POSIX Threads, Mutexes, Condition Variables, Semaphores, Producer-Consumer-shared-circular-Buffer

Victor Buica, Michael Sandrin, Alain Ballen, Brandon La

Lassonde School of Engineering

York University

Toronto, ON M3J 1P3

Emails: victobui@my.yorku.ca
msandrin@my.yorku.ca
alain612@my.yorku.ca
brandla@my.yorku.ca

Abstract

The main purpose of this assignment was utilising POSIX threads in order to create a program that will use POSIX unnamed semaphores. Similarly, in the previous assignment there were instructions and sample code given to us by the instructor which would be used to generate a makefile that would run our program and give the required outputs – as indicated in the README file. We had to test multiple outputs to ensure the program was working and in the end supply a report which elaborated on every part of the assignment in great depth.

The requirements of the assignment indicate an implementation of two alarm requests. These requirements were handled as follows in the program: New_Alarm_Cond.c.

The first requirement included 2 objectives:

1. a string message with a message number following it
2. the second type of alarm request which was prefixed with the key word Cancel.

If neither of the above commands was typed out then an error message would be displayed thus resulting in the request to be discarded.

The second requirement would implement a main thread in

New_Alarm_Cond.c determining whether the format of each alarm request is consistent with one of the two objectives listed above. If the imputed message string exceeds 128 characters, the rest of the message will be cut off. The main thread first checks whether the format of each alarm request is consistent with one of the two formats above, otherwise the alarm request will be rejected with an error message. However, if the objectives above are met then the alarm request would be submitted into an alarm list the same way that alarm_cond.c submits requests into an alarm list.

The third requirement was to ensure that alarm_thread in New_Alarm_Cond.c checks the updated alarm requests in the alarm list whenever the list has been changed. If the alarm request is NOT prefixed with the keyword Cancel in the alarm list, then the alarm_thread will immediately create a periodic_display_thread. Once the alarm request prefix Cancel was found then the alarm_thread will remove all the data corresponding to the alarm along with the message number from the alarm list.

The fourth requirement was to implement another thread where you were to retrieve the alarm requests with a priority number from a shared circular_buffer and insert this new message in a separate alarm_display_list where all outstanding requests would be placed based on their message number.

The last requirement indicates a use of a thread in order to look up an alarm request in a periodic manner with a specific message number in the alarm list, then print said alarm message on the display every (T) time seconds where (T) time is the first parameter in an alarm request.The display thread shall continue to be listed according to message number.

Once all of the above requirements are met then the synchronisation of the threads will act as readers and writers within New_Alarm_Cond.c. The required synchronisation threads access the following shared data located in the alarm list–which is implemented based on the Readers-Writers problem indicated on page 243 of the textbook and *Chapter 7* of the class notes.

Background

Semaphores

In programming and computer science, a *Semaphore* is an unique variable that is used to manage concurrent processes, controlling access to threads, and is commonly used in multithreaded operations with its use in solving the critical section problems. There are two types of *Semaphores*, *Binary* and *Counting Semaphores*.

Semaphores are denoted by the following operations: *Operation P* is used to decrement the *Semaphore* value until it reaches 0. *Operation V* is used to increment the *Semaphore* value. A *Semaphore* can only be modified by using either *Operations P or V* which first implements the *Operation P*, contains the *Critical Section* in the center, and ends with the *Operation V* as shown in the pseudocode snippet below:

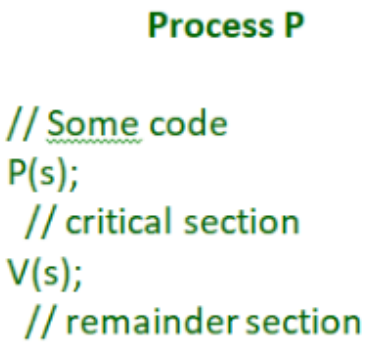

Figure 1: Semaphore Structure [2]

By standard, the *Wait* and *Signal* functions essentially act as *Operations P* and *V* respectively and make up the *Semaphore* process structure:

- Wait: If the value of a *Semaphore* (*S*) is not equal to 0 or invalid, the value decrements by 1 (*S*--). The *"Bust Wait"* occurs until the *While Loop* ends as shown in the code snippet below:

```
wait(S){
    while (S<0)
        ;//busy wait
    S--;
}
```

Figure 2: Wait or P Procedure

- Signal: The value of the *Semaphore* (*S*) increments by 1 after the *Critical Section Problem* is resolved. There is no *"Busy Wait"* time in *Signal Operation* as it is not required to reach a specific value as shown in the code snippet below:

```
signal(S){
    S++;
}
```

Figure 3: Signal or V Procedure
Set for change

Function

Two types of "Alarm requests"

Implementation:

As indicated in the above requirement we were to increase the type of requests the program can handle compared to the previous assignment. This was achieved by simply adding another sscanf to the main function. Along with this we implement a new void function: alarm_cancel (alarm_t *alarm) used to handle the second type of alarm requests.

Problems:

In terms of first implementing the *Cancel* command, there was no issue, however, since *Cancel* cannot be used as a standalone alarm we need to separate it from the normal list of alarms thus implementing a search through the list where the message number inputted by the user and if there was an alarm with that message number we just pulled it off the list.

Justification:

The sscanf is the fastest way of extracting information as it uses the built in scanning function of c in order to retrieve the inputs themselves. Along with this the utilisation of a second void function to read and detect cancel is better than implementing a void function: alarm_insert (alarm_t *alarm) as this will cause issues differentiating when the word *Cancel* is being used.

The main thread

Implementation:

The implementation is to simply cancel the alarm that is being replaced first then adding a new alarm to the list in order to preserve the manner in which the alarm_insert function is being conducted. The alarm_cancel will also be implemented in a similar way to alarm_insert where the calling will be dropped so that the order of the alarms can be maintained. Along with this a new condition is to be added to the main thread to cancel the alarm that is running in the waiting list so that the cancelled

condition can signal to the thread when to drop the waiting alarm and take the next one in the order processed.

Problems:

Initially if left untouched, there would be an expiration issue with the alarm timing as it does not reset the time itself and the following issue would occur if the following lines of code were implemented:

- 100 Message(1) 100
- 200 Message(2) 200
- 300 Message(3) 300
- 120 Message(3) 120

The order of alarm expiry should be 1,3,2 but instead 2,3 expire at the same time after 200 seconds.

Justification/solution:

To overcome this you could simply search through the list once it has been finalised (including the alarm in the waitlist) and then moving to the correct position in the list, the linked list is arranged based on the user input time from lowest to highest. This function will handle any changes made to the alarm, increasing or decreasing in time. This is a good solution but it raises the issue of making the alarm_cancel insufficient to handle all inputs. Therefore, we utilise our solutions of simply cancelling the alarm that is being replaced first by then adding a new alarm to the list in order to allow for the alarm_cancel to handle all inputs.

The alarm thread

Implementation:

The alarm_thread checks the alarm requests in the alarm list whenever the alarm list has changed. Once a new alarm request is NOT prefixed with the word *Cancel* within the alarm list, then the alarm_thread will immediately create a periodic_display_thread that will be specified in the last section.

When an alarm request is found with the prefix *Cancel*: the alarm_thread will remove all the data corresponding to the alarm request with the corresponding message number from the alarm list, including both the alarm request with the format specified in the 2 objectives listed earlier in the report.

Problems:

The main problems that were viewed will be discussed in the next part of the assignment.

Justification:

Since there is another section at the very end that will indicate a display to be utilised in order to remove the request, then simply calling that function act as a “kill two birds with one stone” type of scenario benefiting us by having a much more fluid and efficient code then possible coding other objects that could raise more errors and issues when debugging.

The consumer thread

Implementation:

The statement calls the circular_buffer and adds an alarm whilst incrementing the status of the circular alarm all the way till the buffer is full.

Problems:

The problem was synchronising the circular buffer for both the alarm_thread and the consumer_thread as there were segment errors for the messageNumbers when we were utilising the priority number from the circular buffer.

Justifications:

Since both the alarm_thread and the consumer_thread share the same buffer then there is no simple alternative method for implementation as it will simply result in the program to just loop around itself instead.

The periodic display thread

Implementation:

As indicated from the alarm_thread, the implementation of the new thread is done by printing the alarm and any changes made to it the moment the changes are applied. This will be accomplished by having a display thread for each alarm in the assignment.

Problems:

Since this is similar to our previous assignment there was not a real issue when implementing this thread.

Justification:

Since we utilise this thread in the alarm_thread, then it makes sense to implement it the way it was indicated above in order to run in tandem to make the resetting and removing of the alarms quite a simple task.

Testing

The following was imputed in the command prompt for testing the threads and it reads as follows:

1. red 40 % cc New_Alarm_Cond.c -D_POSIX_PTHREAD_SEMANTICS -lpthread
2. red 41 % a.out
3. Alarm> 10 Message(1) 10
4. "Alarm Recieved at 1449548506: 10 10"
5. Alarm> "Alarm Processed at 1449548506: 10 10"
6. 7 Message(1) 7Alarm displayed at 1449548516: 10 Message(1) 10
7.
8. "Alarm Recieved at 1449548517: 7 7"
9. Display thread exiting at 1449548517: 10 Message(1) 10
10. Alarm> "Alarm Processed at 1449548517: 7 7"
11. Alarm displayed at 1449548524: 7 Message(1) 7
12. 9 Alarm displayed at 1449548531: 7 Message(1) 7
13. Alarm> 9 Message(2) 9Alarm displayed at 1449548538: 7 Message(1) 7
14.
15. "Alarm Recieved at 1449548539: 9 9"
16. Alarm> "Alarm Processed at 1449548539: 9 9"
17. Alarm displayed at 1449548545: 7 Message(1) 7
18. Alarm displayed at 1449548548: 9 Message(2) 9
19. 13 Alarm displayed at 1449548552: 7 Message(1) 7
20. Message(3) 13
21. "Alarm Recieved at 1449548555: 13 13"
22. Alarm> "Alarm Processed at 1449548555: 13 13"
23. Alarm displayed at 1449548557: 9 Message(2) 9
24. Alarm displayed at 1449548559: 7 Message(1) 7

When printing the alarm list:

a. Alarm displayed at 1449548566: 9 Message(2) 9
b. Alarm displayed at 1449548566: 7 Message(1) 7
c. Alarm displayed at 1449548568: 13 Message(3) 13

d. Cancel Message(1)Alarm displayed at 1449548573: 7 Message(1) 7
e.
f. Display thread exiting at 1449548574: 7 Message(1) 7
g. Alarm> Alarm displayed at 1449548575: 9 Message(2) 9
h. Alarm displayed at 1449548581: 13 Message(3) 13
i. CaAlarm displayed at 1449548584: 9 Message(2) 9
j. ncel Message(3)
k. Display thread exiting at 1449548587: 13 Message(3) 13
l. Alarm> Alarm displayed at 1449548593: 9 Message(2) 9
m. Alarm displayed at 1449548602: 9 Message(2) 9

Descriptions of test cases in order:

In order to cover each testing issue and case that will arise we need to ensure that for the initial two types of alarm part we will run 3 cases:

1. When the alarm message displayed is one from the saved list
2. When the would Cancel is used as prefix
3. When the error is to be thrown

Essentially in order to ensure that each works we need to see if an error is thrown or not listed above. This is visible at lines 3, 5, 6, & 24 d.

In order to view the main thread test we used lines to write as many possible characters in order for the characters to be viewed as if they are getting cut off or not. This is visible on line 6 for example.

For this we simply need to see if on the output screen the following is printed in the format “Message(<Message_Number>):<Message>” this can be found on lines: 9, 23, 24, and most of the alphabetical ones as well.

The consumer thread can be tested by utilising the temporary listed objects and seeing when the overflow error will occur.

The periodic display thread is listed when the test cases reach the alphabet order to show when the alarmsted below in all of the alphabetical configurations as these will indicate when the alarm was created, what message it had and what message number it has corresponding to it.

References

[1]D. Butenhof, *Programming with POSIX threads.* Harlow: Addison-Wesley, 1997.

[2] “Semaphores in Process Synchronization - GeeksforGeeks,” GeeksforGeeks, Sep. 07, 2017. https://www.geeksforgeeks.org/semaphores-in-process-synchronization/