UNIVERSITY of York

Systems programming for ARM
extending DocetOS

Y1471938

January 15, 2019

Abstract

This document outlines the extensions and modifications made to the DocetOS operating system. The focus of the extensions is on scalability with the aim of allowing the extended docetOS to work with a large number of tasks efficiently. This results in a slight overhead in memory usage but allows for a stochastic priority scheduler that can efficiently switch between many tasks, and prevent starvation of individual tasks by ensuring that even the lowest priority task gets occasionally allocated CPU time. The scheduler incorporates an efficient heap/hashtable based task sleep and waiting mechanism which aims to minimise the operations that have to be performed when a task transitions from one state to another. I have provided the user with tools to easily set up inter-task communication through the OS channel manager, which is responsible for providing one or more tasks with access to the correct channel, and recycling the channel once the tasks no longer need it. In order to allow the user to easily allocate memory I have created a "memory cluster" that simultaneously aims to prevent memory being wasted and tasks having to wait for memory to become available. The memory provided by the cluster is guaranteed to be 8-byte aligned and can therefore also be used for task stacks. The OS itself uses the memory cluster to allocate the vast majority of its own internal resources such as hash tables, queues, heaps etc. All data structures used by the OS are standalone and available to the user for their own use. They come with convenience functions to quickly create and destroy them e.g "new_hashtable(...)" allocates and initialises a hash table with the desired size. To prevent priority inversion the operating system supports priority inheritance for tasks using the mutex struct that comes with the OS.

Contents

1 The Stochastic Scheduler

My plan was to allow DocetOS to support a large number of tasks, which made me come up with the following key requirements for the priority scheduler:

- The scheduler should be pre-emptive, but allow tasks enough time to run to avoid wasting CPU time on frequent context switching.
- CPU time allocated per task should be dependant on the tasks priority, but the highest priority task should not be the only task getting CPU time (not just standard Fixed priority pre-emptive scheduling behaviour where only the highest priority task is selected)
- Task starvation should be avoided, even the lowest priority task should have a non-zero probability of getting selected during task switch.
- Task switching and status changes of tasks should not result in a large overhead of CPU time.
- The scheduler needs to avoid priority inversion.

1.1 Implementation

At the end of this section you will find an example illustration of the internal state of the scheduler at a particular point in time. The scheduler contains 2 heaps and 5 hashtables which are used to keep track of the various states a task can be in, and allow quick access to any task no matter its state.

- The **Scheduler Heap** is used by the scheduler to determine what task to select for execution. This heap can contain tasks that are active and tasks that are waiting, sleeping or have tasks that have run to completion (see highlight 1). Tasks that are not active are only moved/removed from the scheduler heap if they are encountered in the heap during task selection. This is done to save cpu time on operations that might be unnecessary, a task that goes to sleep might wake up long before the scheduler comes across it in the heap, so moving it to the sleep heap would only waste cpu cycles on restructuring the sleep and scheduler heaps. When the scheduler comes across a sleeping/waiting task it checks if this state is still applicable and only then moves the task to a different heap/hashtable.

- The **Sleep Heap** stores tasks that are sleeping and have been removed from the scheduler heap. In this heap the tasks position in the heap is not determined by its priority but by its remaining sleep time (highlight 2). The scheduler removes the first node, updates its remaining sleep time by referring to the time it started sleeping and checks if it has woken up, if it has it is added back to the scheduler heap and the process is repeated. If the first node has not woken up yet then none of the other nodes will have woken up either. This approach means only one sleeping node has to be updated.

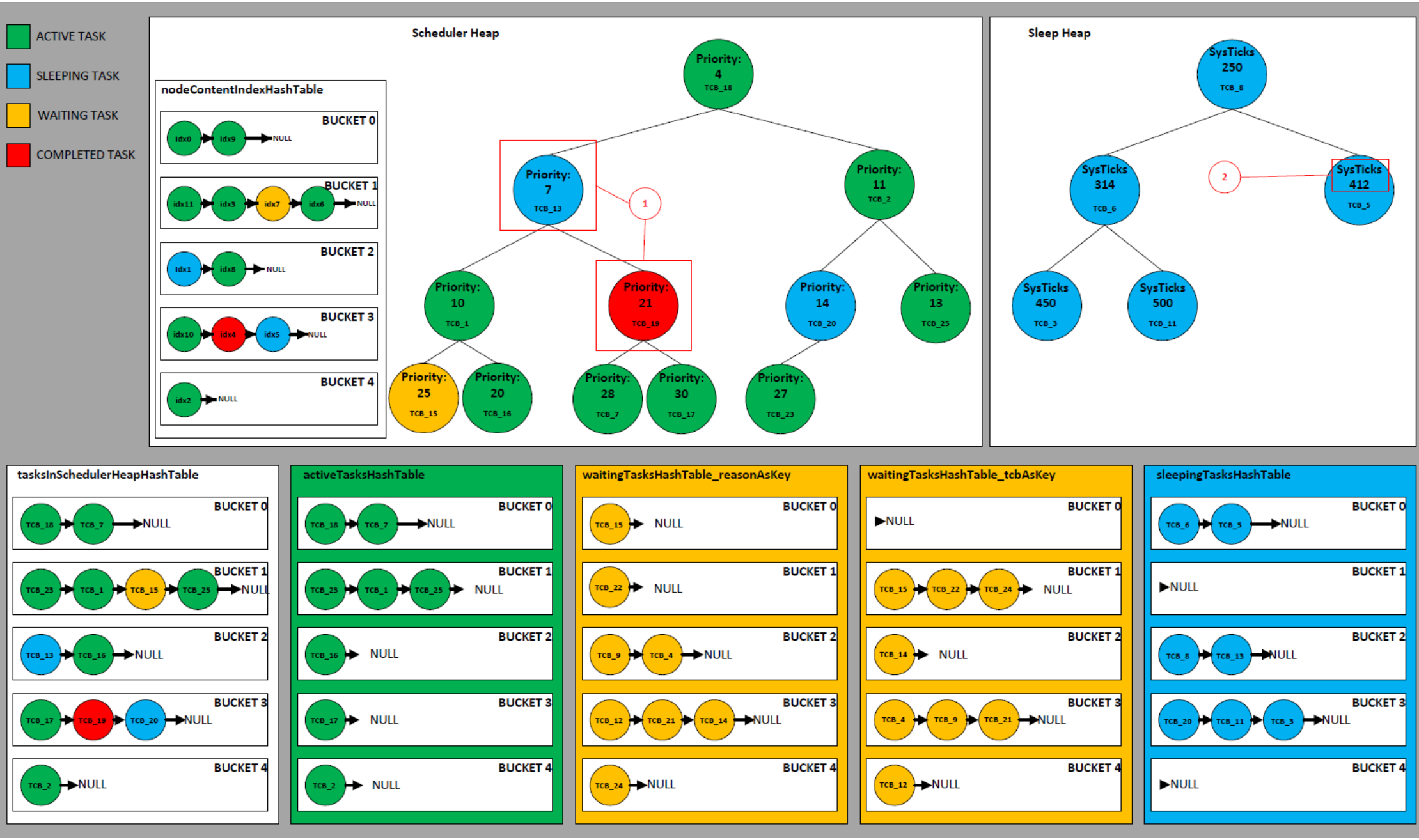
- The **tasksInSchedulerHeapHashTable** keeps track of all the tasks currently present in the scheduler heap. Used to check what tasks have to be added back to the scheduler heap after wake/notify and which ones were never removed.

- The **activeTasksHashTable** keeps track of all tasks that are currently executable.

- The **waitingTasksHashTable_reasonAsKey/_tcbAsKey** both keep track of tasks that are waiting. The _reasonAsKey table is used during notify to change the state of a task waiting for "reason" without needing to check each waiting task. the _tcbAsKey allows the OS and user to check if a given task is waiting.

- The **sleepingTasksHashTable** keeps track of all tasks that are sleeping, no matter what heap they are in.

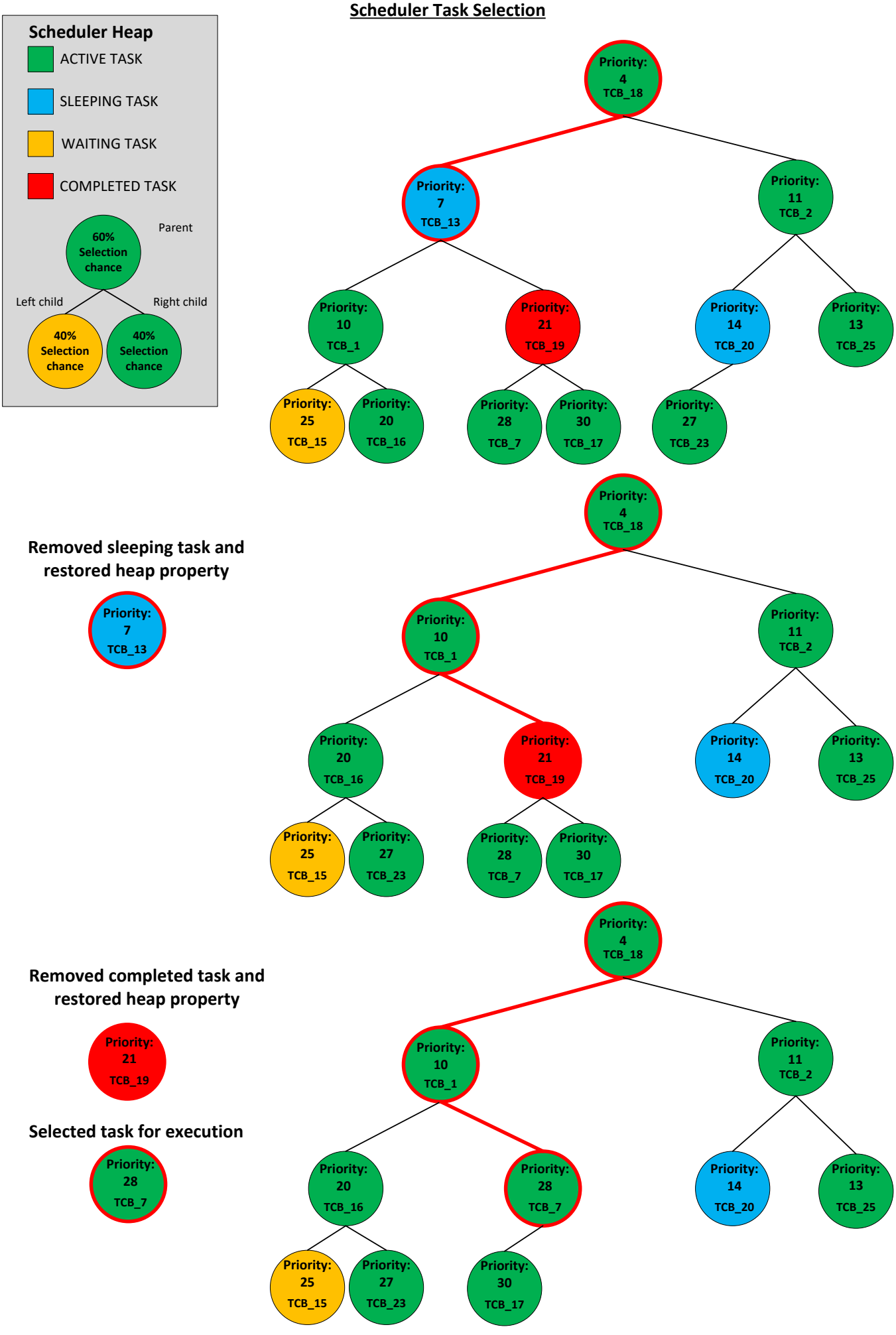
- The **nodeContentIndexHashTable** internal (optional) hashtable of heap data structure, can be used to quickly determine at what index a node with a specific content is located in the heap. This is used during priority inheritance.


ACTIVE TASK
SLEEPING TASK
WAITING TASK
COMPLETED TASK
Scheduler Heap
nodeContentIndexHashTable
BUCKET 0
idx0
idx9
NULL
BUCKET 1
idx11
idx3
idx7
idx6
NULL
BUCKET 2
idx1
idx8
NULL
BUCKET 3
idx10
idx4
idx5
NULL
BUCKET 4
idx2
NULL
Priority: 4 TCB_18
Priority: 7 TCB_13
Priority: 11 TCB_2
Priority: 10 TCB_1
Priority: 21 TCB_19
Priority: 14 TCB_20
Priority: 13 TCB_25
Priority: 25 TCB_15
Priority: 20 TCB_16
Priority: 28 TCB_7
Priority: 30 TCB_17
Priority: 27 TCB_23
1
Sleep Heap
SysTicks 250 TCB_8
SysTicks 314 TCB_6
SysTicks 412 TCB_5
SysTicks 450 TCB_3
SysTicks 500 TCB_11
2
tasksInSchedulerHeapHashTable
BUCKET 0
TCB_18
TCB_7
NULL
BUCKET 1
TCB_23
TCB_1
TCB_15
TCB_25
NULL
BUCKET 2
TCB_13
TCB_16
NULL
BUCKET 3
TCB_17
TCB_19
TCB_20
NULL
BUCKET 4
TCB_2
NULL
activeTasksHashTable
BUCKET 0
TCB_18
TCB_7
NULL
BUCKET 1
TCB_23
TCB_1
TCB_25
NULL
BUCKET 2
TCB_16
NULL
BUCKET 3
TCB_17
NULL
BUCKET 4
TCB_2
NULL
waitingTasksHashTable_reasonAsKey
BUCKET 0
TCB_15
NULL
BUCKET 1
TCB_22
NULL
BUCKET 2
TCB_9
TCB_4
NULL
BUCKET 3
TCB_12
TCB_21
TCB_14
NULL
BUCKET 4
TCB_24
NULL
waitingTasksHashTable_tcbAsKey
BUCKET 0
NULL
BUCKET 1
TCB_15
TCB_22
TCB_24
NULL
BUCKET 2
TCB_14
NULL
BUCKET 3
TCB_4
TCB_9
TCB_21
NULL
BUCKET 4
TCB_12
NULL
sleepingTasksHashTable
BUCKET 0
TCB_6
TCB_5
NULL
BUCKET 1
NULL
BUCKET 2
TCB_8
TCB_13
NULL
BUCKET 3
TCB_20
TCB_11
TCB_3
NULL
BUCKET 4
NULL

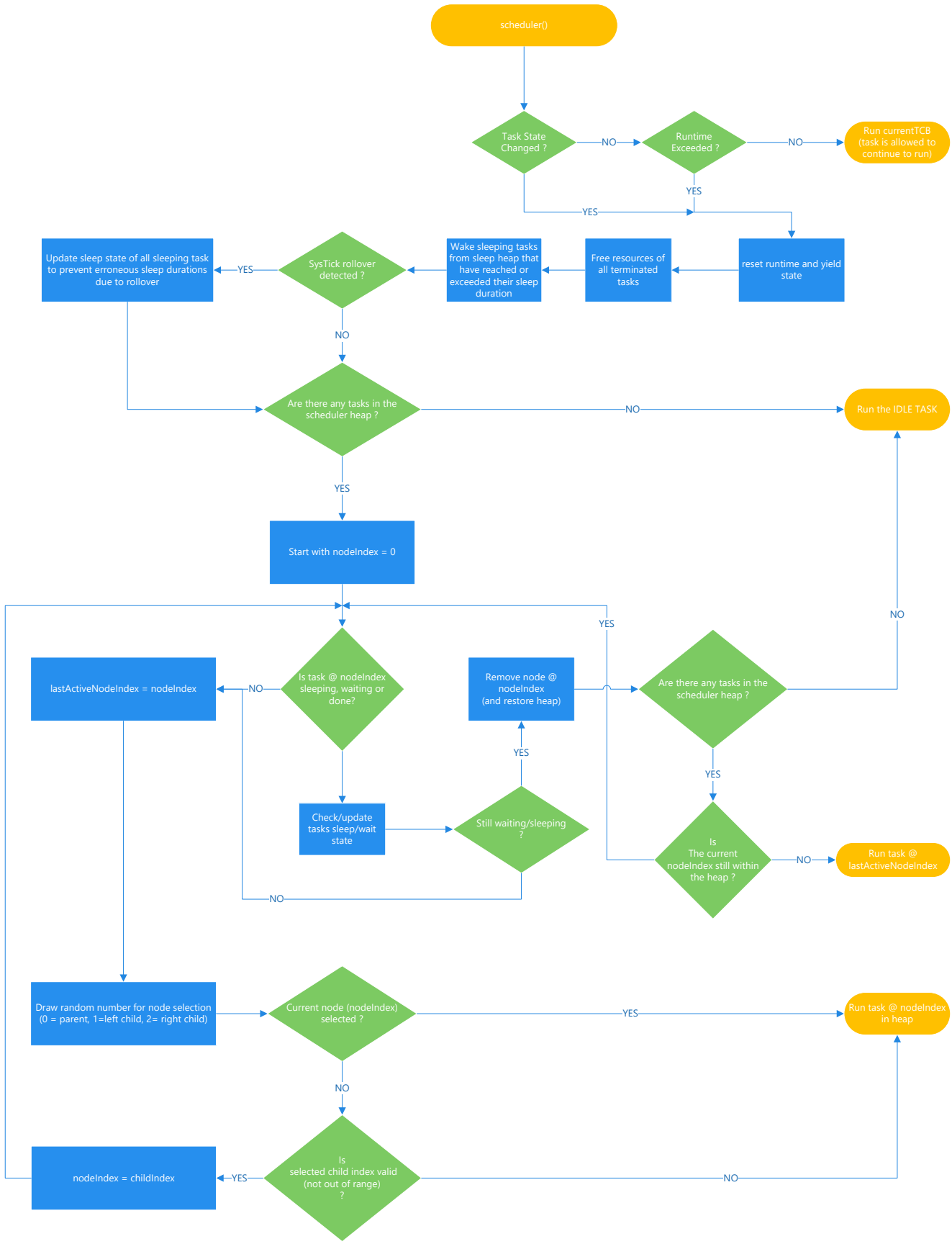
1.2 Selecting a Task

Before a task switch is initiated the scheduler checks if the current task has changed state (from active to sleep, wait etc.) or if it has exceeded its maximum runtime (set in the scheduler header). The scheduler does not force a task switch every SysTick to avoid wasting cycles on the task selection process. The scheduler task selection process uses a min heap which stores the tasks in ascending order of priority (lower number means the task has a higher priority). Starting from the top node of the heap the scheduler decides at each node (via pseudo random number) if the current node should be selected or if the process should be repeated for one of the child nodes. If during this process the scheduler comes across a task that is not in the active state it will remove it from the scheduler heap and place it in the corresponding heap/hashtables. The reason for not removing a task instantly when it goes into the sleep or wait state is that the task could wake/get notified before the scheduler encounters it during task selection, which would mean that moving it from the scheduler heap would have just wasted cycles.

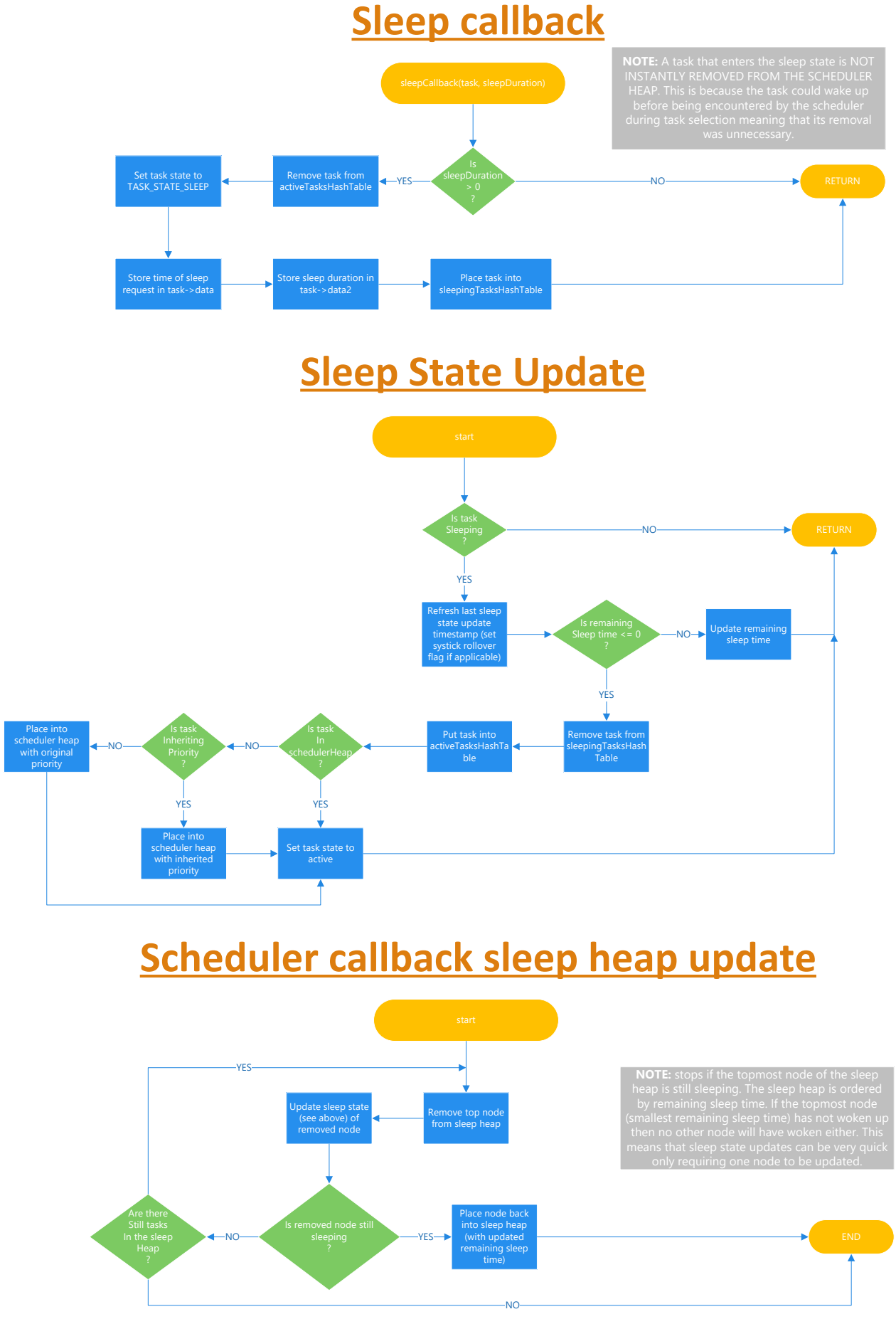
The reasoning behind using a heap and an element of randomness for task selection (with the chance of a task getting selected being correlated with the tasks priority) is that it avoids task starvation by giving every task a non-zero probability of getting selected, no matter how low their priority. My heap data structure also has a modification that the user can enable which allows the heap to store the index where certain content can be located in the heap, this allows for quick insertion/removal of nodes at any heap index. This is especially useful during priority inheritance process.


Scheduler Task Selection
Scheduler Heap
ACTIVE TASK
SLEEPING TASK
WAITING TASK
COMPLETED TASK
Parent
60% Selection chance
Left child
Right child
40% Selection chance
40% Selection chance
Priority: 4 TCB_18
Priority: 7 TCB_13
Priority: 11 TCB_2
Priority: 10 TCB_1
Priority: 21 TCB_19
Priority: 14 TCB_20
Priority: 13 TCB_25
Priority: 25 TCB_15
Priority: 20 TCB_16
Priority: 28 TCB_7
Priority: 30 TCB_17
Priority: 27 TCB_23
Removed sleeping task and restored heap property
Priority: 7 TCB_13
Priority: 4 TCB_18
Priority: 10 TCB_1
Priority: 11 TCB_2
Priority: 20 TCB_16
Priority: 21 TCB_19
Priority: 14 TCB_20
Priority: 13 TCB_25
Priority: 25 TCB_15
Priority: 27 TCB_23
Priority: 28 TCB_7
Priority: 30 TCB_17
Removed completed task and restored heap property
Priority: 21 TCB_19
Selected task for execution
Priority: 28 TCB_7
Priority: 4 TCB_18
Priority: 10 TCB_1
Priority: 11 TCB_2
Priority: 20 TCB_16
Priority: 28 TCB_7
Priority: 14 TCB_20
Priority: 13 TCB_25
Priority: 25 TCB_15
Priority: 27 TCB_23
Priority: 30 TCB_17

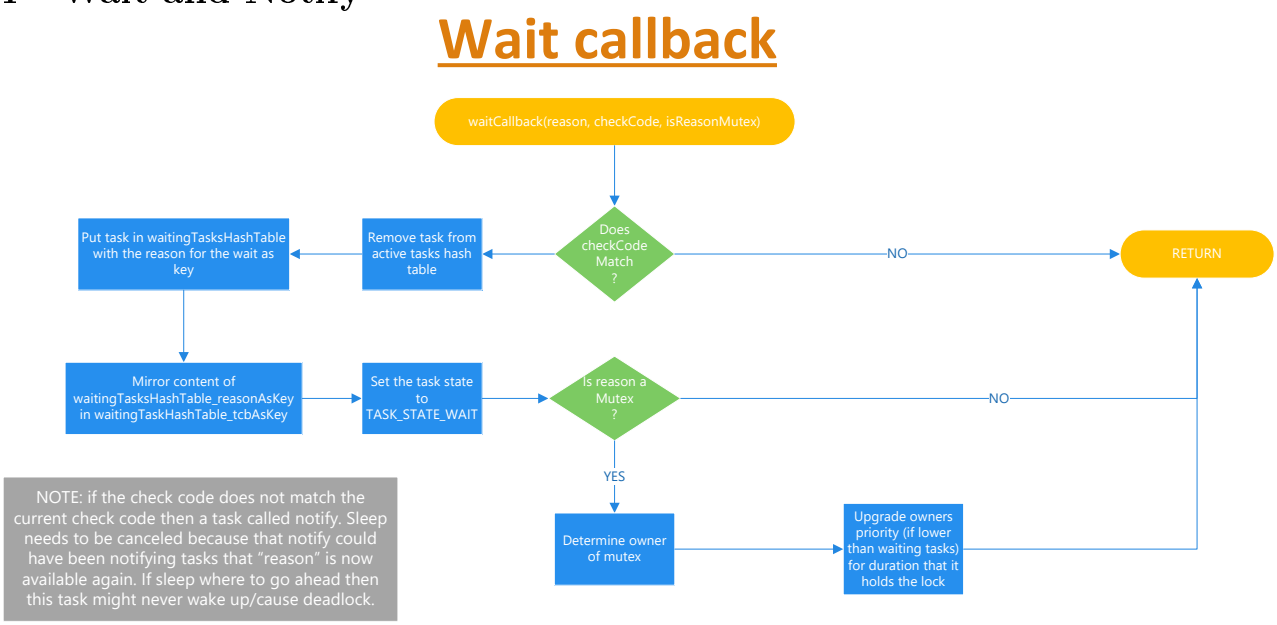
Scheduler Task Selection

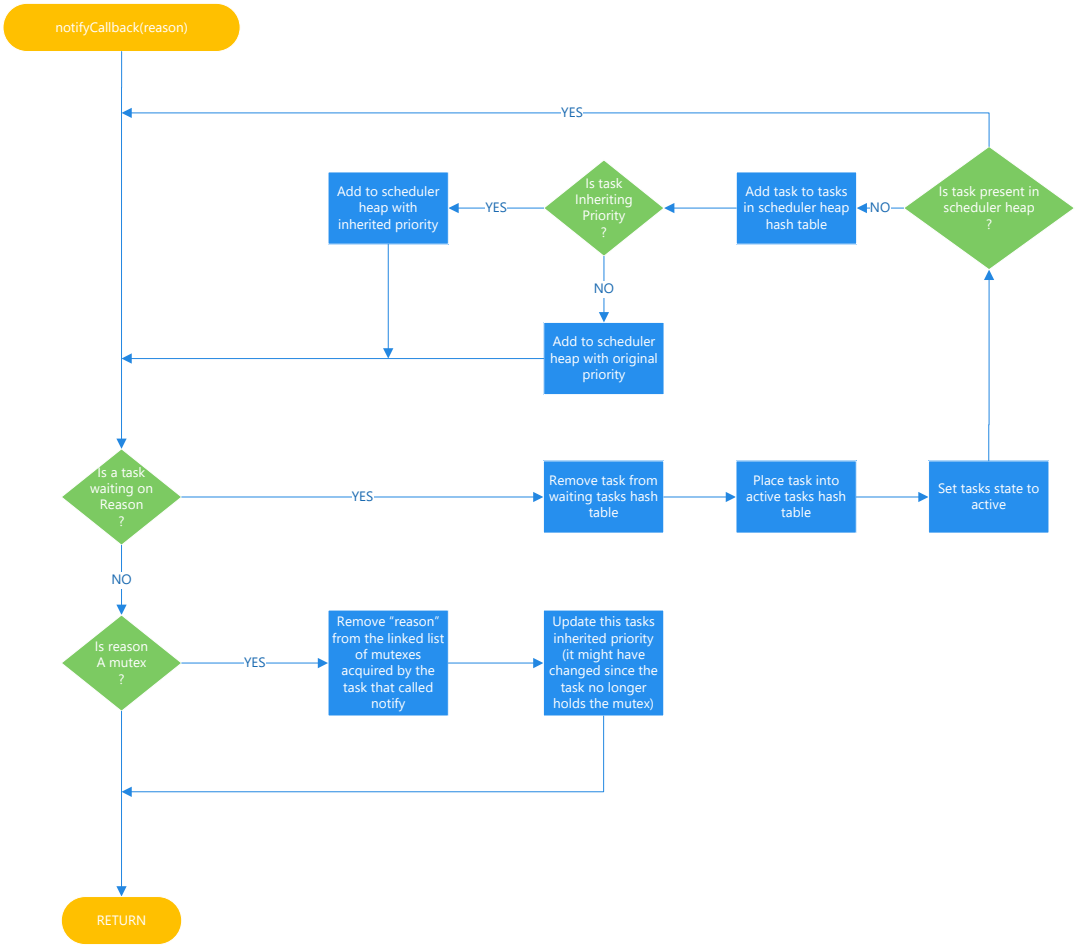


1.3 Sleep


Sleep callback
NOTE: A task that enters the sleep state is NOT INSTANTLY REMOVED FROM THE SCHEDULER HEAP. This is because the task could wake up before being encountered by the scheduler during task selection meaning that its removal was unnecessary.
sleepCallback(task, sleepDuration)
Is sleepDuration > 0 ?
YES
NO
RETURN
Remove task from activeTasksHashTable
Set task state to TASK_STATE_SLEEP
Store time of sleep request in task->data
Store sleep duration in task->data2
Place task into sleepingTasksHashTable
Sleep State Update
start
Is task Sleeping ?
NO
RETURN
YES
Refresh last sleep state update timestamp (set systick rollover flag if applicable)
Is remaining Sleep time <= 0 ?
NO
Update remaining sleep time
YES
Remove task from sleepingTasksHash Table
Put task into activeTasksHashTa ble
Is task In schedulerHeap ?
NO
Is task Inheriting Priority ?
NO
Place into scheduler heap with original priority
YES
Place into scheduler heap with inherited priority
YES
Set task state to active
Scheduler callback sleep heap update
start
YES
Remove top node from sleep heap
Update sleep state (see above) of removed node
NOTE: stops if the topmost node of the sleep heap is still sleeping. The sleep heap is ordered by remaining sleep time. If the topmost node (smallest remaining sleep time) has not woken up then no other node will have woken either. This means that sleep state updates can be very quick only requiring one node to be updated.
Are there Still tasks In the sleep Heap ?
NO
Is removed node still sleeping ?
YES
Place node back into sleep heap (with updated remaining sleep time)
END
NO

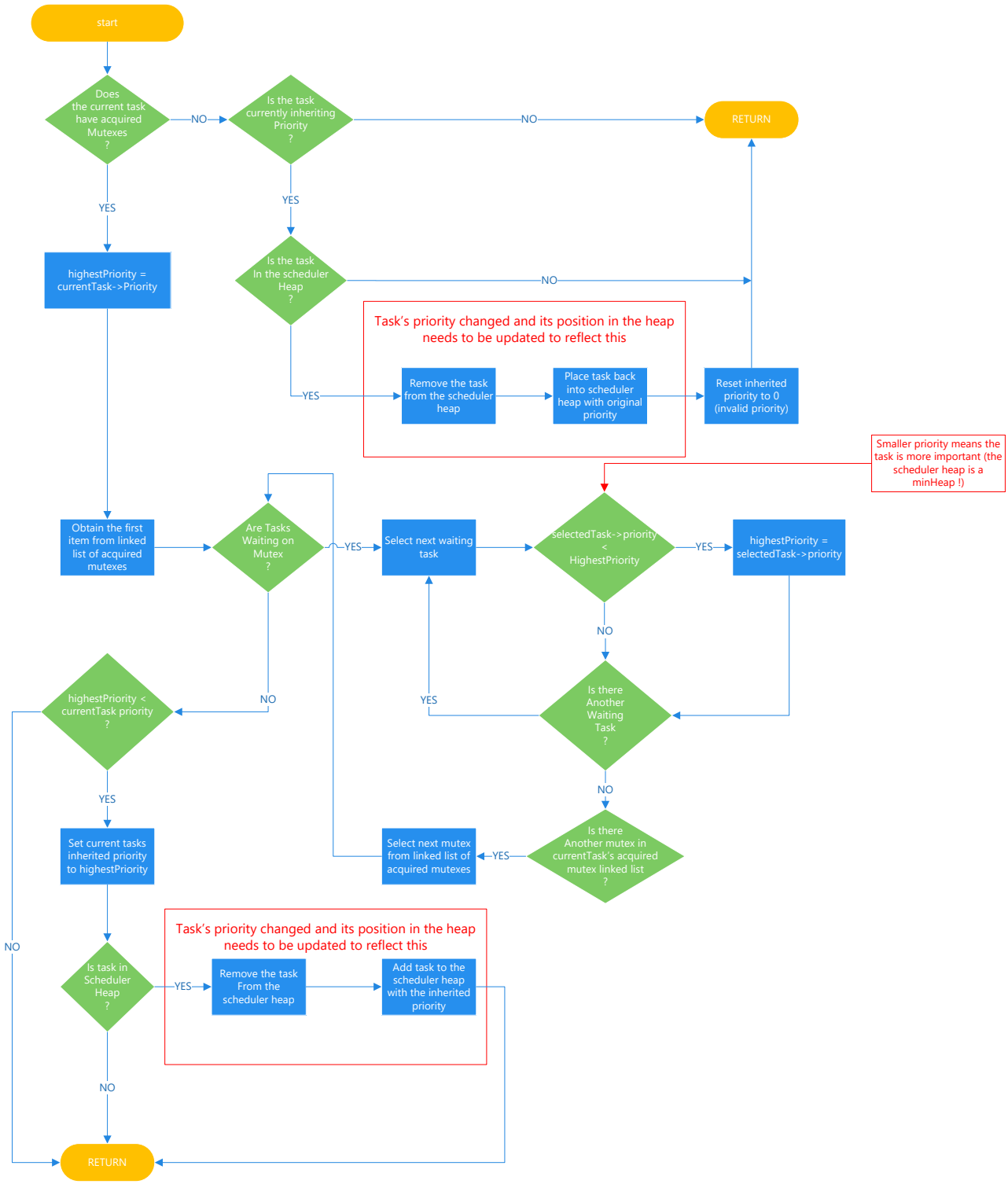
1.4 Wait and Notify





1.5 priority Inheritance

Updating the priority inheritance



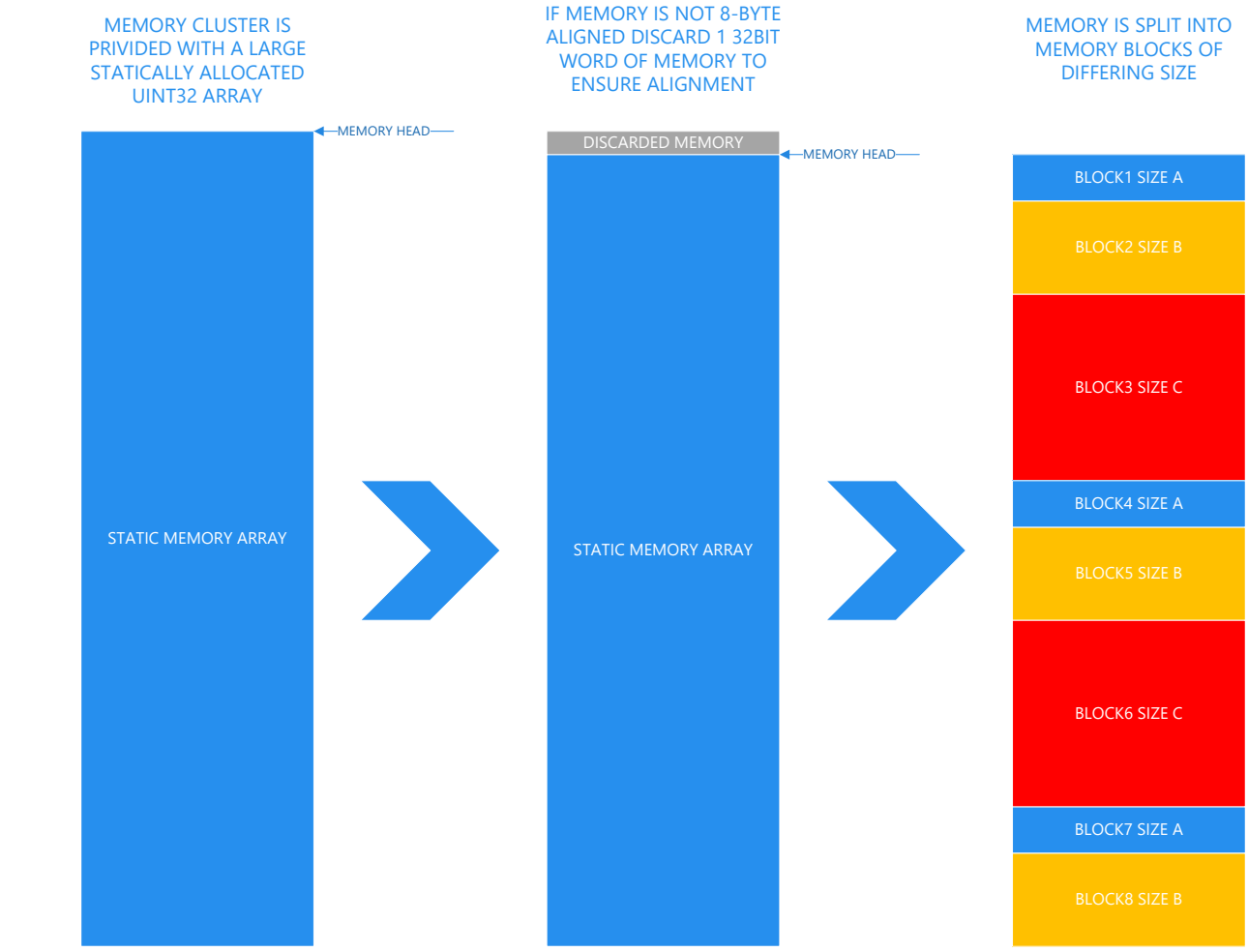
2 The Memory Cluster

To allow the user and the operating system to efficiently allocate memory on the fly I have implemented a "memory cluster". This cluster consists of multiple independent memory pools which store a number of memory blocks of a certain size. The advantage of this system is that it is more flexible/efficient in terms of the memory it allocates. If a task requests a very small memory block a normal memory pool would only be able to allocate it the standard full sized block, wasting a lot of memory. The memory pools block size could be decreased to avoid this problem, but then it is no longer able to satisfy requests for large memory blocks. The memory cluster in comparison has multiple different sized blocks and can satisfy large and small requests efficiently by allocating the smallest memory block that fits the request. It also has an advantage in terms of wait time, if task a is currently obtaining a block from pool A, and pool A is therefore locked, task 2 can still obtain memory from the cluster even if it requests size A. The cluster will realise that pool A is busy and simply provide task 2 with a larger sized block B. This does waste some resources but the trade-off is that task 2 is not blocked from obtaining its requested memory.

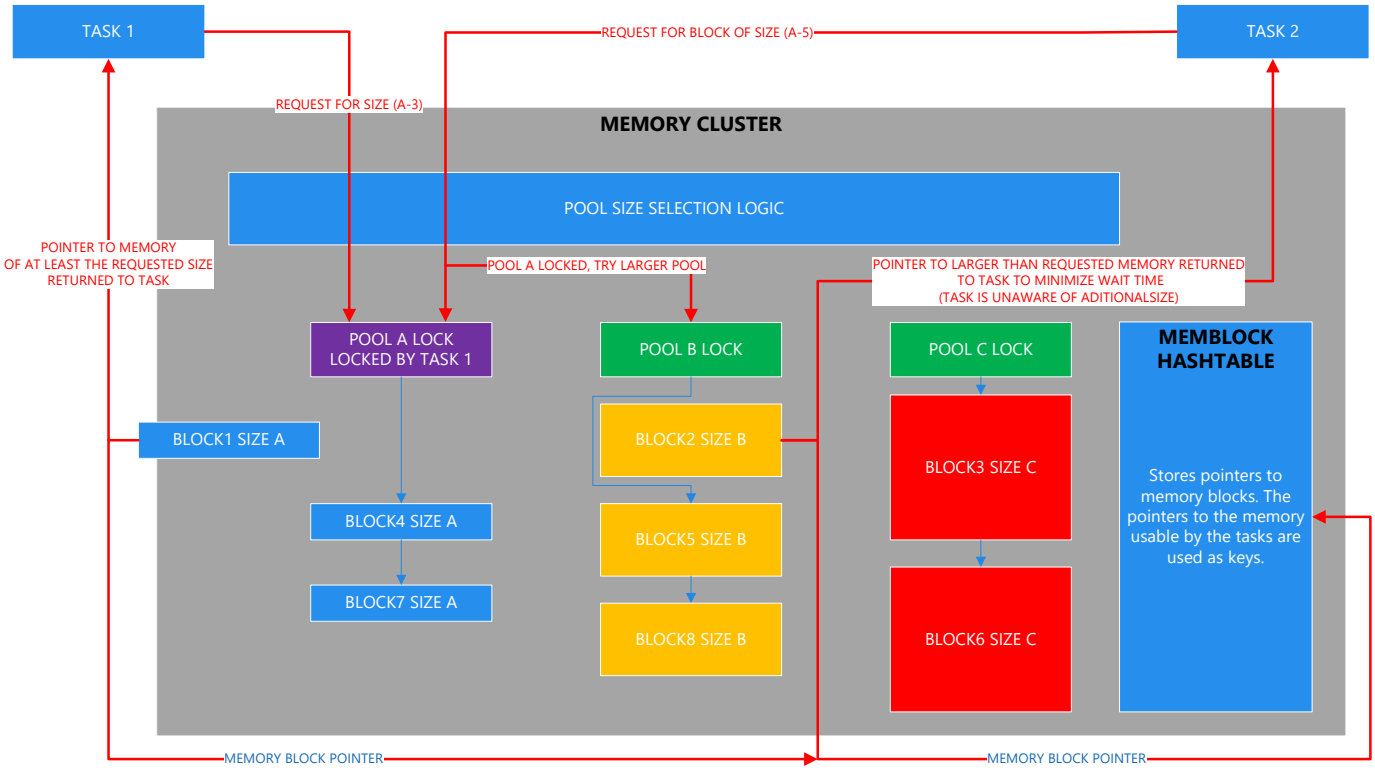
2.1 Cluster Structure

The memory cluster consists of one mutex and one linked list per memory pool. The cluster also contains a hash table and a mutex to lock said hash table. The hash table is used to keep track of the blocks that have been allocated and their size. The use of the hash table is twofold. 1) It can be used to prevent a user from accidentally deallocating memory that the memory cluster did not allocate by comparing the the provided pointer to all pointers belonging to blocks known to the memory cluster (this operation is very quick, time complexity of O(1) on average due to use of the hash table). 2) by using the memory pointer to store a pointer to the entire memory block (memory + memory block struct that contains information such as block size) the memory pointer provided to the deallocate function can be associated with the correct block, which can then be placed back into the correct memory pool.

2.2 Cluster Operation Overview



EACH BLOCK SIZE HAS THEIR OWN MEMORY POOL WITH AN INDEPENDENT LOCK SO THAT MEMORY OF DIFFERENT SIZES CAN BE ALLOCATED SIMULTANEOUSLY AND TIME SPENT WAITING FOR MEMORY IS MINIMIZED.



3 The Channel Manager

3.1 Design Overview

3.2 The Channel

3.3 Initialisation Process

4 Data Structures

4.1 Mutex

4.2 Semaphore

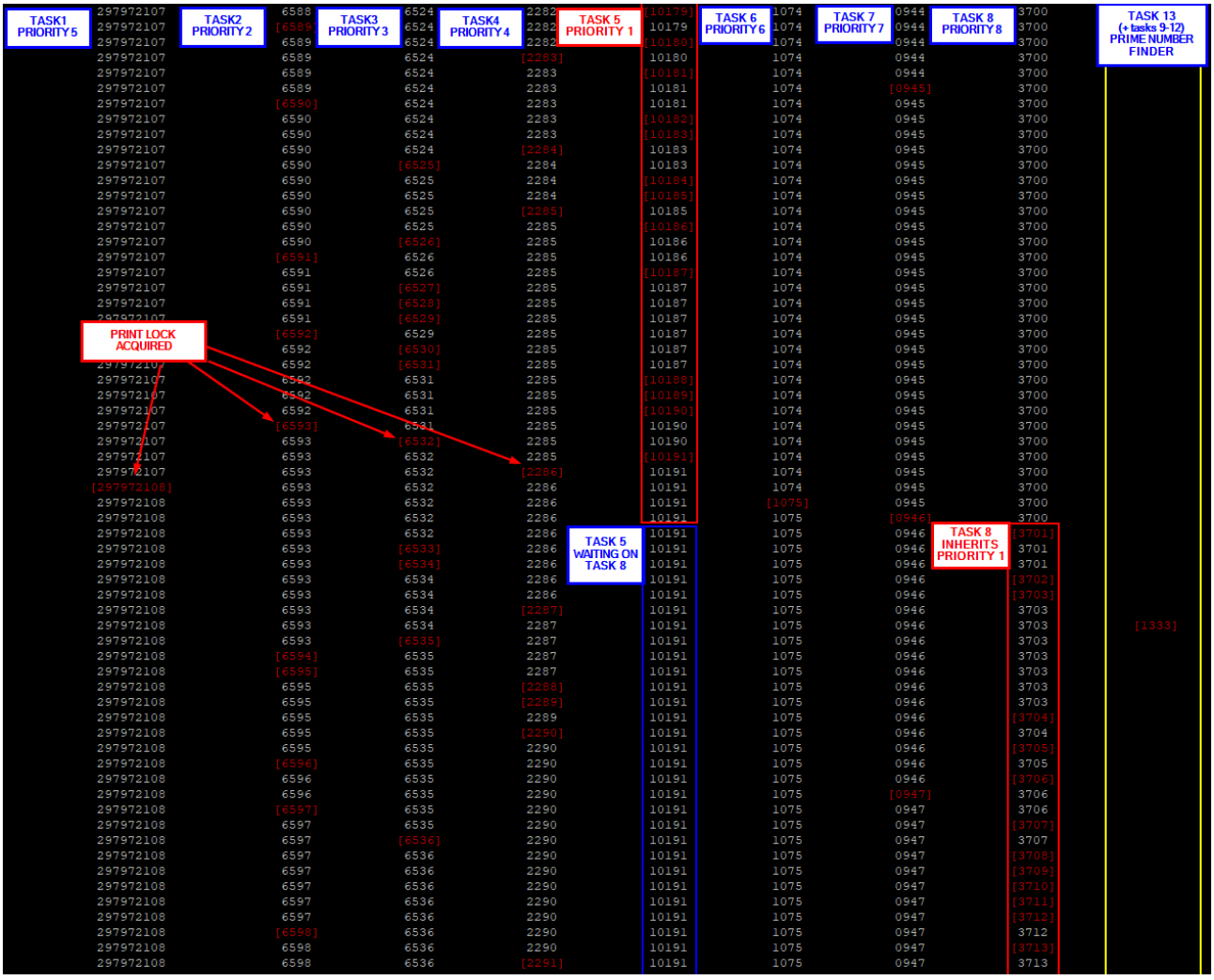
4.3 Queue

4.4 Hashtable

4.5 Heap

5 Demonstration Code Overview

The demonstration code contains 14 tasks that together highlight the various features of the modified OS. The demonstration begins with the creation and initialisation of a single task (task0). All task resources are allocated using the memory cluster. This is possible because the memory cluster guarantees that the memory blocks it returns are 8-byte aligned. The OS is then started and task 0 begins its execution. It allocates, initialises and adds 9 further tasks to the scheduler from within the running task itself. After this task0 exits and the resources associated with it (TCB and stack) are deallocated by the OS. Below this paragraph you can see an annotated screen shot of the demo output. The screenshot shows the state of the counter of each of the tasks. The bracketed red numbers indicate which task holds the print Lock and executed the printf.



Tasks 1 through 4 and tasks 6 and 7 demonstrate the operation of the mutex datatype. Task 1 is an exception because rather than wait whilst another mutex owns the lock it uses the OS_mutex_acquire_non_blocking function which enables it to skip the printf section if it is not successful in obtaining the lock because another tasks holds it. This allows it to continue increasing its counter when other tasks would have to wait. By inspecting the count that tasks 2,3,4,6 and 7 have obtained in the screen-shot and comparing this count to their priorities it can be seen that cpu time allocated to the task is dependant on the task's priority. The screen-shot also highlights that the scheduler does not only allocate time to the highest priority task and thereby prevents task starvation of lower priority tasks (see how task 4 with priority 4 still

occasional increases its count despite task 5 with priority 1 also being active). Tasks 6 and 7 showcase the operating system's sleep functionality after printing to the serial bus by going to sleep for a random interval between 0 and 100 SysTicks.

Task 5 and 8 demonstrate the schedulers priority inheritance functionality. Priority inheritance is a feature that prevents a condition known as priority inversion where a lower priority task can potentially starve a higher priority task. This is avoided by giving the lower priority task the priority of the higher priority task for the duration that the lower priority task holds the lock to a shared resource that both tasks use.

Task 13 (spawned from task0) is an implementation of the Sieve of Eratosthenes prime number finding algorithm which I use here to demonstrate the inter task communication abilities of the OS. The OS allows task to communicate using channels. A task can obtain a channel from the OS channel manager by stating a channel ID (the user can also create and use channels not managed by the channel manager if they so desire). Two tasks that request the same ID both get given the same channel which they can now use to pass data back and forth. The use of the channel manager lets tasks create and use channels dynamically rather than the user having to create all channels beforehand (the channel manager uses the OS memory cluster for channel allocation). Task13 allocates, initialises and adds task 9-12 to the scheduler, it also creates 4 distinct channels. Tasks 9-12 connect to one channel each. task 9 has a counter starting at 2, all numbers where (num % 2 == 0) get written to the channel it shares with task 13. tasks 10, 11 and 12 do the same with 3, 5 and 7 respectively. task13 reads from the 4 channels and compares the numbers. If all 4 numbers match then it shows that that number is a prime and task 13 prints it to the serial bus.

6 Conclusion

I have extended DocetOS with a range of features that allow the user to implement complex tasks with relative ease, abstracting away/hiding lower level operations such as memory allocation and channel setup from the user by providing convenience functions (e.g OS_channel_connect) that handle these procedures. I think that the OS components I have added are well suited for running a larger number of tasks efficiently and allow the system to be scaled to up significantly (provided that certain parameters are tweaked, e.g memcluster block sizes and maximum task runtime before scheduler forces a task switch).