Assignment 2 Report

Important Aspects:

Main

The main function first gets the input from the command line. The main function calls get_cmd_type function; this function is responsible for parsing the line to 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 from the thread list, 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 by increasing MessageType. If it is none of the above, a 'Bad Command' message is shown.

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; pthread_mutex_unlock unlocks mutex for when thread is cancelled. 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. If the list contains the same MessageType 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 looks into 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 it replaces the old alarm, and it is outputted when time is up.

Issues and Solutions:

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 locks were still locked when the thread had been terminated; no other alarm thread was able to be run as a result. To overcome the issue, 'pthread_cleanup_push(pthread_mutex_unlock, &alarm_mutex)', and 'pthread_cleanup_pop(1)' and been implemented in the alarm thread. Whenever the main thread had called cancel, the pthread_mutex_unlock function had been able to unlock the threads before the thread s terminated.

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 Erros 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.