Introduction

The purpose of this assignment was to create an alarm application which supports the following command types: “Type A”, “Type B”, and “Type C”. In Particular, “Type A” commands create a new alarm or replace an existing one. Each alarm has a time delay, a message type, a unique message number and a message. A message type is used to categorize different alarms. Time delays denote the amount of time to wait between consecutive prints of the alarm message. Message numbers represent the ID associated with each alarm. Finally, message represents the corresponding alarm message to be displayed when the alarm has been activated. “Type B” commands create a new alarm thread of the given message type which handles the specified alarms of that message type. Finally, “Type C” commands terminate a single message with the specified message number. If an alarm thread becomes idle because of the alarm cancellation, then the thread self-terminates. The goal of this assignment was to use the **pthreads** library in a more advanced way by using **semaphores** to implement the solution to the readers-writers’ problem, **conditional waiting**, and much more.

Design Decisions

Throughout this assignment, there were several components that provided us with an opportunity to make our own design decisions. The following section is informative.

Design Decision #1: Handling Input Strings

- We make use of the **read_line** defined in **std_ulilities.h** to read a full line of input without making any assumptions about its length. It allocates memory as needed to hold all of the currently read characters. We then parse the read line for all required inputs. For example, in “Type A” commands, we look for a message which satisfies the requirements and obtain the first 50 characters while truncating the string if needed, by not considering any remaining characters.

Design Decision #2: Maintaining Commands List

- We separate the commands list into three sublists each containing different commands types, namely “Type A”, “Type B”, and “Type C”. This makes our program more modular since we have separated the command lists allowing us to process commands of that type separately. Furthermore, it also prevents having a “**superman-struct**”, meaning that there is no single data structure containing all command attributes. If one were to implement the “**superman-struct**” design, they would encounter the following issue. With the increase in the number of commands, the one structure encapsulating all of them would need to grow as well to the point that each command may only use a tiny fraction of the attributes (i.e., memory) allocated to it. The modularity of separate command lists also makes it easier to process since we are not required to check the type of the command.

Design Decision #3: Waiting for New Commands

- The command thread, (the thread that handles commands) waits on a **conditional variable**(new_cmd_insert_cond_var) until it has been signaled by the main thread signifying a new valid command. This ensures that there is at least one **valid** command that should be processed. This is a significant design decision since it prevents the wastage of CPU usage since the command thread would otherwise be continually searching through an empty commands list, finding nothing to execute.

Design Decision #4: Locks and Synchronization

- Since there are **only two writers to the commands list** (the main and the command threads), we use a **single mutex** (cmd_mutex) to synchronize access. The **alarms list,** however, has **a single writer** (the command thread) but also **many readers** (the main and alarm threads) which is why we implemented the solution to the **readers-writers'** problem using **binary semaphores** (alarm_rw_bin_sem initialized with value 1, alarm_r_bin_sem initialized with value 1, and reader_count set to 0).

Design Decision #5: Main Thread Obtaining Locks

- The main thread is a **writer to the commands list and a reader to the alarms list**. It never accesses both lists at the same time, and as such it only needs to hold one set of locks at any given point in time. It locks **cmd_mutex** whenever it requires (write)access to the commands list and unlocks when it is done with the writing. It makes use of the **obtain_alarm_read_lock** and **release_alarm_read_lock** functions implemented in lock.c. These functions respectively obtain and release a reader lock on the **alarms list** for the calling thread which may be the main thread but also may be one of the alarm threads as described in Design Decision #6.

Design Decision #6: Alarm Thread(s) Obtaining Locks

- The alarm threads **only read the alarms list**. They only require a reader lock to read from the list. Unlike the previous version, in this version, the alarm threads are not required to periodically check the list for new alarms to handle. Therefore, we decided to implement the solution to the **readers-writers'** problem as we potentially will have many readers on the alarms list, however, the only writer to this list is the command thread. The reason why we only require a **reader lock** on the **alarms list** is that we are not changing any globally shared data on the design level. The link_handle attribute allows us to **embed** an alarm handler's local list into the global list for faster runtime and memory allocation efficiency. The **is_assigned** flag makes it so that different alarm handler threads do not even attempt to access alarms that have already been assigned. Message types(**msg_type**) further help us in making the data disjoint by only allowing alarm handlers to manage alarms that have the same type as them. Finally, since we **enforce unique threads** of a given message type (this is enforced by the main and command handler threads), we arrive at the following conclusion: The local lists are accessible globally but remain **solely accessed by the owning** alarm handler thread. However, note that even without one of the above, we cannot be able to guarantee that a reader lock would be enough. For example: If we allowed multiple threads of a given type, then there would actually be an issue with the reader lock since many alarm handler threads of a given type could find an unassigned Alarm structure of their desired type at the same time, and all of them would be able to overwrite each other's selection process.

Design Decision #7: Command Thread Obtaining Locks

- The command thread is a **writer to both the alarms list and the commands list**. Unlike the previous threads that do not require simultaneous locking of both lists, the command thread needs to **access both lists at the same time** which means that it has a potential of becoming a bottleneck on the whole application. Therefore, we decided to be very careful about the **order** that this thread **obtains locks** and the **length** of its **critical section**. The command thread **first** acquires the lock that is **more challenging** (meaning more competitors exist for that lock) to get. Namely, the **writer lock on the alarms list**, since potentially there will be multiple alarm threads each handling their own set of messages. However, the **mutex**(cmd_mutex) on the **commands list** is only fought over by the main and the command threads. We also decided to **release and obtain locks multiple times** during one loop of execution of the command thread. The reason being that the command thread is **only in a state of executing in its critical section**. This means that if it were allowed to hold both locks for the entirety of its critical section, it would prevent the application from being responsive by **starving** all other threads. For more detail, visit the lock.c file and read the comments in the cmd_handler_obtain_locks function.

Design Decision #8: Processing “Type A” Commands

- This is achieved by the main thread adding “Type A” commands to the “Type A” commands list and then signaling the command thread for execution (mentioned in **Design Decision #3**). Once the Command thread has been signaled it processes the newly inserted “Type A” command, which then determines if the message number associated with the command is unique. If so, it will insert the alarm into the alarms list in sorted order (by message number). If not, the existing message in the alarms list with the specified message number will be canceled, and the new alarm will replace it. All of this is achieved by using the

insert_alarm function defined and implemented respectively in **alarm_def.h** and **alarm_def.c**. Finally, the “Type A” command previously contained in the commands list is removed.

Design Decision #9: Process “Type B” Commands

- When receiving a valid “Type B” command, the main thread stores the command in the “Type B” commands list. The command thread is then woken up by a signal on the conditional variable. It will create a new alarm thread and **encapsulates its ID** in the corresponding “Type B” command node and stores this **until the thread’s cancellation/termination**. This alarm thread now handles the alarms associated with the specified message type.

Design Decision #10: Process “Type C” Commands

- When the command thread has been woken up to execute a “Type C” command, it will search for the alarm that needs to be canceled. If the alarm so happens to be **unassigned** to an alarm handler thread, then the cancellation will be performed **immediately**. Otherwise, a **flag** will be set, and the command thread will sleep on a **conditional variable waiting** for the responsible alarm handler thread to detach and remove the canceled alarm from its list. The alarm handler thread will then signal the same conditional variable (**alarm_cancel_cond_var**) waking up the command thread thus allowing it to finish the alarm cancellation.

Design Decision #11: Application Log File

- To provide a better user experience, we decided to **separate some** of the **application output** and automatically **redirecting it to a file**. The reason is that the application is intended to be executed from the command line. In the terminal, all three main streams (input: **stdin**, output: **stdout**, and error: **stderr**) are seen together in the one screen even though they are in fact separate. In fact, the user's input will be littered with error messages, application log messages but also alarm prints. In the design, however, much of that is printed to the specified **log file** which means that the user can continue entering inputs without being interrupted.

Design Decision #12: Application Termination

- When the main thread reaches the **End Of File (EOF)**, or when an **error** has occurred during its run, it attempts to **clean up** the entire process. It cancels the command thread, all active alarm threads while also freeing all existing alarms in the alarms list. It also destroys all created mutexes, semaphores, and conditional variables described in the above design decisions used to create all of the needed synchronizations.

Design Decision #13: Why Use goto!!!

- As Professor Donald Knuth pointed out in his "*Structured Programming With Goto Statements*" written in 1974, **goto** is a powerful tool which can be used properly to write more elegant and optimized code if one follows all of the following principles as we **<u>clearly have</u>**.
    - All **goto** uses have been well documented.
    - There are very few **tag**s used.
    - **Tag** names are semantic (as all names should be) but also **memorable** in terms of code location. Here, memorability means that any programmer who has seen the code for a very short amount of time can easily remember what the first executed statement after a given tag is.
    - There are very few **goto** statements.
    - **goto** statements only go in one direction (usually downward). Downward jump, just means that each **goto** goes from a smaller line number to a larger line number.

    - No nested **goto**s. This means that if a jump was executed, then there are no more immediate jumps from the new location
    - **goto** statements are only used when it makes the code more elegant and/or optimized. You should make sure that there is **no other simple way** to achieve the same result. Some will say that all **goto** statements should be **replaced** with break and/or continue. However, notice that both **break** and **continue** are also jumps in machine language which means that replacing a **goto** with a break or continue is not necessarily better.

Finally, note that all **powerful** things can be misused. However, this does not mean that they should not be used. It only means that one must truly understand how and when to use them properly. The argument that **condemns goto** (avoid using a powerful thing since it can be misused) can also be extended to the **C** and **C++** programming languages but as we all know, abandoning these languages would be a terrible mistake. **With great power comes an even greater responsibility!!!**

Design Decision #14: Uses of goto in the main thread

- In the main thread, we only have a **single tag** (RESET_AND_READ_NEXT_LINE). Therefore, we can see that some of the principles have already been satisfied (documented, few tags, semantic and memorable, only downward, nested goto). All **goto** uses, allow the programmer to avoid creating a command validity flag and continuously nest the entire code at every stage with an if statement. Instead, as soon as the program determines that the current read line of input is not valid, it proceeds to reset certain variables involved in the input parsing and reads the next line by jumping to the appropriate location. Therefore,

goto uses have made the code more elegant and easier to maintain and understand. Note that these gotos could have been replaced with **continue**s and moving some code around (would have to reset previously mentioned variables at the beginning of the loop instead of the end). However, if one were to do so, they would make the code less elegant due to being less logical.

Design Decision #15: Uses of goto in the command thread

- The only other uses of **goto** in the entire application is in the command handler thread. There is a total of **four tags** (TYPE_B_AFTER_OBTAIN_LOCKS, TYPE_B_BEFORE_RELEASE_LOCKS, TYPE_C_AFTER_OBTAIN_LOCKS, and TYPE_C_BEFORE_RELEASE_LOCKS) used. So again, we can see that some of the principals have already been satisfied (documented, few tags, semantic and memorable, only downward, few gotos, nested goto). The main reason why these tags were used was due to **Design Decision #7**. These jumps allow the command thread to short-circuit its execution and process commands faster and thus create a smaller bottleneck on the application. Therefore, **goto** uses have made the code more optimized and efficient in a way that would not have been possible otherwise.