POSIX Threads - Assignment #2 Report

EECS3221 N, W23

Group members:

Abbasi, Muhammad	danyal07@my.yorku.ca
Ahmadzai, Jamal	jamal123@my.yorku.ca
Ali, Abdallah	aa7@my.yorku.ca
Mohammed, Ibrahim	ib15@my.yorku.ca
Sajjad, Abdullah	sajjadab@my.yorku.ca

Table of Contents

1. DESIGN & IMPLEMENTATION OF PROGRAM

1.1 Summary of the program

The program is a modified version of the original alarm_mutex.c program. The new_alarm_mutex.c program includes two types of alarm requests which are: Start_Alarm and Change_Alarm. Since we are working with POSIX threads, the main thread of the program should initially create an alarm thread and for every alarm request it receives, it checks if the format of the alarm request is consistent with the assignment's specified formatting.
If the request matches the format, it is valid and then the created alarm thread is inserted into the alarm list.
For every Start_Alarm request, the main thread prints a message which tells the user that the alarm was inserted into the alarm list.
For every Change_Alarm request, the thread changes the Time and Message values in the alarm based on the respective Alarm_ID. It then should print a message which indicates that the alarm was changed.
When an alarm is expired, the thread ensures it is removed from the alarm list and prints a message stating that the alarm was removed.
The Alarm thread is meant to communicate with the display thread which causes it to print a message for the alarm or newly changed alarm every 5 seconds.
More specific details can be found in the program's comments & implementation of the program below.

1.2 Implementation and Design of the program

The design contains 2 structs "alarm_t" and "display_d". The alarm_t represents a simple alarm while the display_d is used to control the output through the display threads. The alarms are stored in a linked list and are ordered based on their expiration time. When a new alarm is inserted, the list is traversed and the alarm is inserted such that the expiration time sort is maintained. The access to the list structure is controlled via a mutex variable "pthread_mutex_t". The "Start_Alarm" function is used to input alarms as mentioned before, while the "Change_Alarm" function is used to change the contents and/or time expiration of the existing alarms based on their id. The "display_selector" checks how many current alarms are assigned to the 3 display threads via the "display_details" struct. It then assigns the new to be assigned alarm, to the thread with the smallest count. The display threads run in infinite loops which all wait for the conditional variable's ("display_cond1", "display_cond2", "display_cond3") signal from the main thread. Once a signal is received, the threads display the alarm message every 5 seconds until expiration. Finally, once the alarm has expired, the display thread unlocks the list structure and the alarm is removed from memory.

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


// struct for alarm list
typedef struct alarm_tag
{
  struct alarm_tag *link;
  int id;
  int seconds;
  time_t time; /* seconds from EPOCH */
  char message[128];
  int changedFlag;




} alarm_t;


// struct for tracking display information
typedef struct display_details{
  int id1;
  int count1;
  int id2;
  int count2;
  int id3;
  int count3;
  int current_display;
```

1.3 Declared constants & structs used

We have the following structs: alarm_tag and display_details.

struct alarm_tag: in this struct, there is a linked list of alarms. The struct stores the id of the alarm, the seconds until alarm expiry, a message to go along with the alarm which is truncated to 128 characters if it goes over..

struct display_details: this struct is used for tracking information for the display threads that are used throughout the program.

1.4 Main Thread

From lines **651** and onwards, we have the main thread. The main thread contains any initializations that must be made of variables and required structs. The key design decisions used in the main thread are that the variable declarations take place here, the display thread and alarm threads are initialized here and we have the while loop at line **688** onwards which is what the user interacts with in order to use the program itself. Specific details can be found in the comments of the code.

```
448  void *alarm_thread(void *arg){
449
450      int status;
451      alarm_t *alarm;
452      int sleep_interval;
453      time_t current_time;
454      pthread_t display;
455
456
457
458
459
460      while(1){
461
462
463
464
```

1.5 Alarm Thread

Starting at line **448** onwards, there is the alarm thread. The alarm thread runs continuously, and on each iteration will unlock the mutex, and check if there are any alarms in the alarm list. If the list is empty, the thread sleeps for 1 second to allow for input. If there are associated alarms, the thread checks to see if the alarm is one that was recently changed. If yes, it informs the alarms associated display thread that it should start printing out the new message. Otherwise, the thread makes use of the display_selector function to choose the right display thread to be used, incrementing the number of associated alarms to the respective thread and calling the respective condition signal as it does so. The display_selector function chooses the thread with the least number of alarms. Finally, the thread unlocks the mutex and returns.

1.6 Display Thread

Starting at line **161 to 445** are the 3 alarm threads that are called in rotation according to which thread has less than 3 alarms associated with it. The display threads run in a continuous loop. They first lock the alarm mutex, after which they wait for their respective display conditions to be signaled by the alarm_thread.
Each thread does a check to see if the alarm it is currently associated with has a time value that is greater than the current time value (i.e. the alarm has expired). While this is false, it will print the message associated with that alarm, or it will print the changed message for when the alarm had been recently changed, and the thread had been notified to print the changed message. Once an alarm has expired, the thread will print a message indicating that the message had expired, followed by the alarm being removed from the alarm list. Once removed, the number of alarms associated with the thread is decremented by one and a check is done to see if there are any more alarms associated with the thread. If there are no more alarms, the thread prints a message indicating that it has been terminated. Finally, the mutex is unlocked.

2. TESTING

2.1 Testing Process

There is a test output document file which is included in the package for the submission of this Assignment #2. It details the testing process and inputs and expected outputs when using the program. The testing shows that the program is able to handle incorrect input as well, which is a requirement under part **C. Additional Requirements.** To sum it up we tested our code with manual testing, inputting correct and incorrect inputs on behalf of the user, and checking if the terminal displays the correct response given these. The testing was also split up into each command the user can use. Examples of test cases include, a test that displays “Bad Command” output when the user is missing an argument, and a variety of tests that display correctly with different types of inputs, for example if the user inputs a decimal number. We do this to deal with every potential input that the user uses in the program.

2.2 Make file

Included within the submission is a make file which will allow for easy compilation of the program so that it may be used on your local device.

3. Difficulties Encountered

A primary issue we faced during implementation was devising a way to keep track of the respective display thread ids, and alarm counts. In order to solve this issue, we implemented a display_details struct that would keep track of all these different fields as attributes of a global variable display_details. This way we would be able to access the exact number of all the alarms associated with a display thread, as well as the respective ids of the threads for notification purposes.

Next we faced the issue of knowing which display thread to add a new alarm to when an alarm was inserted. To combat this issue, we developed a display_selector function that would use the values of the number of alarms associated with out 3 main alarm threads, and would select the most suitable display thread for the current alarm.

One of the toughest problems our group encountered was separating the display thread and the Start_Alarm and Change_Alarm input threads. The issue we faced was that whenever one would start an alarm, the user could no longer put in new commands to change or start another periodic alarm. To fix this we would have to somehow be able to override the display thread and have it not affect the thread that administers the command. However we were unable to implement the solution for this problem in time.

Another problem we faced was that the display thread did not display multiple alarms at the same time. Instead of the alarms all displaying at the same time, we had to wait for one alarm to finish before the other started. This could be solved by once again separating the threads effectively, to allow them to run at the same time rather than running consecutively.