SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

ELE00138M
Systems Programming for ARM Assessment

Y3884541

January 16, 2024

1 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.

Key Modifications:

1. Fixed-Priority Scheduling
    - Introduction of a fixed-priority scheduling algorithm to prioritise tasks based on predetermined priorities
    - Enhances control over task execution
2. Task Sleeping
    - Integration of a sleep orientated wait list to pause the execution of tasks for set periods
    - Expands functionality of tasks
3. Re-entrant Mutex
    - Introduction of a re-entrant mutex to facilitate exclusive access to a shared resource
    - Ensures consistency in data access and prevents race conditions
4. Mutex Priority Inheritance
    - Integration of priority inheritance of tasks owning a mutex with a higher priority task in the mutex waiting list
    - Prevents priority inversion issues and streamlines the execution of critical tasks
5. Wait and Notify System
    - Overhaul of the system managing waiting and notification of tasks by splitting the centralised waiting list into individual sorted waiting lists per blocking item.
    - Enhances speed of task verification and notification
6. Memory Pool
    - Introduction of a memory pool allowing the reservation of memory for use during system run time
    - Consolidates memory management and gives the ability to define maximum system OS memory allocation at runtime
7. Task Communication
    - Integration of a queue-based task communication system, utilising a memory pool to allow the transfer of data across tasks
    - Introduces cooperation between tasks towards a common goal

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.

Contents

2 Introduction

2.1 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 rotate which task is being executed between each system tick. The tasks rotate execution in a round-robin style. Due to the simplicity of the current system, at this point the system 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 individual 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 slows 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 and safeguarding against unexpected behaviour, we can modify their operation 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.

2.2 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 purpose-built waiting 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.

2.3 Structure of Report

The subsequent sections of this report provide information on each modification, including the rationale and specification 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 in terms of mutual exclusivity that needs to be kept in mind if further modification was to take place in the future.

3 Fixed-Priority Scheduler

3.1 Purpose

Currently, there are no systems in place within DocetOS to control the order of task execution. All tasks are treated with equal priority in relation to task execution and resource allocation. While this is acceptable for basic system usage, in real-time operation systems (RTOS), it is important for a task to be capable of meeting strict task deadlines when necessary.
By implementing Fixed-Priority Scheduling in DocetOS, we allow prioritising execution of critical, time sensitive tasks above regular tasks and the preferential assignment of resources to tasks,

without the complex overheads that come with implementing Dynamic-Priority Scheduling. Additionally, Priority Scheduling increases the operating systems responsiveness to external inputs such as button presses or timer triggers that require immediate response from the system.

3.2 Specification and Design Considerations

3.2.1 Task Priority

Tasks in the embedded system need to be assigned priorities based on their importance and timing requirements. Priorities must be assigned on initialisation and remain 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 must be configurable through a system definition.

3.2.2 Pre-emption

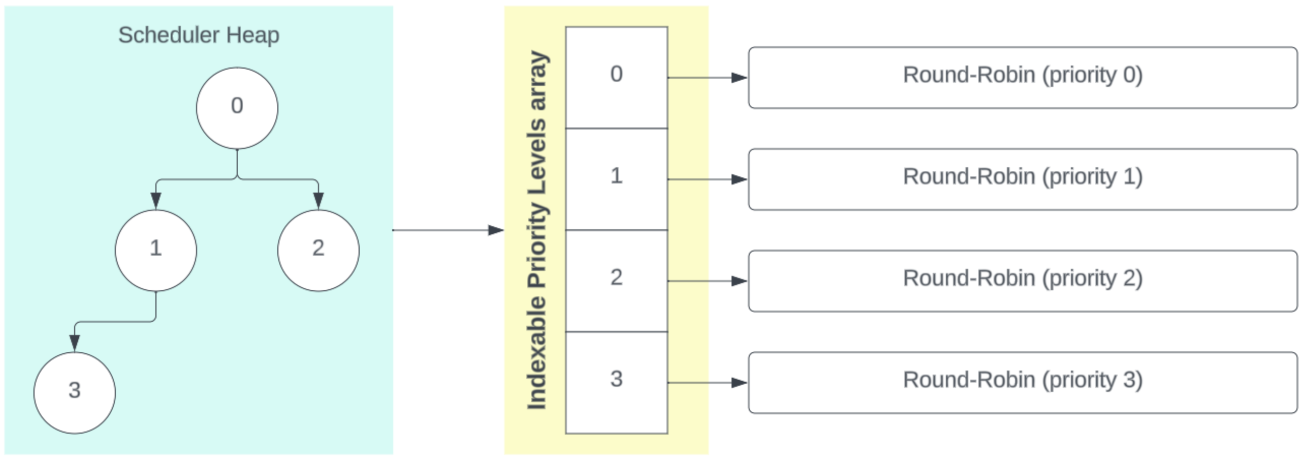
To handle tasks with varying execution times, the fixed-priority scheduler must support pre-emption. When a higher-priority task becomes ready to run, it will pre-empt the currently executing task, interrupting the lower-priority task allowing shorter response times to critical higher-priority tasks.

3.2.3 Task Synchronisation

To aid task synchronisation mechanisms 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.

3.3 Implemented Design and Functionality

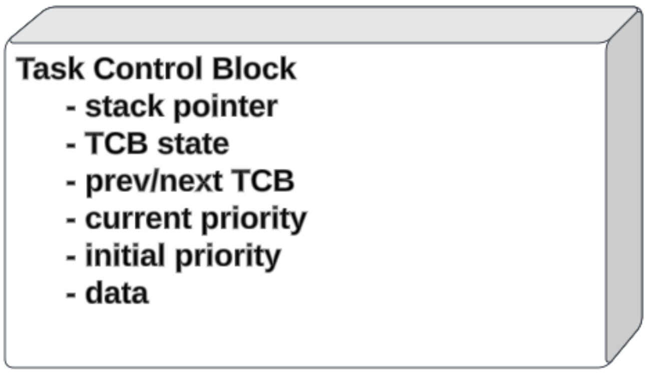
3.3.1 Scheduler Data Structure



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. This allows indexing priority levels with a time complexity of O(1) instead of linearly searching for specific priorities an a regular ordered 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 levels array when extracting the highest-priority task, to find the next highest-priority level containing a 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.

3.3.2 Scheduler Initialisation



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 added to TCBs which are initialised with a fixed priority that remains constant throughout runtime, with the exception of when influenced by mutex priority inheritance. The priority of the TCB is constrained within the range defined by the number of priority levels specified by the OS.
TCB priority is represented by an unsigned integer, with zero indicating the highest priority. Priority decreases 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.

3.3.3 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.

3.3.4 Task Control Block (TCB) Management

The below operations are implemented to move TCBs within the scheduler:

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.

The binary heap operations and the scheduler add/remove functions do not have mutual exclusivity systems in place. Due to the size of the operations, using LDREX and STREX to safeguard these operations is unsuitable however a mutex cannot be used as mutex's themselves use these operations, creating circular dependency. Therefore, these operations are implemented as SVC handler functions. The SVC interrupt halts thread mode and shifts the processor to handler mode to conduct the TCB movement, preventing other tasks or interrupts from clashing with the TCB management functions. When the movement completes, the processor is returned to thread mode to resume system execution.

4 Conclusion