## Executive Summary

This report outlines several significant enhancements to the operation of the DocetOS embedded operating system, aimed at expanding the toolset available to users while maintaining robustness and compatibility with low-performance systems. The focus of these modifications is the implementation of fixed-priority scheduling, mutual exclusivity through the integration of a re-entrant mutex, and task management systems such as sleeping and inter-task communications.

TODO: expand below key points

**Key Modifications:**

1. Fixed-Priority Scheduling
2. Task Sleeping
3. Re-entrant Mutex
4. Mutex Priority Inheritance
5. Wait and Notify System
6. Memory Pool
7. Task Communication

**Benefits:**

1. Improved Determinism
2. Enhanced Reliability
3. Optimised Resource Utilisation
4. Compatibility with Low-Performance Systems
5. Memory footprint clarity
6. System Synchronisation

These modifications to DocetOS are a significant step towards meeting the current demands required of real-time embedded systems and provide an opportunity to investigate known techniques used in the professional industry. This report provides a detailed account of the modifications, their rationale, and the anticipated benefits, resulting in a product that can be modified for use in future applications.

## Introduction

### DocetOS

DocetOS is a simple embedded system operating system created with the intention of providing a basic skeleton framework to teach the basics of operating system operation. It consists of the routines to initialise task control blocks, and a context switcher to switch between tasks each system tick. Thus, the system at this point is vulnerable to bugs and limitations that necessitate changes to OS functionality to equip it for real-world implementation.

The current state of DocetOS, allows multiple tasks to run concurrently to achieve their goals which works completely fine for a small number of tasks. But with each new task added to the system, the runtime of all other tasks slow down proportionally, and access to shared resources becomes more complex and vulnerable to race conditions. By implementing functionality that provides more in-depth control of tasks, we can modify their behaviour within the scheduler and provide support for the concurrent execution of many additional tasks with minimal impact on performance and risk of unexpected race conditions, removing current limitations.

### Objectives of modifications

Throughout this report we will increase the functionality of DocetOS by completing the follow objectives:

* Efficient task manipulation capabilities through the implementation of a fixed-priority scheduler with extensive task waiting routines to move tasks to and from custom wait lists.
* Mutual exclusion through the implementation of a re-entrant mutex with task priority inheritance functionality
* Memory management and inter-task communication using a memory pool, utilising memory reserved for OS usage within the embedded system.

### Structure of report

The subsequent sections of this report provide information on each modification, including the rationale of the modification, an exploration and justification of the design considerations taken during the design process, an overview of the modifications final implemented design and functionality, and if required, information related to the safe usage of the implemented design related to mutual exclusivity that needs to be kept in mind if further modification was to take place in the future.

### Modifications to OS

### A diagram of a scheduler heap Description automatically generatedFixed priority Scheduler (Compulsory Task)

A screenshot of a diagram

Description automatically generated

### Design Considerations

#### Task priority

Tasks in the embedded system will need to be assigned priorities based on their importance and timing requirements. Priorities must be assigned on initialisation and static throughout runtime, unless mutex priority inheritance is triggered. Higher-priority tasks must be scheduled to run before lower-priority ones, ensuring timely execution of time-sensitive processes. Where two tasks share the same priority, they need to operate in round-robin. The number of priority levels should be configurable through a system definition.

#### Pre-emption

To handle tasks with varying execution times, the fixed-priority scheduler must support pre-emption. When a higher-priority task becomes ready to run, it pre-empts the currently executing task, pausing the lower-priority task allowing shorter response times to critical higher-priority tasks.

#### Task Synchronisation

To aid task synchronisation within the system such as semaphores and mutexes, the scheduler must support the movement of tasks from the running task list to waiting lists and vice versa. When notifying tasks and returning them to the task list, checks must be in place to verify task priority and ensure pre-emption triggers where necessary.

#### Scheduler Data Structure

In adapting the initial round-robin structure to support fixed-priority scheduling, we replicate the round-robin for each priority level, akin to a bucket queue. Traditionally, a bucket queue includes a pointer to the highest-priority level with a task. However, this approach necessitates a linear search through the priority list when extracting the highest-priority task, to find the next highest-priority task.

To enhance efficiency and eliminate the need for this search, we employ a binary min-heap. The min-heap tracks priority levels in the scheduler that contain tasks. Upon removing the highest-priority task, instead of searching the priority list, we identify the next highest priority level by extracting the root of the min-heap.

#### Fixed Priority Scheduling Algorithm

Before initiating the operating system, the scheduler is preloaded with all tasks to be executed by the user, up to a maximum task limit defined within the scheduler head file. Tasks are initialised with a fixed priority that remains constant throughout runtime, except when influenced by mutex priority inheritance. The initial priority is constrained within the range defined by the number of priority levels specified in our scheduler header file.

Task priority is represented by an unsigned integer, with zero indicating the highest priority. Priorities decrease as the integer value increases, up to the defined limit of priority levels. This design allows assigning a priority of zero to system-critical tasks, ensuring they always hold the highest priority. Increasing the number of priority levels does not then require adjusting the priority of these critical tasks since, at priority zero, they consistently remain at the highest priority, regardless of the total number of priority levels.

#### Context Switching

In the context switch process, several safety checks facilitate the smooth operation and transition between tasks. Firstly, any priority levels devoid of tasks are pruned from the scheduler heap to prevent repetitive polling of the priority level to verify it is empty. Subsequently, the highest priority level containing tasks is retrieved from the heap using a peek operation.

To conclude the context switch, the head of the highest priority level is incremented, introducing round-robin behaviour within the priority level if multiple tasks exist at the level, and the new head is returned for execution.

#### Task Management

Various operations are employed to move tasks within the scheduler, including:

* **Task Waiting:** Removing a task from the scheduler and placing it in a waiting list.
* **Task Notification:** Removing a task from the waiting list and returning it to the scheduler, interrupting the currently executing task if of higher priority.
* **Task Priority Modification:** Changing a tasks priority by extracting it from the scheduler, updating the priority, and returning the task to the scheduler at the new priority level.

#### Mutual Exclusivity

To safeguard against deadlocks and race conditions, alterations to the scheduler heap are restricted to the context switch interrupt. Similarly, modifications to individual tasks within priority levels, as detailed in the task management section above, are reserved for SVC interrupts. An exception is made for sleeping tasks, which are polled and returned to the scheduler during the context switch. This ensures the regular comparison of the current system tick counter against the wake time of sleepings tasks.

### Sleeping task module (Compulsory Task)

### Purpose

At times, there is a need to temporarily pause a task, such as when polling a memory section one per second. Although measuring an exact second internally is challenging without an external timer, a viable alternative is to make a task wait for a designated number of system ticks, referred to as ‘sleeping’ in this implementation.

### Sleeping Task Representation

Sleeping tasks are organised in a sleep list using the same binary min-heap structure as the scheduler. Unlike sorting by priority, tasks in this list are sorted based on their wake-up time. This arrangement ensures that the task with the earliest wake-up time is always at the root of the heap, streamlining the scheduler’s task of checking whether any tasks need to be awakened.

### Sleeping Mechanism

Before being placed in the sleep list, a task’s wake-up time is calculated according to the specified sleep duration and is then attached to the task structure. Subsequently, the task is removed from the scheduler and transferred to the sleep list through the ‘wait task’ SVC interrupt.

### Wake-up Mechanism

During each context switch, the scheduler examines the root of the sleep list to determine if the system tick counter has reached the specified wake-up time for any tasks. When it is time to wake a task, the scheduler removes the task from the sleep list and reintegrates it into the scheduler, making it ready to resume task execution.

### Re-entrant Mutex (Compulsory Task)

### Purpose

A re-entrant mutex ensures exclusive access to a shared resource in a multi-tasking environment. This mechanism permits only one task, the owner, to access the resource at a time, preventing interference from other tasks. The re-entrancy feature allows the owner task to re-enter the critical section, even if it already holds the mutex, facilitating nested locking without risking deadlock. This is particularly valuable in situations where a task invokes a function requiring the same mutex.

### Obtaining Mutex

Before utilizing a resource prone to race conditions, a task can acquire a mutex for that resource. Upon attempting to obtain the mutex, the task checks its lock status. If unclaimed, the task becomes the owner, with an internal counter set to 1. For a task already holding the mutex, the counter increments. If the mutex is owned by another task, the current task is temporarily removed from the scheduler and sent to a wait list, awaiting mutex availability. Once obtained, the task can use the resource exclusively, safeguarding against unintended race conditions.

### Releasing Mutex

Upon completing the critical section requiring a mutex, the task releases the mutex, decrementing the internal counter. If the counter reaches zero after multiple releases, signalling completion of mutex use, ownership is relinquished. This prompts notification of the wait list, enabling the highest priority waiting task to rejoin the scheduler. This task can then attempt to acquire the mutex and utilize the resource it guards once again.

### Mutex Wait List Operation

The Mutex Wait List employs a binary min-heap, facilitating notification and removal of only the root task when the mutex becomes available. This heap is organized based on both task priority and the time a task joined the wait list. In the event of two tasks sharing the same priority, the task added earlier takes precedence for notification and release from the wait list. This approach ensures fairness by favouring the task that has been waiting longer when priorities are equal.

### Priority inheritance for mutexes (Optional Task)

### Purpose

In scenarios where a high-priority task seeks access to a mutex held by a lower-priority task, the higher-priority task may experience prolonged wait times on the mutex wait-list. This delay, caused by the lower-priority task's ongoing execution, poses a risk to the efficient operation of the system.

To mitigate this, the mutex identifies when a higher-priority task is queued in the wait list. In response, the mutex dynamically elevates the priority of the current owner to match that of the waiting task. This adjustment allows the lower-priority task to release the mutex promptly, treating it with the same urgency as the waiting task. Consequently, this mechanism prevents the blocking behaviour of mutexes in a priority scheduler and ensures timely execution of high-priority tasks.

### Implementation

In our scheduler implementation, we incorporate an SVC interrupt for task priority adjustment. By regularly assessing the priority of the mutex owner in relation to the root task in the wait list during each insert operation, as the root task holds the highest priority among waiting tasks, this process ensures that the owner task receives an elevated priority when necessary. Upon releasing the mutex, the priority of the owner task is reset to its original value established at the beginning of runtime.

### Wait and notify system (Optional Task)

### Purpose

Currently, running the notify SVC interrupt will wake all currently waiting tasks, no matter what they are waiting for. Many of the tasks will then immediately re-enter the wait list after discovering they still need to wait for a resource to become available. By separating the main wait list into a separate queue for each blocking item (mutex, semaphore, etc.) we can greatly improve the efficiency of task notification.

### Implementation

Our Fixed-Priority Scheduler design accommodates the wait and notification of tasks from specific SVC interrupts, allowing any blocking item to declare its own wait list min-heap, and then notify only the task at the head of the wait list.

### Design Justification

A min-heap structure is used for waiting lists as it allows us to choose our own comparator functions for each wait list and arrange tasks according to a chosen priority. Mutexes are then able to arrange the min-heap so the task with the highest priority, that has been waiting the longest is at the root of the min-heap while in other blocking cases such as sleeping, tasks can instead be arranged in the queue by wake-up time, ignoring task priority or wait time.

### Memory pool (Optional Task)

### Purpose

**TODO**

### Task communication (Optional Task)

### Purpose

**TODO**

### Generic Binary Heap library (Generic Library)

### Purpose

Throughout the modifications to DocetOS to implement fixed-priority scheduling and designing efficient wait list and notification systems, in many areas it is necessary to utilise a sorted queue. By designing and supplying a generic binary heap library, it becomes simple to make use of an efficient and optimised data structure that operates similarly to a sorted queue, while also supplying an interface for operating system users to also utilise.

### Usage

To use the heap library, on initialisation the structure must be supplied with a storage space and a comparison function used to compare and sort heap items, ensuring the root is always the desired minimum or maximum product. The library then provides optimised behaviour for inserting and removing items from the heap, and additional functions for peaking the heap state.

## Demonstration

Use mutex for printing.

Every second, create eight random 0-1 values and add to memory pool with high priority. Then read with another task and print.

## Conclusion