Assessment Report

System Programming for ARM

# Introduction

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, a fixed-priority task scheduler, a more efficient task sleeping mechanism, a more efficient wait and notify system, priority inheritance for mutexes, and finally, counting semaphores.

# 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 acquisition 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. A zero value signifies that no task holds the mutex, and the task can safely obtain said mutex. A mutex's 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's task field points to the same task as the one acquiring, then this represents a recursive acquisition, and only the counter requires modification with an 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 calling the OS yield function to yield the task requesting an occupied mutex. 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 only the mutex-owning task is requesting this release to prevent the unsafe behaviour of tasks releasing mutex owned by other tasks. Once confirmed that the task holding the mutex is making the request, the function can safely decrement the mutex's 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

To safeguard the mutex TCB pointer field from corruption due to multiple simultaneous modifications, exclusive load, and store CMSIS intrinsics can be utilised. The LDREX intrinsic loads the mutex's 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 OS ‘notify’ function must increment this counter. Before starting the logic, the mutex acquire function must retrieve the check code. When the acquisition logic decides that the requesting task must wait, the function passes the code to the OS wait delegate, which then compares the code with the global code. Matching codes signify that a context switch triggering the notify function 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 and trigger a context switch. Therefore, the mutex acquire logic will iterate to restart the acquisition logic.

Including an OS 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 in case of a successful release, preventing mutex hogging and allowing for other tasks to run appropriately.

# A More Efficient Task Sleeping Mechanism

In the base DocetOS, when a task enters a sleep state, staying within the task list, a status flag is set in the task control block alongside the wake time by the operating system. The round-robin scheduler function will cycle between all tasks in the task list, checking whether it needs waking each time.

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 to implement a list for sleeping tasks due to its simple programmatic logic. However, an 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 .

## 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.

The implementation of a generic variant of the heap structure can improve storage reusability for other parts of the operating system, for example for use with implementing a fixed-priority task scheduler. A generic implementation approach simplifies the organisation of code files. The definition of the heap structure, alongside the insert, extract and empty check 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, the initialisation of heap storage in memory takes place before the heap is initialised, 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 to extract tasks from the heap and insert them into the scheduler task list. Additionally, the sleep delegate can perform task list removal and sleeping heap insertion.

Initially, during prototyping, this implementation involved a sleep header and source file within the project which consisted of a heap structure for TCBs solely and a sleep delegate function. While this approach involved simpler logic, the implementation would have resulted in disorganisation because of 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, as well as 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 take place 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 takes place 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. The implementation of a fixed-priority task scheduler ensures the completion of the highest-priority tasks before scheduling lower-priority tasks.

## Design

The implementation of an additional ‘priority’ field can make it possible to store the priority level within the TCB itself. The ‘OS\_initialiseTCB’ function can accept an additional argument, which extra logic within this function can store in the TCB structure. Although there already exists a ‘data’ field which can implement the storage of priority levels, it stores the wake time when a task is sleeping. Although it is possible to preserve priority levels to protect overwritten ‘data’ fields, such as by implementing array task lists to retain priority levels, it is far easier to create an additional ‘priority’ field. This priority field can take a type of ‘uint\_fast8’ for performance benefits.

Modification to convert the currently implemented task list to an array of multiple task lists can enable the separation of tasks given a priority level. Using a pre-processor directive allows for the hard coding of the number of priority levels, which equals the number of array elements. The ‘OS\_addTask’, ‘\_OS\_taskExit\_delegate’, ‘\_OS\_wait\_delegate’, and ‘OS\_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. To maintain code readability, priority levels should be in 1-index when assigning to tasks, then converted to 0-index within internal code before array access logic, with smaller numeric values denoting higher priorities.

## Safety

The same logic for the ‘add task’ function will be present, albeit with slight modifications to allow functionality with an array of doubly linked lists, therefore, this function will still be thread-unsafe. The logic for removals of tasks from the task list array is situated in SVC delegates, therefore, we can assume that it will be safe from corruption due to concurrent modification.

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 largest possible priority number defined by the pre-processor directive must clamp to the highest priority number. Compiler errors and warnings protect against the instance of a user not passing in a value, or passing in a negative value.

Since there are a variable number of lists in the task list array, the initialisation of the head pointer fields in each list is more 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 which can affect code portability across different compilers. I have chosen to not initialise the fields since as per the C specification, without an explicit initialisation, an object with static storage duration initialises to a null pointer if it has a pointer type[[1]](#footnote-1).

# A More Efficient Wait and Notify System

With the current implementation of re-entrant mutexes, a task that requests for an acquired mutex moves to the wait list. On the release of a mutex, the notification logic notifies all tasks on the wait 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, this will be a massive improvement from the current logic of notifying all waiting tasks.

## Design

One approach for a mutex-specific wait list is a first-in-first-out-based ordering system, where tasks that request for an acquired mutex are queued in order of attempt. Another approach is a priority-based ordering system, where the ordering is 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 tasks 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 potentially solve priority inversion by preventing higher-priority tasks from waiting for a mutex in a queue before a lower-priority task.

As high responsiveness and suppressed priority-inversion pose major benefits to an embedded operating system, the modifications will feature the priority-based approach. 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, 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.

An implementation of a mutex creation function can replace the requirement for a static initialiser. This function, existing in the mutex source code, will have access to the heap comparator function implementation for mutexes, therefore it will simply need to 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, the function cannot return a pointer since the mutex is a local variable. To resolve the issue of stack crowding, the mutex create function will only initialise a mutex given a pointer to it. To ensure this logic works correctly, the heap implementation must allow modifications to the heap store at runtime.

## 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 wait list, guarding the system against tasks imprisonment in the wait 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 instance of a task’s refusal to sleep on the release of a completely unrelated mutex.

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 scheduler’s task list and inserting it into the mutex wait heap. This new notify function no longer needs to be in the scheduler source files, it is better suited alongside mutex logic, to achieve this, the pending list and all push/pop functionality must be accessible from the mutex source code. The OS wait function, despite being a mutex-specific implementation, will stay in the scheduler source files. it is ensuring that the scheduler task list is not accessible from other sources 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 thread safety of both 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 which holds a mutex blocks higher-priority tasks that are 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 to first check 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-holding task’s priority to the level it was previously at before the inheritance to prevent the permanent promotion of lower-priority tasks. To facilitate this implementation, the TCB structure must contain an additional field to preserve the original priority level.

The current mutex acquire function calls the OS wait SVC delegate function in the instance where another task already holds the mutex that a task is requesting. The wait function, after confirming check codes, 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, with the inclusion 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 to the pending list. The context switch invoke will run the scheduler which will then 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 presence of TCB retrieval via the OS\_currentTCB() 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 function will be unused in the OS, a complete generic heap package would be beneficial for the end-user.

The mutex release function resets the mutex task field and calls the mutex-specific notify function in the instance of a complete mutex release from the notification counter being equal to zero. Implementing the task priority restoration functionality is ideal right before the notify function call in the mutex release function. Before notifying the waiting task, 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 which calls the OS yield function which invokes the context switch.

## Safety

The initialisation of the additional priority field must take place at the same time as the initialisation of the TCB’s normal priority field in the TCB initialise function. With this, the additional priority field can properly 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 would’ve been possible to restore task priorities within here and save the complexities of implementing an additional function to achieve this. However, a large 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 easily, unless with the removal of the static type qualifier for these objects. Since it is challenging to ensure thread safety for doubly linked lists, 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

Like a re-entrant mutex, a counting semaphore allows concurrency control to prevent race conditions when accessing shared resources between threads. 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 as soon as a token becomes available. A binary semaphore implementation can utilise this counting semaphore logic by initialising with a single token.

## Design

* Semaphore structure:
  + uint32\_t tokenCounter – holds the total available number of tokens.
  + uint32\_t notificationCounter – holds a count of the number of notifications.
  + \_OS\_tasklist\_t waiting\_list – SL list holding waiting tasks.
* Semaphore initialise function:
  + Takes in a pointer to semaphore structure and initialises all fields.
* Semaphore acquire function:
  + WHILE (1):
    - Cache notificationCounter value
    - LDREX tokenCounter value.
    - IF tokenCounter is non-zero:
      * IF (Decrement tokenCounter and STREX) is zero
        + BREAK (out of WHILE)
    - ELSE IF tokenCounter is zero:
      * Add task to wait list (list\_sortedPush\_sl)
      * Set PendSV bit
* Semaphore release function:
  + DO:
    - LDREX tokenCounter
    - Increment tokenCounter
  + WHILE: STREX fails
  + Call semaphore-based notify
* list\_sortedPush\_sl:
  + This function needs to be present to sort a SL list of waiting tasks with highest priority first.

1. C99 Draft §6.7.8 ¶10 p126 (<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>) [↑](#footnote-ref-1)