# EECS 3221 3.0A Operating Systems
# Assignment # 2

**Jason Hoi**
Student Number: 210989523
CSE ID: cse31078

**Raghad Khudher**
Student Number: 208451098
CSE ID:cse93170

**Heaten Mistry**
Student Number: 211869476
CSE ID:cse13131

**Jhan Perera**
Student Number: 211502671
CSE ID: cse13187

Date Submitted: November 2nd 2015

# Table of Contents

# 1. About

In this report we will discuss our program, *My_Alarm.c*, including but not limited to, the design implementations, design methodology(s), hardships in implementing and modifying the code, and the trials we faced debugging our program. Now, before we continue, we must explain that our program, *My_Alarm.c,* is a modification of another program, *alarm_mutex.c*, given to us (along with other files) from our professor.

In this report we will describe our journey of implementing and modifying a multi-threaded user application. At the end of this report will be the source code of all the files and dependencies that we either developed or used to help solve this problem. Included with the source file(s), will be comments that will help explain, in quick details, how each function or section of code works and what it does.

# 2. Introduction

The purpose of this assignment was to modify and develop a program, given to us, that will run an "*alarm*" simulation using a multi-threaded approach. This means, that at the end of our implementation the user will be able to input any number of *alarms requests, which* in turn will all execute in parallel. Each alarm will consist of a time (in seconds) and a message (a string of characters), in which after the elapsed time has passed, the program will display the message associated with the alarm.

Let's go into more detail of what happens when a user's interacts with our program. When the program is executed the user is prompted to input a integer and a message. The integer is the amount of time, in second at which the alarm will go off. Once the appropriate time has elapsed the program will display the message back to the user via a terminal. Each time an alarm request is made the program creates a separate thread just for that alarm. This allows the user to create as many alarm requests as possible and each request will be running on its own independent thread. Once the alarm request has ended the thread is killed (by executing the *pthread_exit()* command) and memory is freed (by executing the free() command), allowing the newly freed up memory space to be used by other programs and/or data.

Since we are using threads over processes we are gaining a lot of benefits.For one, when the thread is created the thread will be using a lot less memory than an individual process. This helps by allowing the user to make more and more alarm requests without worrying about running out of memory. Another benefit is time. Since threads can run in parallel it's easy for the to keep making request and not worry about if the other threads are still executing. Since the user doesn't have to wait for a thread unless it's for a return value, the user is free to keep feeding alarm requests into the program. Next we shall analyze the initial program that was provided to us.

# 3. alarm_mutex.c

*alarm_mutex.c* is the original file we were given to modify. In this program there are two threads: The main thread, which executes the main function, and a second thread, which executes the alarm_thread function.

## 3.1 main(int argc, char *argv[])

In this program, the main function, which is executed by the main thread, first creates a secondary thread that will execute the function alarm_thread, in parallel to the main thread. Then the main function will enter a while loop.

In this loop, the main thread, upon each iteration will prompt the user to type in an alarm and then parse the newly inputted alarm into a new *alarm_t* structure (a linked list in our case), which would then be stored into the alarm list, *alarm_list*. This loop will continue until the program ends, either normally or abnormally.

## 3.2 alarm_thread(void *arg)

In this function, which is executed by a secondary thread in parallel with the first, the body of the loop is executed. Within this loop, the function, gets the first alarm off of the alarm list and then sleeps until the specified amount of time has passed in regards to the alarm. Once done, the function prints out a message and free's the memory allocation of the alarm. The function then iterates and begins executing the body of the while loop again.

Now due to the alarm list being accessed by two different threads, synchronisation would become a problem. But in this program a mutex lock is used, by both threads, such that race conditions will not occur.

# 4. My_Alarm.c

*My_Alarm.c* is an modified version of *alarm_mutex.c*, which initially would take in an alarm request, iterate through the alarm_list, wait $n$ seconds, then print out the alarm message, and finally terminate the thread. *My_Alarm.c* works of this idea and adds multi threaded functionality to all the alarm request.

## 4.1 new_thread_function(void *arg)

We used one new method in our implementation of *My_Alarm.c.* This method is used to create the child threads in the *alarm_thread* function. This function takes one void pointer argument. Each time this function is called, the head of the *alarm_list* is removed and that element is used as the argument for this function.

Once the function is called the first thing that happens is that the argument passed is cast to an *alarm_t* data type. This allows us to extract all the valuable information and save it to local variables. This in turn allows us to allocate the correct amount of memory and to have direct access to all the information we need to execute the alarm. The other variables are *sleep_t* (holds the value 1 for a one second sleep timer) , *start* ( a seconds counter), and *end* (help calculate when the alarm timer expires). There is one while loop that will check if start is less then end, it will then increment start by 1 second. If start < end, then the child thread will sleep for one second and then print out the specified message in the while loop body and then iterate once more. Once the start >= end the alarm has expired. The message is printed on the screen and the memory is freed to be used by the next alarm request. Finally the method terminates by calling *pthread_exit(0)* which cleans up the rest of the thread.

## 4.2 alarm_thread (void *arg)

This function has been modified slightly to assist with creating a child thread every time an alarm request is made by the user. The main while loop is waiting to get the alarm request from the user. As each request is read in we added the “Alarm Retrieved” message that is displayed on the screen to inform the user that the alarm request is retrieved from the list and is being passed to the child thread. At this point the program jumps into the *pthread_create()* method and the child thread is born. The main thread continues to wait for more alarm request from the user while the child threads are doing all the work.

## 4.3 main()

In the main thread we did a little bit of modification to notify the user of what is going on with the request that has been made. Simply we added a print statement that will print out “Alarm Received” to notify the user that the alarm request was successfully received and added to the list of requests. After that it is again added to the list and passed to the main thread.

# 5. Issues Encountered

This assignment was a bit of challenge to get the exact mechanics to work as intended. The first issue we came across was how we would implement the child thread in a simple and efficient way. After a few issues dealing with *pthread_create()* function we decided to implement a method that will take care of all the details specified in the instructions. This allowed us to focus on one small segment of code rather than the whole program. This also allowed us to keep the code simple and localized.

Another issue we had was how we would get the child thread to print out the specified message, once every second. Thinking about it, it is known that instructions are executed within nanoseconds, if not faster. Meaning, assuming it takes around a nanosecond to execute a single line of code, in order to make our child thread print out a statement every second, we would have to make the child thread execute the instructions inside a while loop, to which it will sleep for one second each iteration and then perform a printf statement. Once done it would check the condition of the loop and iterate yet again. Overall the implementation used in regards to this problem offers an effective and efficient solution to that requirement of our program.

# 6. Conclusion

In conclusion, this assignment helped us understand a lot about how threads work. Even though the threads we created are detached from one another it still similar to how joint threads would work in a real world scenario. We were able to create child threads and make them run in parallel without waiting or depending on the other threads to finish. This allows the the user to input as many alarm request as possible without worrying about the other threads finishing first or running out of memory. On our next assignment this idea might be expanded to have the child threads all synchronised and be joined together. None the less this assignment helped us completely understand how threads are handled in an operating system and how they play an important role in our modern system.

# 7. Source Code