SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

ELE00138M
Systems Programming for ARM Assessment

January 23, 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 the ability to pause the execution of tasks through sleeping.

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

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

1 Executive Summary **2**

2 Introduction **4**
2.1 DocetOS . . . 4
2.2 Objectives of Modifications . . . 4
2.3 Structure of Report . . . 4

3 Fixed-Priority Scheduler **4**
3.1 Purpose . . . 4
3.2 Specification and Design Considerations . . . 5
3.2.1 Task Priority . . . 5
3.2.2 Pre-emption . . . 5
3.2.3 Task Synchronisation . . . 5
3.3 Implemented Design and Functionality . . . 5
3.3.1 Scheduler Data Structure . . . 5
3.3.2 Scheduler Initialisation . . . 6
3.3.3 Context Switching . . . 6
3.3.4 Task Control Block (TCB) Management . . . 7

4 Wait and Notify System **7**
4.1 Purpose . . . 7
4.2 Specification and Design Considerations . . . 7
4.3 Implemented Design and Functionality . . . 8
4.3.1 Waiting List Data Structure . . . 8
4.3.2 Task Waiting and Notification . . . 8

5 Sleeping Tasks **8**
5.1 Purpose . . . 8
5.2 Specification and Design Considerations . . . 9
5.2.1 Operation . . . 9
5.2.2 System Efficiency . . . 9
5.2.3 Potential causes of error . . . 9
5.3 Implemented Design and Functionality . . . 9
5.3.1 Sleeping Tasks . . . 9
5.3.2 Waking Tasks . . . 9

6 Re-entrant Mutex **9**
6.1 Purpose . . . 9
6.2 Specification and Design Considerations . . . 10
6.3 Implemented Design and Functionality . . . 10
6.3.1 Obtaining Mutex . . . 10
6.3.2 Releasing Mutex . . . 10
6.3.3 Mutex Waiting List Operation . . . 10

7 Priority Inheritance for Mutex's **10**
7.1 Purpose . . . 10
7.2 Specification and Design Considerations . . . 11
7.3 Implemented Design and Functionality . . . 11

8 Memory Pool Module **11**
8.1 Purpose . . . 11
8.2 Specification and Design Considerations . . . 11
8.2.1 Operation . . . 11
8.2.2 Mutual Exclusion . . . 11
8.3 Implemented Design and Functionality . . . 12

9 Demonstration **12**

10 Conclusion **14**

11 References **14**

2 Introduction

2.1 DocetOS

DocetOS is a simple embedded 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 protection through the use of 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 purpose 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

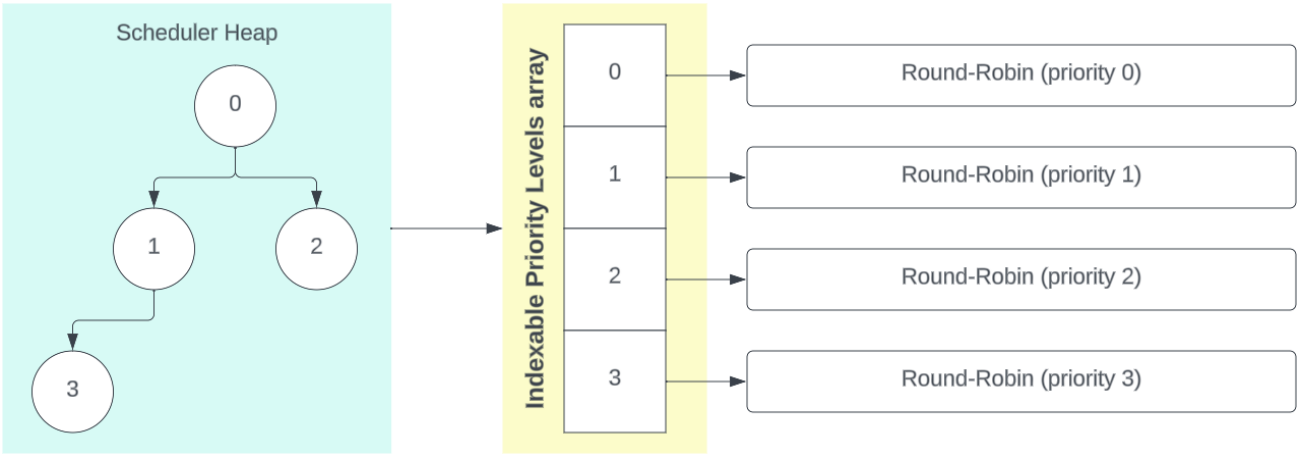
To handle tasks with varying execution times, the fixed-priority scheduler must support pre-emption. When a higher-priority task becomes ready to run, it will pre-empt the currently executing task, interrupting the lower-priority task allowing shorter response times to critical higher-priority tasks.

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

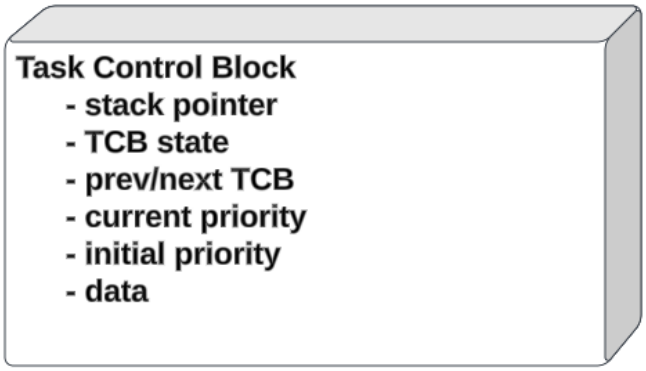
3.3.1 Scheduler Data Structure



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 in 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 header file. Tasks are added to Task Control Blocks (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 complexity of the operations, using the instructions provided by the Cortex-M for mutual exclusion, 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 execution 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 Wait and Notify System

4.1 Purpose

In the current state of DocetOS there is one universal waiting list which, when the notify SVC interrupt is triggered, will wake all currently waiting task regardless of whether the item blocking the task is free. Most tasks from the waiting list will then immediately re-enter the waiting list after discovering they still need to wait for a resource to become available. By separating the main waiting list into a separate waiting list for each blocking item (mutex, semaphore, etc.) we can greatly improve the efficiency of task notification.

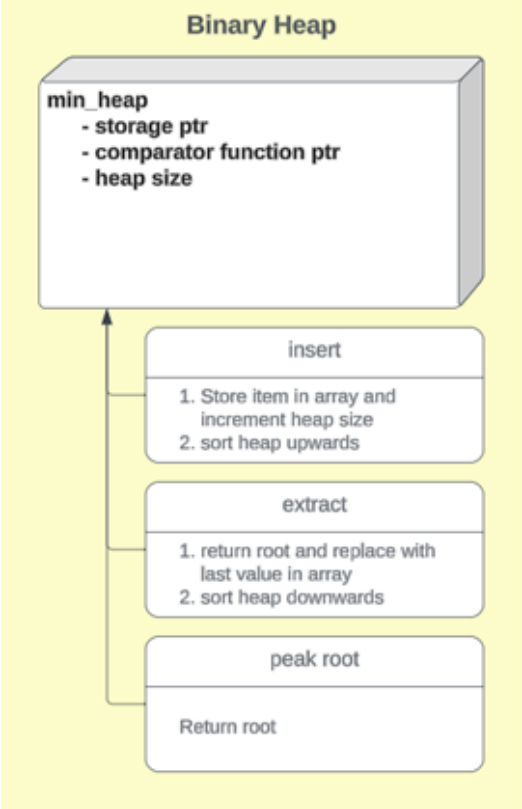
4.2 Specification and Design Considerations

Tasks must be moved from the scheduler to a specified waiting list, and the blocking item containing the waiting list needs to be capable of notifying the task at the front of the waiting list, returning the task to the scheduler, pre-empting the currently running task if the notified task has a higher priority. To retain mutual exclusivity of the scheduler, task movement to/from waiting lists must be done using SVC interrupts.

4.3 Implemented Design and Functionality

4.3.1 Waiting List Data Structure

To implement the improved waiting system, a binary heap module is provided that contains the functionality required to initialise and manage an ordered heap which can be individualised to specific use cases by providing the heap with a comparison function. Using a heap, gives the ability to notify the single item that needs to be notified next instead of the entire waiting list, improving notification efficiency.
For a blocking item to integrate the waiting list feature, it must declare a waiting list and provide a comparator function, choosing to order the waiting list by a range of task properties including time in the queue, priority, and wake-time.
The binary heap data structure is used in various applications throughout the operating system, including within the scheduler. So, by consolidating binary heap logic into a generic heap module, the usage of heaps is simplified system wide.



4.3.2 Task Waiting and Notification

To give blocking items the ability to wait/notify tasks, SVC interrupt handlers are added to the Fixed-Priority Scheduler that facilitate the movement of tasks between waiting lists and the scheduler. By utilising an SVC interrupt, we maintain the mutual exclusivity of the scheduler while providing a function that can be easily called anywhere within the OS.

5 Sleeping Tasks

5.1 Purpose

In scenarios when tasks are not required to run constantly, suspending the tasks execution for a set period can aid in CPU load, power efficiency and freeing up allocated resources. A good example of this is when required to poll a memory location once per a second. In the current state of DocetOS the task would constantly poll the memory location with near zero delay depending on processor load. By implementing task sleeping we give tasks the ability to suspend their own execution and free up processor time for other, possibly lower priority tasks.

5.2 Specification and Design Considerations

5.2.1 Operation

When sleeping a task, the user passes the number of system-ticks the task should sleep for into the sleep function. The task should then be removed from the scheduler for the defined number of system ticks, returning the task to the scheduler after the set period.

5.2.2 System Efficiency

When the system verifies the waiting list of sleeping tasks, it would be advantageous to retain an ordered queue of tasks allowing the operating system to only check if the task that will wake up soonest is ready to be returned to the scheduler, reducing verification time considerably.

5.2.3 Potential causes of error

During sleep function design, care must be taken to handle integer overflow of the system tick counter and calculated wake-time of tasks. Additionally, the movement of tasks to/from the scheduler must be carefully managed to retain mutual exclusivity of the scheduler.

5.3 Implemented Design and Functionality

5.3.1 Sleeping Tasks

To implement Task Sleeping into DocetOS, we utilise the waiting list functionality provided by the binary heap module introduced in the improved waiting/notification modification. During task insertion into the sleep waiting list through the related scheduler SVC handler, we calculate and store the task wake-time in the TCB. By providing the generic heap with a comparison function that sorts the heap by TCB wake-time, the TCB to be woken the soonest will be sorted to the heap root, so we can poll only the heap root to verify whether a task is ready to be woken and returned to the scheduler.

5.3.2 Waking Tasks

Each system tick, the OS polls the sleeping list to verify whether a task is ready to be woken. To maintain mutual exclusivity and simplify behaviour, this is done within the context switch. Before the context switch fetches the next task for execution, the sleeping list is verified and all tasks ready to be woken are returned to the scheduler. This ensures the timely execution of high priority woken tasks.
To handle system tick counter overflow, when comparing task wake time to the system tick counter, both values are cast to signed integers, avoiding unexpected behaviour.

6 Re-entrant Mutex

6.1 Purpose

When multiple tasks are attempting to access a resource, they are susceptible to race conditions causing unexpected behaviour within the accessed resource. A re-entrant mutex ensures exclusive access to the shared resource in a multi-tasking environment. Safeguarding the resource against race conditions.

6.2 Specification and Design Considerations

This mechanism will permit only one task, the owner, to access the resource at a time, preventing interference from other tasks. Being re-entrant will allow the owner task to re-enter the critical section, even if it already holds the mutex, to facilitate nested locking without risking deadlock. This will be particularly valuable in situations where a task invokes a function requiring the same mutex.
The comparator function created for the mutex waiting list must sort tasks by both priority and time in the waiting list, ensuring the highest priority tasks receives earliest access to the resource and preventing task deadlock.

6.3 Implemented Design and Functionality

6.3.1 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.

6.3.2 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. The released task can then attempt to acquire the mutex and utilize the resource it guards once again.

6.3.3 Mutex Waiting List Operation

The Mutex Wait List employs the use of our binary heap module, facilitating notification and removal of only the root task when the mutex becomes available. The 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.

7 Priority Inheritance for Mutex's

7.1 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 in the mutex waiting list. This delay, caused by the lower-priority task's ongoing execution, poses a risk to the efficient operation of the system. Hence implementing priority inheritance of the mutex owner facilitates the freeing of resources in a timely manner for higher priority tasks to utilise.

7.2 Specification and Design Considerations

The mutex will identify 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.

7.3 Implemented Design and Functionality

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 waiting 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.

8 Memory Pool Module

8.1 Purpose

In an embedded RTOS, it is bad practice to dynamically allocate memory as considering the limited memory often available, it is required to know the operating system memory allocation at run time. This limits the capabilities of the operating system. To overcome this, a memory pool can be used to reserve memory at runtime for use by the operating system. This allows deterministic memory allocation, reduced fragmentation of the system and enhanced memory management and protection.

8.2 Specification and Design Considerations

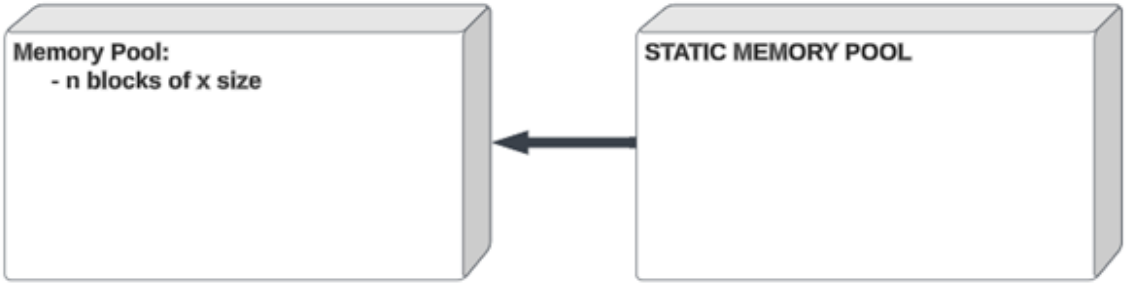
8.2.1 Operation

A static memory pool will statically allocate a predefined size of memory at run time for use by the system. Users can then initialise memory pools with a number of memory blocks of a requested size. Care must be taken to ensure memory allocated from the static memory pool is aligned appropriately to prevent memory fragmentation.

8.2.2 Mutual Exclusion

In our operating system, it is possible for multiple tasks to attempt to allocate memory from the memory pool simultaneously. To protect against concurrent modification, mutual exclusion techniques must be utilised.

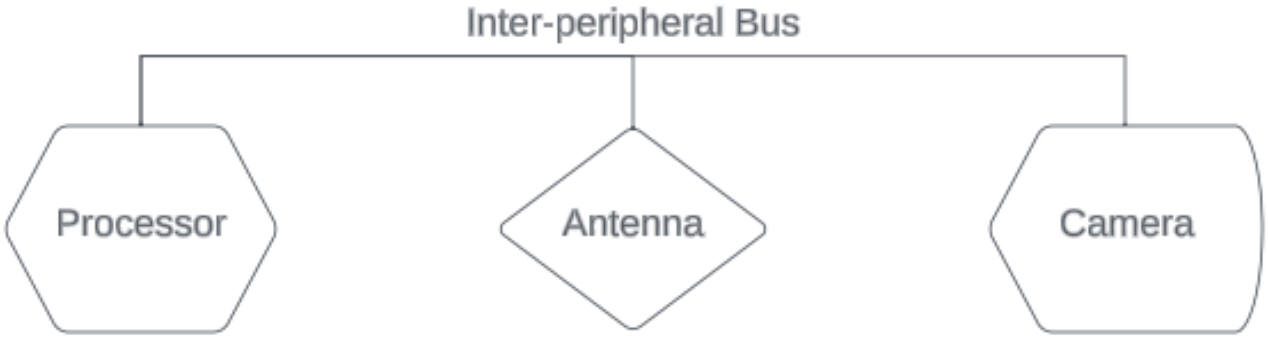
8.3 Implemented Design and Functionality



To implement a memory pool into DocetOS, a static memory pool firstly allocates a predefined block of memory at runtime. Blocks of memory of the required size can then be requested from the static memory pool, which will be aligned to the correct size to prevent fragmentation, and be sent to the memory pool.
Users can then request a block of memory from the pool and cast it to the desired data type, and then return the memory block to the pool when no longer required. This acts much like malloc() and free() do in a regular system, though as our memory pool defines memory at run time, our implementation is faster.
Memory blocks are stored within the memory pool as a linked list, having being cast to a struct data type with a pointer to the next block of memory in the list. As modifications to the memory pool required only one store operation in the pool, we protect the memory pool and static memory pool against race conditions using the Cortex-M LDREX and STREX instructions.

9 Demonstration

To demonstrate the capabilities of the modified version of DocetOS, we will attempt to simulate a portion of the priority inversion problem faced by the Mars Pathfinder mission [1], and show that our program is not only able to replicate the problem, but overcome it.
To accomplish this, we will consider the following scenario:



We have a simple system with a processor, antenna, and camera. The processor is responsible for sending commands to peripherals, along with a myriad of other devices it is simultaneously connected to, and hence is running multiple tasks concurrently. The camera is responsible for capturing images to be sent back to earth, which it sends to the antenna. The antenna waits until it has received a specified number of images (in our demonstration, we will use 10), filling its internal buffer, and then sends the images to earth for analysis.

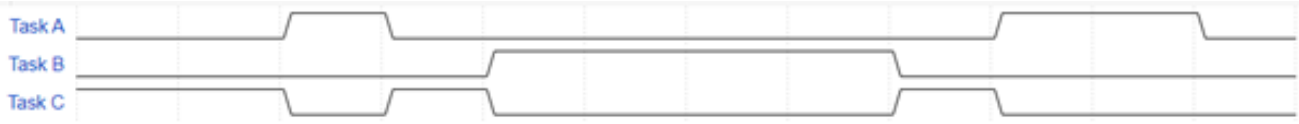
The challenge with this system, is that the processor, antenna, and camera all share the same bus. So, a mutex is required to protect the bus against being accessed by multiple tasks simultaneously. In the Mars PathFinder mission, it led to the following priority inversion problem when software engineers accidentally did not set the mutex to operate with priority inheritance. Consider 3 tasks of ascending priority:

Task A (high priority) – polls the antenna regularly to verify the internal buffer level. When at capacity, it sends a command to the antenna to send all stored data, emptying the buffer.

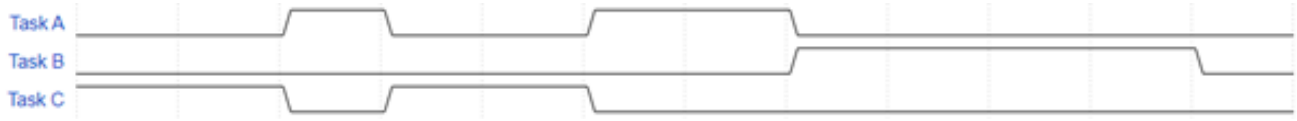
Task B (Medium priority) – a miscellaneous task related to other areas of the system which take up processing time.

Task C (Low priority) – Sends an instruction to the camera to take 10 images in succession and send them to the antenna, filling up the antennas buffer.

Without mutex priority inheritance, the combination of the timing of these tasks causes the following to occur:



As we can see due to Task B becoming active during the execution of Task C, the completion of Task C and consequently task A is delayed considerably, slowing down the filling of the antenna buffer. While this is not exactly how the Mars PathFinder priority inversion incident occurred, during the mission this task incident caused the automatic watchdog timer on the craft to trigger, resetting the entire system daily. In our demonstration, turning off mutex priority inheritance causes the same behaviour, slowing down the population of the antenna buffer with data, making the system unsuitable for real-world use.
To remedy the priority inversion vulnerability, we activate our mutexes priority inheritance functionality, correcting the system to the intended behaviour seen below:



Instead of Task C being interrupted by Task B, Task C inherits a higher priority due to the mutex priority inheritance, allowing it to run to completion freeing up the shared resource for Task A to complete. Then the miscellaneous Task B can freely run. The corrected behaviour demonstrates the enhanced functionality offered by our modifications. Specifically, we can see:

- The priority scheduler correctly scheduling higher priority tasks above lower priority tasks.
- Task sleeping, as the higher priority task only runs when it polls the antenna state, instead of constantly, avoid system deadlock.
- Our re-entrant mutex, protecting the shared resource and disallowing concurrent modification. Correctly blocking the higher priority task when the resource is in use by the lower priority task.
- Mutex priority Inheritance, assuring our lower priority task run with a higher priority than our medium priority task, freeing up the shared resource quickly to allow the antenna to send data to earth at an increased rate.

While not obvious in the wave diagrams, the data packets are blocks of memory that are dynamically allocated and released, which is not possible with the basic DocetOS functionality. The use of dynamically allocated memory is available due to our memory pool reserving and providing memory to be assigned to data packets to fill the buffer. Additionally, the improved wait and notify system is greatly enhancing the overall speed of the system.

10 Conclusion

To summarise, we have enhanced the capabilities and expanded the functionality of the real-time operating system DocetOS, along with reducing the systems vulnerability to bugs and unexpected behaviour by implementing:

- Fixed-Priority Scheduling
- Task Sleeping
- A Re-entrant Mutex
- Mutex Priority Inheritance
- An improved Wait and Notify System
- A Memory Pool

These features then allow us to demonstrate the expanded operation of the operating system by simulating the collection and transmission of data from a device that has multiple tasks of various priority attempting to operate peripheral resources concurrently, while miscellaneous tasks operate in the background. A function not easily accomplished by the original system, if at all.

11 References

[1] R. M. Pathan, "Report for the seminar series on software failures - mars pathfinder: Priority inversion problem," 2024. [Online]. Available: `https://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf`.