POSIX Threads, Semaphores, Readers-Writers Problem

# Assignment 3: Report

## EECS 3221: Operating System Fundamentals (Winter 2017)

## Instructor: Dr. Jia Xu

**Team Members**

Jose Cardona

Derek Li

Minh Nguyen

Harpreet Kaur Saini

**1. Program Design**

**1.1 Assumptions and Constraints**

In our program, we give users the capability to continuously input alarm requests while the displaythreads, after receiving a request, are still counting down. We have ensured thread-safety within our program at every critical section, and it is devoid of race conditions with the help of flags and semaphores, using the readers-first paradigm for synchronization.

**1.2 Constants, Semaphores and flags**

The program My\_Alarm has been designed with thread-safety and precision in mind. To start, we have defined our constants and structs:

#include <pthread.h>

#include <time.h>

#include <semaphore.h>

#include "errors.h"

#include <stdio.h>

#define TYPE\_A 0

#define TYPE\_B 1

#define FIRST\_ALARM 0

#define REPLACEMENT 1

#define NO\_MATCHING\_ALARM 2

#define MULTIPLE\_CANCEL 3

#define CANCEL\_REQ 4

/\*

\* The "alarm" structure now contains the time\_t (time since the

\* Epoch, in seconds) for each alarm, so that they can be

\* sorted. Storing the requested number of seconds would not be

\* enough, since the "alarm thread" cannot tell how long it has

\* been on the list.

\*/

**typedef struct** alarm\_tag {

**struct** alarm\_tag \*link;

**int** seconds;

**int** alarm\_number;

**int** request\_type;

**char** message[128];

**int** changed;

} alarm\_t;

/\*

\* The append\_list structure contains the alarm to move to a new thread and append.

\* This structure is useful in the case of flooded alarm requests, so it will append all in a batch,

\* or simply one by one.

\*

\* The reference to last allows us to append to the end of the list

\*

\*/

**typedef struct** append\_list{

alarm\_t \* alarm;

**struct** append\_list\* next;

**struct** append\_list\* last;

} append\_list;

/\*

\* Structure to hold the display thread's alarm object and

\* information regarding the alarm's removal (aka free() called on alarm).

\*

\*/

**typedef struct** display\_thread\_alarm {

alarm\_t \* alarm;

**int** removed;

} thread\_alarm;

/\*

\* Linked list holding a reference to the current threads

\* and their respective data structures.

\*

\*/

**typedef struct** thread\_alarm\_list {

**struct** thread\_alarm\_list \* next;

**struct** thread\_alarm\_list \* previous;

thread\_alarm \* data;

} display\_thread\_list;

**int** alarm\_thread\_flag = 0;

**int** reader\_flag = 0;

//for alarm list cleanup

alarm\_t \*alarm\_list = NULL;

append\_list \*list\_to\_append = NULL;

sem\_t main\_semaphore;

sem\_t display\_sem;

**int** append\_flag = 0;

**int** delete\_flag = 0;

display\_thread\_list \* thread\_list = NULL;

The constants relate to two things:

1. The type of alarm request
2. The status code from processing the alarm request (i.e whether it is new, a replacement, a cancel or an invalid attempt);

The structs and flags are explained as follows:

* alarm\_t: This structure holds a linked list of alarms carrying both the length of the alarm interval, the alarm message number, the message, the alarm request type (type A or type B request), and a value that indicates whether the value has been overwritten.
* thread\_alarm: This struct holds the alarm passed to a specific display thread spawned.
* display\_thread\_list: This struct holds the list of display threads. This data structure is a doubly linked list for simplicity in allowing us to remove an element from any arbitrary part of the list.
* append\_list: This struct holds the list of alarms to be mapped to display threads. IT works as a queue, with a constant time append to the last element, constant time polling of elements.
* alarm\_thread\_flag: This flag is used for the alarm\_thread to busy wait on critical section work.
* list\_to\_append: Holds the list of elements to append
* sem\_t main\_semaphore: The semaphore indicating entry to the critical section.
* sem\_t display\_sem: The semaphore used for the display threads readers-first synchronization
* append\_flag: indicates whether the alarm\_thread has a queued append action. Actions can arrive rapidly (in a batch), thus it’s entirely possible that threads are appended in a batch.
* delete\_flag: Same concept as append\_flag, just for deletions

During program initialization, the alarm alarm\_thread\_flag is set to 0, ensuring it will not busy wait for the semaphore for no reason.

**1.3 The Main Thread**

**int** main (**int** argc, **char** \*argv[]){

**int** status;

**char** line[160]; //Messages with higher allocated size

**char** msg[10];

**char** cancellation[10];

**int** err\_num, message\_num;

time\_t now;

alarm\_t \*alarm;

pthread\_t thread;

append\_list \* to\_append;

**if**(sem\_init(&main\_semaphore,0,1) < 0){

printf("Error creating semaphore!");

exit(1);

}

**if**(sem\_init(&display\_sem,0,1) < 0){

printf("Error creating semaphore!");

exit(1);

}

//Create the alarm thread

status = pthread\_create (

&thread, NULL, alarm\_thread, NULL);

**if** (status != 0)

err\_abort (status, "Create alarm thread");

**while** (1) {

printf ("Alarm> ");

**if** (fgets (line, **sizeof** (line), stdin) == NULL) exit (0);

**if** (strlen (line) <= 1) **continue**;

alarm = (alarm\_t\*)malloc (**sizeof** (alarm\_t));

**if** (alarm == NULL)

errno\_abort ("Allocate alarm");

//Parse input if it's of type A, check format correctness

**if** (sscanf (line, "%d %10[^(](%d) %128[^\n]", &alarm->seconds, msg, &alarm->alarm\_number,alarm->message) == 4) {

//Check if message is in the right format

**if** (strcmp(msg, "Message") == 0) {

//This is a type A message

now = time(NULL);

alarm->link = NULL;

alarm->request\_type = TYPE\_A;

alarm->changed = 0;

//Main thread always counts as a writer, never a reader.

status = sem\_wait(&main\_semaphore);

**if** (status != 0)

err\_abort (status, "Lock mutex");

err\_num = alarm\_insert(alarm);

//Check the return type of the function.

**switch** (err\_num) {

**case** FIRST\_ALARM:

printf("First Alarm Request With Message Number (%d) Received at %d: %d Message(%d) %s\n",

alarm->alarm\_number, (**int**) now, alarm->seconds, alarm->alarm\_number, alarm->message);

//Add the element to the append list

to\_append = (append\_list \*) malloc(**sizeof**(append\_list));

to\_append->next = NULL;

to\_append->alarm = alarm;

to\_append->last = NULL;

//If the list is null, make the list reference the element

**if**(list\_to\_append == NULL) {

list\_to\_append = to\_append;

} **else** {

//Otherwise, append in the next available

**if**(list\_to\_append->next == NULL){

list\_to\_append->next = to\_append;

list\_to\_append->last = to\_append;

} **else** {

list\_to\_append->last->next = to\_append;

list\_to\_append->last = to\_append;

}

}

//Set the append flag, telling the alarm\_thread we want to create threads

append\_flag = 1;

**break**;

**case** REPLACEMENT:

printf("Replacement Alarm Request With Message Number (%d) Received at %d: %d Message(%d) %s\n",

alarm->alarm\_number, (**int**) now, alarm->seconds, alarm->alarm\_number, alarm->message);

**break**;

**default**:

err\_abort(1, "Alarm added an incorrect type");

**break**;

}

alarm\_thread\_flag = 1;

status = sem\_post(&main\_semaphore);

**if** (status != 0)

err\_abort (status, "Unlock mutex");

} **else** {

printf("Error: Incorrect format\n");

}

//Else, check if of type b

} **else if**(sscanf (line, "%[^:]: %10[^(](%d)", cancellation, msg, &message\_num) == 3){

**if**(strcmp(cancellation, "Cancel") == 0) {

//Alarm is of type b

now = time(NULL);

alarm->alarm\_number = message\_num;

alarm->request\_type = TYPE\_B;

status = sem\_wait(&main\_semaphore);

**if** (status != 0)

err\_abort (status, "Lock mutex");

err\_num = alarm\_insert(alarm);

//Check the return type of the function.

**switch**(err\_num){

**case** NO\_MATCHING\_ALARM:

printf("Error: No Alarm Request With Message Number (%d) to Cancel!\n",

message\_num);

**break**;

**case** MULTIPLE\_CANCEL:

printf("Error: More Than One Request to Cancel Alarm Request With Message Number (%d)\n",

message\_num);

**break**;

**case** CANCEL\_REQ:

printf("Cancel Alarm Request With Message Number (Message\_Number) Received at %d: Cancel: Message(%d)\n",

(**int**) now, alarm->alarm\_number);

//Set the alarm thread flag

alarm\_thread\_flag = 1;

//Set the delete flag, telling the alarm\_thread we want to delete items.

delete\_flag = 1;

**break**;

**default**:

err\_abort(1, "Alarm added an incorrect type");

}

status = sem\_post(&main\_semaphore);

**if** (status != 0)

err\_abort (status, "Unlock mutex");

} **else** {

printf("Error: Incorrect format\n");

}

} **else** {

fprintf (stderr, "Bad command\n");

free (alarm);

}

}

}

The main thread is comprised of the following elements:

* Stack variable declaration (including error check variables).
* Semaphore initialization
* Alarm thread initialization.
* Main Event Loop.

Within the main event loop, it reads the line from stdin and matches it against the correct input formats. If it is a match, the line is parsed and processed. The function `*alarm\_insert (alarm\_t \*alarm)`* adds the new alarm to the global list, replaces an alarm currently in the list, appends a new cancellation request, or returns an error code based on incorrect inputs such as a cancellation request for a non-existent alarm. Whatever code is returned from `*alarm\_insert`,* the correct message is output and the main thread releases the alarm\_thread for busy wait.

Any alterations to the alarm list are critical alterations, so on a successful request parse, the main thread waits on the semaphore, then when the critical section is done, it will signal the semaphore.

**1.4 The Alarm Thread**

**/\***

\* The alarm thread's start routine.

\*/

**void** \*alarm\_thread (**void** \*arg) {

thread\_list = (display\_thread\_list \*) malloc(**sizeof**(display\_thread\_list));

**if** (thread\_list == NULL)

errno\_abort("Out of memory\n");

display\_thread\_list \* last = thread\_list;

/\*

\* Loop forever, processing commands. The alarm thread will

\* be disintegrated when the process exits.

\* The default alarm\_thread\_flag will be 0, so the thread will busy wait

\* until there are tasks for it

\*

\*/

**while** (1) {

//Busy wait while the flag is 0

**while**(alarm\_thread\_flag == 0);

sem\_wait(&main\_semaphore);

**if**(append\_flag == 1) {

//Create display threads

last = create\_display\_threads(last);

append\_flag = 0;

}

**if** (delete\_flag == 1){

alarm\_delete();

delete\_flag = 0;

}

alarm\_thread\_flag = 0;

sem\_post(&main\_semaphore);

}

}

The alarm thread follows the following execution sequence:

* Initializes the list of threads via malloc.
* Places a reference to the first element of the display thread list in a pointer on the stack
* Main event loop

**1.4.1 Alarm Thread Main Event loop**

After the initializations, the alarm thread waits for the alarm\_thread\_flag to signal it can wait on the critical section semaphore.

Once the alarm thread successfully acquires the semaphore, it checks whether the append flags and delete flags are set. If the append flag is set, it will create the batch of threads queued for creation. If the delete flag is set, it will free the memory allocated to the alarm objects, set a flag for the corresponding threads to exit, and free the memory used for the elements in the thread\_list linked list. Given the cancels can be in arbitrary positions of the list, the list needed to be a doubly linked list for simplicity of removal.

During the batch adds, the append\_list is cleared, and during the batch deletes, the references to the threads in thread\_list are freed. Both batch deletes and batch adds can happen in one critical section, and it will never result in a deadlock, as the display threads will check for the removal flag before any sort of read operation to avoid a segfault.

**1.5 Display Threads.**

During the display thread design, the target was making a generic design that allowed for fast checking of whether the thread has been terminated, as well as fast access to changes to the alarm structure. Thus, changes were made to `*alarm\_t` and `thread\_alarm`.* To accomodate for these changes. The tradeoff was made in regards to removals, in that batch removals will look through the thread\_list n times for n removals for an O(n^2) maximum time for a batch removal, but allowing constant time for removal checking which happens in n threads much more often and has to happen per iteration. This could be improved with a hash table implementation of bounded size, or of amortized size doubling, which would allow for constant time lookups to remove a thread.

/\*

\* The display thread

\* arg gets cast into the thread\_alarm struct containing information

\* pertaining to the specific thread

\*

\*/

**void** \* display\_thread(**void** \* arg) {

thread\_alarm \* alarm = (thread\_alarm \*) arg;

**char** msg[128];

strcpy(msg, alarm->alarm->message);

**int** alarm\_num = alarm->alarm->alarm\_number;

**int** interval = alarm->alarm->seconds;

**int** has\_changed =0;

time\_t now;

time\_t display\_interval = time(NULL);

**while**(1){

//Readers-first synchronization

sem\_wait(&display\_sem);

reader\_flag++;

**if**(reader\_flag == 1)

sem\_wait(&main\_semaphore);

sem\_post(&display\_sem);

now = time(NULL);

//If alarm was removed, exit.

**if**(alarm->removed == 1){

flockfile(stdout);

printf("Display thread exiting at time %d: %d Message(%d) %s\n",

(**int**) now, interval, alarm\_num, msg);

fflush(stdout);

funlockfile(stdout);

//Exit safely and not abruptly

//If this is the last thread to exit, make sure it unlocks the semaphore as well

sem\_wait(&display\_sem);

reader\_flag--;

**if**(reader\_flag ==0)

sem\_post(&main\_semaphore);

sem\_post(&display\_sem);

//Free the thread\_alarm struct used in this thread

free(alarm);

pthread\_exit(NULL);

}

//If the alarm has been altered, display the message and then set the altered flag.

**if**(alarm->alarm->changed == 1){

printf("Alarm With Message Number (%d) Replaced at %d: %d Message(%d) %s\n",

alarm->alarm->alarm\_number, (**int**) now, alarm->alarm->seconds,alarm->alarm->alarm\_number, alarm->alarm->message);

interval = alarm->alarm->seconds;

display\_interval = now + interval;

has\_changed = 1;

alarm->alarm->changed = 0;

strcpy(msg, alarm->alarm->message);

} **else if**(now >= display\_interval){

//Check if the interval has been satisfied and

**if**(has\_changed == 1){

printf("Replacement Alarm With Message Number (%d) Displayed at %d: %d Message(%d) %s\n",

alarm->alarm->alarm\_number, (**int**) now, alarm->alarm->seconds,alarm->alarm->alarm\_number, alarm->alarm->message);

} **else** {

printf("Alarm With Message Number (%d) Displayed at %d: %d Message(%d) %s\n",

alarm->alarm->alarm\_number, (**int**) now, alarm->alarm->seconds,alarm->alarm->alarm\_number, alarm->alarm->message);

}

display\_interval = now+ alarm->alarm->seconds;

}

//Readers-first synchro

sem\_wait(&display\_sem);

reader\_flag--;

**if**(reader\_flag ==0)

sem\_post(&main\_semaphore);

sem\_post(&display\_sem);

}

}

A generic display thread follows the structure:

* Stack variable initialization
* Main event loop

During the main loop, the display threads solves the synchronization prioritizing readers. The thread will check whether the alarm it’s periodically showing has been removed, and will exit gracefully by covering the edge case that the exiting thread happens to be the last thread to read, releasing the semaphore and returning the reader flag to the baseline value.

**2. Thread safety.**

The program ensures mutual exclusion of data, as explained in the previous section for each thread, via flags and mutexes. Thus, no thread ever dereferences a null pointer as it is freed.

The thread safety involved ensures no readers will ever read when a writer is modifying the alarm list. The only modifiers are the alarm\_thread and main thread, thus simplifying the problem via the data structures employed. Even when multiple alarms are freed, the readers will never access the alarm structures directly before checking whether they’ve been deallocated via a flag, thus no memory is ever null defererenced, and the readers clean up the memory related to the struct assigned to them.

**3. Challenges:**

Ensuring that the program could handle batched input forced a major design shift, as the original implementation was a one by one thread creator. Thus, on batched inputs, many threads would simply never be created, and batched deletes would delete only one thread, which made the program never end despite being given a batched file that was supposed to create and destroy all alarms in one pop.

Lastly, handling many different data structures in such a way that they would only be hierarchically accessed by readers/writers turned out into a bigger challenge than expected, as we needed to hold references to memory in such a way that we could efficiently handle multiple data structures to handle batched inputs while maintaining the invariant that threads would clean up any references to memory allocated exclusively to itself while avoiding memory leaks.

**4. Testing:**

For testing, We have developed a python script to check whether the input was passed to a correct display. The following python script tests input:

#!/usr/bin/python

**import** time

**import** pexpect

**import** os

**import** random

# Spawn My\_Alarm as a child process

child = pexpect.spawn("./New\_Alarm\_Cond")

#open the program log

**if not** os.path.isfile("program\_log.txt"):

os.system("touch program\_log.txt");

program\_log = open("program\_log.txt", "w")

child.logfile\_read = program\_log

**with** open("test\_file.txt", "r") **as** f:

lines = f.readlines()

responses = ["First Alarm Request With Message Number",

"Replacement Alarm Request With Message Number",

"Cancel Alarm Request With Message Number",

"Error: Incorrect format",

"Bad command"]

responses\_cancel = ["Error: No Alarm Request With Message Number",

"Error: More Than One Request to Cancel Alarm Request With Message Number",

"Display thread exiting at time"]

#do a flooded test

**for** l **in** lines:

**try**:

# Send input to process

child.sendline(l)

# Parse which thread received the request

**if** "Cancel" **in** l:

index = child.expect(responses\_cancel)

**print** "Line sent: " + l.replace("\n","")

**print** "Expected Response: " + responses\_cancel[index]

**else**:

index = child.expect(responses, timeout=30)

# Output results

**print** "Line sent: " + l.replace("\n","")

**print** "Response: " + responses[index]

**except** KeyboardInterrupt:

**break**

#do a non flooded test

**for** l **in** lines:

**try**:

# Send input to process

child.sendline(l)

# Parse which thread received the request

**if** "Cancel" **in** l:

index = child.expect(responses\_cancel)

**print** "Line sent: " + l.replace("\n","")

**print** "Expected Response: " + responses\_cancel[index]

**else**:

index = child.expect(responses, timeout=30)

# Output results

**print** "Line sent: " + l.replace("\n","")

**print** "Response: " + responses[index]

#Do not flood messages.

time.sleep(random.random())

**except** KeyboardInterrupt:

**break**

program\_log.close()

This script sends a list of commands testing both replacements, single adds, incorrect input, cancellation and double cancellations (which should never happen and return an error) in both a flooded fashion (as fast as python can send them) and with a random delay. The program output is piped to “program\_log.txt” and is included in the submit file.

**5. Conclusion**

We have crafted a thread safe, concurrent, multi-threaded periodic alarm using only two semaphores and one flag that fulfills the software requirements. Moving forward, improvements that could be made would be directing the output to either a file or a different display method, so the user may enter as many alarms as necessary without having stdout flooded as well as a hash table implementaiton for constant time adds and removes.