Assignment 2 Design and Implementation Report

November 12, 2019

Daniel Kozlovsky

Jada Jones

Ahaz Goraya

EECS 3221E

Prof. Jia Xu

Summary of Program

My_Alarm.c takes alarm requests as user input and parses the duration of the alarm, in seconds and the expiry message. After the specified time passes, the program shows that the alarm has expired and displays the alarm's message. The user can input as many requests as they want and as fast as they want. The alarms will all be tracked and display their messages will display as each one expires.

The alarms are tracked by two displays. One handles the even requests, and the other shows the odd numbered requests. Each display will count down the remaining time of the next, earliest alarm that each display handles respectively. Once an alarm expires, a message is shown and the remainder of the next earliest alarm starts counting down. No messages will be displayed while there are no alarms.

Design

The program consists of four different threads:

- The main thread, which collects user input and sorts alarms in a master list
- The alarm thread, which organizes all the alarms into sub lists based on request number (odd/even)
- Display thread 1, which displays information about and counts down all the odd alarms
- Display thread 2, which displays information about and counts down all the even alarms

Flow

1. User inputs an alarm, specifying number of seconds until expiry and the message to show

2. Alarm is added to master list, in the form of a linked list. It is inserted based on its expiry time from least to greatest

3. Alarm thread reads the first (soonest to expire) entry from the list and based on the request number, sorts it into a secondary list also based on expiry.

4. Display thread 1 and 2 continuously check their respective lists for new entries. D1 checks odd request, while D2 checks for even requests

5. Starting with soonest to expire first, the display threads show how many seconds are left in the alarm, every two seconds.

6. Once an alarm expires, the corresponding display thread notifies the user and displays the message and initial alarm number of seconds.

7. If there are more alarms in a sub list, the corresponding thread will now count down the soonest one.

8. Steps 5-7 repeat until there are no alarms in a sub list

9. The program is constantly accepting new alarms from the user

Notable Design Decisions

1. There are two sub lists in addition to the master list. One for odd requests and one for even.

   This was chosen so that each display thread could access their respective list and continuously poll for alarms. This increases speed and quality of the program. This also takes advantage of the alarm thread for processing.

```
alarm_t *alarm_list = NULL; /*intermediate storage for new alarm requests */
alarm_t *even_alarm_list = NULL; /*even numbered requests */
alarm_t *odd_alarm_list = NULL; /* odd numbered requests */
```

2. Three mutex tokens are used. One to manage changes per each linked list.

   Instead of locking all data at any one time, different mutexs are be used to lock a respective list. This way threads have more running time since they aren't waiting for an irrelevant list to be unlocked.

```
/* one mutex for each list because they may be accessed at different
times. This way is faster and more efficient */
pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t even_alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t odd_alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
```

3. Each list is sorted based on expiry time.

   Since each list is sorted by expiry, by taking the head of a list, a thread has access to the next alarm to expire. This way it can do the countdown, and remove the head, moving on to the next list entry. The main idea is that the user is interested in the next alarm that will expire. This also is a clever way to “track” each alarm. Instead of having extra data per alarm, they are just ordered so that they expire in order and each one doesn’t have to be checked all the time.

```
/*
 * Insert the new alarm into the list of alarms,
 * sorted by expiration time.
 */
last = NULL;
next = alarm_list;
if(next == NULL)//first entry into list
{
    alarm_list = alarm;
    alarm->link = NULL;
}
else
{
    while(next != NULL)//iterate through list
    {
        //insert before smallest time that's still larger than alarm.time
        if(alarm->time <= next->time)
        {
            alarm->link = next;
            if(last != NULL)
                last->link = alarm;
            else
                alarm_list = alarm;

            break;
        }
    last = next;
    next = next->link;//iterate

    }
    if(next == NULL)
    {
        last->link = alarm;
    }
}
```

Challenges

1. It was a challenge to decide how to display the alarms. Initially we had a FIFO implementation, yet it didn't make sense with alarms and required a lot of changes to successfully implement. We decided to go with sorted lists to make it easy to keep track of the alarms as they expire.

2. We had trouble displaying the countdown for each alarm. Our initial idea was to display the countdown of each alarm from start to finish. We changed this to showing the remainder of the next expiring alarm. This was influenced by our sorted list implantation. To do this, we also had to base the countdown on current time minus alarm time as opposed to keeping track of every two seconds.

```
/* Repeat every two seconds while alarm hasn't expired */
if(current_alarm->time - now > 0 && now - prev_timestamp >= 2)
{
    fprintf(stdout,"Display Thread 1: Number of Seconds Left %d : Alarm Request Number: (%d) Alarm Request: (%d) [\"%s\"]\n",
        current_alarm->time - now, req_num, req_seconds, str);

    /*reset timestamp */
    prev_timestamp = now;
}
else if(current_alarm->time - now < 0) //Alarm expired
{
    fprintf(stdout,"Display Thread 1: Alarm Expired at %d : Alarm Request Number: (%d) Alarm Request: (%d) [\"%s\"]\n",
        time (NULL), req_num, req_seconds, str);
    printf ("(%d) %s\n", req_seconds, str); /*print alarm after expired */

    /* remove first node */
    odd_alarm_list = current_alarm->link;
    //free(current_alarm);
}
```

Testing

All test were designed in order to cover many extreme cases, some base cases and incorrect cases. They are done through user input to the program.

Test 1:

Two alarms of same length; both are handled by different threads and expire at once.

Test 2:

Incorrect input; program indicates bad argument

Test 3:

A long alarm, followed by two shorter alarms; the countdown will show the shorter alarm first and then continue the remainder of the long alarm

Test 4:

4 alarms of same length; each display thread handles them in order of input

Test 5:

Negative number for alarm length; program indicates bad arguments

Test 6:

Ten alarms input very quickly with variable lengths; all are handled accordingly and in desired order