# Assignment 2 Report

## Important Aspects:

### Main

The main function first gets the input from the command line. The main function calls the get_cmd_type function; this function is responsible for parsing the line into their variables, and determining what type of message was given (Type A,B,C, or none of the above). If message is Type B (Create Thread), switch case 1 is responsible for dealing with it. It creates the tread using pthread_create, and it inputs the thread into a list of threads. If it is Type C(Terminate), switch case 2 is responsible. It removes the threads with the same Message Type from the thread list, and unassigned alarms for the alarm_list, and frees memory. It also terminates the thread using pthread_cancel. If the message is Type A(create alarm), Case 3 is responsible. It allocates memory space for an alarm and assigns the struct with the variables from the command. It is placed in the alarm_list through function alarm_insert, which sorts by increasing MessageType. If it is none of the above, a 'Bad Command' message is shown. At the end of the main function, the main thread yields so other threads have CPU time.

### alarm_thread

The alarm thread holds local variables to the thread such as a sublist of messages that have been assigned to it. The alarm thread also pushes and pops a function to stack for when the thread is terminated; thread_terminate_cleanup frees memory allocated to the thread's alarms and unlocks mutex for when thread is cancelled, preventing a deadlock situation.

At the start of the while loop, it locks the mutex. If the alarm list is empty, it goes to pthread_cond_wait, which waits for the pthread_cond_broadcast in alarm_insert; when an alarm is inputted into the list, all threads are made known, so they look in the list. The mutex is unlocked immediately afterwards allowing other threads to be made aware of the alarm, and it does not lock the mutex twice, which would lead to undefined behaviour.

If the alarm_list contains the same message type as the thread, and it has not been assigned, the alarm is assigned to the thread; if not, the thread waits for another alarm in pthread_cond_wait. If a suitable alarm is found, it is placed in the threads sublist by increasing time. If the alarm is ready to be outputted, it is outputted. If the alarm is not ready to be outputted, the thread searches the list again, and assigns them if they are suitable. If a new alarm is assigned, then it is put into the thread sublist. If the new alarm has a shorter time than the current alarm, than it replaces the old alarm, and it is outputted when time is up; if time is not up, the loop continues.

The thread has 2 cancellation points (pthread_cond_wait and sleep) so the thread can be cancelled if the list is either empty or not empty.

### thread_terminate_cleanup

This function is responsible for cleaning up a thread after it has been cancelled by the main thread. The function frees up the memory occupied by its alarms and unlocks the mutex so that other threads have access to the mutex, so that it does not lead to a deadlock. This function will only be called if the mutex is locked, and this function is only called if the main thread calls pthread_cancel, and the thread is at a cancellation point; either at pthread_cond_wait(mutex is locked after condition arrives) if list is empty, or sleep(0)(mutex is locked at this point) if the list is not empty, since they are the only cancellation points in the threads.

**alarm_insert**

This function is responsible for inputting an alarm into a global alarm list. This function puts the alarms in order of message type. The function locks the mutex so that a thread does not have access to the list as it is being modified and unlocks it afterwards.

**alarm_remover**

This function is responsible for removing an alarm from the global list after is has been assigned to a thread. It locks the mutex before modifying the list and unlocks it afterwards.

**pthread_mutex_t alarm_mutex and pthread_cond_t alarm_cond**

The alarm_mutex is locked whenever a function is reading or modifying the global alarm_list. This is done to synchronize the data(alarm_list) between the threads. The alarm_cond is broadcasted whenever a new alarm has been inserted into the alarm_list by function alarm_insert to make the thread aware that a new alarm has been inputted

**<u>Issues and Solutions:</u>**

One problem we faced was the inability to terminate the thread correctly. When ‘pthread_cancel(temp_thread->thread_id)’ had been called by the main function, the mutex lock might have been still locked when the thread had been terminated; no other alarm thread was able to be run as a result as a result of deadlock. To overcome the issue, ‘pthread_cleanup_push(thread_terminate_cleanup**,** (void*)thread_alarm_list)’, and ‘pthread_cleanup_pop(1)’ and been implemented in the alarm thread. Whenever the main thread had called cancel, the thread_terminate_cleanup function had been able to unlock the threads before the thread is terminated, and free memory space utilized by the threads alarms.

Another issue we had faced was if a message was entered into the command line before a thread had been created. The time given to the alarm will continue to run even if no thread had been able to service it. To overcome this issue, whenever the alarm does get assigned, the alarm’s time is updated, so it is outputted correctly.

An issue we faced was Segmentation Errors that had occurred due to the inability to traverse the alarm_list, and the thread list for when a Type C message had been inputted into command line. This was overcome by debugging the code and printing out messages for where the code was working.

In order for the data to be synchronized, the data has to be locked before accessing it. If the data is not locked, it would lead to inconsistencies if the thread order is not the same as intended. The alarm mutex is locked before the alarm_list is modified or read, and unlocked after modification.

Passing an int to the thread was not straight forward. The value of the message type (was passed in the pthread_create function) changed if multiple threads are created at the same time by the user through the command line. To overcome this issue, the message type variable was allocated to memory space, and freed upon the thread receiving it.

If the main thread had multiple inputs at the same time, it might hog up more CPU time in-between messages than intended. In order to combat this, the thread was yielded, allowing the other threads to access CPU time.

**Quality of Design**

The program has modular design, so that functions that are known to work are kept separate from the thread function as well as main. Mutex are used appropriately so that it is not locked if it is already locked, or unlocked if already unlocked, which would have led to undefined behaviour. The mutex is made sure to be unlocked after the termination of a thread. All memory space that had been allocated, is freed upon termination of the thread.

**Testing**

All 4 of us had created test cases to see if the output of the program is desired. The test cases vary widely so see if the program hangs, or a segmentation error occurs. Proper design choices were made so that this did not occur.

Sample input.txt

Create_Thread: MessageType(2)

Create_Thread: MessageType(2)

Create_Thread: MessageType(3)

Terminate_Thread: MessageType(2)

5 MessageType(2) Will meet you at Grandmas house at 3

5 MessageType(3) Will meet you at Grandmas house at 3

Terminate_Thread: MessageType(2)

Create_Thread: MessageType(2)

5 MessageType(3) Will meet you at Grandmas house at 3

5 MessageType(6) Will meet you at Grandmas house at 3

invalid message

Create_Thread: MessageType(6)

5 MessageType(3) Will meet you at Grandmas house at 3

5 MessageType(6) Will meet you at Grandmas house at 3

5 MessageType(3) Will meet you at Grandmas house at 3

5 MessageType(6) Will meet you at Grandmas house at 3

Terminate_Thread: MessageType(3)

5 MessageType(2) Will meet you at Grandmas house at 3