Technical Report

System Programming for ARM (ELE00138M) 2023-24

Contents

Executive Summary

This technical report supplies an executive summary of the modifications and additional features implemented to the base DocetOS. The pre-emptive operating system now includes mutual exclusion via a re-entrant mutex and counting semaphores, a fixed-priority task scheduler, a more efficient task sleeping mechanism, a more efficient wait and notify system, and priority inheritance for mutexes.

Mutual Exclusion via a Re-Entrant Mutex

Control of concurrency to prevent race conditions when accessing shared resources between threads is possible with mutual exclusion. A re-entrant mutex enables recursive locking, allowing a thread to lock a resource multiple times without causing deadlock. When acquiring a pre-acquired mutex, the requesting task must enter a waiting state to block the attempt. Tasks move out of the wait state on the release of a mutex.

Design

Only one task can hold a re-entrant mutex at any given time, but any task can request a mutex many times. We can define a mutex as a structure holding a field that points to the task control block (TCB) that owns this mutex and a counter to track the recursive acquisitions. There must be acquire and release functions that tasks can call to obtain and release a given mutex. A notify function must be present for calling during the release of a mutex to alert all tasks currently in a waiting state waiting for a mutex.

The mutex acquire function will load the pointer of the task stored in its field, only if zero can the task safely obtain said mutex. A mutex TCB pointer field value that is non-zero and not equal to the current OS TCB pointer signifies that another task owns the mutex. If the mutex task field points to the same task as the one acquiring, then this represents a recursive acquisition, denoting a counter increment. When a task requests an already acquired mutex by another task, the OS should force the requesting task to enter a waiting state. The most basic implementation consists of the mutex acquire function simply calling the OS yield function to switch context. However, for optimised efficiency, waiting tasks will move into a separate task list, away from the primary task list, so the scheduler will not try to switch to it.

Any task owning a mutex can safely release it using the release function. When a task calls the mutex release function, the function must first verify that the mutex owner is requesting this release to prevent the unsafe behaviour of tasks releasing mutex owned by other tasks. Once confirmed, the function can safely decrement the mutex counter. Once the counter reaches zero, the function can unset the TCB pointer field in the mutex structure. The notify function, which the release function calls now, will move all waiting tasks into the scheduler task list.

Safety

Safeguarding the mutex TCB pointer field from corruption due to multiple simultaneous modifications is possible with exclusive load and store CMSIS intrinsics. The LDREX intrinsic loads the mutex TCB pointer field, and when changes occur on this field before storage, STREX will fail. If STREX fails due to a mutex modification part-way through the acquire function, the function should restart by reloading the altered mutex task pointer field.

In operation, a task requesting an owned mutex will call the acquire function, which confirms that the mutex is unavailable and calls the OS wait function to send the requesting task into a waiting state. However, if a context switch occurs before the OS wait function call and the release of the mutex, the task will continue to enter a wait state, causing a deadlock. The mitigation of this potential bug is possible with the inclusion of check code logic.

The declaration and initialisation of a counter, alongside a global getter function for this counter, can form the foundation of the check code logic. Each call to the notify function must increment this counter. The mutex acquire function must retrieve the check code, and to wait for the requesting task, the function must pass the code to the OS wait delegate, which then compares the code with the global code. Matching codes signify that a notification did not occur, and the task can enter the waiting state without any issue. If there is a code mismatch, the wait delegate must not send the task to the waiting list. Therefore, the mutex acquire logic will iterate to restart the acquisition logic.

Including a yield function call within the mutex release function corrects the spinlock bug when tasks acquire and release a mutex in a tight loop. This call triggers a context switch, allowing another task to run appropriately in case of a successful release, preventing mutex hogging.

A More Efficient Task Sleeping Mechanism

Developing a more efficient sleeping mechanism consists of adding sleeping tasks into a separate sorted list, on which the scheduler will only need to check the head to verify if any sleeping tasks need waking. An insertion-sorted list is a workable solution due to its simple programmatic logic. However, an $O(n^2)$ average time complexity shows insertion-sort inefficiency. A binary heap, on the other hand, is much more efficient for sorted insertion with an average time complexity of $O(1)$.

Design

Any task that requires sleep for a definite amount of time will call the OS sleep function, passing in the duration of ticks. This function will first calculate the wake time for the sleeping task, storing this value within the data field of the TCB. The sleep flag will be set high within the TCB state field. The function must also remove the task from the scheduled task list and insert it into the sleeping task list. Finally, setting the PendSV bit in the Interrupt Control and State Register invokes a context switch to schedule the next task.

Implementing a generic variant of the heap structure can improve storage reusability for other parts of the operating system, greatly simplifying the organisation of code files. The heap structure's definition, with the insert, extract, and empty check utility functions should be present in the project via a specific source and header file for OS-wide usage. The definition of heap storage size should be present in the scheduler header file, allowing access from the scheduler source file, where the definition of a heap comparator function for a sleeping task-specific use case exists.

Given the nature of heaps, heap storage initialisation happens before the heap initialisation. Therefore, the number of entities in the heap is pre-defined without means of changing at runtime. The initialisation of the heap store array and the sleeping heap within the scheduler source file should allow the scheduler to check whether there are any sleeping tasks to wake and extract from the heap and insert them into the scheduler task list. The sleep delegate can also perform task list removal and sleeping heap insertion.

During prototyping, this implementation involved a sleep header and source file within the project consisting of a heap structure for TCBs and a sleep delegate function. While this approach involved straightforward logic, the implementation would have resulted in disorganisation from distributing sleeping task logic between the scheduler and sleep codes. Confinement of all sleep logic to the scheduler is possible with a generic approach.

Safety

With this implementation, the scheduler task list and the sleeping heap should have protection from multiple simultaneous modifications. Removal of tasks from the scheduler task list occurs in the sleep function, and since this doubly linked list is not thread-safe, the sleep function must be an SVC delegate for privileged access. Heap insertion logic should only occur within the sleep delegate function for thread-safe removals from the scheduler task list. Heap extract logic should only take place in the OS scheduler function to wake tasks by moving from the heap to the task list. The heap is thread-safe since the modification logic is within SVC interrupts and the OS scheduler.

Fixed-Priority Task Scheduler

In the base DocetOS system, the scheduler performs a round-robin switch between tasks within the scheduler task list. Implementing a fixed-priority task scheduler ensures the completion of the highest-priority tasks before scheduling lower-priority tasks.

Design

Implementing an additional priority field allows storing the priority level within the TCB itself. The TCB initialise function can accept an additional argument to store in the TCB structure. Although a data field already exists which can implement the storage of priority levels, it stores the wake time when a task is sleeping. Albeit the possibility to preserve priority levels to protect overwritten data fields, such as by implementing array task lists to retain priority, it is far easier to create an additional priority field.

Converting the currently implemented task list to an array of multiple lists can enable the separation of tasks given a priority level. Using a pre-processor directive allows for the hard coding of priority levels, which equals the array elements. The task add, exit, wait delegate, and sleep delegate functions must be able to handle the array. The scheduler must incorporate a loop to iterate through the priority levels, ensuring the schedule of the highest-priority tasks before the lower-priority ones. This logic requires extra care to ensure long-running tasks do not possess a higher priority, allowing lower-priority tasks to get the opportunity to run. For better code readability, priority levels should be in 1-index during user assignment, then converted to 0-index within internal code before array access logic, with smaller numeric values denoting higher priorities.

Safety

The existing task add function will feature slight modifications to allow functionality with an array of doubly linked lists, resulting in thread-unsafe logic. The placement of task list removal logic in SVC delegate functions can assume thread safety against concurrent modification corruption.

There also needs to be logic to prevent the accidental assignment of invalid priority levels to tasks. The assignment of a priority number null or greater than the highest possible priority number defined by the pre-processor directive must clamp to the highest priority number. Compiler errors and warnings protect against the user not passing in a value or passing in negative values.

Since there are a variable number of lists in the task list array, initialising the head pointer fields in each list is difficult. An option is to use a for-loop to iterate through all array elements, setting each head field to zero. Another option is to use GNU array range extensions to initialise the field in a single line of code. The latter option utilises a GNU extension and can affect code portability across different compilers. I have chosen not to initialise the fields since the C specification states that without explicit initialisation, an object with static storage duration initialises to a null pointer if it has a pointer type [1].

A More Efficient Wait and Notify System

On mutex release, the notification logic notifies all items in the waiting list. Maintaining a list of waiting tasks within each mutex can drastically improve efficiency. By having one waiting list per mutex, the mutex release only needs to notify one task from the head of the list, thus massively improving the current logic of notifying all waiting tasks.

Design

One approach for a mutex-specific wait list is a first-in-first-out-based (FIFO) ordering system, with the queue of tasks requesting an acquired mutex in order of attempt time. Another approach is a priority-based ordering system, based on task priority levels, granting the highest priority task first on mutex release. The FIFO-based approach is much fairer as it will service on a first-come-first-serve basis. Although this prevents the starvation of lower-priority tasks, it will result in higher-priority tasks waiting for lower-priority tasks to complete. A priority-based approach will improve OS responsiveness by servicing high-priority tasks first. Furthermore, it can solve priority inversion by preventing higher-priority tasks from waiting for a mutex in a queue before a lower-priority task.

High responsiveness and suppressed priority-inversion provide substantial benefits to an embedded operating system. The OS-wide heap implementation from earlier will benefit this implementation, as it can provide a highly efficient ordering system based on the priority parameters. The declaration of a heap can be within the mutex structure to create a unique heap store for each mutex instantiation, and there needs to be a heap store declaration inside the mutex structure. The static initialiser implementation from the base DocetOS will no longer work since the heap needs initialising with the heap store and comparator function pointers.

[1] 7.8 ¶10 p126 (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

Implementing a mutex creation function can replace the static initialiser. The initialiser function in the mutex source code will have access to the heap comparator function implementation for mutexes. Therefore, the initialiser function must instantiate a new mutex and return it. A drawback to this method is the crowding of the stack from the function return of the entire mutex structure, and the function cannot return a pointer since the mutex is a local variable. A function to create a mutex which only initialises a mutex given a pointer to it resolves stack crowding. The heap implementation must allow modifications to the heap store at runtime to ensure the mutex creation logic operation.

Safety

In the mutex-wide wait list implementation from earlier, mutex-wide check code logic was present to ensure mid-operation mutex releases will not affect tasks that are about to enter the waiting list, guarding the system against tasks imprisonment in the waiting list. Modifications to convert this safety mechanism to mutex-specific logic will incorporate mutexes containing the check code within their structure, with the notify functions incrementing and wait function checking on a mutex-specific level. This update will prevent the refusal to sleep when an unrelated mutex releases.

A mutex-specific notify function will replace the current notify all function, now taking in a pointer to the mutex of the waiting tasks to notify. Since the head of the waiting list heap contains the highest priority task waiting for the mutex, this will be the task to notify. Modifications to the wait delegate function consist of accepting the mutex in question as an argument in addition to the check code, using the mutex-specific check code and waiting list heap to remove the current task from the task list and inserting it into the mutex wait heap. This new notify function no longer needs to be in the scheduler source files as it is better alongside mutex logic. To achieve this, the pending list and all push/pop functionality must be accessible from the mutex source code. Despite being a mutex-specific implementation, the wait function will stay in the scheduler source files to ensure the scheduler task list is local to the scheduler due to its safety-critical nature.

Preserving the pending list logic safeguards the thread-unsafe scheduler task list, with the mutex-specific notify function adding tasks to a pending list to ensure the scheduler task list will not succumb to corruption. The SVC delegate-based implementation of the OS wait function preserves the thread safety of the scheduler task list and the mutex-specific heap-based waiting list. Furthermore, considerations to preserve thread safety are paramount during the heap extract function call in the mutex-specific notify function. Converting this notify function into an SVC delegate ensures atomic operation execution.

Priority Inheritance for Mutexes

The combination of the current re-entrant mutex and fixed-priority task scheduler implementations incurs the possibility of priority inversion, where a lower-priority task holding a mutex blocks higher-priority tasks waiting for this mutex. A mutex-based priority inheritance implementation aims to resolve this defect by granting the mutex-holding task the priority of the highest-priority task that requests for said mutex if the priority is higher, achieving prompt mutex release.

Design

The implementation must contain logic with an initial check on whether the priority level of the requesting task is higher than that of the task that holds the mutex. If it is greater, the logic must remove the mutex holding task from the previous task list, add it to the new priority task pending list for the scheduler to sweep and process correctly, and finally add the requesting task to the mutex heap-based waiting list. Secondly, there must be logic to revert the mutex-holder priority to the level it was previously at before the inheritance to prevent the permanent promotion of lower-priority tasks. The TCB structure must contain an additional field to preserve the original priority level.

The current mutex acquire function calls the wait SVC delegate function in the instance where another task already holds the mutex that a task is requesting. After confirming check codes, the function removes the requesting task from the round-robin, inserts it into the mutex-specific heap-based waiting list, and then sets the PendSV bit to invoke a context switch. The wait function is the ideal location for the priority promotion logic consisting of a priority comparing if-statement and further functionality that updates the priority level of the mutex-holding task, preserving the original level and then adding it to the pending list. The context switch invokes the scheduler to schedule all pending tasks observant of the promoted priority level.

The initial idea to retrieve the highest priority waiting task was to develop a peek function in the generic heap implementation. Given the TCB retrieval via the current TCB function within the wait function, there is no need for this peek function. However, I will still develop the peek function to produce a more complete generic heap implementation. Though this peek function will be unused in the OS, a complete generic heap package would benefit the end-user.

The mutex release function resets the mutex task field and calls the mutex-specific notify function on a complete mutex release from the notification counter equal to zero. Implementing the task priority restoration functionality is ideal before the notify function call in the mutex release function. Before the waiting task notification and yielding the OS, additional logic should be present that restores the priority field value, removes the task from the scheduler task list, and then pushes it to the pending list only if the priority field differs from the original priority. There should not be a need for a PendSV bit setting to invoke a context switch inside the notify function since once the function completes, the logic returns to the mutex release function and calls the yield function, invoking a context switch.

Safety

The initialisation of the additional priority field must be simultaneous with the initialisation of the standard priority field in the TCB initialise function. With this, the additional priority field can accurately preserve the original priority level, even with multiple priority promotions, provided the logic does not modify this field elsewhere.

Initially, the idea for logic placement of the priority restoration implementation was in the mutex-specific task notify function. As it is already an SVC delegate, it is possible to restore task priorities within here and save the complexities of implementing an additional function to achieve this. However, a drawback of containing this logic within the mutex source files is that the doubly linked task list and its utility functions would not be accessible due to the static type qualifier for these objects. Since ensuring thread safety for doubly linked lists is a challenge, it makes more sense for them, including the utility functions, to be local. Making these objects globally accessible will bring some thread safety concerns. Therefore, it makes more sense to implement a standalone task priority restore delegate function within the scheduler source files, local to the doubly linked task list.

Mutual Exclusion via a Counting Semaphore

A counting semaphore implementation holds a fixed number of tokens which functions can obtain and release one by one, with the ability for any task or ISR to add tokens to the semaphore container. When requesting a semaphore token when none are available, the task must enter a waiting state. These tasks move out of the wait state once a token becomes available. A binary semaphore implementation can utilise this counting semaphore logic by initialising with a single token.

Design

A definition of a semaphore structure should contain three fields, two of which are integer variables that count the number of tokens and the number of notifications, and the third is a singly linked waiting list. To ensure logical similarities with the mutex interface, a semaphore initialisation function should be present to take in a semaphore pointer and total token count to initialise all fields for use. A semaphore acquire function must enable tasks to obtain a semaphore token, utilising the notification count acting as a check code and the decrement of the token count field of the semaphore. If there are no tokens left, the logic must call the semaphore wait function, sending the requesting task to the semaphore waiting list with the check code to assess whether a semaphore token has become available, ensuring tasks are not sent to the waiting list when it can obtain a token. The semaphore release function must be able to release tokens from any task or ISR. Therefore, there must not be any SVC delegate calls to ensure ISR compatibility. The function must increment the token counter and notify a waiting task by popping from the semaphore waiting list.

For simplicity and thread-safety considerations, instead of implementing a heap for a waiting list, like that in the mutex implementation, the semaphore logic can utilise a singly linked list, with the waiting tasks in a first-in-first-out arrangement. Currently, the singly linked utility function only contains push-to-head and pop-from-head logic. Therefore, to ensure FIFO, there must be a pop-from-tail function. The semaphore notify function must be a non-SVC delegate for mindfulness of ISRs calling to release tokens. The function should increment the notification counter of the semaphore and must pop the waiting task at the tail of the waiting list to schedule it by adding it to the pending list. The semaphore wait function, which the semaphore acquire function calls if no tokens are available, is safe as an SVC delegate since only tasks can acquire tokens. This function must verify the check code notification counter to prevent wrongful task waits. Once confirmed, however, the task can wait by entering the waiting list, and finally, the function must invoke the scheduler. For further simplicity, there will not be an additional field in the semaphore structure to prevent releasing more tokens than a maximum limit. Therefore, the end-user must be mindful of acquiring and releasing in pairs to eliminate accidental increases in the token pool. This implementation also allows the user to modify the token pool at runtime.

Safety

The design of the heap implementation does not consider thread safety to achieve a generic solution, providing difficulties when attempting to fuse with the semaphore waiting list logic. Despite the aspect of simplicity, utilising singly linked lists ensures the necessary thread safety for the semaphore requirements of giving ISRs the ability to release tokens. A thread-safe pop-from-tail function for a singly linked list must implement aspects of thread safety with exclusive load and store CMSIS intrinsics, ensuring the safe handling of situations where the list updates mid-way through the operation, such as when traversing to the end. The semaphore acquire, release, and notify functions must utilise exclusive load and store intrinsics to provide atomic read and write access to the counter variables. Checking the IPSR register allows for handler mode detection. Using the PendSV bit set in handler mode and the OS yield delegate in ordinary function permits safe yielding regardless of releasing from ISRs or ordinary functions.

Feature Demonstration and Conclusion

An emulation of a digital thermostat implementation aims to demonstrate all new features and improvements to this version of DocetOS. Although the thermostat design isn't optimal, it provides an interactive presentation showcasing the improved OS features. The primary duty of the thermostat is to sense the temperature via a sensor, control the heater to ensure the desired temperature is maintained, and broadcast its data to various peripherals, such as an LCD. The thermostat has a few threads to facilitate the connection with multiple remote devices for 'smart' controls. The thermostat outputs a log of all its actions to a console via the serial interface.

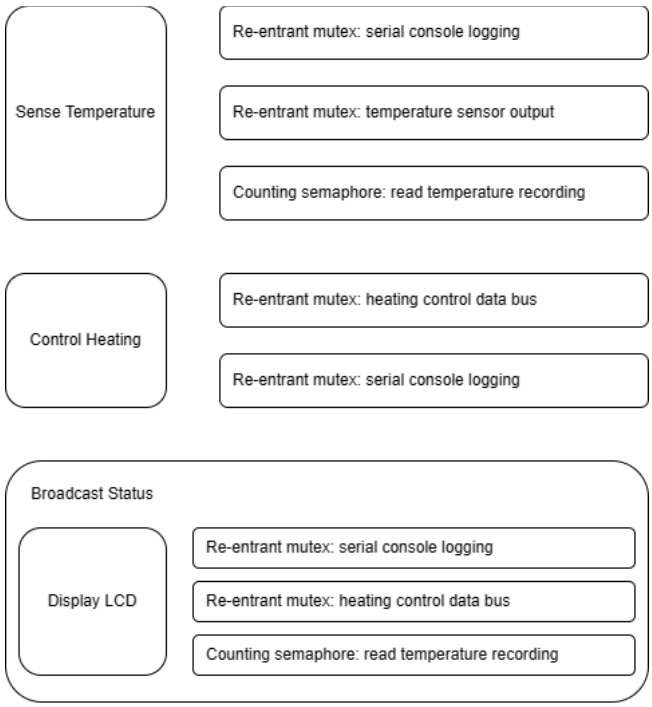

Figure 1 Priority 1 tasks.

Figure 1 shows the highest-priority tasks for execution by the thermostat. The thermostat communicates with an external temperature sensor using a mutex to prevent simultaneous write access to this data bus to emulate a temperature reading. A counting semaphore implementation allows for limited yet simultaneous read access to the temperature sensor data bus for other threads to read the temperature. The temperature sensing function utilises this semaphore and the serial output mutex to log a status output. The serial output mutex prevents simultaneous access to the serial connection, ensuring clean and readable outputs. The heating control function uses the temperature sensor semaphore to read the desired and current temperatures to set the heating on or off. The function utilises a mutex to communicate with the heater to ensure asynchronous data bus usage and finally logs to the serial console with a mutex. The broadcast status function utilises nested mutex usage to output to various peripherals and to log to the console. The thermostat senses the temperature every 10 seconds, controls the heater every 15 seconds, and broadcasts data every 3 seconds.

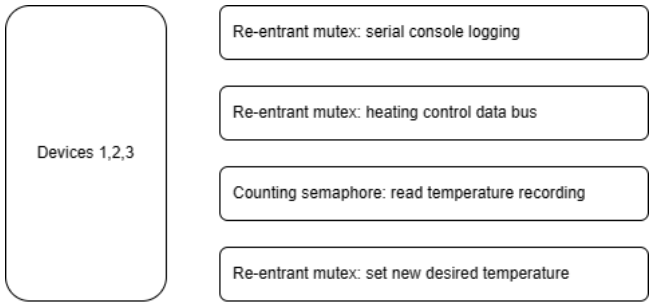

Figure 2 Lower priority tasks.

The thermostat accepts simultaneous remote connections from three specific devices. Each remotely connecting device is different and takes varying CPU time, so the thermostat must handle them with different priorities. Figure 2 shows this in detail. Each task uses a counting semaphore to read the temperature sensor recording, a re-entrant mutex to set the new desired temperature, and finally, another re-entrant mutex to log this event to the serial console. The device one thread waits for 5 seconds before emulating a constant temperature change every 10 seconds with logging. The device two thread does precisely the same, with a 10-second wait before emulation starts and a 15-second wait before changing temperature. The device three thread emulates a poorly handled task with a faulty remote connection to showcase the priority inheritance feature by hogging the mutex.

In conclusion, the enhanced DocetOS version successfully integrates new features and improvements, offering a comprehensive solution for real-time embedded systems. Implementing re-entrant mutexes and counting semaphores ensures effective mutual exclusion, while the fixed-priority task scheduler prioritises critical tasks. The more efficient task sleeping mechanism and the revamped wait and notify system optimise resource utilisation. Additionally, the introduction of priority inheritance for mutexes mitigates priority inversion issues. The demonstration using a digital thermostat interactively showcases all feature implementations of the base DocetOS system.