Systems Programming for ARM Assessment Report

Introduction

This report covers the task of improving the provided Operating System (OS), DocetOS, running on an ARM Cortex-M, by increasing its functionality. This was achieved by implemented these features: an efficient priority scheduler, improve the efficiency of wait and sleep, and a mutex implementation. These features were to be designed to work effectively and efficiently, such that there be no race conditions or deadlocks. Prior to this task, I undertook 10 weeks of lectures and labs dedicated to systems programming for ARM. This covered a vast variety of subjects, such as the ARM assembler, structures and collections, multitasking and specifics of ARM Cortex-M.

Modifications to DocetOS

Design Feature

Priority Scheduler

I have created a scheduler that orders tasks by priority, starting with the task that has the highest priority; In my implementation 0 is considered the highest priority, as the priority increases with how far back the task will be in the list. I used heaps to store my list of tasks, as it allows fast access to the top of the heap, which is task with the highest priority task; More about heaps in Data Structure. Task’s are also given two priority files: Assigned and Affected. On initialisation of a task both these priorities fields are set with the priority past into the initialisation function, however the affected priority is subject to change. If the task was to enter a wait or sleep state. The affected priority will increase by the ‘scheduler max priority’. This allows the scheduler to only check for tasks to run that have an affected priority lower than ‘scheduler max priority’; The scheduler has this set to 10. As the scheduler uses a heap to store the list of tasks, once it has reached the first task with a priority that exceeds the ‘scheduler max priority’ it knows it doesn't need to waste time checking the rest of the tasks in the heap.

The scheduler first checks the sleeping heap for any tasks that needs to be woken up. This doesn't take every longe, as all the sleeping tasks are stored in a heap; with the task to be woken up first at the top of the heap. Therefore the scheduler only needs to check the top element. If it is time to wake a task, it is simply extracted from the heap and its affected priority set back to the assigned priority. This is done so that the scheduler can check it again. Then it checks whether the top element in the running heap has the right priority to run. The diagram in figure 1.1 also displays this behavior.

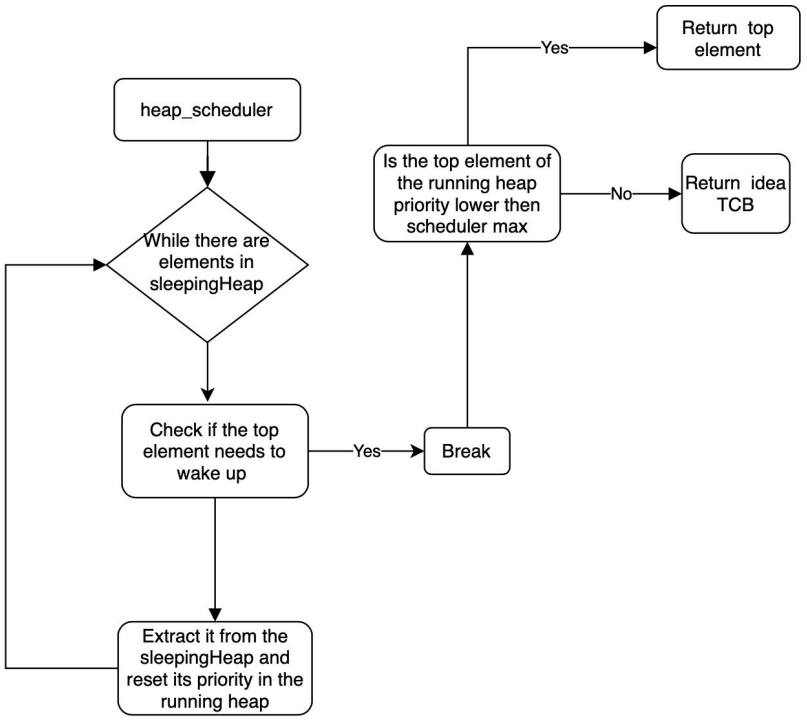

Figure 1.1

Wait and sleep

The wait callback function can be found in heapScheduler.c and is used to set the current task to a wait state. In my implementation this meant to set the wait bit in the state filed. Also to prevent the scheduler from checking a waiting task it increases the priority by the scheduler max priority. There is also a checksum to prevent tasks from missing a notify. This works as the check code is always sent on a wait call. If the checksum past to wait does not match the true checksum value, then a notify has been called since starting the function. Therefore wait does nothing to the task and returns it. This is very inefficient as a check needs to be done every time the wait function is called. With more time and a better understanding I would have liked to implement a method of storing a list of waiting task, and notify would start to work through the list waiting task.

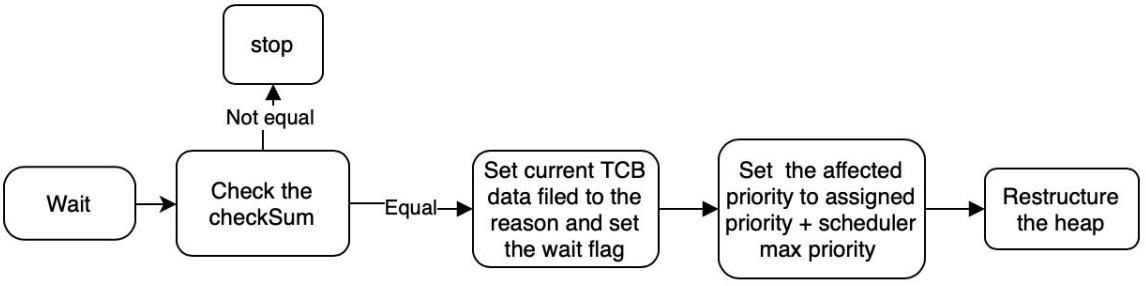

Figure 1.2

The sleep function can be found in heapScheduler.c and it simply stores the time of when a task should be woken in the it's data field, and then stores this in the sleeping heap. This heap is used in the scheduler for checking if a task should be woken, and using a heap allows the option to order the tasks so that the first task to be woken is at the top, saving the scheduled time checking all the sleeping tasks. The sleep function starts by fetching the current task and then setting its data field to the current tick value plus the additional ticks value, passed into the function. Then it sets the sleep bit in the task state filed and inserts the task into a sleeping heap, that holds all the sleeping task. To prevent the scheduler from checking this task it's affected priority is changed and the running heap is restructured.

Mutex

The mutex controls when a tasks can run, if the mutex is busy with another task, the task requesting access will be put in a waiting state. There are two functions associated with the mutex: Mutex Acquire and Mutex Release. The Mutex acquire firstly loads the value in the mutex, if it is empty then, try to store the current task in mutex. If the store fails start again, if it passes increase the counter. If the mutex is not empty, check the mutex doesn't have the current field and then set the current task's wait and increase the counter. This can be seen in figure 1.3.

The mutex release, first checks that the mutex is owned by the task trying the release it. Then decrease the counter, checks that the counter is equal to zero. If it is then reset the mutex and call notify.This can be seen in figure 1.4. If I had more time and a better understanding I would have modified the mutex to function as a semaphore, and created a list of waiting . Then when the mutex is released, complete the tasks from the waiting list.

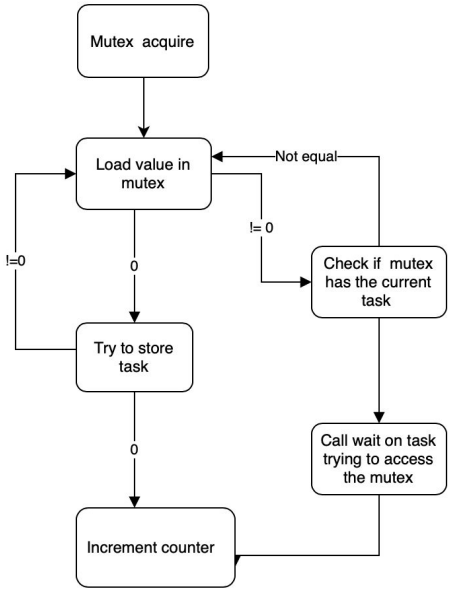

Figure 1.3

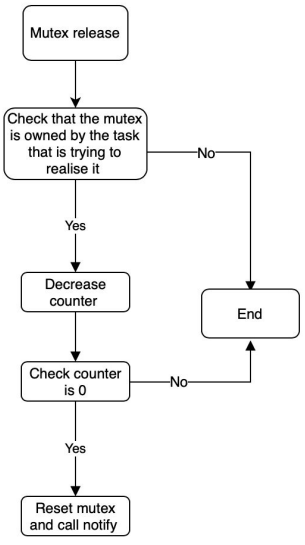

Figure 1.4

Data Structure

Heap

A simple binary heap is a data structure that implants a binary tree, such that each value can be a parent to children. In a heap the child node's are considered to be of a lower priority than their parent node, also a parent can only have up to two children. This feature is why I have chosen to implement this structure, as I can create a scheduler that order tasks by their priority. Another advantage is when adding or removing tasks from the heap, the heap will restructure itself to maintain the correct order. If the heap is implemented correctly, the top element will always be the task with the highest priority. Making it simple and fast to extract the next task.

I have used two heaps in my implementation: running heap and sleeping heap. The running heap is used in the scheduler and orders the tasks by their priority. Meaning that the highest priority task will always be at the top of the heap. As the heap is designed in that way I took this opportunity to manipulate the order of tasks, such that the scheduler will only look for tasks that are demanded runnable. I took a similar approach with the sleep heap, however the task's are ordered such that the task that will wake up the soonest is at the top of the heap. Therefore then checking for the new task to wake up the scheduler just needs to check the top element.

There are two main functions the heap uses: Heap Up and Heap Down. I have modified these functions, such that the element can be in any position in the heap. The heap up started by checking if it is the first element, if it is then break otherwise compare it with its parent. If the parent is smaller stop, else if the child is smaller swap the parent and child. This is repeated until the element is in the right place. This can be seen in the diagram in figure 2.1.

The heap down first checks if the element as any children, if no then stop otherwise find out if it has one or two children. If it has two compare both children and find the smallest , then compare the smallest child with the parent. If the parent is bigger then swap the child and parent around and start again with the parent in the child's position. . If the parent only has one child, compare the child with the parent. And again if the parent is bigger then swap the parent and child and start again with the parent in the child's position. This can be seen in figure 2.2

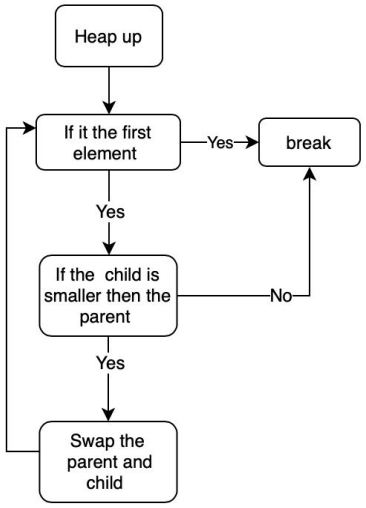

Figure 2.1

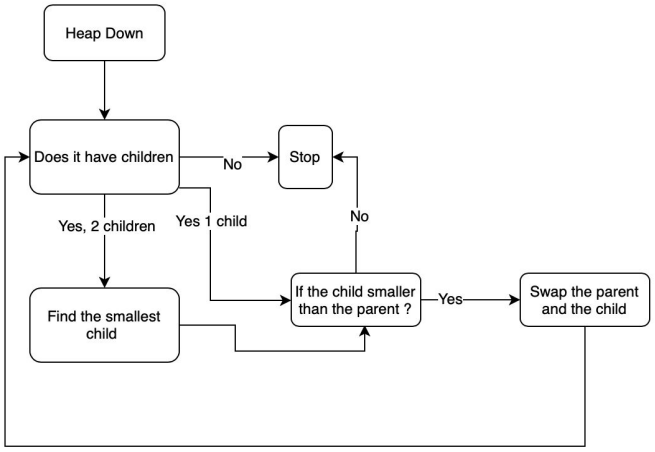

Figure 2.2

Conclusion

To conclude I have modified replace the simple round robin scheduler with my heap scheduler, that order a list a task by their priority. To element time constraint the scheduler will only check task's are of a certain priority (0 - scheduler max priority). I have used heaps as they allow for a fast response time when retrieving the next task and order the list in priority. I have implemented a mutex that controls when to run a task and when to wait it. And finally a sleep function that allow tasks to sleep and then woken again.

As interesting as I have found this module and task I have found some of the concepts and task challenging. With more time and a better understanding I would have liked to of explored other implements that improve DocetOS.