was to have it handle multiple alarm types as well as implement a main method that would check to see if alarm inputs were valid. We would also create and handle alarm threads that worked on the alarms themselves. This document will further discuss the implementation and thoughts that went behind it.

# Design

## Design Challenges

* Difficult to understand how to implement semaphores
* Many small bugs that weren’t conceptually hard but proved challenging to debug given the development environment
* Strategic mutex and semaphore placement was crucial to allowing fluid data access by threads
* Waiting methods needed to be modified from assignment 2, sadly they would not work well in this assignment due to the varied requirements
* There was an issue with making Type B alarms able to print without summing up the time till the next print call. This would have lead to print calls in increasing increments that would become unusable if too many types or alarms where assigned.

## Design Decisions

Our design decisions include many of the challenges we faced while writing this program. The following are the implementations we used to solve them. Starting with thread handling, our group realized that deleting threads would pose a challenge if we had no way of storing thread ID's. When going over potential data types to store them we realized that a linked list would be best suited for both performance and management reasons. However, since C does not have built-in support for linked lists we had to create one ourselves.

Figure 2. Linked List used to store threads

The last large decision we had to make was dealing with deleting locked threads. If this happened, we could be perma-locked out of the system and the mutex would be unable to unlock. This was solved using deferred cancellation types. The two cancellation types, asynchronous and deferred were what we had planned to use to handle thread deletion. The asynchronous, however, was vulnerable to the perma-lock effect mentioned earlier. The way the deferred cancellation worked was by putting the cancellation requests in a queue. After this was done, if the thread's status said it was unlocked, it was okay to follow through with the cancellation, however, if that was not the case then it would stay in the queue until ready to cancel.

The program differs from the previous assignment in that it creates threads to display different messages of different types at varying intervals. It handels all alarm requests as well by putting them in a singular list that is mutable at any time. So a user could create a new thread or terminate it as they so choose. To map these threads the above linked list was used however this new linked list has increased efficiency due to some design changes made around accessing the structure. In order to do this efficiently much of the implemented code is helper code.

# Implementation

## New\_Alarm\_Cond.c

### insert\_type\_A

This method was implemented to add new alarms of type A to this list structure. It takes in the parameters of the alarm threads: current, next and last pointer. The method locks the mutex such that no thread can access the alarm list. When the alarm is added, the mutex is unlocked.

### test

This method was created for debugging purposes.

### terminate

This method traverses through the two-dimensional matrix structure we used and searches for messages of the input type given as a parameter. After locating the type, we cancel the thread, however, if it is not successful we use "err\_abort" method to help us debug.

### remove\_alarms

This method removes alarms but not in the same way terminate does. This works specifically on the alarm\_list. . Many lines of code are for managing the linked-list and pointers for when an alarm is removed. Also locks and unlocks the mutex to prevent threads from accessing it when alarms are being deleted.

### alarm\_thread

This method works almost the same way as the one in mutex\_alarms.c. the main deference is that it disables and enables the cancel stat of that thread to prevent it from locking the mutex and potentially being terminated. This was done to prevent possible deadlock as there would be no was to unlock a mutex if the thread that locked it was cancelled.

### main

The main thread is also similar to the original main thread from mutex\_alarm.c the main difference is that it delegates some of the work that the old main did to helper methods like insert\_type\_a(). Another difference it that it parses inputs of 3 specific types. Any input that doesn’t match one of the 3 types is ignored and the system prints "bard command".

## README

## makefile

1. New\_Alarm\_Mutex:  New\_Alarm\_Mutex.c  cc  New\_Alarm\_Mutex.c  -  D\_POSIX\_PTHREAD\_SEMANTICS  -  lpthread