Systems Programming for ARM Assignment

January 2025 - ELE00138M

***Abstract*—The base design of the Real-Time Operating System, DocetOS, has been expanded to add mutual exclusion, synchronisation, dynamic memory, and inter-task communication. Efficiency and thread-safe operation have been prioritised through improvements to the scheduler, and the addition of semaphores and a re-entrant mutex.**

CONTENTS

***Index Terms*—ARM, embedded software, operating system, synchronisation, time-sharing.**

I. INTRODUCTION

ARM processors are commonly found in many consumer electronics devices and have long been a preferred architecture for devices such as mobile phones, which favour low power consumption. The microcontroller STMF4 Discovery Board contains an ARM Cortex M4 processor, which uses the ARMv7E-M 32-bit architecture [1]. Programs for the M4 are specified in the Thumb and Thumb-2 Instruction Set Architectures [2]. These can either be written directly in Assembly language, or compiled into Assembly language from a higher level language such as C.

In order to run multiple program tasks on a device at the same time and make development more straightforward, it is appropriate to develop a single Operating System (OS) in which different tasks can be run concurrently. The OS then handles CPU time-sharing to enable tasks to each use processing and memory resources in an appropriate way. An effective OS will allow all required tasks to run continuously whilst also minimising the time each task spends waiting for resources to become available. Especially important to design is minimising the chances of a crash occurring, upon which the CPU is no longer capable of running tasks, and minimising the chances of a deadlock occurring, wherein tasks enter an endless wait cycle and cease to be productive.

A. DocetOS

DocetOS is a simple Real Time Operating System (RTOS) written by A. Pomfret, which features basic task switching and time-sharing using a round-robin scheduler. This base OS has been further developed to improve the efficiency of the core scheduling system, whilst providing features such as memory allocation and mutual exclusion for tasks. Each task is scheduled using its unique Task Control Block (TCB), implemented in the OS_TCB_t custom data type. Sleep, Wait and Notify functionality, and a re-entrant Mutex were first added, and then improved upon in a new design. Fig. 1 shows the original architecture of DocetOS at this stage.

From a brief of nine potential improvements, six have been implemented, and these are:

- Mutual Exclusion using a Re-Entrant Mutex.
- A Fixed Priority Scheduler, with a variable number of priority levels.
- A Memory Pool that is shared between tasks.
- Efficiency improvements to the handling of Sleep functionality by the scheduler.
- Semaphores, and Binary Semaphores.
- A queue based Task Communication System.

Support for additional abstract data types was included in the designs of these features, which are suitable for use by future modules.

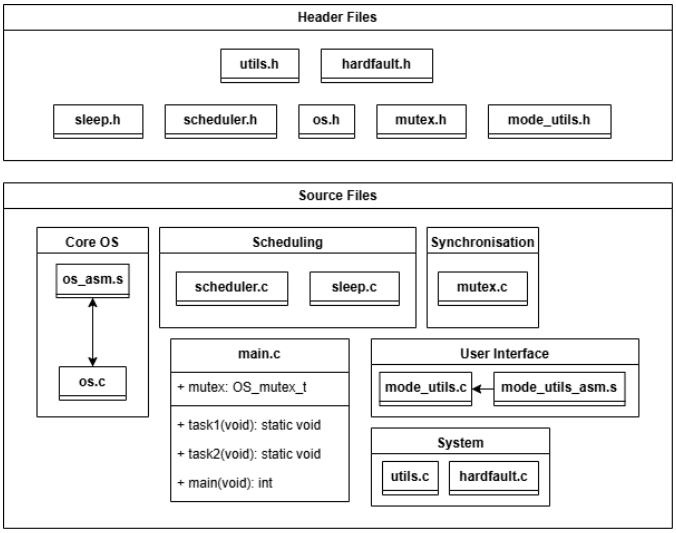

Fig. 1. Architecture of the original version of DocetOS inclusive of sleep, wait and notify features.

B. Key Design Choices

The design of the Re-Entrant Mutex was not altered significantly from the original implementation, although the Wait and Notify mechanisms were changed to accommodate the new scheduler design. The new scheduler design reused the double-linked list approach of the original round robin scheduler, however separate lists are now used for each priority level. The new common memory pool uses a pointer queue design allowing tasks to pass memory addresses to one another. Efficiency of the scheduler was improved by moving sleeping tasks to a new binary heap structure to sort them by their required wake times. Semaphores were used in the design of the memory pool to manage memory block availability, and binary semaphores are used to prevent simultaneous access to data structures.

II. Wait and Notify using the Re-Entrant Mutex

At any point during their runtime, tasks can remove themselves from the current schedule whilst they are waiting for system resources to become available. Using a SuperVisor Call (SVC), the task can ask to be added to a list of waiting tasks and force a context switch to a new task, while they wait to be notified. Tasks also have access to a notification function, through which they can notify waiting tasks that they have recently finished using resources which other tasks may have been waiting for.

In order to manage and simplify the implementation of this process within tasks a Re-Entrant Mutex has been written which may be held by only one task at a time. This mutex has been used to implement thread-safe printing to the USART output pins on the Discovery Board. The ARM synchronisation primitives LDREX and STREX are used to protect against two different tasks attempting to acquire the mutex at the same time.

This mutex does not support priority inheritance, where a task holding the mutex may be temporarily granted the priority of the highest priority task currently waiting for the mutex. Therefore the same mutex must not be used by tasks with different priority levels, to prevent a low priority task from acquiring the mutex, and then becoming unable to run to release it again because a higher priority task is running and looking for the mutex. To accommodate this, one mutex is used for printing by tasks on each priority level.

To ensure that all resources acquired have been released prior to a task relinquishing the mutex, a counter is incremented every time the mutex is called by the holding task, and decremented whenever a call to release is made. When the number of release calls matches the number of acquire calls made, the counter will be zero and the mutex released. This makes the mutex suitable for calling to block the mutual acquisition of multiple system resources using the same mutex.

If a task notifies the scheduler it is releasing a system resource whilst another task is partway through acquiring the mutex, it is possible that a task could be set to wait after the notification it is waiting for is sent, risking a perpetual wait. To prevent this, a notification counter is kept, to ensure that no additional notifications have occurred whilst the mutex was being acquired. If the notification counter has changed between the call to acquire the mutex and the call to wait, the call to wait is aborted, allowing the task to check if the notification was for the mutex.

III. The Fixed Priority Scheduler

The fixed priority scheduler provides a set number of priority tiers for tasks. The number of tiers is set at compile time using the *maxPriorities* macro in scheduler.h.

This new design reuses the double-linked list round robin from the original version of DocetOS, but now creates a separate round robin list for each priority level. As illustrated in Fig. 2 these round robin lists are accessed by the scheduler through a single pointer array, with the pointer to the highest priority task appearing at the head of the array.

The order of tasks completed by the scheduler is as follows:

1) Check the top of the Sleep Heap and move the tasks which are due to wake up into the Pending List.
2) Sort all tasks in the Pending List into the round robins for their priority level.
3) Iterate through the priority array from highest priority (0), through to lowest.
4) If a priority level contains a round robin list then it is checked.
5) The task at the head of the round robin is returned, and the head of the round robin progressed to the next task in the list.

If there are no runnable tasks at a given priority level, the array will store a null pointer at that index. This means that the scheduler does not need to spend time accessing a list directly to see if it is empty, though in this implementation there is a small efficiency loss in the time to access lower priority tasks. However, each priority having its own unchanging array address means that pointers never need to be moved between array indices for insertion and removal. This was judged to be a more effective design than using a linked list, as in that case

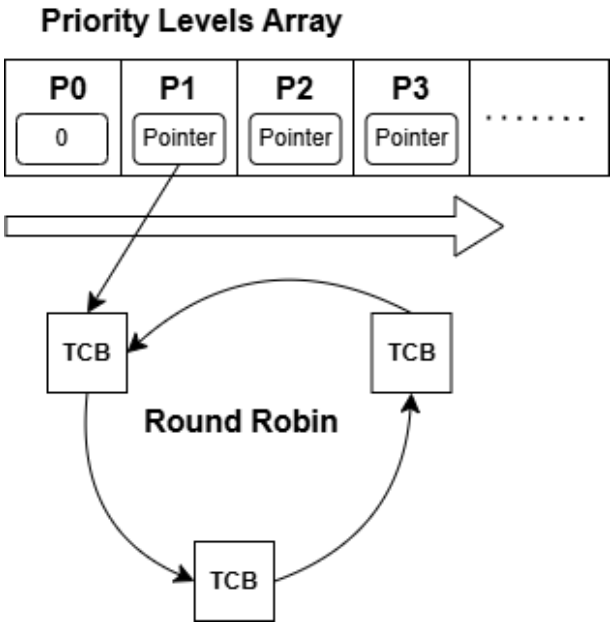

Fig. 2. Fixed Priority Scheduler; the priority array contains pointers to round robins for all levels which contain tasks. In this example 'Priority 0' is empty and contains a null pointer.

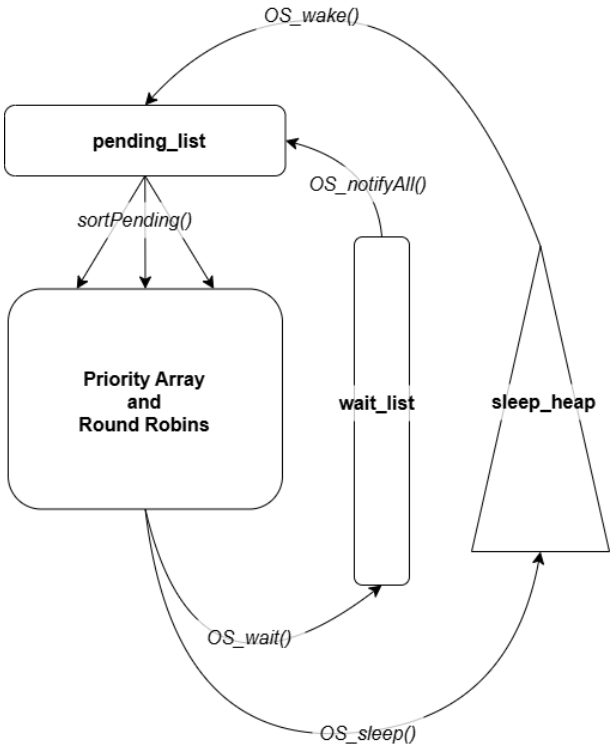

Fig. 3. Tasks can be temporarily removed from the scheduler and added to the Wait List or the Sleep Heap. They are sorted back into the round robins from the Pending List.

insertion and removal of priority lists would require traversal of the list.

In addition to the round robins, TCBs for tasks may also be stored in the Wait List if they are waiting for resources

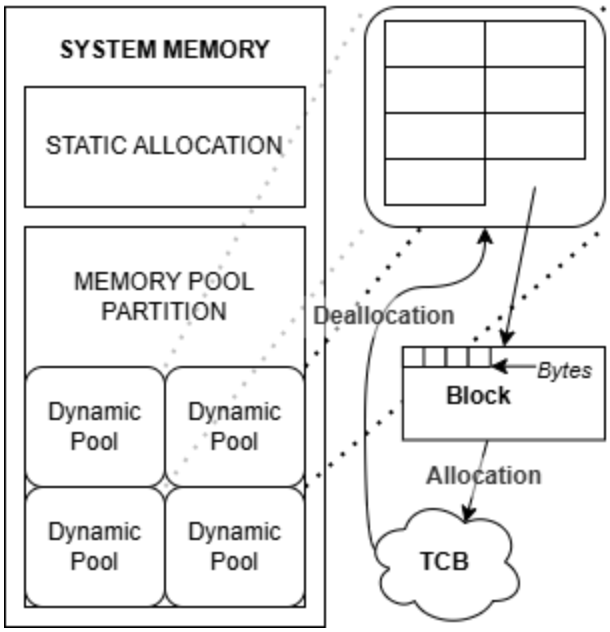

Fig. 4. The Pointer Queue structure used for Inter-Task Communication.

to become available, or in the Sleep Heap, if they have been set for a timed sleep. As illustrated in Fig. 3 When they are ready to be re-entered into the scheduler round robins, TCBs are first moved into the pending list, which is checked each time the scheduler is run.

IV. Shared Memory Pool

Two memory allocation mechanisms have been included in the design which serve separate functions. The first is a static allocation system, which is used to allocate memory for a variable number of task lists in the scheduler. The second is the dynamic allocation system, which serves as a shared memory pool which all tasks can use. As all tasks have access to this pool, it can also be used to pass data between tasks by sharing block addresses.

Fig. 4 shows the structure of the memory system and how it is organised. Within the system memory, a partition of 16KB is made (variable through a macro in mempool.c). Within this partition, multiple dynamic memory pools can then be set up using the same pool structure. Pools are divided into blocks of the size of the data type they are initiated with, therefore a new memory pool needs to be used for each data type that needs to be stored.

The size of each dynamic pool is set at compilation with the total number of available blocks recorded within the mempool_t type using semaphores (see Section V for details on their operation). Blocks from the dynamic pools (which are stored in the mempool_item_t type) are contained within a singly-linked list, with only the current head and semaphores made accessible. Blocks can then be allocated and deallocated by different tasks as they require, by accessing the head of the list. After a block is allocated, the head of the list is moved to the next block in the pool, and when a block is deallocated it is made the new head of the list.

A. Protections Against Concurrent Modification

The dynamic memory is accessible to all tasks whilst they are running the CPU in Thread Mode, meaning that context switching is enabled. As a result of this is is possible for multiple tasks to be attempting to access the pool at the same time. The linked-list structure of the pool is modified when it is accessed, and it is therefore necessary to prevent concurrent access to avoid corruption of the structure.

Two different features have been included in the design to protect against concurrent modification, whilst maintaining efficiency. These features address two potential causes of data corruption:

1) Two tasks attempting to change the head of the list at the same time.
2) Two tasks trying to access the final block in the pool at the same time.

1) Exclusive Monitors: The ARM synchronisation primitives LDREX and STREX are accessed using the CMSIS C functions '__LDREXW()' and '__STREXW()'. These use exclusive monitors to ensure that the current modification is the only one which has occurred in between a read and a write to the data structure taking place. If another access is detected the write is aborted and the read step is repeated.

These functions are invoked for both allocation and deallocation to memory pools when the pool head is to be accessed and changed.

2) Counting Semaphores for Blocks: Prior to a task being allowed to access the head of a pool with the use of exclusive monitors, it must first acquire a semaphore for a block. Semaphores are limited to the number of blocks in the pool and so failure to access a semaphore will block access to the pool structure. This ensures that there will still be a block to allocate before the task attempts to access the list head. When a task has successfully deallocated a block back to the pool, it then will return its semaphore to show that the block is now available for allocation again.

The design of the semaphores themselves is discussed in Section V.

V. Semaphores

Two different semaphore types have been implemented for resource management within DocetOS. Counting semaphores are used to track the number of blocks available for allocation within dynamically allocated memory. Binary semaphores are used to give exclusive access to shared data structures which are vulnerable to corruption through concurrent modification.

Counting semaphores could also have been used to implement binary semaphores by simply setting the maximum number of semaphores to 1 upon initialisation. However, from a design perspective it was judged to be better to implement them separately so that it is always clear in code which is being used. This method ensures that each declared semaphore type is used correctly wherever it is reused within the program, preventing bugs. The separation of the two also allowed for slight differences in operation. However, the two types share the same source code and header files, and are called in similar ways.

A. Counting Semaphores

Counting semaphores are declared using the OS_semaphore_t type. Using the static initialiser they are initialised to zero, and then need to have a quantity explicitly allocated in their implementation. The quantity of semaphores remaining is treated as volatile, forcing the value to be re-accessed and checked whenever it is used, as opposed to being cached for use within the same task.

Semaphores have not been used outside of larger data structures in this build of DocetOS; they are accessed by passing the structure's semaphore pointer to the function OS_semaphore_acquire(), and returned using OS_semaphore_release(). Tasks are allowed to make a call to acquire or release semaphores at any time, and are not required to have acquired a semaphore previously in order to be able to release one. This means that tasks are not prevented from moving blocks between different memory pools with different semaphore pointers.

When a call to acquire a counting semaphore is made, the task will continue to attempt to acquire until a successful attempt is made and the counter is decremented. If no semaphores currently remain the task is set to wait and will try again after a notification occurs. When a call to release is made, the task is immediately allowed to access the pointer to increment the counter. Once complete OS_notifyAll() is called and all waiting tasks are added back to the schedule at the next SysTick.

The internal structure of the semaphore functions follows a similar logic to the re-entrant mutex, with synchronisation primitives used to prevent two tasks from incrementing the counter to the same number at the same time.

B. Binary Semaphores

Binary semaphores operate under the same design as counting semaphores, with some small differences. Binary semaphores must be declared using the OS_semBinary_t data type and have their own acquire and release functions. Binary semaphores are statically initialised as 1 at runtime. After a binary semaphore is acquired the counter value is always set to 0, though when a binary semaphore is released the counter is always incremented by 1. This means that no more than 1 task can ever acquire the binary semaphore at a time. Though there is no explicit check against multiple releases, it is possible to see if they have occurred by checking the value of the counter and seeing if it is more than 1.

VI. Efficient Task Sleeping

In the original design of DocetOS, the round robin scheduler would contain tasks which were both asleep and awake, and would have to check the status before deciding whether or not to run a task. This meant increased time spent on list traversal before returning a new task to run. Now that DocetOS has multiple round robin lists for different priority levels it is desirable to offset the additional time taken to check each priority level by improving the efficiency of the sleep system.

For any given sleeping task, it can be assumed that it does not need to be returned again by the scheduler until at least the

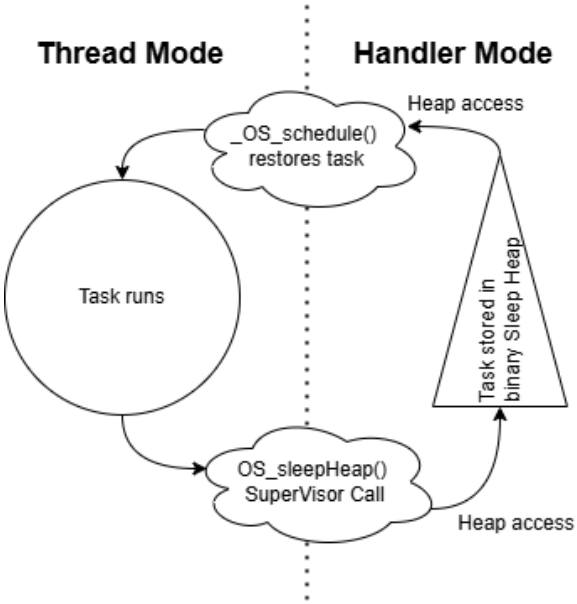

Fig. 5. Sleep heap access is kept thread-safe by accessing in Handler Mode code using a SuperVisor Call (SVC).

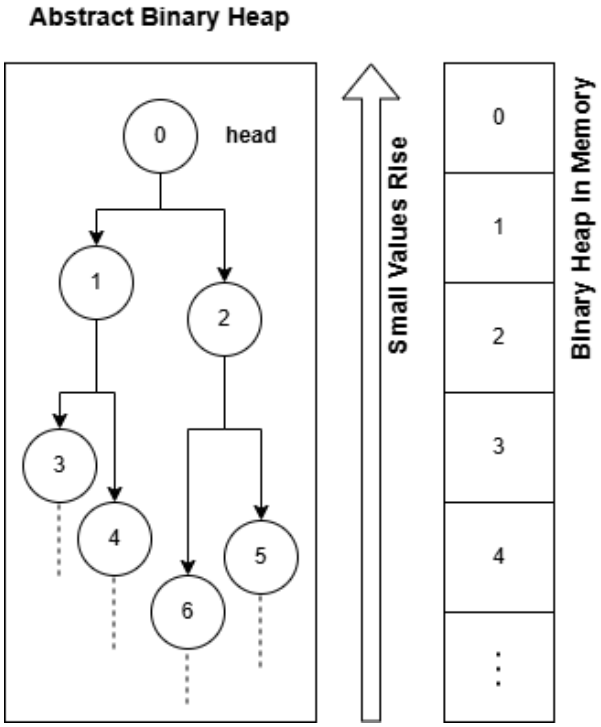

Fig. 6. The binary heap data structure is sorted so that TCBs with lower wake time values rise to the top. TCBs are stored within an array in physical system memory.

wake time. If it does not need to be run after this time, another task will be ahead of it in the schedule and the scheduler will not spend time checking it unnecessarily. Therefore sleeping tasks are now removed from the round robins and place in a wake time-sorted binary heap. The scheduler then checks if any tasks need to be moved back to the round robins before searching for the next task to run.

In order to make the process of adding tasks into the Sleep Heep as efficient as possible, an additional SVC has been designed and added to the OS_sleep() mechanism. This allows the TCB to access the scheduler to be sorted into the sleep heap immediately in thread-safe handler mode and then trigger a context switch, as illustrated in Fig. 5.

The Sleep Heap has been implemented using a generic binary heap structure contained in heap.c, illustrated in Fig. 6. The implementation of this structure requires the writing of an application specific comparator function, which is contained in scheduler.c. Using this comparator, tasks are sorted within the heap by wake time, with the lowest wake times rising to the top of the heap. This means that the tasks that need to wake up the earliest are always at the head of the heap, and easily accessible.

The Sleep Heap structure also contains an access token, implemented using a binary semaphore, which can be used to guard against concurrent access if it needs to be accessed by Thread Mode code. This could be used, for example, to extend a task's sleep time whilst it is still asleep. However, in the current implementation it is only accessed using Handler Mode code, and so the access token is not used.

As previously discussed in Section III, the Scheduler checks for tasks that need waking at every SysTick interrupt. When a task's wake time arrives, the Scheduler automatically adds it to the Pending List, from where it can be sorted back into the round robins. As this process occurs in Handler Mode only, it is completely thread-safe and no concurrency checks are needed.

VII. Communication Between Tasks

An Inter-Task Communication system has been designed and implemented which makes use of the memory pool system described in Section IV. Tasks are enabled to share allocated memory blocks with one another by using a common pointer queue. The design allows for a high number of concurrent communications, however the synchronisation of communications must be implemented within the tasks themselves.

In this simple design the producer-consumer problem is not specifically addressed, and it is up to the user to ensure that items are added to and removed from the queue at a similar rate. If the queue is overloaded, tasks will hang whilst they try to add to the full queue, and likewise if the queue is empty, tasks will listen until a message is added to the queue. These factors have been considered in the development of the demonstration code (see Section VIII), which shows how the system can be used in a stable way.

A. The Communications Queue

The Communications Queue, which operates as a pointer queue, has been designed as an efficient and thread-safe method for tasks to pass memory pointers between them. Transfer is quick because only a small 64-bit data structure needs to be copied during each transaction. The Queue structure used also means that access to an array holding the pointers only occurs at the start and end points. An endless

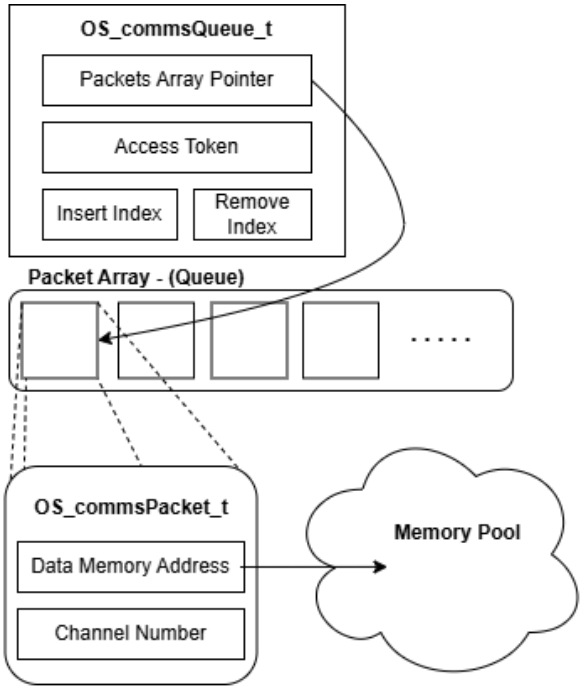

Fig. 7. The Pointer Queue structure used for Inter-Task Communication.

queue is used where if the end of the array is reached, the start of the array is used for the following value. This prevents pointers from having to be moved within the array during access.

The Queue is defined by the OS_commsQueue_t data type. The Queue structure when accessed contains a pointer to the zero index of the queue array, the index at which new items should be enqueued, and the index of the next item to be dequeued. The type also contains a pointer to a binary semaphore access token, which is used to protect against concurrent access.

The binary semaphore should be claimed by tasks before attempting to read or write to the queue. As there is only a single semaphore, if the semaphore is unavailable when a task tries to access the queue, it will be set to Wait. Whenever the semaphore is returned, OS_notifyAll() is called to prompt waiting tasks to try again. In this way tasks should always take turns when using the queue.

B. Communications Packets

Each index within the Queue array contains a Communications Packet (contained by the OS_commsPacket_t data type). This is a small 64-bit structure containing two 32-bit unsigned integers only. The first is the pointer to the memory location being shared between tasks, and the second is a channel number. Tasks will only dequeue a communications packet if the channel number matches the one they have requested. In this way, it can be ensured that messages will only be received by the correct tasks by setting the sending and receiving tasks to use the same channel number. Furthermore, this makes concurrent communications between two tasks possible by employing different channel numbers for different purposes. The data structures used for the Communications Packets and queue are illustrated in Fig. 7.

C. Other Considerations

The size of the Communications Queue, and so the number of items which can be simultaneously transmitted between tasks is initialised at compile time, through a #define statement QUEUE_SIZE, which can be found in queue.h.

In order to make the process of sending and receiving packets simpler for tasks, two functions are made available externally, OS_sendPacket() and OS_receivePacket(). These functions both require the task to input its requested channel number. The sending task can then directly pass the memory block pointer to the function as a void pointer, and the same void pointer is then returned to the receiving task. Packing and unpacking of the pointer into Communications Packets is handled by these functions, and does not need to be completed at task level.

It is important to note that no type information for the allocated memory is passed by the pointer queue, which stores memory addresses in void pointers. Therefore it is necessary for tasks to correctly cast the memory address to its anticipated type when received. Furthermore, the queue does not perform any checks to make sure that memory allocation and deallocation is performed correctly. After sending a memory pointer to another task, it is important that the sender does not deallocate the pointer back to the memory pool until the receiving tasked has finished using it. Therefore it is recommended that the responsibility for deallocation is always given to the receiving task.

VIII. Demonstration Program

The new build of DocetOS is being provided with a demonstration program which makes use of the new features which have been implemented. Functions for tasks have been written in tasks.c, and initialised to TCBs in main.c. These tasks will run concurrently when the program is initialised. The task functions which have been written are:

- veryHighPriorityTask
- lowPriorityTask
- greedyTask
- chattyTask
- listeningTask

The outputs of these tasks are written to the USART pins on the Discovery Board and can be monitored using a PC serial COM port.

1) veryHighPriorityTask: The very high priority task is designed to demonstrate how the scheduler always gives preference to the highest priority task. Every 20 seconds this task will interrupt all other tasks and print its output to USART. It has been initialised to a TCB with the highest priority level, Priority 0.

2) lowPriorityTask: This task, which has been initialised to the lowest priority level, Priority 4, will try to display the current system time once every second. However, it will always be interrupted when another task is run.

3) greedyTask: The greedyTask is designed to show the operation of the dynamic memory pool. After an initial wait of 15 seconds it will allocate 90 memory blocks of size packet_t to itself. Once completed it will sleep, and then deallocate all pools back to the pool, before repeating the allocations.

4) chattyTask and listeningTask: The chattyTask and listeningTask demonstrate intertask communication, with chattyTask acquiring a memory block of type packet_t, copying a string to it, and then sending it over either channel 1 or channel 5 via the Communications Queue. The listeningTask waits to receive messages on channel 1 and channel 5, and when it does, prints the resulting data to confirm correct receipt. After this, listeningTask will always deallocate used memory blocks back to the pool. It is worth noting that chattyTask and listeningTask use the same memory pool as greedyTask, and they often use the pool simultaneously.

When the two tasks are operating together, they are relying on their syncronised behaviour to continue to operate correctly. If they are initialised to the same priority level, sometimes the operations of higher priority tasks will cause them to fall out of synchronisation. This can cause listeningTask to hang briefly if it is waiting for a task to be sent by chattyTask. However, this should be temporary, and resolve when chattyTask sends another message.

5) Other Features Demonstrated: Tasks make use of the Re-Entrant Mutex whilst performing print statements. As different mutexes are used at each priority level, it is worth noting that higher priority tasks can interrupt the printing of lower priority tasks, but not the other way around.

Use of semaphores is demonstrated by tasks which use the Memory Pool and Communications Queue. The Sleep Heap is also used by all tasks each time they are set to sleep, using a variety of sleep times, showing that the sorting of the heap works as intended.

IX. Conclusion

An existing RTOS, DocetOS, has been expanded to improve scheduling, to add dynamic memory allocation and inter-task communication. This has been done whilst maintaining the thread-safe design through the use of exclusivity monitors, a Re-Entrant Mutex, and Binary and Counting Semaphores. The scheduler now uses fixed priority levels for tasks and a more efficient sleep, wake design. These features allow for more effective, efficient, and flexible time-sharing between tasks using the OS, as demonstrated in code provided with the new build. The components contained within the new build are shown in Fig. 8.

Many elements of the design are still simplistic, and must be used carefully in order to ensure stable operation. Further development of the new features through the use of Mutex Priority Inheritance and a more complex Wait and Notify system in particular would remove some cases of instability and inefficiency. Additional mechanisms could be put in place to prevent tasks from trying to send messages when the Communications Queue is full, or set them to wait more effectively when trying to receive messages.

The existing implementation is stable when running the demonstration code. It is hoped that the new design is a stable and logical platform on top of which further complexity may be added. However, it is likely that bugs and design oversights remain. It was found to be difficult to design demonstration code which operated as expected. In some cases it was difficult to determine whether problems experienced were a result of the OS design, or its implementation in the demonstration tasks. The new features should continue to be scrutinised during future development, for potential design flaws, but also considering that they were not always explicitly developed with expandability in mind.

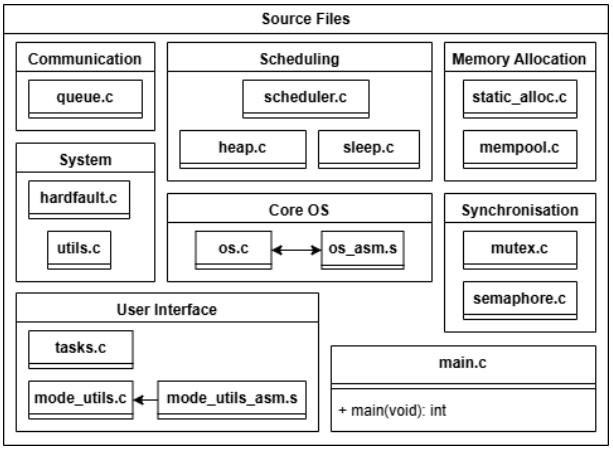

Fig. 8. The components of the new design of DocetOS.

References

[1] *Arm Cortex-M4 Processor Datasheet*, ARM, 2020. [Online]. Available: https://developer.arm.com/documentation/102832/latest/

[2] *ARMv7-M Architecture Reference Manual*, ARM, 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0403/ee/?lang=en