

# EECS 3221 - ASSIGNMENT 2

## *POSIX THREADS REPORT*

Group Members: Michael Baker (215757982), George Giannopoulos (213468442) , Matthew Draws (215690944), Nicolae Semionov (216468498), Brandon Morin (214880306)

## Contents

|          |                                  |           |
|----------|----------------------------------|-----------|
| <b>1</b> | <b>User Input</b>                | <b>2</b>  |
| <b>2</b> | <b>Main Thread Processing</b>    | <b>5</b>  |
| <b>3</b> | <b>Display Thread Processing</b> | <b>6</b>  |
| 3.1      | Design Overview . . . . .        | 6         |
| 3.2      | Critical Details . . . . .       | 8         |
| <b>4</b> | <b>Testing and Debugging</b>     | <b>9</b>  |
| <b>5</b> | <b>Process Flow Diagrams</b>     | <b>11</b> |

# 1 User Input

The main design ideology for section 3.1 included being responsible for the input from the user when the Alarm prompt was presented after executing the program. There are four main interlinked possibilities of input commands that get be executed by the user when running the program.

There are four main interlinked possibilities of input commands that get be executed by the user when running the program. These four include:

- **Start\_Alarm**
- **Change\_Alarm**
- **Cancel\_Alarms**
- **View\_Alarms**

These four commands are known as alarm requests and they cohesively represent the main functionality of the Alarm program. They are all case sensitive keywords that need to be written as specified by the assignment instructions. They must be written in the proper format such as:

→ **Alarm\_Request(AlarmID): Time Message**

The AlarmID represents any positive integer inputted by the user to represent a number ID for that specified alarm. For this assignment, the AlarmID was coded as an integer type in C, where the value of it must be  $> 0$  ensuring it is a positive value. The **time** represents the amount of seconds that must elapse until an alarm will go off. This time value was scanned in by input as a positive integer as well. The **message** is scanned in as an array of characters of length 128. An example input line is shown below, This line describes initialization of an alarm of ID "100" that will go off in 60 seconds and print the message "Hello World".

→ **Start\_Alarm(100): 60 Hello World**

One of the main challenges for this section of the assignment was simply handling input at the alarm prompt. There were four possible alarm requests, we had to ensure proper handling for each possible request. The way this was done by our group was by having four preset strings for each request. These four preset strings are shown in the code below, they are initialized by static constant chars, there is one string declared for each of the four requests.

**Figure 1:** Static constant characters used by strcmp

```
300 static const char* STR_CHANGE_ALARM = "Change_Alarm(";
301 static const char* STR_START_ALARM = "Start_Alarm(";
302 static const char* STR_CANCEL_ALARM = "Cancel_Alarm(";
303 static const char* STR_VIEW_ALARMS = "View_Alarms";
304 static const char* STR_EXIT = "Exit";
```

The design idea of this section was as follows; the input that the user would provide on the alarm prompt would be scanned and checked with each of these four predefined strings. Depending on which of the four the input would be matching to, that would represent a specific process to be executed. Looking at the main\_thread.c file, we have a function called interpret which is responsible for this exact

process. We see at the beginning of this function that the alarm requests strings that were identified above are declared here. The interpret function takes as parameters the entire input from the user and essentially interprets this information to see which alarm request path to follow. This is done by using string compare (strcmp) operations. To compare the scanned input and the predefined strings.

**Figure 2:** sscanf function separating the input into 3 words

```

312
313     int wordcount = sscanf(trimmed_line,"%s %s %128[^\n]",words[0],words[1],words[2]);
314     if(wordcount >= 3)
315     {
316         //second string (time) is not positive integer
317         if(!strIsNumeric(words[1]))
318             return 0;
319

```

The figure above shows where the input was scanned within the code, this figure comes from the main\_thread.c file and it is clear that the three strings that are scanned are the alarm type and ID for the words[0], the time for words[1] and the message for words[2]. One of the main design difficulties to overcome for this specific part of the program was the fact that the first string inputted by the user included information of both the alarm request and the alarm ID. As an example, if the user wanted to start an alarm they would write:

→ Start\_Alarm(213):

As seen above, this string contains information for both the AlarmID as well as the Alarm request type. In order to extract all information necessary, we needed to first process the characters of the string from the starting index until before the first bracket "(". We would then check this substring to see if matched any of the following four: Start\_Alarm, Change\_Alarm, Cancel\_Alarm, or View\_Alarm. If it didn't match then the input would be considered incorrect and the program would reset with the alarm prompt again. If it did match then we would scan the index from the first bracket "(" until the last bracket ")" and extract this substring. This substring would be converted to an integer and stored as AlarmID. Finally the character for the colon ":" would be checked after the last bracket to make sure it is the last character within this string.

The final design challenge we faced within our code for section 3.1 refers to string issues found while testing code. The problem relates to a proper reading of input from the user, as shown below in figure 3.

**Figure 3:** Bug occurring when extra text added after the the alarm id

```

[user@localhost 3221assignment2-AlarmThread]$ ./NewAlarmThread

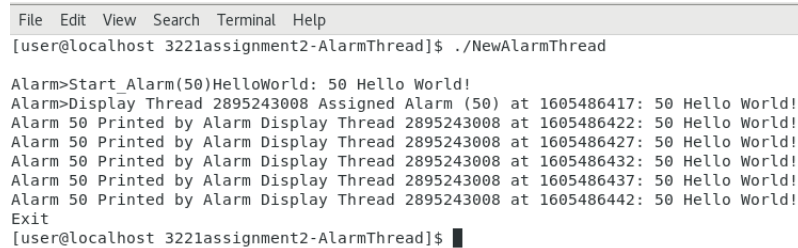
Alarm>Start_AlarmHello(50): 50 Hello World!
Alarm>Display Thread 177964800 Assigned Alarm (50) at 1605487041: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 177964800 at 1605487046: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 177964800 at 1605487051: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 177964800 at 1605487056: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 177964800 at 1605487061: 50 Hello World!
Exit
[user@localhost 3221assignment2-AlarmThread]$ █

```

The program was disregarding any input before the first bracket was inputted. This issue was caused due to the fact that we coded it in a way where after the alarm request type was inputted, the program would just wait until it sees the "(" . This unfortunately caused a problem because between the alarm request and the first bracket anything could be written without getting an invalid error prompt. We solved this by appending a "(" that is seen at the end of each of our predefined strings in figure 1 except for View\_Alarms.

Initially, we didn't have the bracket appended, but this would ensure that the first bracket would have to be typed immediately after the alarm request type. This led us to finding a very similar issue happening after the last bracket and the colon symbol as shown below in figure 4

**Figure 4:** Bug occurring when extra text added after the the alarm id

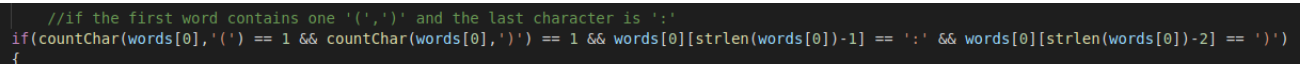


```
File Edit View Search Terminal Help
[user@localhost 3221assignment2-AlarmThread]$ ./NewAlarmThread

Alarm>Start_Alarm(50>HelloWorld: 50 Hello World!
Alarm>Display Thread 2895243008 Assigned Alarm (50) at 1605486417: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 2895243008 at 1605486422: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 2895243008 at 1605486427: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 2895243008 at 1605486432: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 2895243008 at 1605486437: 50 Hello World!
Alarm 50 Printed by Alarm Display Thread 2895243008 at 1605486442: 50 Hello World!
Exit
[user@localhost 3221assignment2-AlarmThread]$
```

This was the same problem as the one mentioned above, however it was happening in a different part of the string input, specifically nearing the end between the last bracket ")" and the colon ":" To fix this problem we looked at the interpret function, and added this to one of our if statements.

**Figure 5:** Improved filtering ensuring brackets were in the proper positions



```
//if the first word contains one '(',')' and the last character is ':'
if(countChar(words[0], '(') == 1 && countChar(words[0], ')') == 1 && words[0][strlen(words[0])-1] == ':' && words[0][strlen(words[0])-2] == ')')
{
```

The -1 and the -2 will make sure that the two last characters of the input are a closing bracket and a colon and this solved our problem.

A design choice we made in our code is a simple Exit command for our program, This was not a feature that was specified in the instructions for the assignment but was one that we felt was an appropriate quality of life change when working with the alarm program. The only way to exit the program previously was to hold ctrl + c in the linux terminal. This would force quit the program. However with this simple feature implemented, the user can just simply type the keyword "Exit" at the alarm prompt and the program would end. The exit feature is implemented in Main\_Thread.c within the interpret function as an 'else if' statement as shown below.

**Figure 6:** Additional Exit command provided for convenience

```

415         else if(strcmp(STR_EXIT,words[0]) == 0)
416         {
417             //terminate the program
418             exit(0);
419         }

```

**Figure 7:** The modified version of the alarm\_tag

```

38 typedef struct alarm_tag {
39     struct alarm_tag *link;
40
41     time_t          time; /* seconds from EPOCH. Holds the absolute time the alarm will expire*/
42     char            message[128]; //increased from 64 to 128 characters
43
44     unsigned int     alarm_id; //alarm id stored as a positive integer
45     time_t          create_time; //holds the time that the alarm was created
46     time_t          assign_time; //holds the time the alarm was assigned to a thread
47     char            status; //0=unassigned 1=assigned 2=modified (see define at top of this file ALA
48 } alarm_t;

```

## 2 Main Thread Processing

Ideally, when a thread prints a message on a new line the prompt should be reprinted at the bottom of the screen with whatever has already been typed. For example, if the user is trying to type the 'View\_\_Alarms' and a display thread also has output while you are typing the screen would look like the following:

```

Alarm>View__AlaDisplay Thread 1426388736 Assigned Alarm (5) at 1605202645: 60 test
rm

```

You can see from the above line that the 'View\_\_Alarm' string is separated by the output of the thread. This sort of behaviour is confusing not allowing the user to clearly see what they have already typed. An ideal version of the program could reprint whatever had been typed as a new line for clarity. The output in the previous example would change as follows:

```

Alarm> View__Ala Display Thread 1426388736 Assigned Alarm (5) at 1605202645: 60 test Alarm>__Ala

```

Allowing the user to easily see what they have already typed. Such a simple problem proved difficult to solve. We tried the following approach:

- Create a buffer for the list of characters as they are typed
- Erase the last character if backspace is pressed
- Clear the buffer on the enter key

Following the aforementioned rules, whenever a thread needs to print to the screen it has access to the previously typed keys which allows the thread to print the previous line. C interprets the stdin buffer in a very abstract manner. In modern Linux operating systems, the input is buffered. The input buffer is only 'dumped' when the return key is pressed. In effect this limits C's ability to modify a buffer on every key press. This is because the operating system prevents access to the input buffer until the enter key is pressed. Clearly this presents a problem because we need to have a copy of the typed characters before the enter key is pressed.

It turns out that there is a more advanced display library that can be used with c called 'Ncurses'. Ncurses allows you to disable input buffering. With input buffering disabled a function called getch() can be used to get one character at a time. Unfortunately, Ncurses is not a thread safe library and because our program needed to use threads we had to live with the possibility of user text being split.

## 3 Display Thread Processing

### 3.1 Design Overview

The display thread process flow operates on a constant loop that is initiated by the creation of an alarm and terminated when there are no alarms left. The way that the loop eventually terminates is through a flag that is set as the condition of the while loop. On every iteration of the while loop, the program checks the number of alarms that are active. If that number is 0, the flag is changed to 1, the loop exits and the process stops according to the requirements (A3.3.3). You can see the code for changing the flag called `exitcode` in figure 8 below.

**Figure 8:** The subsection of the display process that exits the while loop.

```
317 //if the number of assigned alarms is 0
318 if(mythreadtag->num_alarms == 0)
319 {
320     //display thread exit message
321     printf("Display Alarm Thread %u Exiting at %d\n",mythreadtag->thread_id,(int)time(NULL));
322     //remove thread tag from the thread list and free allocated memory
323     removefromThreadList(thread_list,mythreadtag->thread_id);
324     //unlock mutex
325     status = pthread_mutex_unlock (alarm_mutex);
326     if (status != 0)
327         err_abort (status, "Unlock mutex");
328     //set loop exit status
329     exitcode=1;
330 }
```

In addition to determining whether the loop needs to exit, there is additional functionality that was required to be built into the display thread process. The process is also required to both assign and remove alarms from the thread. The number of alarms per thread was implemented as a constant variable defined in `alarm_threads.h`. For our purposes it was defined to be 2 as this was the requirement given to us. On every iteration of the loop, the process checks to determine whether the number of alarms in the thread matches the number of alarms available.

If there are extra alarms that have not been assigned to the thread, it will assign the alarms to the thread through a function called `tryAssignAlarm()`. This functionality is required by A3.3.1. The function `tryAssignAlarm()` iterates through the linked list of alarms to find a spot for any unassigned alarms. It will then update the links in the linked list and add this alarm to the list.

If there are less alarms than expected, i.e. an alarm got canceled in the main thread, the process also needs to handle this (see A3.3.2). To do this, the process cycles through the alarms assigned to the threads and checks whether the alarm is either nonexistent or the time on the alarm has expired. It will then display the "removed alarm" output text with the appropriate reason. You can see the code referring to how A3.3.1 and A3.3.2 were implemented figure 9 below.

Another requirement of the display thread process given by the design criteria is to display the alarm every 5 seconds(A3.3.4). This is implemented through code that triggers every 5 seconds by updating itself to trigger in 5 seconds every time it triggers. On every trigger, the process checks for valid alarms and cycles through the list of them and displays the status of each. This section of the code is included in figure 10 below as reference.

**Figure 9:** The code for assigning and removing alarms from the display thread

```

251 //if this thread needs to assign alarms to it still
252 if(mythreadtag->num_alarms < MAX_THREAD_ALARMS)
253 {
254     //try to assign this thread to an alarm
255     tryAssignAlarm(*alarm_list,mythreadtag);
256 }
257 //iterate through all of the assigned alarms
258 int i;
259 for(i=0;i<MAX_THREAD_ALARMS;i++)
260 {
261     //alarm id 0 is reserved for alarms that are not assigned
262     if(mythreadtag->assigned_alarms[i]!=ALARM_STATUS_UNASSIGNED)
263     {
264         //get the current alarm
265         alarm=getAlarm(*alarm_list,mythreadtag->assigned_alarms[i]);
266         //if alarm is expired or alarm is not in the alarm list (cancelled)
267         if(alarm == NULL || time(NULL) >= alarm->create_time + alarm->time)
268         {

```

**Figure 10:** The display alarm code in accordance with A3.3.4. regarding triggering

```

292 //display time has expired (5 seconds )
293 if(time(NULL) >= last_display_time + ALARM_THREAD_DISPLAY_INTERVAL)
294 {
295     //print a message for each assigned alarm for this thread
296     int i;
297     for(i=0;i<MAX_THREAD_ALARMS;i++)
298     {
299         //if the assigned alarm is valid
300         if(mythreadtag->assigned_alarms[i] != 0)
301         {
302             alarm_t* thisalarm = getAlarm(*alarm_list,mythreadtag->assigned_alarms[i]);
303             if(thisalarm->status == ALARM_STATUS_MODIFIED)
304             {
305                 printf("Display Thread %u Starts to Print Changed Message at %d: %d %s\n",
306                     ,mythreadtag->thread_id,(int)time(NULL),(int)thisalarm->time,thisalarm->message)
307                 thisalarm->status = ALARM_STATUS_ASSIGNED;
308             }
309             printf("Alarm %d Printed by Alarm Display Thread %u at %d: %d %s\n",
310                 mythreadtag->assigned_alarms[i],mythreadtag->thread_id,(int)time(NULL),(int)thisalarm->time,
311                 thisalarm->message);
312         }
313     }
314     //update last display time to the current time
315     last_display_time=time(NULL);
316 }

```

A final portion of the display process thread that is crucial, but is not technically listed as a requirement of the process, is the management of the alarm memory locations. In order to do this, the process employs the usage of mutex locks. At the start of every iteration through the loop, the process attempts to claim the mutex lock. If it cannot do so, the remainder of the logic will not run. At the end of each iteration, the process releases the mutex lock which allows for other processes to pick it up. This ensures that the memory is properly managed and there are no memory glitches in the program.



### 3.2 Critical Details

Based on the design criteria there were many design challenges that presented themselves. One of the considerations in the design is to allow the main thread to be able to remove items from the linked list of alarms. When an alarm is removed from the linked list of alarms there needs to be some mechanism to inform the display thread that it has been removed or conflicts could happen. Take the following example:

The program is running and there are 3 alarms in the list of alarms. A display thread has been assigned to the second alarm in the list and is printing alarm messages every 5 seconds. The user decides to remove the second alarm and types the appropriate command. The following events take place:

1. The main thread immediately removes the alarm from the list of alarms
2. The display thread assigned to that alarm decides to print a display message
3. the display thread tries to access members of the alarm but the reference to that alarm is now pointing to unallocated memory

Pointers referencing random places in memory will most likely crash the program or produce unpredictable results. To overcome this issue we decided to use the unique alarm id field of the alarm. Instead of storing the address of the alarm, we simply store the alarm's id number. Then whenever a thread needs access to an alarm, it must walk the linked list of alarms and find the alarm with the matching id. If the alarm is not found we can infer that the alarm was cancelled. Our implementation of the search function can be seen in figure 11

**Figure 11:** The getAlarm function used to find an alarm in the list of alarms

```
alarm_t* getAlarm(alarm_t *tag, int alarm_id)
{
    //if the tag is initially null the list is empty so return NULL
    while(tag != NULL)
    {
        if(tag->alarm_id == alarm_id)
            return tag;
        //go to the next element
        tag = tag->link;
    }
    return NULL;
}
```

Now there is a way to avoid memory corruption and detect cancel alarms. This is great, however, there still needs to be a way to keep track of which alarms have been assigned to a display thread. Taking into consideration the View\_Alarms command, it would be convenient to have all alarms that are assigned to each thread in a list. Then when the View\_Alarms command is issued, the list can simply be walked and printed. All of these conclusions led to the development of a second list of thread structures. (see figure 12 below)

**Figure 12:** The thread\_tag structure used to track a thread's assigned alarms

```
typedef struct thread_tag
{
    int num_alarms; //the number of alarms assigned to the thread
    unsigned int thread_id; //the thread's id number
    struct thread_tag* next; //The next element in the thread tag linked list
    unsigned int assigned_alarms[MAX_THREAD_ALARMS]; //an array of alarm ids that are assigned
}threadtag_t;
```

From the thread tag structure another linked list can be made representing each thread and its assigned alarms. Also, the number of assigned alarms is stored along with the thread id of the thread that the structure belongs. Additionally this provides an easy way for the display thread to know when to stop assigning alarms to itself.

## 4 Testing and Debugging

One of the first things we realized was that in order to test our program we were going to need a script to input values as the display thread prints to the command line every 5 seconds and so typing needs to be extremely quick if you want to add more than 5 alarms. Because we wanted to test many more than 5 alarms, we created the script below that creates 1000 alarms. In addition we added cases that change 3 alarms for every 20 created, cancel 3 alarms for every 51 created, and executes "View\_Alarms" for every 99 created alarms (every 99 and 51 were chosen because if it were 100 and 50 they would coincide with the Change alarm at multiples of 100 and would never execute).

**Figure 13:** Our test script designed to test various features of the program

```

massalarms.sh
1  #!/bin/bash
2  numalarms=10
3
4  #wait 3 seconds for the program to start running
5  sleep 3
6
7  #generate the alarms
8  for (( id = 1 ; id < numalarms ; id++ ));do
9      sleep 0.25;
10     #generate another alarm
11     change=$(( $id % 20 ))
12     cancel=$(( $id % 51 ))
13     view=$(( $id % 99 ))
14
15     if [[ $change -eq 0 ]]; then
16         echo "Change_Alarm($(( $id - 12 ))): 40 GOODDAY World!!! How you doing?"
17         sleep 0.25;
18         echo "Change_Alarm($(( $id - 3 ))): 23 GOODDAY World!!! How you doing? Im doing GREAT!"
19         sleep 0.25;
20         echo "Change_Alarm($(( $id - 19 ))): 36 GOODDAY World!!! THE WEATHER is VERY NICE today ;)"
21     elif [[ $cancel -eq 0 ]]; then
22         echo "Cancel_Alarm($(( $id - 30 )))"
23         sleep 0.25;
24         echo "Cancel_Alarm($(( $id - 35 )))"
25         sleep 0.25;
26         echo "Cancel_Alarm($(( $id - 15 )))"
27     elif [[ $view -eq 0 ]]; then
28         echo "View_Alarms"
29     fi
30     sleep 0.25;
31     echo "Start_Alarm(${id}): 60 Hello world"
32
33 done
34
35 echo "Exit"
36
37 exit 0

```

This script was piped into our alarm program using:

```
./massalarms.sh | ./NewAlarmThread
```

As well we used gdb to debug any issues we found with some modifications to the test script and input command.

Many issues were found using these methods, and general testing. One example of such is a segmentation fault that we found. The issue was with the linked list that holds the alarms. The last item in this list points to null, and we had an issue where if you deleted the last item in the list the 2nd last item (now the new last item) was not changed to now point to null. This was fixed very easily.

Another issue we discovered with testing was high CPU usage. This was not so much a bug but the effect of checking the alarm queue too much. With the combination of the `sched_yield()` and `sleep` functions appended at the end of the thread loop, each queue now performs properly without consuming the CPU's resources.

## 5 Process Flow Diagrams

**Figure 14:** Flow diagram outlining the main process.

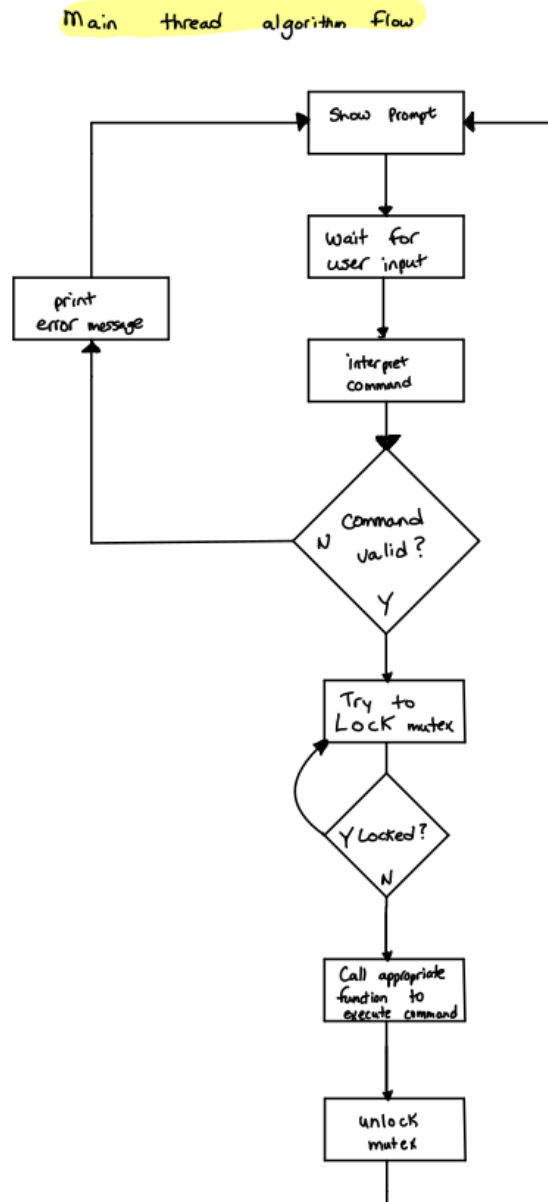
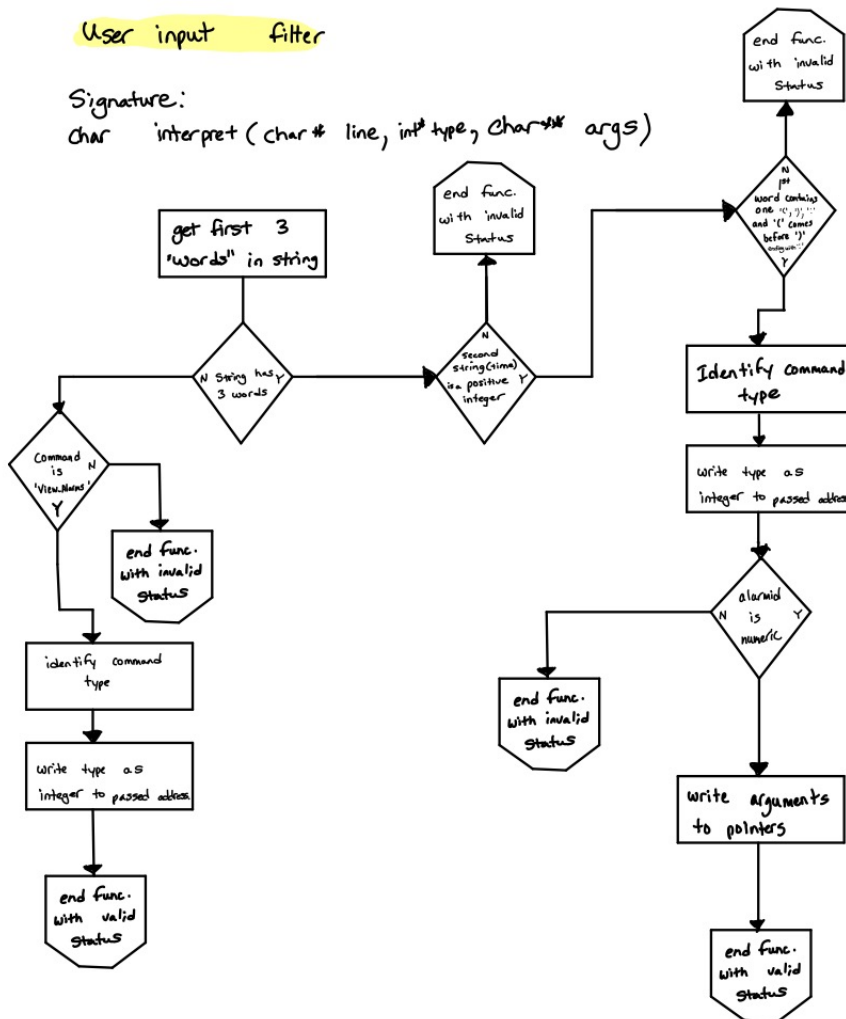


Figure 15: Flow diagram outlining user input



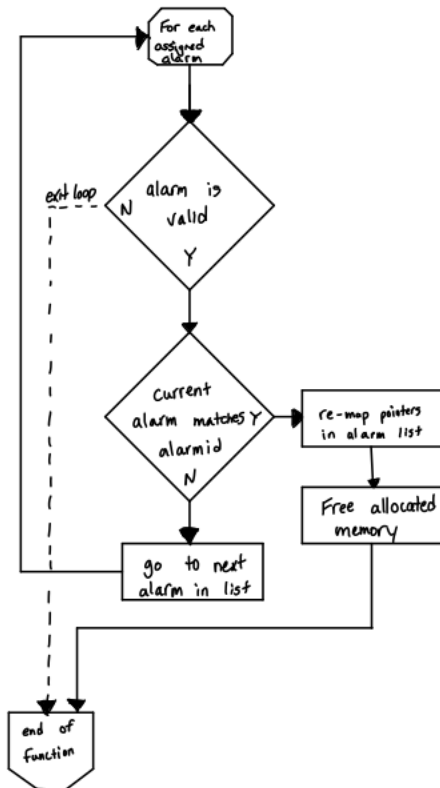
| Command        | Command Value |
|----------------|---------------|
| Start - Alarm  | 0             |
| Change - Alarm | 1             |
| Cancel - Alarm | 2             |
| View - Alarms  | 3             |

Function Returns:

0 ≡ invalid command  
1 ≡ valid command

Figure 16: Flow diagram outlining key processs in the display threads

Cancel Alarm (alarmid provided)  
Function signature:  
void (alarm-t \*head, int alarmid)



Change Alarm (alarmid provided)  
void changeAlarm(alarm-t \*head, int alarmid, int time, char \*msg)

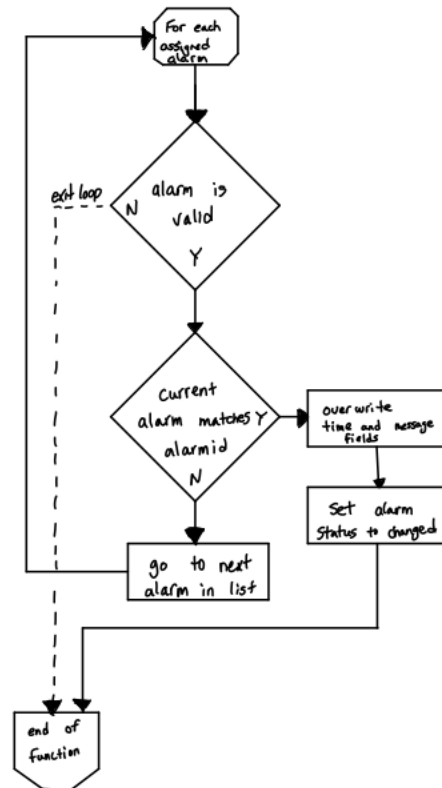


Figure 17: Flow diagram outlining key processs in the display threads

