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, **[binary/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 trying to acquire 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.

A more efficient sleeping mechanism can be developed by 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. With a generic implementation approach, the organisation of code files is greatly simplified. A heap source and header file should be present in the project, where the heap structure is defined alongside the heap insert, extract, and empty check functions for OS-wide usage. The definition of heap storage size should be present in the scheduler header file, which can then be accessed in the scheduler source file, where the heap comparator function is defined for a sleeping task use case, where data fields need to be compared.

Due to the nature of heaps, the initialisation of the heap storage in memory is defined beforehand, therefore, the number of entities in the heap is pre-defined and cannot be easily changed 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 be awakened, 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 both the scheduler code and sleep code. With a generic approach, all sleep logic is confined to the scheduler.

## Safety

With this implementation, the scheduler task list, as well as the sleeping heap, should be protected 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, as tasks are removed from the scheduler task list. Heap extract logic should only take place in the OS scheduler function as tasks are woken by the move 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. To ensure highest priority tasks are completed first, a fixed-priority task scheduler can be implemented.

## Approach 1

* Currently, task\_list is a doubly-linked of type ‘\_OS\_tasklist\_t’ list which stores entities of type ‘OS\_TCB\_t’.
* Add and remove task (doubly linked list functions) need to be modified to take in the priority of the task to add and remove tasks to and from the correct priority level list.
* Use similar methodology to the task wait logic – use an intermediate ‘pending’ list when adding to the task list, scheduler function will go through pending list and add to task list.