UNIVERSITY of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

ELE00138M
Systems Programming for ARM Assessment

January 20, 2025

1 Executive Summary

This report covers the modifications made to the embedded C DocetOS operating system. These modifications were implemented to increase it's functionality and improve it's efficiency. The core modifications in question cover the following areas; including mutual exclusion via the use of a re-entrant mutex, a fixed-priority scheduler, and a memory pool protected against concurrent modification. In addition to these core modifications, a number of additional features were implemented to either complement the existing features and expand the toolset to the users, or to improve the quality and efficiency of the internal OS. These features were as follows: An efficient task sleeping mechanism, a counting semaphore to allow task synchronisation, and a queue based task communication system via a pointer queue implementation.

1.1 Objectives and Methodologies

Throughout the implementation of each feature, the primary methodologies used involved firstly designing each system and analysing how it would work and integrate with the users requirements, then starting to implement the basic feature and ensure that worked properly before expanding it to improve any robustness or efficiency. Prioritising design analysis and planning before writing any code ensured that informed decisions could be made about the optimal structure of each feature. This approach also facilitated improved integration and interaction between different components, leading to a more efficient and robust system. Additionally, by spotting any situations which may cause deadlocks or race conditions, these issues could be avoided and care taken when implementing the sections.
During the code implementation, every modification or newly added feature was thoroughly tested at various stages to ensure each component functioned correctly before proceeding to the next stage of development.
Finally a number of example tasks were setup to test the final features implemented and to allow for a demonstration of what DocetOS offers.

1.2 Key Findings

The modifications made to DocetOS have proved to be a success and provide the user with an efficient set of tools which ensure better task management, efficient use of system resources and increased system reliability.
DocetOS successfully implements the features described above without any errors. From a stand-alone perspective of each feature, they work as intended to the required specification. However, there are a number of improvements which could be made to add additional features which when combined with the existing features would improve the efficiency and reliability of DocetOS. For example, the wait and notify system current has to notify all the waiting tasks each time, whereas only the specific task that was waiting could be notified to improve efficiency. This will be discussed in more detail later in the report.

Contents

2 Introduction

DocetOS is a lightweight embedded system operating system designed to provide a simplistic but efficient set of tools for users to run varying tasks. The basic framework allows for the creation of task control blocks (TCB) which allow specific user tasks to operate. These TCBs are then added to a simple round robin scheduler which determines the next task to run based on their priority, with higher priority tasks taking precedence over lower priority tasks.
Each individual task provides the user with flexibility to do any standard operations of C, as well as utilise the features of DocetOS to enable task communication as well as pauses in operation via an `OS_sleep` function. For example, a user could make one task to periodically measure a peripheral, such as a sensor, and then store that data. A second task could then retrieve that stored data and use it to provide some functionality, and one such case could be to utilise a distance sensors' measurement, to then turn a motor faster or slower to move a robot away from or closer to an obstacle.
DocetOS allows for these functions to operate in an efficient and robust environment, whilst maintaining a modular design which would allow for many more features to be implemented at a later date. These options will be covered later in the report.

This report is organised to provide a comprehensive overview of the modifications made to DocetOS, starting with each of the implemented features, covering their purpose and specification and including a description of the requirements for successful operation.
Following the feature description, the report explores the design and development processes of each feature. This section outlines the steps taken to design each feature, accompanied by the specifications which aided the development.
A short description of the code demonstration will be included at the end to help understand the system as a whole, which is designed to supplement the demonstration code in the main file.
Finally, a conclusion will summarise DocetOS's effectiveness and robustness, providing a critical analysis addressing any current issues and future improvements that could be made in further code revisions.

3 Modifications to DocetOS

3.1 Mutual Exclusion, Via a Re-entrant Mutex

3.1.1 Intent & Specification

Mutual exclusion is a key component to any operating system, it provides an important mechanism for locking a task to prevent any other interrupts or tasks running while a mutex is acquired. A mutex is often used when a task or function requires access to a resource which if interrupted may cause unwanted errors or data loss. A re-entrant mutex can only be held by one task at a time, but may be requested several times. Upon a task requesting a mutex which has already been acquired, if the task is the same task that already has a mutex, then there are no issues, and it can be re-acquired by the same task. However, if another task that doesn't already have the mutex tries to acquire one, then it will not be able to and will have to wait until the mutex is released by the original task.

3.1.2 Design Choices & Development Process

The design of the mutual exclusion mechanism in DocetOS was centred around the implementation of a re-entrant mutex to ensure that critical sections could be accessed by tasks without causing race conditions or data corruption. As mentioned above being able to re-acquire the re-entrant mutex is appropriate for DocetOS, as it prevents any deadlocks in the system.
The decision to use a re-entrant mutex provides a number of benefits that were considered:

- Support for recursive locking by the same task.
- If the same tasks requests the mutex again, re-entry logic is simplified.
- A re-entrant mutex is simple and lightweight to maintain an efficient system.
- The mutex can only be acquired if the task is the current holder of the mutex, or if it is free, preventing deadlocks.

The flow chart design in Figure 1 provides a helpful representation of each scenario when a task tries to acquire and release a mutex. It's important to note the atomic CMSIS operations used in the mutex, as these are essential to prevent any interruptions during the acquisition of the mutex, otherwise the purpose of the mutex is defeated.

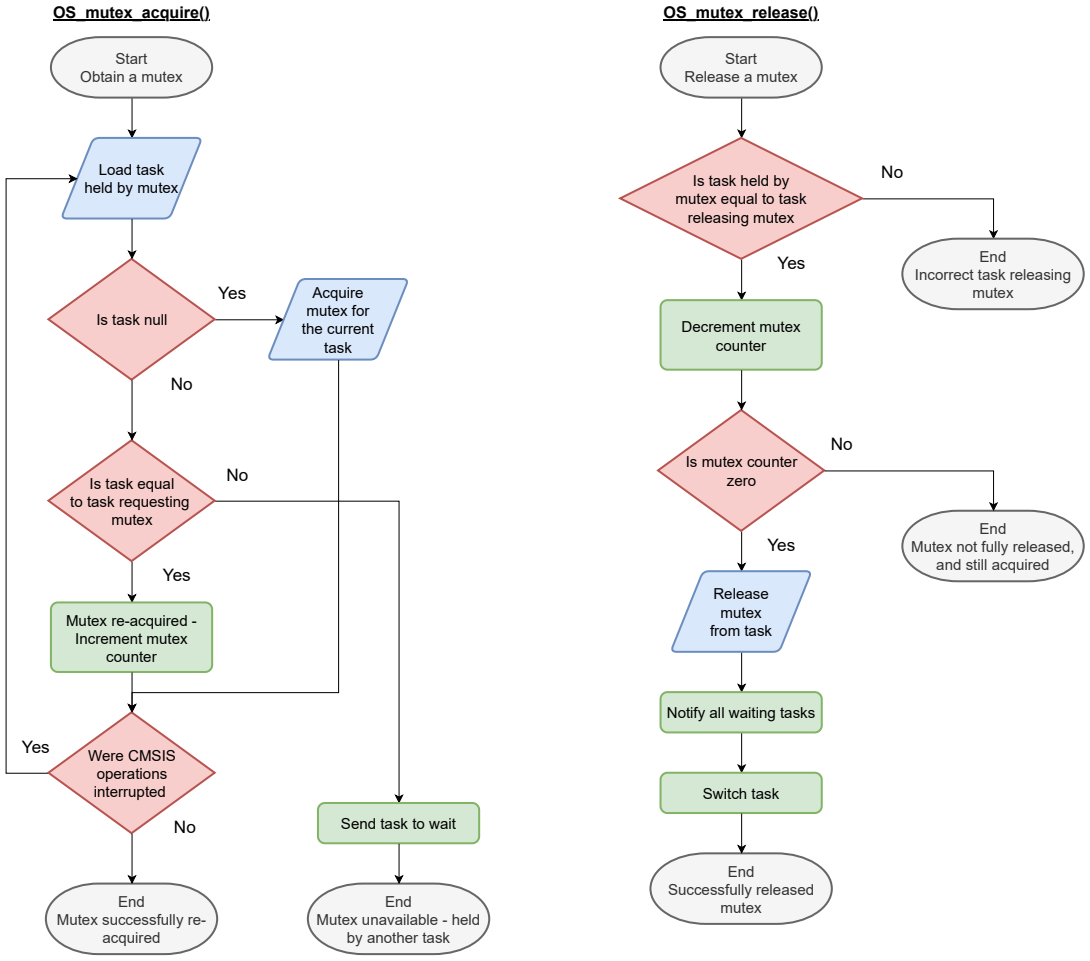

Figure 1: Re-Entrant Mutex Flow Chart

3.2 Fixed-Priority Scheduler

3.2.1 Intent & Specification

The fixed priority scheduler aims to provide functionality which allows certain tasks with higher priority to have more or all of the CPU time. This could be useful for multiple reasons, but particularly in the case of a task that is time sensitive, then setting it to the highest priority would ensure that it will run to completion before other tasks. Without a task priority, all tasks run without any order, and the scheduler just iterates over each task in the list equally.
By utilising fixed priorities in this scheduler the design and implementation is largely simplified compared to a dynamic priority scheduler. Since all task priorities are known at compile time, so the scheduler can be optimised before anything runs.

The specification for this fixed-priority scheduler is as follows:

- The number of priority levels should be configurable through a `#define`
- Priority levels are fixed and should not change at runtime
- The highest-priority task in the list should always be run, and if there are two or more tasks with the same priority, these should operate in a round-robin procedure.

3.2.2 Design Choices & Development Process

The first choice made when designing the fixed-priority scheduler involved deciding what would be counted as the highest priority and how the number of priority levels would be defined. By choosing the highest priority as being "1", it allows easy checking of tasks to see if they are already the highest priority, and flexibility to add as many lower priority tasks as the user desires up to the priority level limit. In contrast, if the highest priority is defined by a higher number, then at a glance it isn't possible to tell if a task is the highest possible priority unless you know the number of priority levels defined.
In order to understand how the OS would handle tasks having varying priorities, visualising the interactions between different program "objects" can be helpful. A sequence diagram helps with this, and the interaction process for the priority scheduler can be seen in Figure 2.
The process of creating this diagram allows the opportunity to think through each interaction and decide upon each specific edge case that could be encountered within the implementation, reducing the chance of missing these later on.

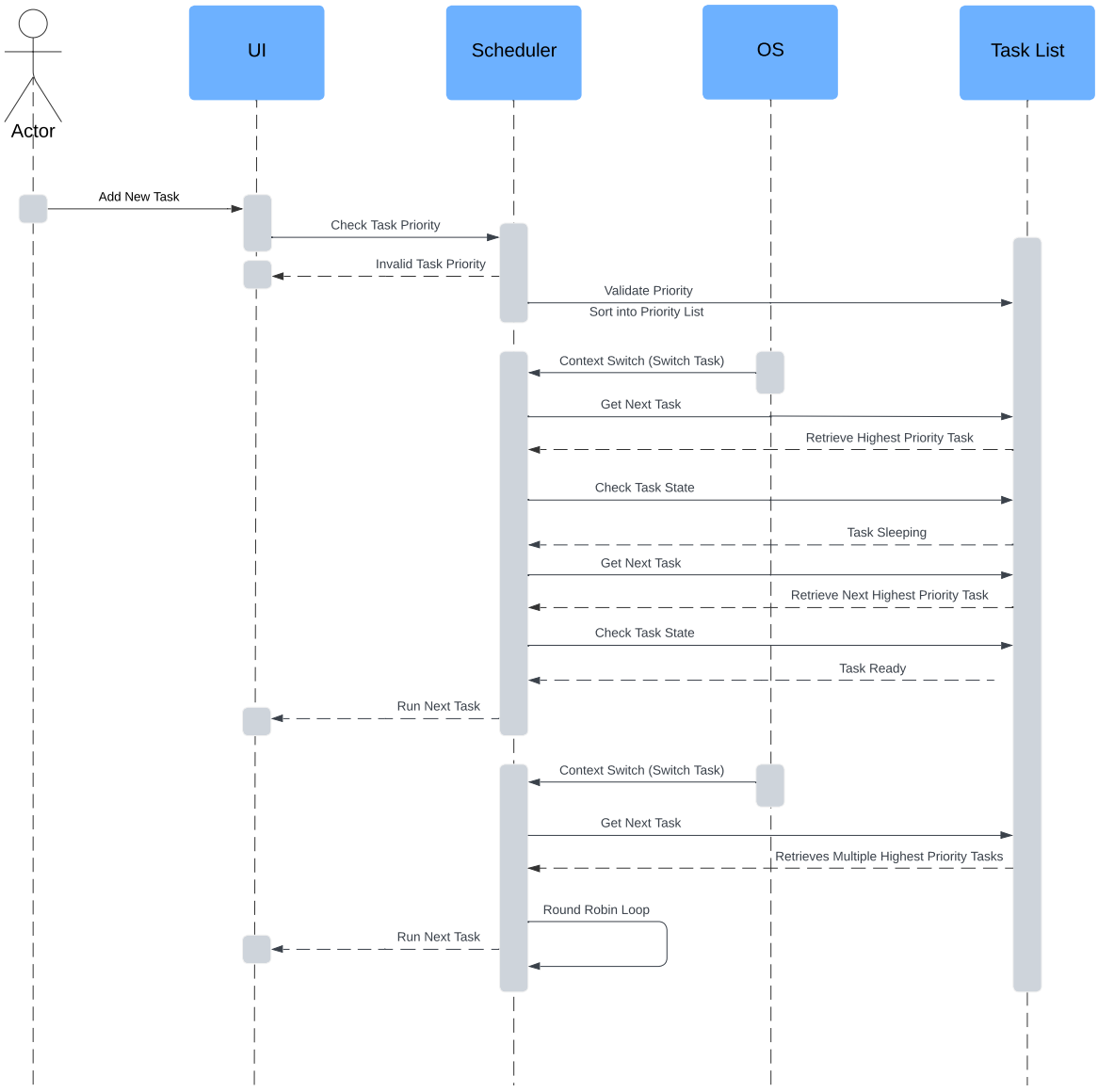

Figure 2: Fixed-Priority Scheduler Sequence Diagram

The design of the active task list utilising a doubly linked list, the simplest method for sorting tasks by priority is to use the insertion sort method. This involves comparing the priority of the task to be inserted with a current task in the list, and iterating over the list until the priority value fits between two others. This is relatively efficient and simple to implement so it is a good choice for this system.

The round robin scheduler function previously operated on tasks with no priority at all. Hence an improvement to this scheduler was required. By considering all the scenarios that can be encountered, it was possible to put together a simple pseudocode design for the fixed-priority round robin scheduler. The 5 use cases predicted are as follows:

- The OS is operating on the highest priority task
- Operating on a task with priority 1 and another task with priority 1 is added (either by waking up or by being notified)
- Operating on a task with priority 3 and a priority 1 task is added
- Operating on a task with priority 3 and a task with priority 1 and 2 are now added
- Operating on a priority 3 task and another priority 3 task is added

Scheduler Pseudocode

if *prev.priority* **higher than** *current.priority* **then**
    Loop to highest priority and return task
else
    if *next.priority* == *current.priority* **then**
        Return next task
    else
        Loop until same priority found and return task
    end if
end if

By following this logic, the scheduler ensures that it is always running the highest priority task within the task list, or if there are multiple tasks with the same priority and that priority is the highest in the list, then those tasks will operate a round robin sharing the CPU time between them equally until they either finish or are instructed to sleep or wait.

3.3 Memory Pool Protected Against Concurrent Modification

3.3.1 Intent & Specification

Memory pools provide an alternative method to standard C `malloc` and `free` functions. The advantage of memory pools is that they are faster and smaller than traditional memory allocation. To use a memory pool, a certain sized "pool" must be allocated from the main memory pool, where the number of blocks and the size of each block is provided pre-determined by the user. This creates a specified number of blocks of memory available to use which all are the same size as each other, allowing for a very efficient implementation. Linked lists are the simplest implementation of the memory pool, allowing for a modular design where only the head of the list is modified each time memory is allocated or deallocated.
The standard memory pool implementation is not protected against concurrent modification (when two tasks try to modify the same memory), so for the specification required in DocetOS, the memory pool must utilise a method to protect it against concurrent modification.
The simplest and most reliable solution is to utilise the already implemented re-entrant mutex. This will provide synchronisation as well as mutual exclusion preventing any errors or data loss from concurrent modification.

3.3.2 Design Choices & Development Process

The design of the memory pool was made significantly easier due to being able to base the design off the memory pool previously experimented with earlier in the module. However, all aspects of the design were still considered to ensure nothing was forgotten and all edge cases accounted for.
The sequence diagram for this system can be see in Figure 3, and shows the full high level overview of allocating and deallocating memory. By designing the memory pool in this manner, it simplified the choice of where to acquire and release the mutex to ensure mutual exclusion, and as can be seen, this was handled internally rather than by the user acquiring a mutex before allocating. The benefit of this allows more flexibility and reduces the chance of forgetting to release the mutex and creating a situation where nothing else will operate.

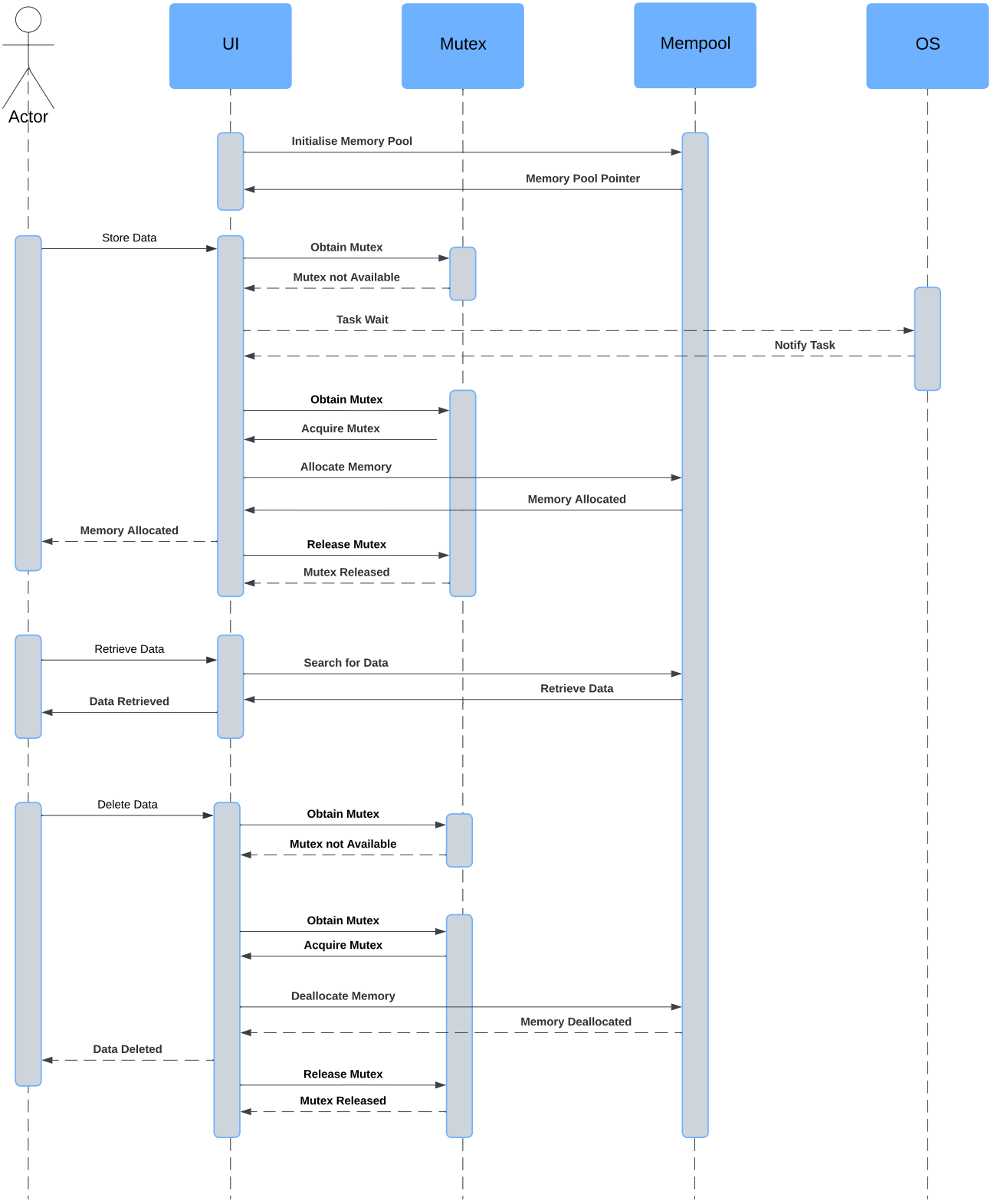

Figure 3: Memory Pool Sequence Diagram

An important aspect when designing the memory pool involved deciding on how the memory pool would be initialised so blocks of memory could easily be allocated and deallocated whenever required. Initially the memory pool had to be initialised in every single task desiring use of it, due to requiring a mutex to be used when initialising the pool. However, this proved to be inefficient as multiple memory pools were often initialised when only small sections of them were required.

The solution to this involved initialising the memory pool in the main function before the operating system started and declaring a global static pool variable for all tasks to share.
By initialising the pool before the OS started, the requirement for the mutex in the initialisation could be removed, since prior to starting the OS, there would not be any context switches causing concurrent modification. This greatly reduced the amount of redundant memory allocated during the lifetime of the OS.

3.4 Efficient Task Sleeping Mechanism

3.4.1 Intent & Specification

An efficient task sleeping mechanism optimises the task scheduling process by ensuring tasks which are asleep do not consume processing resources unnecessarily within the scheduler. By temporarily removing sleeping tasks into their own list, the main task list now solely contains active tasks ready to run next. This mechanism minimises the overhead associated with checking for sleeping tasks every step through the main task list. Additionally if the separate sleeping task list is sorted by wakeup time, with the task expected to wakeup the soonest at the top of the list, all the scheduler is required to do is to "peek" at the top task and check if it should wakeup. If the task is not ready to wakeup, then no further processing is needed, greatly improving the efficiency.
This is the requirement for an efficient task sleeping mechanism for DocetOS, where tasks put to sleep must be sorted into a separate sleep list, and then only re-added into the main task list upon waking up.

3.4.2 Design Choices & Development Process

The initial design for task sleeping involved setting a sleep bit in each task to indicate a task was asleep, and then setting a variable with a calculated wakeup time so that once a certain time had elapsed the task would be woken up. As described above, this becomes inefficient due to the scheduler requiring to check every single tasks sleep bit, and only run it if it isn't asleep.
Designing an improvement to this involved considering a few use cases:

- When a task is put asleep, it should be moved from the main task list to a separate sleep list
- Tasks inserted into the sleep list should be sorted by wakeup time.
- Every single call to OS_schedule(), the scheduler should check the top task in the sleep list, to determine if any tasks should wakeup

The process of creating the sequence diagram design shown in Figure 4, allowed the refining of the implementation of the task sleeping mechanism, and visualised the order of each operation when sorting and retrieving sleeping tasks.
When designing the algorithm to sort tasks being put to sleep into the sleep list. The simplest method was an insertion sort, which provided the most efficient method whilst also being the simplest. Additionally the insertion was wrapped up in atomic operations which ensured that if the sorting was ever interrupted at all, the sorting operation would start again each time.

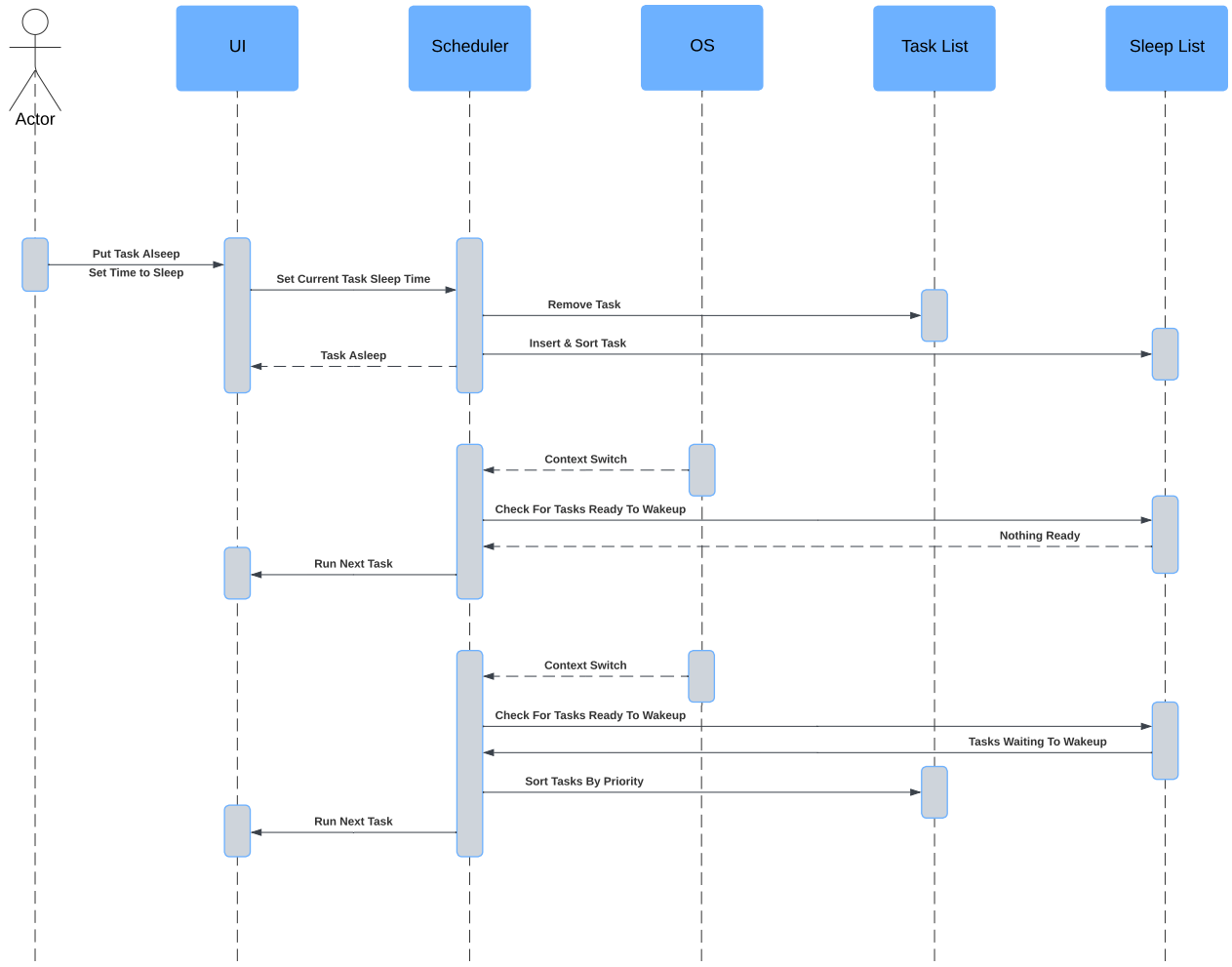

Figure 4: An Efficient Task Sleeping Mechanism Sequence Diagram

3.5 Binary and Counting Semaphores

3.5.1 Intent & Specification

Binary and counting semaphores are two common synchronisation devices. They both work via the same principal where a container holds a number of tokens (for a counting semaphore) or a single token (for a binary semaphore). When a task requires a semaphore such as when accessing memory or operating on a data structure, it obtains a token from the semaphore which reduces the number of available tokens by one. Once the task is finished it returns the token back into the semaphore. Particularly for binary semaphores, this ensures that when a task has a token, nothing else is able to acquire a token and must wait until the token is added back in.
In DocetOS, semaphores can be implemented as an optional functionality, however to fully utilise some of the other features, semaphores are required to ensure synchronisation between tasks.

3.5.2 Design Choices & Development Process

When designing the semaphore, the existing mutex was considered as a template, however a large proportion of the complexity could be removed to simplify the design and implementation. The design outline in Figure 5 helped to spot how the system would flow.
Initially the semaphore would cause an infinite loop when trying to acquire a token if there wasn't one available or the token container was full. This was tested and confirmed, before re-designing part of the system so that an initial number of tokens was passed as a parameter during initialisation. By checking the current tokens against the initial tokens, it prevented the loop infinitely waiting, as if the current tokens was equal to the initial tokens, then the semaphore could call the current task to wait for a token to become available.

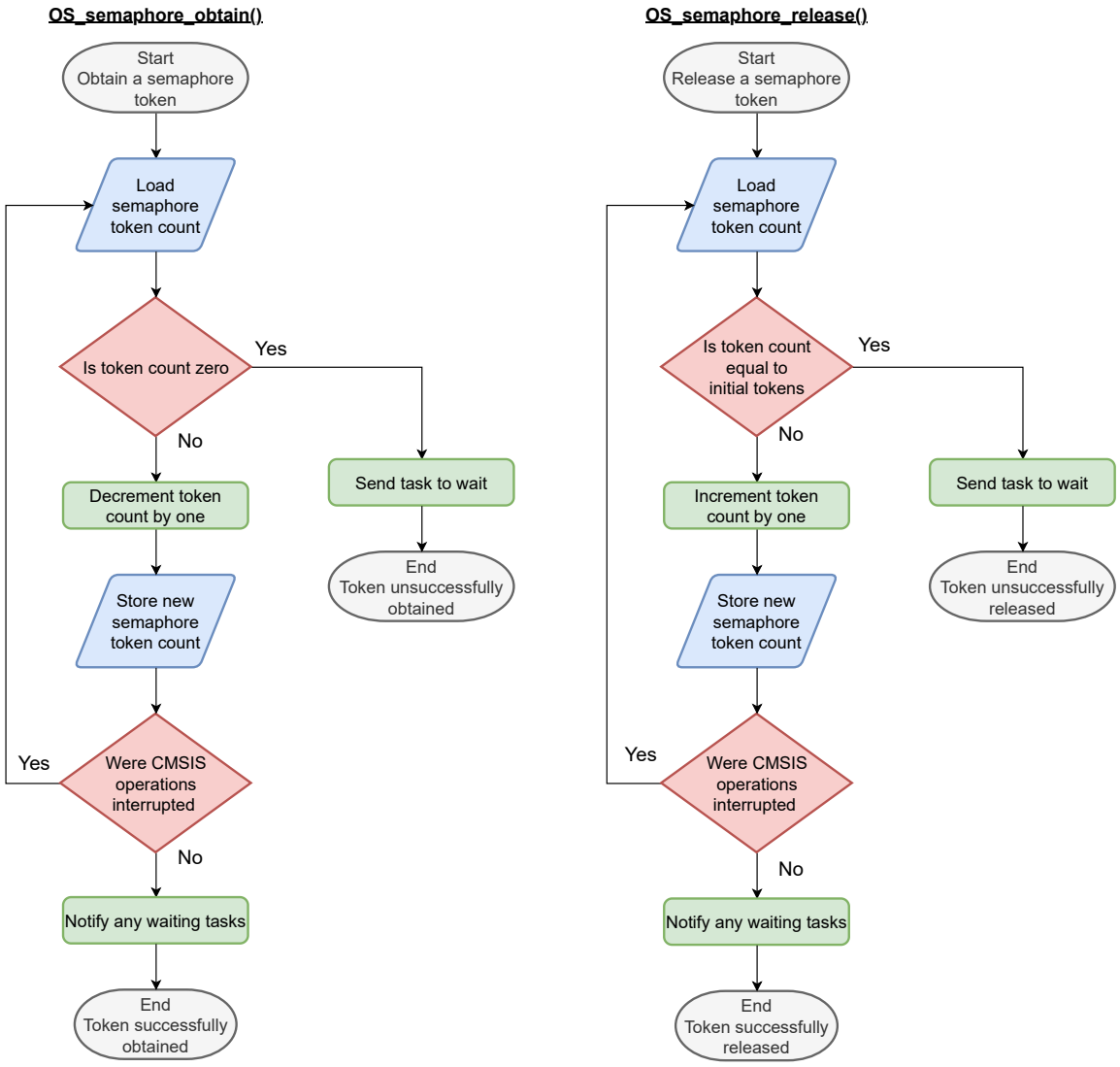

Figure 5: Flow chart presenting the semaphores operation

3.6 A Queue-Based Task Communication System

3.6.1 Intent & Specification

Communication between tasks is challenging to manage efficiently, as there are many situations for shared variables or other data structures where the question as to who owns the variable/data structure arises. Queues are one method to help fix this issue. Various implementations of queues exist to store the data so that it can safely be accessed in multiple tasks without worrying about who owns the data.
Two common methods are copy queues and pointer queues. By default copy queues have no issues with ownership of data, however due to them operating via copying all the desired data from one queue into another, these have effectively double the memory usage. Therefore pointer queues are generally more efficient, as they operate by the "producer" storing a pointer to the data and the "consumer" retrieving the data at that pointer.

3.6.2 Design Choices & Development Process

To maximise efficiency for DocetOS, a pointer queue is more desirable rather than a copy queue. This will allow tasks to communicate with shared data via an efficient implementation and smaller memory requirement than a copy queue. The biggest issue with a basic pointer queue that needs addressing in the design, is that the "producer" does not know when the "consumer" has finished reading from a block of memory. Since a memory pool has already been implemented this is therefore the most efficient method without requiring anything else.

By pairing a pointer queue with a memory pool, the data that is to be shared should be stored in a block allocated from the memory pool. The pointer to this block is then stored in the queue. And the next time the same task requires some memory, it can allocate a new block from he pool. The advantage of this is that when the consumer task receives the pointer to the block, the data can be used as desired and then once finished the block of memory can be deallocated back into the pool thereby freeing up the memory for use later on, and indicating that the task has finished with the data.
When the queue is being operated on, pointer values are being modified and an interrupt at this point could cause errors, so to prevent this a re-entrant mutex is used for mutual exclusion and to prevent concurrent modification.
Additionally, since the queue has a fixed size, there should be a method to detect when it is full and empty. This prevents access of data which doesn't exist, or overflowing the queue and overwriting data. To solve this two counting semaphores can be used, one to track when the queue is full and one to track when it is empty. When in operation this results in the task accessing the queue waiting if the operation attempted is not possible (i.e. the queue is full or empty). The design thought process can be seen in Figure 6 and Figure 7 shows the operation of how the queue is designed and how the semaphores prevent these undesired operations.

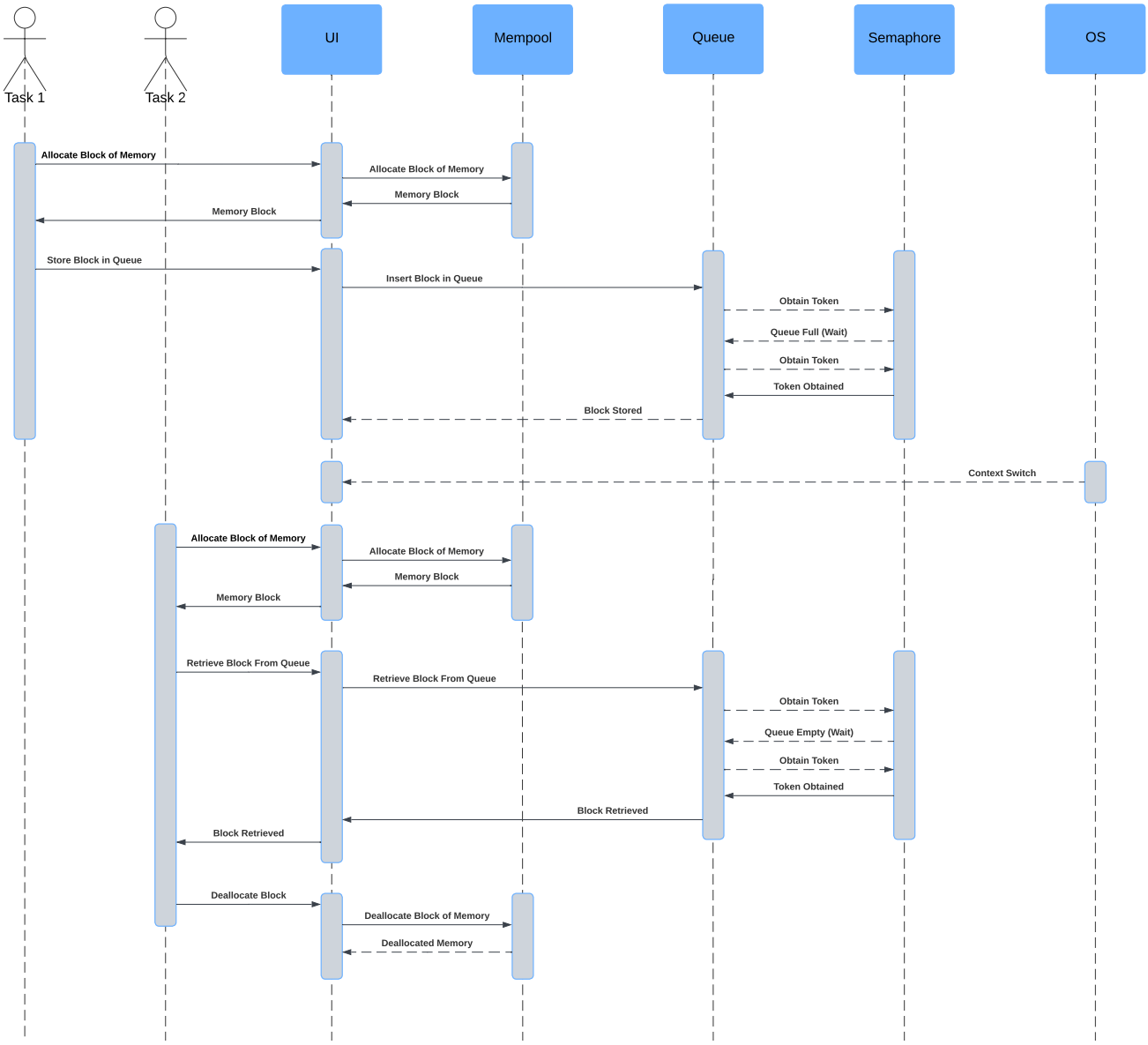

Figure 6: Task communication via a priority queue

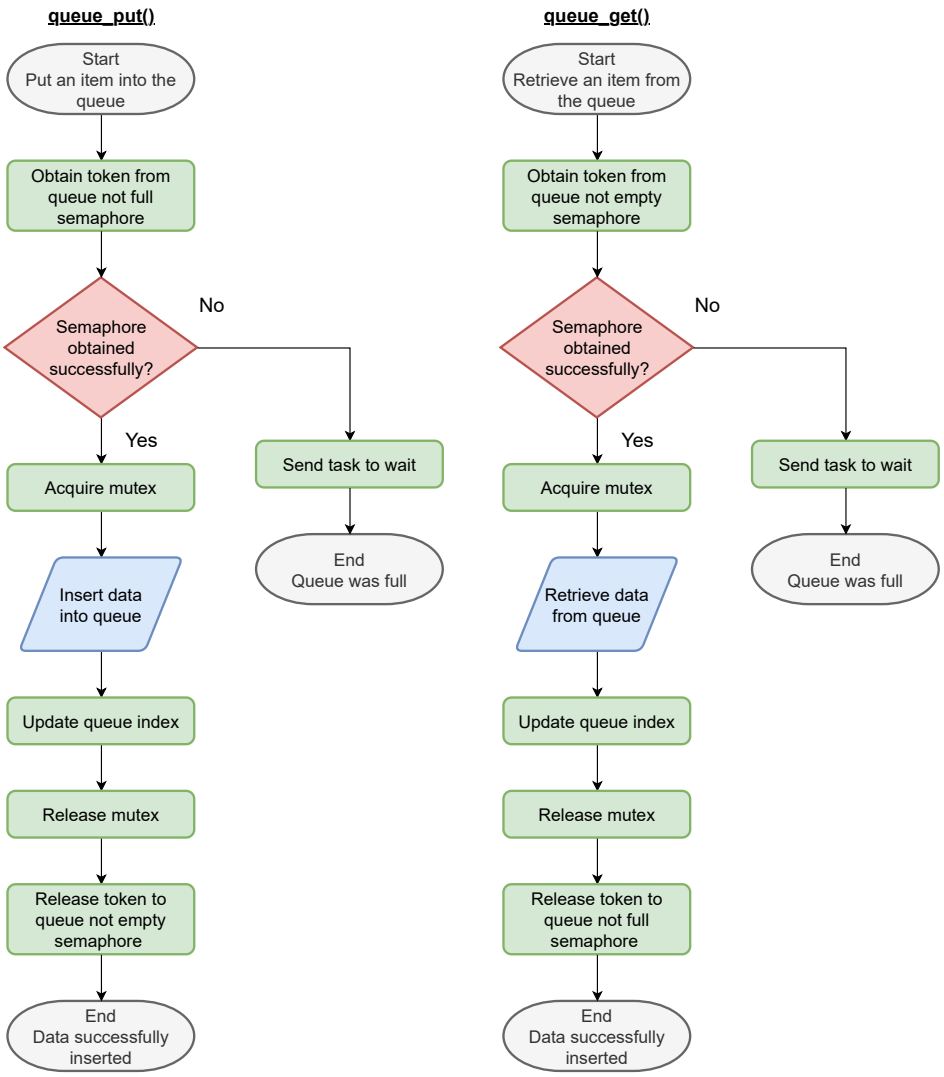

Figure 7: Flow chart presenting the Queue Operations

4 Demonstration

Within the code in the main file, there are a number of tasks which demonstrate the features of DocetOS, this section will cover a short description of the demonstration and how it showcases the features.

Mutual Exclusion Via a Re-entrant Mutex

Throughout each task there are a number of mutex's used to provide mutual exclusion and prevent race conditions, deadlocks or concurrent modification. These sections can be identified by the function calls to `OS_mutex_acquire` and `OS_mutex_release`. For example when printing out text, this is always surrounded by a mutex to prevent a context switch interrupting the data being printed out.

Fixed-Priority Scheduler

During initialisation of tasks in the OS, a selection of priorities are given to the different tasks. Some of the tasks all have the same priority and some have higher or lower priorities. This showcases all conditions where highest priority tasks will always run first, and multiple tasks which have the same priority will share the CPU time and the scheduler will round robin these tasks. By watching the order in which the tasks print out data, it is possible to observe the priority scheduler operating.

Memory Pool Protected Against Concurrent Modification

Some of the data printed out is allocated as memory stored in the memory pool, which showcases how the pool can be setup to store a custom amount of data, allocating and deallocating this memory when required.
Additionally due to the pointer queue utilising a memory pool implementation, there are blocks of memory which are allocated and shared between tasks to display the memory pool usage when communicating between tasks with the queue.

Efficient Task Sleeping Mechanism

Throughout the tasks, there are a couple of places where tasks are sent to sleep for a short period of time. When the data printed out is observed, it can be seen that some of the tasks pause and other tasks run in the meantime, before the sleeping tasks wake up and continue their operation later on.

Binary and Counting Semaphores

Although less obvious from the demonstration code in main, the counting semaphore implemented in DocetOS can be observed in the pointer queue implementation. Two semaphores are used to keep track of when the queue is full and empty, and operates whenever an item is added or removed from the queue.

Queue-Based Task Communication System

Tasks 1, 3, and 4 use the queue in one situation with a producer and consumer situation, whereas task 5 and 6 utilise the queue to communicate between each other and continuously update a variable to increment each loop.
Task 1 allocates memory for some data packets via the memory pool then stores pointers to these on the queue. When task 3 and 4 are run, they retrieve one of the two data packets off the queue and display the data before deallocating the memory pool block.
Task 5 and 6 operate in a round robin fashion due to them both having the same priority, and then each iteration of a loop retrieve the value of a counter, increment it and display it before placing it back on the queue for the other task to retrieve and increment. This operates for 30 loops each task to showcase the queue operating over a longer period of time

5 Conclusions and Summary

The enhancements made to DocetOS have significantly improved its functionality, efficiency and reliability. Key modifications such as the implementation of a re-entrant mutex, fixed-priority scheduler, efficient sleeping tasks, and semaphore have strengthened the system's core operations. The memory pool, and pointer queue for task communication has expanded the set of tools available to the user, allowing for additional functionality and efficiency when performing tasks.
By taking a design first approach to the features implemented into DocetOS, more reliable and robust solutions have been achieved reducing the chances of errors, race conditions and deadlocks during runtime. While the current implementation has met the specified requirements, there is room for further improvements, both to existing features as well as new features that would complement the existing OS.
A number of features rely on parameters being set before starting the operating system, which although reliable and functional, still limits the users flexibility. A future improvement could involve implementing dynamic parameters and operations. One such example could be to implement dynamic priorities for the tasks via priority inheritance. This would allow flexibility to modify tasks priority if they have reached certain milestones or only need to be checked every so often.
An improved wait and notify system could greatly enhance the efficiency and responsiveness of DocetOS by refining how tasks are managed during blocking operations. Currently when a resource such as a mutex becomes available, all waiting tasks are notified, even if only a small subset may be waiting for that one mutex. So, by improving the wait and notify functionality, only the tasks ready to proceed would be notified, leaving the remaining tasks waiting for their turn.
Additionally, DocetOS is currently limited in it's scalability capabilities. As the number of tasks and resources increases, the system will require to operate on large numbers of tasks all at once. An improved wait and notify system ensures that DocetOS can handle a growing number of tasks and resources with

minimal performance impact, making it more robust and responsive.
In summary, the enhancements to DocetOS have created a solid, efficient embedded operating system with improved task management and resource use. Future improvements, like dynamic parameters and an optimised wait and notify system, will further enhance its scalability and responsiveness, ensuring it remains a flexible and reliable tool for embedded systems programming.