POSIX Threads

# Assignment 2: 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 mutexes. It may look out of order because the console is used for both input and output. However, the program is fully functional.

**1.2 Constants, Mutexes 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 "errors.h"

#include <stdio.h>

#define DISPLAY\_ONE 1

#define DISPLAY\_TWO 2

#define PRINT\_INTERVAL 2

#define DATEFORMAT\_SIZE 50

**typedef struct** alarm\_tag {

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

**int** seconds;

**struct** timespec time; /\* seconds from EPOCH \*/

**char** message[64];

**char** time\_retrieved[DATEFORMAT\_SIZE];

} alarm\_t;

//Structure to pass onto display thread

//Contains a thread number, the alarm list specific to the thread, and the latest request in the

**typedef struct** display\_struct {

**int** thread\_num;

alarm\_t \* alarm\_list;

alarm\_t \* latest\_request;

} disp\_t;

//MUTEX for alarm thread

pthread\_mutex\_t alarm\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

//MUTEX for display threads

pthread\_mutex\_t display\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

//Global alarm list. this list should be written into then deleted from.

alarm\_t \*alarm\_list = NULL;

//Display flag

**volatile int** display\_flag = 0;

//Alarm flag

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

//Date format

**const char**\* date\_format\_string = "%Y-%m-%d %H:%M:%S";

These are explained as follows:

* struct alarm\_tag: This structure holds a linked list of alarms carrying both the length of the alarm (in seconds), a struct timespec that holds the exact time in seconds and nanoseconds of the expiry of the request, the message to display, which is truncated if the input is too long, and a message indicating the time at which the message arrived at the display thread in time\_retrieved.
* struct display\_struct: This structure is a generic structure for display threads to receive data from the thread that spawned it, by typecasting from a void pointer. One of the biggest benefits for this data structure is mutual exclusion of data for each thread. After data-passing, threads holding a specific alarm n will not have that alarm ever accessed by another thread.
* alarm\_mutex: The critical section mutex for interactions between the alarm\_thread and main\_thread.
* display\_mutex: The critical section mutex for interactions between the alarm\_thread and a specific display thread.
* display\_flag: This is a flag for the busy wait loop for each alarm thread. It precedes the pthread\_mutex\_lock operation and keeps the thread in the waiting loop. It either counts down an alarm it has received or, in the case of no alarms, does nothing. The value 0 indicates the flag belongs to the alarm thread, and anything >0 corresponds to a specific display\_thread; i.e: display\_flag = 1 references a disp\_t struct display where display->thread\_num = 1. The moment this flag is set to the corresponding thread number, the thread will exit the busy wait loop and ask for the mutex, thus simplifying the need to signal which thread will ask for a lock, avoiding all race conditions.
* alarm\_flag: This flag is set between 0 and 1, 0 being the main thread is allowed to query the user for an alarm, and 1 meaning the alarm thread currently is allowed to ask for the mutex.

During program initialization, all flags are set to 0, meaning that while the other threads initialize, none will enter a critical section no matter what order they are spawned in.

**1.3 The Main Thread**

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

{

**int** status;

**char** line[128];

alarm\_t \*alarm;

pthread\_t thread;

**struct** tm main\_local\_time, \* err\_check;

**char** main\_local\_str[DATEFORMAT\_SIZE];

//Create the alarm thread;

status = pthread\_create (

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

**if** (status != 0)

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

/\* Main Event loop

\* Wait for stdin, parse if correct, then allocate to an alarm

\* otherwise, try again.[

\*

\*/

**while** (1) {

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

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 line into seconds (%d) and a message

\* (%64[^\n]), consisting of up to 64 characters

\* separated from the seconds by whitespace.

\*/

**if** (sscanf (line, "%d %64[^\n]",

&alarm->seconds, alarm->message) < 2) {

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

free (alarm);

**continue**;

} **else** {

//Lock the thread

status = pthread\_mutex\_lock (&alarm\_mutex);

**if** (status != 0)

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

//Allocate the time

clock\_gettime(CLOCK\_REALTIME, &(alarm->time));

//Set alarm time

alarm->time.tv\_sec += alarm->seconds;

//get the local time string

err\_check = localtime\_r(&(alarm->time.tv\_sec),&main\_local\_time);

**if**(err\_check == NULL)

fprintf(stderr, "Error Acquiring local time\n");

strftime(main\_local\_str,DATEFORMAT\_SIZE,date\_format\_string,&main\_local\_time);

flockfile(stdout);

//Output message to console

printf("Main Thread Received Alarm Request at %s: %d seconds with message: %s\n",

main\_local\_str, alarm->seconds, alarm->message);

fflush(stdout);

funlockfile(stdout);

//Set alarm list to the current alarm. NULL the next in sequence

alarm\_list = alarm;

alarm\_list->link = NULL;

//Set flag for alarm thread to wait on the mutex

alarm\_flag = 1;

//Unlock the alarm thread

status = pthread\_mutex\_unlock (&alarm\_mutex);

**if** (status != 0)

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

}

}

}

The main thread is comprised of the following elements:

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

The main thread first initializes all necessary variables needed locally on the stack: the input variable line, the pthread\_t thread id, the alarm struct that will be parsed from the input, the local time structures, and the string holding the local time.

The main thread first initializes the alarm\_thread (with error checking), then proceeds to the main event loop, since alarm\_flag is always initialized as 0. It blocks until the user enters valid input, at which point it acquires the mutex, parses the input into an alarm\_tag structure and prints to stdout the response that it has received the request. Printing to stdout is done atomically via flockfile. Once it has passed the reference of the structure to alarm\_list, the global structure used to pass data between main and alarm threads, it set the alarm\_flag so the alarm thread may wait for the mutex and enter a critical section, then unlocks.

**1.4 The Alarm Thread**

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

{

alarm\_t \*alarm;

**int** status;

//The structures to pass data to the display threads.

disp\_t \* display\_one, \* display\_two;

//The threads

pthread\_t display\_thread1, display\_thread2;

//Precision time checks.

**float** nano\_time;

time\_t sec\_time;

//String format time

**struct** tm alarm\_local\_time, \* err\_check;

**char** alarm\_local\_str[DATEFORMAT\_SIZE];

//Set the struct for the first thread

display\_one = malloc(**sizeof**(disp\_t));

**if**(display\_one == NULL)

err\_abort(EXIT\_FAILURE, "Display one allocation failed");

display\_one->thread\_num = DISPLAY\_ONE;

display\_one->alarm\_list = NULL;

//Set the struct for the second thread

display\_two = malloc(**sizeof**(disp\_t));

**if**(display\_two == NULL)

err\_abort(EXIT\_FAILURE, "Display two allocation failed");

display\_two->alarm\_list = NULL;

display\_two->thread\_num = DISPLAY\_TWO;

//Create thread one

status = pthread\_create (

&display\_thread1, NULL, display\_thread, (**void** \*)display\_one);

**if** (status != 0)

err\_abort (status, "Create display thread 1");

//Create thread two

status = pthread\_create (

&display\_thread2, NULL, display\_thread, (**void** \*)display\_two);

**if** (status != 0)

err\_abort (status, "Create display thread 2");

/\*

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

\* be disintegrated when the process exits.

\*/

**while** (1) {

//Check for the flag to see whether a display thread is currently performing an operation.

**while**(display\_flag != 0);

//Lock the display mutex.

pthread\_mutex\_lock(&display\_mutex);

//Block thread until main thread has actually receive a request.

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

alarm = alarm\_list;

//Receive mutex to assure mutual exclusion.

status = pthread\_mutex\_lock (&alarm\_mutex); //Block

**if** (status != 0)

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

nano\_time = (**float**)alarm->time.tv\_nsec\*1e-9;

sec\_time = alarm->time.tv\_sec;

**if**(nano\_time >= 0.5)

sec\_time += 1;

//Get the current time

err\_check = localtime\_r(&(alarm->time.tv\_sec),&alarm\_local\_time);

**if**(err\_check == NULL)

fprintf(stderr, "Error Acquiring local time\n");

strftime(alarm\_local\_str,DATEFORMAT\_SIZE,date\_format\_string,&alarm\_local\_time);

//If the time is even, send to display two, otherwise

//Send to display one

**if**((sec\_time % 2) == 0) {

display\_flag = DISPLAY\_TWO;

appendToList(&(display\_two->alarm\_list), alarm);

display\_two->latest\_request = alarm;

printf("Alarm Thread passed Alarm Request to Display Thread %d at %s: number of seconds: %d message: %s\n",

DISPLAY\_TWO,

alarm\_local\_str,

alarm->seconds,

alarm->message);

}

**else** {

display\_flag = DISPLAY\_ONE;

appendToList(&(display\_one->alarm\_list), alarm);

display\_one->latest\_request = alarm;

printf("Alarm Thread passed Alarm Request to Display Thread %d at %s: number of seconds: %d message: %s\n",

DISPLAY\_ONE,

alarm\_local\_str,

alarm->seconds,

alarm->message);

}

//Get rid of the reference.

alarm\_list = NULL;

//Unlock the display thread that received the request.

status = pthread\_mutex\_unlock(&display\_mutex);

**if** (status != 0)

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

//Unlock the main thread after the transaction.

status = pthread\_mutex\_unlock (&alarm\_mutex);

**if** (status != 0)

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

}

}

The alarm thread follows the following execution sequence:

* Stack variable initialization (including error check variables).
* Display thread data structure initialization.
* Display thread initialization and creation.
* Main event loop.

**1.4.1 Alarm Thread Main Event loop**

After the initializations, the alarm thread waits for the display\_flag to check if any of the display threads are currently in a critical section. During start-up, this flag is initialized to 0, so it proceeds to acquire the mutex before any of the display threads and busy wait until the main thread is done parsing an entry.

Then, the display thread acquires a reference to the newly passed object via alarm\_list and acquires the mutex lock to enter the critical section. Within this section, calculations from the alarm that was just passed are made to find out whether the time it was passed to the alarm thread is closer to an even or odd number, to then pass to the corresponding display thread.

After a display thread has been decided from the conditional, the display flag is set to the corresponding thread so it may busy wait for the display\_mutex, and the new alarm is appended to the selected display thread’s alarm list, sorted by alarms ending soonest. This is done in the case a user wants to set an alarm that ends quicker than one already set, so the display thread will count that alarm, then switch to the old one after it expires. The display thread cannot count down the alarm until it is received within its critical section thus assuring mutual exclusion to the corresponding display structure and ensuring thread-safety so no null dereferencing is made.

A reference to the latest set alarm is also set within the display structure, as the thread receiving the new alarm must print the alarm it just received, no matter its location on the linked list. Afterwards, a message is printed to stdout indicating the alarm was successfully passed to a thread, then the alarm\_list is set to NULL, to remove any global reference to any pointers that will be freed later during alarm expiry.

Finally, the display mutex that the corresponding thread is waiting for is unlocked, followed by the alarm\_mutex. Note: The alarm\_thread will not enter a critical section or acquire a mutex until the display thread is done receiving the new alarm and printing the event to stdout. The display thread, after it’s done printing to stdout, will unlock the main thread and the alarm threads in that order, so the alarm thread will then busy wait for a new request, and the main thread will wait for new input from a user.

**1.5 Display Threads.**

During the display thread design, the target was making a generic design capable of supporting any number of display threads connected to one alarm thread, with the precondition of the alarm\_flag and display\_flag existing, then delegate any logic related to display thread alarm assignment to the alarm thread.

Thus, only one display thread function was needed. Much of the heavy logic towards display thread assignment is located within the alarm thread.

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

//Variables to print the current and time interval

time\_t print\_time;

//flag for whether the print\_time variable has been reset

**int** print\_flag = 0;

//Reference to previous alarm to be free

alarm\_t \* oldref;

//the display struct.

disp\_t \* display = (disp\_t \*) args;

//Structure to acquire current time with nanosec precision

**struct** timespec now;

//Double precision floating point for time comparisons

**double** time\_nsec, alarm\_time;

//Time printing struct;

**struct** tm local\_time;

**char** local\_time\_str[DATEFORMAT\_SIZE];

**char** expiration\_str[DATEFORMAT\_SIZE];

**while** (1){

/\* If the display flag is not set, loop on the alarm list

\* If it's empty, do nothing. If it's not, loop printing out

\* every two seconds. Free the alarm after the duration.

\* If there's another alarm in the queue, resume that alarm.

\*

\* It is done this way due to sleep() pausing the entire thread.

\*/

**while**(display->thread\_num != display\_flag){

**if**(display->alarm\_list != NULL){

//Calculate the current time in seconds.

//`now = time(NULL);

clock\_gettime(CLOCK\_REALTIME, &now);

time\_nsec = ((**double**)now.tv\_nsec) \* 1e-9 + (**double**)now.tv\_sec;

//Print flag is set in case we break out of the loop, then we know to

//re-set the time interval

**if**(print\_flag == 0){

flockfile(stdout);

printf("Display thread %d: Number of SecondsLeft %d: Time:%s alarm request: number of seconds: %d message: %s\n",

display->thread\_num,

display->alarm\_list->time.tv\_sec - now.tv\_sec,

display->alarm\_list->time\_retrieved,

display->alarm\_list->seconds,

display->alarm\_list->message);

fflush(stdout);

funlockfile(stdout);

//Set the print time, rounding to seconds is okay

print\_time = now.tv\_sec + PRINT\_INTERVAL;

//Set the precise alarm time.

alarm\_time = (**double**)display->alarm\_list->time.tv\_nsec\*1e-9 + display->alarm\_list->time.tv\_sec;

print\_flag = 1;

**continue**;

}

//If the current time is greater than or equal to the target time

//Print and free.

**if**(time\_nsec >= alarm\_time){

//Print alarm done and a newline for the user to display alarm

//Get the local time

localtime\_r(&(display->alarm\_list->time.tv\_sec), &local\_time);

strftime(local\_time\_str, DATEFORMAT\_SIZE, date\_format\_string, &local\_time);

flockfile(stdout);

printf("\nDisplay Thread %d: Alarm expired at %s: %s\n",

display->thread\_num,

local\_time\_str,

display->alarm\_list->message);

printf("alarm>");

fflush(stdout);

funlockfile(stdout);

oldref = display->alarm\_list;

//If there is a next item in the list, free the old reference and move to that one.

**if**((display->alarm\_list)->link != NULL){

display->alarm\_list = (display->alarm\_list)->link;

free(oldref);

}

//Otherwise, just free the reference and null.

**else** {

free(oldref);

display->alarm\_list = NULL;

}

print\_flag = 0;

**continue**;

}

// If the current time has finally reached the print time,

// Print. Note: Should be error checked.

// now should never actually excede print\_time.

**if**(now.tv\_sec >= print\_time){

print\_time = now.tv\_sec + PRINT\_INTERVAL;

flockfile(stdout);

printf("\nDisplay thread %d: Number of SecondsLeft %d: Time:%s alarm request: number of seconds: %d message: %s",

display->thread\_num,

display->alarm\_list->time.tv\_sec - now.tv\_sec,

display->alarm\_list->time\_retrieved,

display->alarm\_list->seconds,

display->alarm\_list->message);

fflush(stdout);

funlockfile(stdout);

}

}

//Do nothing if it's null

}

//Lock the display thread to make sure the append operation to the list is atomic.

pthread\_mutex\_lock(&display\_mutex);

//Set time

clock\_gettime(CLOCK\_REALTIME, &now);

//Get the time the request was received

localtime\_r(&(now.tv\_sec),&local\_time);

strftime(display->latest\_request->time\_retrieved,DATEFORMAT\_SIZE,date\_format\_string,&local\_time);

//Get time alarm expires

localtime\_r(&(display->alarm\_list->time.tv\_sec), &local\_time);

strftime(expiration\_str, DATEFORMAT\_SIZE,date\_format\_string, &local\_time);

//Display the last request received.

printf("Display thread %d: Received Alarm Request at time %s: number of seconds: %d message: %s, ExpiryTime is %s\n",

display->thread\_num,

display->latest\_request->time\_retrieved,

display->latest\_request->seconds,

display->latest\_request->message,

expiration\_str);

//Unlock the display mutex.

pthread\_mutex\_unlock(&display\_mutex);

//Unlock the display flag back to the alarm thread

display\_flag = 0;

//Unlock the main thread print

alarm\_flag = 0;

print\_flag = 0;

}

}

A generic display thread follows the structure:

* Stack variable initialization
* Main event loop

During the main loop, each display thread will busy wait until the flag is set for it to receive a new request, assuring mutual exclusion to acquire the mutex and print the corresponding message to stdout. During the critical section, the display thread will print the message that it has received a new request to stdout, set the time retrieved on the alarm it just received, and unlock all flags to 0 to resume busy waiting for a new alarm.

During the busy wait, the alarm will count down every 2 seconds until the soonest ending alarm expires, after which it displays the time at which the alarm expired, then transitions to the next alarm, or if the list is empty, it will simply busy wait while doing nothing.

**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.

Thread-safe mechanisms were necessary to ensure standard functions such as printf were executed atomically. Any critical print is surrounded by flockfile(stdout), fflush(stdout) (to flush the buffer to avoid unintended behavior, and funlock(stdout), that works like a mutex to the output stream.

The thread-safe reentrant functions, as described by the posix standard, were used during any instance where thread-safety could be compromised. Thus, localtime\_r was used instead of the <time.h> classic localtime, as it the latter references a point in memory which could be subject to race conditions, and is not reentrant. All other posix function calls during both busy waits and critical sections are thread-safe. Thus:

* Calls to localtime\_r are safe, as the function is reentrant.
* Calls to strftime are safe, as no calls to setlocale() are done at all in the program.
* All references to data structures are safe.

**3. Challenges:**

Ensuring the display\_thread function can support any number of display threads managed by an alarm thread was challenging. However, accomplishing this improved simplicity within our code, as behavior among threads was consistent. Debugging was made simple by making the alarm passing to only one thread. Given that the design of the display\_thread function assures that no other threads will experience a race condition due to the display flag, we can observe the other functions within the thread in a vacuum, while remaining assured that this behavior will properly propagate amongst all threads.

Lastly, thread-safety within all standard POSIX functions proved to be documentation heavy and required an in-depth understanding of each function and structures related to the function to assure each function is re-entrant and returns the desired output.

**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/python2.7**

**import** time

**import** pexpect

**import** sys

**import** random

# Sample input

correct\_input = "2 I love operating systems!"

# Number of loop iterations

loop\_iters = 1000

# Responses to be parsed

expect\_arr = ["Display thread 1: Received", "Display thread 2: Received"]

# Spawn My\_Alarm as a child process

child = pexpect.spawn("./My\_Alarm")

#Set error tolerance

offset\_tolerance = 0.015

errors = 0

**for** \_ **in** range(loop\_iters):

# Send input to process

child.sendline(correct\_input)

# Parse which thread received the request

index = child.expect(["Display thread 1: Received", "Display thread 2: Received"], timeout=1)

# Measure the system time immediately after. Is subject to some error, as it's not at the same time stdout was printed

# Also, from the time of receiving the alarm to printing there is a small delay,

# as there are also delays due to locking of stdout.

time\_set = time.time()

# Calculate the time offset to figure which thread the alarm should have gone.

offset = time\_set - int(time\_set)

**if** offset >= 0.5:

time\_offset = int(time\_set) +1

**else**:

time\_offset = int(time\_set)

# Output results

**if** (time\_offset % 2 == 0) **and** index == 1:

**print** "[+] Correct. So far so good. Thread %d time %.3f" % (index+1, time\_set)

**elif** (time\_offset % 2 != 0) **and** index == 0:

**print** "[+] Correct. So far so good. Thread %d time %.3f" % (index+1, time\_set)

**else**:

**if** (offset-0.5 <= offset\_tolerance):

**print** "[~] Error within 15msec tolerance bound in read error"

**else**:

**print** "[!] Uh oh. Error in Thread: %d with time %.3f"%(index+1, time\_set)

errors+=1

time.sleep(random.random())

**print** "[.] Errors found: %d Error percent: %.3f%%" %(errors, errors\*100/loop\_iters)

This script sends a request to My\_Alarm set to expire within 2 seconds 500 times with random delay intervals between requests, reads the time the response from stout was received mapped to which thread, compares the time the thread that received the request printed to stdout vs the current system time and outputs the result. Given that the function expect() has some parsing delay, we set an offset of approximately 15 msec of tolerance for errors. In the file provided with out project, test\_out.txt, we can see that over 500 attempts, there was 1 error (which was 1 millisecond off from the set tolerance). Running this test more times, we are bound to run into situations where there might be a slow parse on the python side, or the calls to flockfile() delay a thread print, but in general, we can see the code works as is required.

Given that the My\_Alarm code is thread safe, the call to timeout in the parser never raises an exception, we can see the code can handle 500 alarms in millisecond intervals while still printing to stdout as alarms expire, thus the code satisfies all the requirements.

Given sscanf is limited to 64 characters and you cannot cause an overflow in one line, then there is no need to test for incorrect input in regards to parsing numbers and text.

**5. Conclusion**

We have crafted a thread safe, concurrent, multi-threaded alarm function using only two mutexes and two global flags 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. Aother path to improving the program would be a more obvious method of the alarm notifications (possibly some sound).