UNIVERSITY of York

Systems programming for ARM
extending DocetOS

Y1471938

January 14, 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 task sleeping 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.

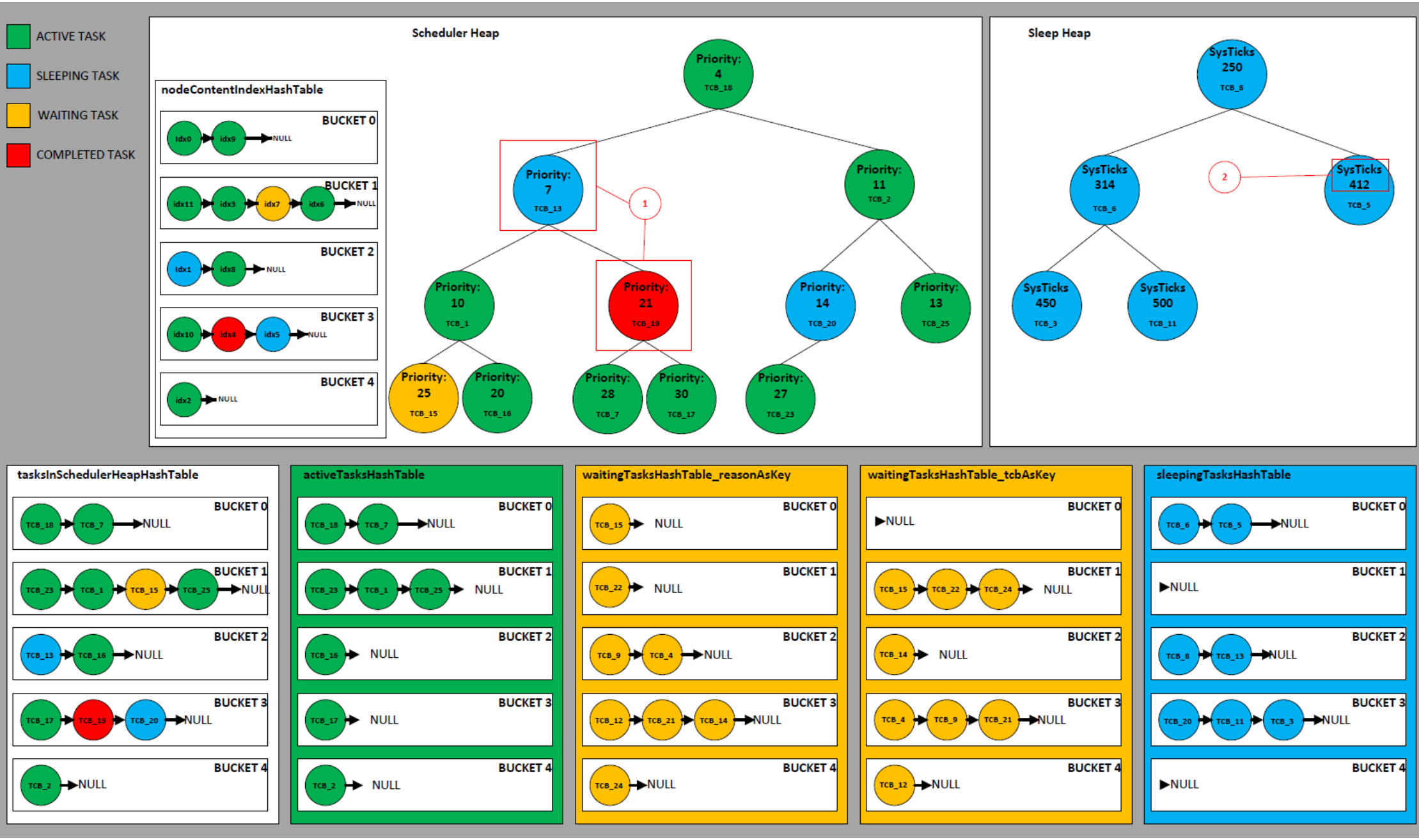
- The **tasksInSchedulerHeapHashTable** keeps track of all the tasks currently present in the scheduler heap. Used to check what tasks have to be added back to the scheduler heap after wake/notify and which ones were never removed.

- 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.

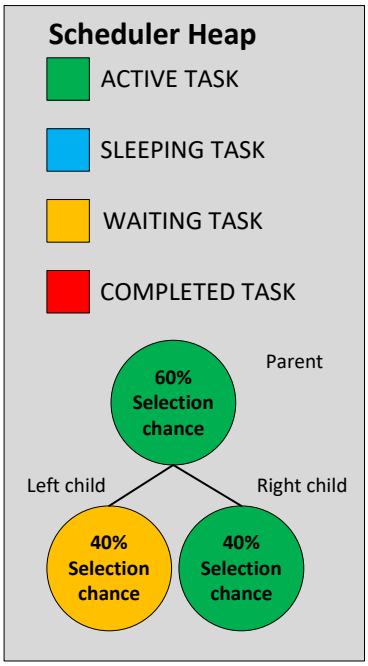

ACTIVE TASK
SLEEPING TASK
WAITING TASK
COMPLETED TASK
Scheduler Heap
nodeContentIndexHashTable
BUCKET 0
idx0
idx9
NULL
BUCKET 1
idx11
idx3
idx7
idx6
NULL
BUCKET 2
idx1
idx8
NULL
BUCKET 3
idx10
idx4
idx5
NULL
BUCKET 4
idx2
NULL
Priority: 4 TCB_18
Priority: 7 TCB_13
Priority: 11 TCB_2
Priority: 10 TCB_1
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

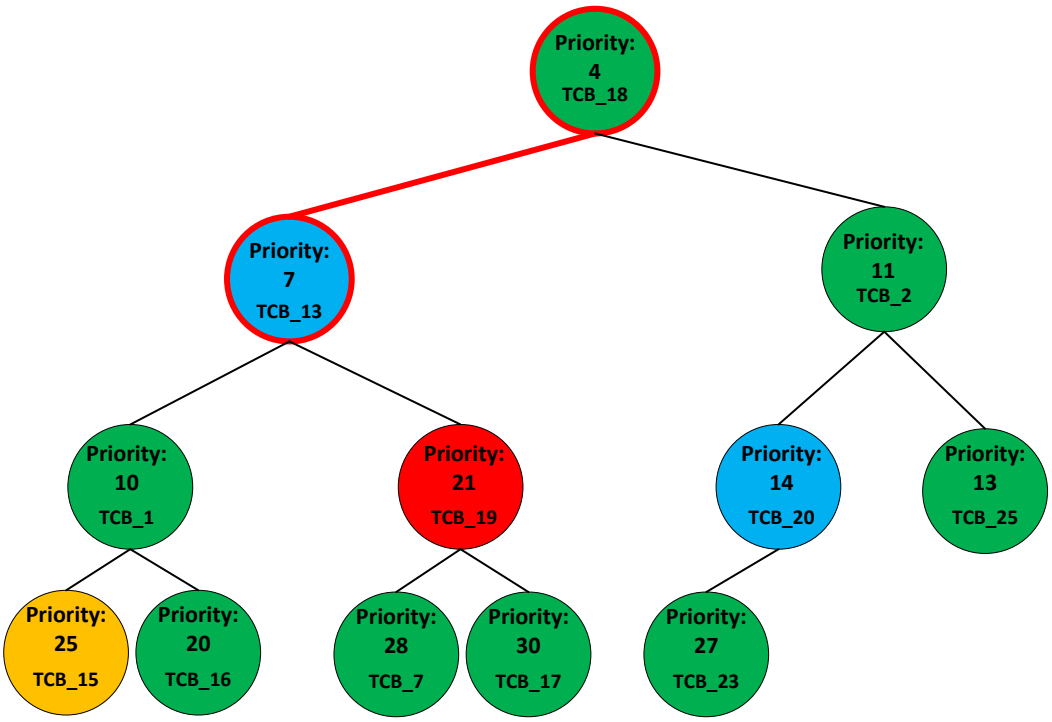
1.2 Selecting a Task

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.

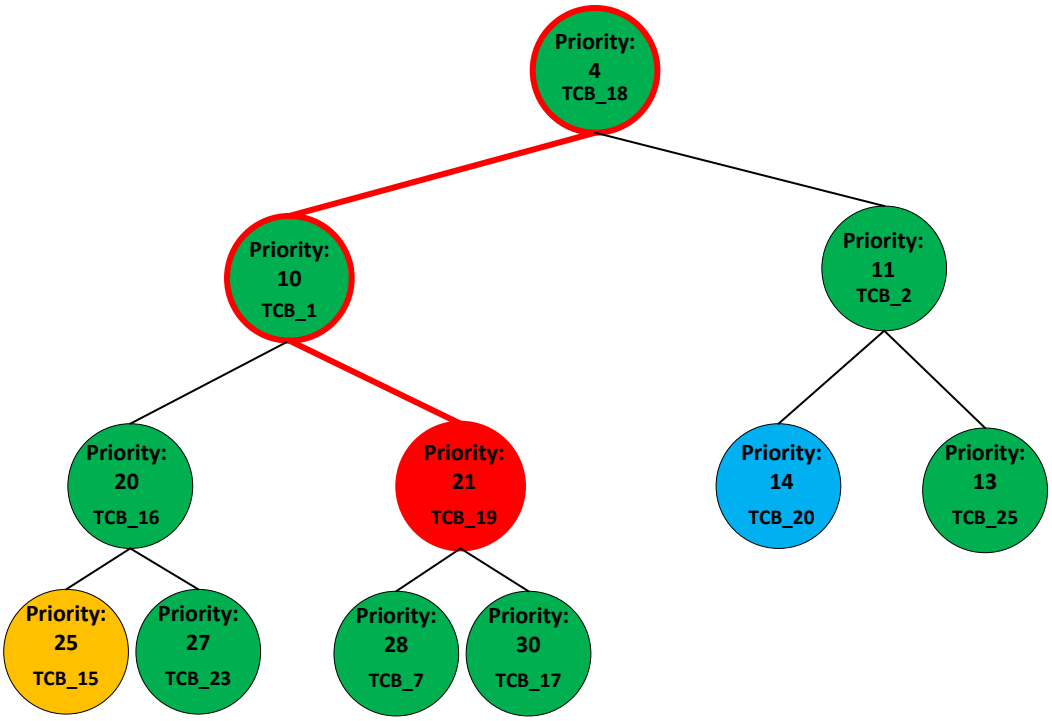
The reasoning behind using a heap and an element of randomness for task selection (with the chance of a task getting selected being correlated with the tasks priority) is that it avoids task starvation by giving every task a non-zero probability of getting selected, no matter how low their priority. 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

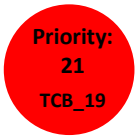




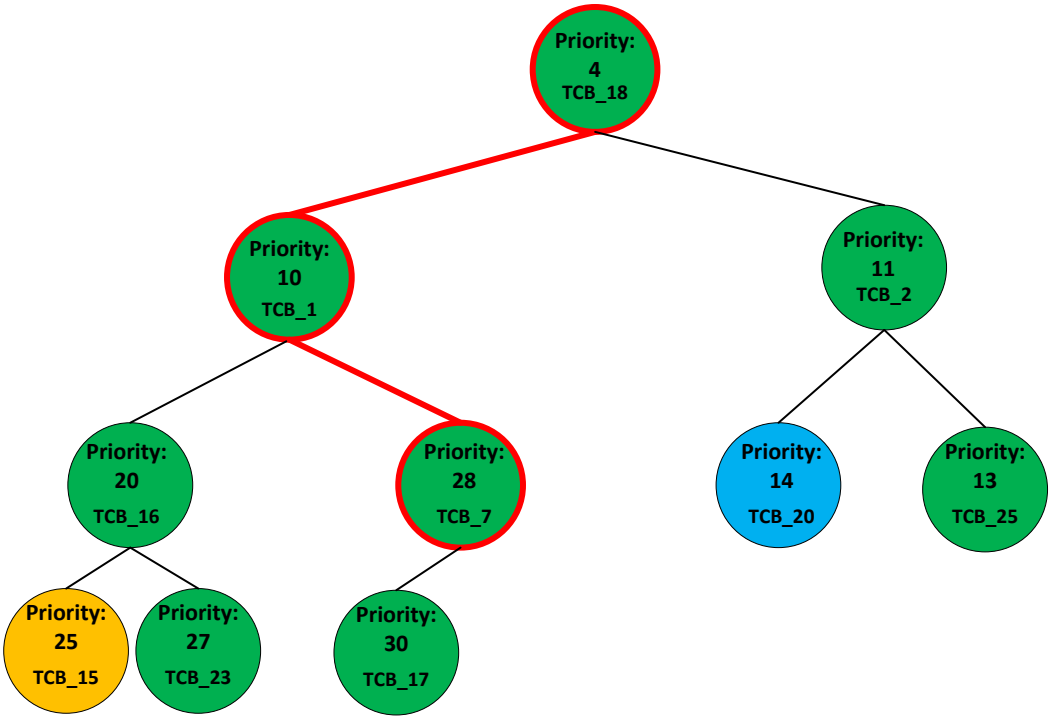
Removed sleeping task and restored heap property



Removed completed task and restored heap property



Selected task for execution



Scheduler Task Selection