Expertise
and insight
for the future

Pavel Dounaev

Desgin and Implementation of Co-operative Task Scheduler

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Innovation Project

16 December 2019

Author(s) Title Number of Pages Date	Pavel Dounaev Desgin and Implementation of Cooperative Task Scheduler 31 pages + 0 appendices 16 December 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Smart Systems
Instructor(s)	Antti Piironen, Principal Lecturer

This report documents the design and implementation of cooperative task scheduler for ARM Cortex-M3 processor core. The report focuses on practical implementation of task scheduler in C programming language that is based on learned theory about operating systems.

The minimal requirements were defined and achieved for this project. End result is working minimal task scheduler. The task scheduler can be used to create tasks that need to be scheduled cooperatively. Additional synchronization between tasks can be provided with use of semaphore.

The task scheduler is open-sourced and current plans for the project is to develop it into a full fledged real-time operating system . Full source code of the task scheduler is available from GitHub repository.

Keywords	operating system, task scheduler, task, semaphore, ARM Cortex-M3

Tekijä(t) Otsikko Sivumäärä Aika	Pavel Dounaev Desgin and Implementation of Cooperative Task Scheduler 31 sivua + 0 liitettä 16.12.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto–ja viestintätekniikka
Ammatillinen pääaine	Smart Systems
Ohjaaja(t)	Yliopettaja Antti Piironen

Tämä on tiivistelmän ensimmäinen kappale. Tiivistelmän kappaleet loppuvat komentoon newline, jotta saadaan yksi tyhjä rivi aikaiseksi.

Tämä on tiivistlemän toinen kappale.

Avainsanat	avainsanat

Contents

List of Abbreviations

CPU	Central Processing Unit.
FIFO	First In First Out.
HDD	Hard Disk Drive.
I/O	Input/Output.
PCB	Process Control Block.
RAM	Random Access Memory.
TCB	Task Control Block.

1 Introduction

The purpose of this report is to explore theory and implementation of minimal cooperative task scheduler. The minimum requirements defined for the task scheduler is the ability to create tasks, schedule them cooperatively and provide task synchronization by utilizing semaphore. The task scheduler is designed and implemented for micro-controllers using ARM Cortex-M3 processor core. The report includes an introduction to all components of an operating system required to understand the implementation and operation of this task scheduler. The task scheduler is created using C programming language, thus the report assumes that the reader is familiar with it.

The goal of the project is not only to implement the task scheduler itself, but also research and understand the fundamentals of operating system design that will be needed in the future for implementing preemptive task scheduler or full fledged real-time operating system. Understanding fundamentals will also help to understand other real-time operating systems that could be used as a reference.

This report is intended for anyone interested in operating systems, schedulers or for who is trying to build own task scheduler or operating system and needs a reference of such software. This task scheduler can also be used for educational purposes to explain one of the main functions of any operation system in introductory level and not delving too deep into other complexities of operating systems.

2 Theoretical background

2.1 Task

2.1.1 Task as a unit of work

Modern operating system is considered a multitasking system since it can perform actions such as concurrent execution of multiple tasks. [1] In operating system theory a term *task* or *process* is often described as a instance of a program in execution on a computer system. Thus a task is a unit of work for most multitasking operating systems. As a unit of work, the main function is accomplishing a certain task using computer system's resources such as Central Processing Unit (CPU) time and memory. [2]

However, the program itself is not a task, it is a passive entity residing somewhere in computer system's non-volatile memory. One example of the non-volatile memory is a Hard Disk Drive (HDD) or flash memory. In non-volatile memory, program is usually stored as a file containing program code for CPU to execute. Such files are often called executable files or binary files. Such executable file becomes an active entity, that is a task once it is brought from non-volatile memory to a volatile memory. In most computer systems that volatile memory is often Random Access Memory (RAM). In brief, program's current location on computer system is often what differentiates a task from a program. [3]

2.1.2 Task Structure

Task is divided into four sections: text, data, stack and heap. All of those sections are placed in a computer system's RAM when task is executed. By accessing those sections, CPU is able to interact with the task. Task's text section contains a program code that CPU will execute. One of the CPU's special registers is used to access that section, that register is called program counter or instruction pointer depending on processor's

architecture. Task's stack section is used to store task's temporary data throughout its life time. Such temporary data include for instance local variables, function calls and return addresses. Each time temporary data is added, it is placed on the top of the stack, and thus stack also grows. Depending on processor architecture, stack can grow upwards or downwards. To keep track of stack CPU contains a special register called stack pointer that always points to top item placed on the stack. Since stack holds only task's temporary data that is not always present, the data section is used for global data. Global data is not temporary and is available throughout task's life time. Heap section is used for dynamic data that is allocated during the task's run time. [3]

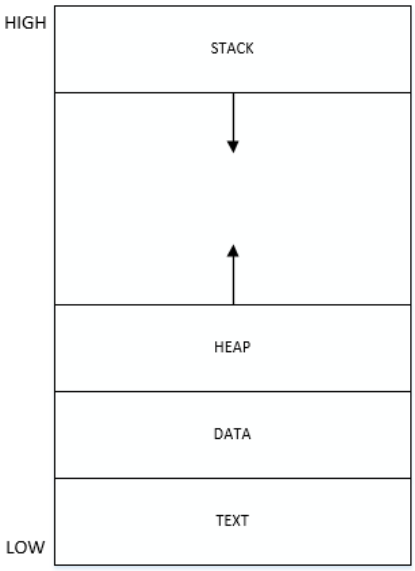

Figure 1: A representation of a task's sections in memory.

2.1.3 Task states

Each task in a operating system must somehow indicate it's current activity. Operating system must now if a task is waiting for some resource, waiting for Input/Output (I/O) operation or maybe it is done executing and is ready for operating system to terminated it. Without it a task could be wasting CPU's precious cycles by doing nothing and thus CPU utilization will also decrease. To solve that problem task states are introduced. Each task state defines task's current activity. As shown in the Figure 2, some of the defined states for a task might be the following. [4]

- Running. Task is in Running state when it is currently executing on CPU. On a uni-

processor computer systems only one task can be executed on CPU and thus only one task can be in Running state.

- Ready. Task is in Ready state when it is ready to be executed but must wait to be scheduled. That might happen because other task is currently executing on a CPU.
- Blocked. Task is in Blocked state when it is waiting for some event to happen. That event might for instance be a resource becoming available. When corresponding event happens task's state waiting for that event will change to ready. When task will be scheduled to execute(task will be in Ready state) it will have access to that resource.
- Terminated. Task is terminated when it finished executing it's program code and no longer uses computer system's resources. Thus no longer is located in RAM.

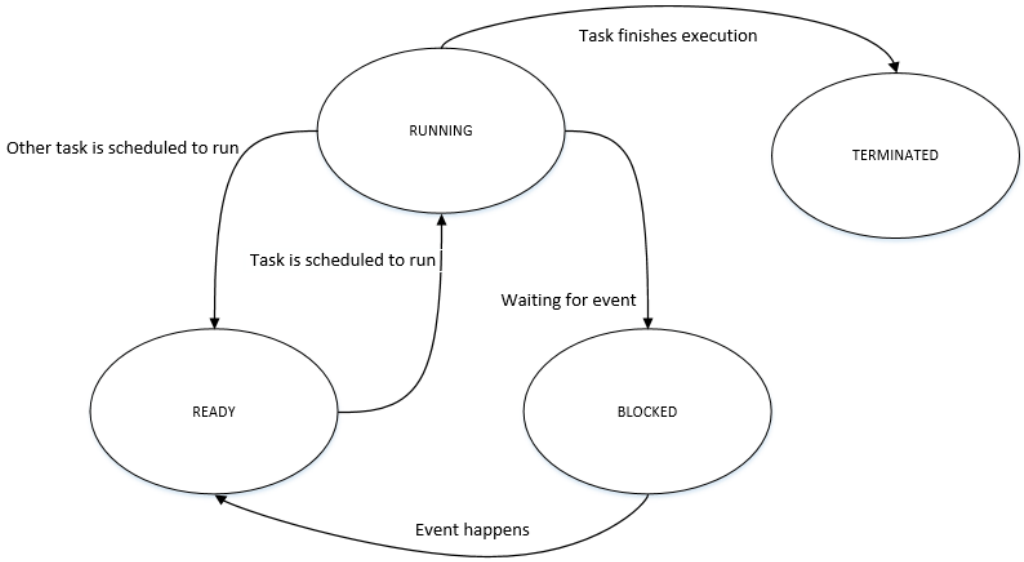

Figure 2: Task state transition diagram with four task states

2.1.4 Task Control Block

Operating systems implement a Task Control Block (TCB) or sometimes called Process Control Block (PCB) to represent a task. Each TCB for each task is created and maintained by the operating system. TCB contains vital information for operating system to distinguish it from other executing tasks. Such information include:

- Task Identifier. A unique identifier usually an integer for each task.
- Task State. Tasks current state. As discussed previously state naming conventions may vary across operating systems.
- Task Priority. Task's scheduling priority.
- Program Counter. Holds the address of the next instruction to execute for this task.
- CPU registers.

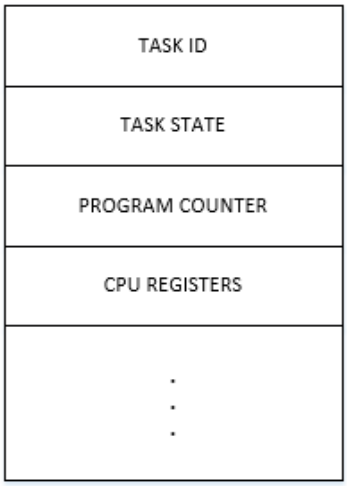

Figure 3: Example of simple Task Control Block

TCB serves as a container for that information. There is no standard of what information TCB should include, so each implementation of TCB may vary across operating systems. [5]

2.2 Task Scheduler

Task Scheduler is the main component of any modern operating system. Scheduler is responsible of operating systems primary function, which is the scheduling of tasks. By doing so, it maintains the main objective of multitasking operating system. Because of this, the scheduler can be considered as a heart of the operating system, and it is why it is often the first part of operating system that is implemented. [6]

2.2.1 Task Queues

Because main function of the task scheduler is scheduling tasks, the scheduler must know from where to choose tasks to schedule. To accommodate that, tasks are queued up into a queue. This task queue is often implemented as a First In First Out (FIFO) linked list. The records in those queues are often references to task's TCB. So each node in that linked list is a TCB which also points to a next task's TCB. Queue structure also maintains two pointers. One pointer points to the front of the queue and other to the tail of the queue. Structure of this task queue is shown in Figure 4. When task is added to a queue it is pushed to the back of the queue. When scheduler picks a task to schedule, it will choose first task from the front of the queue. [7].

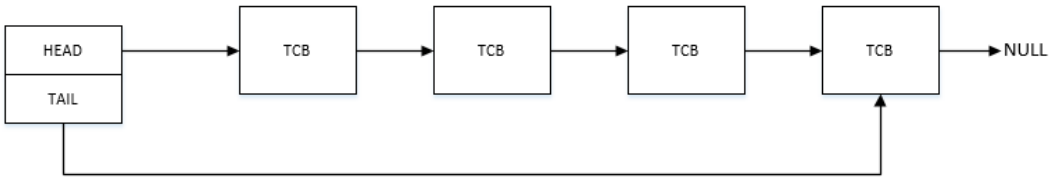

Figure 4: Illustration of task queue

Operating system implements various task queues for scheduler to use. There are different queues for different purposes that vary across operating systems, but the most operating systems implement at least two queues: Ready and Waiting. When task is scheduled to execute it will be added to a Ready queue. That Ready queue consists of tasks that are ready to be executed. Thus in Ready state. When task is waiting for some computer resource to become available it is put into Waiting queue. As the name suggest, that queue contains a tasks that are waiting for some event to happen and so are in Waiting state. Scheduler always chooses a task from the Ready queue and so task in Waiting queue first must be transitioned from Waiting queue to Ready queue to be scheduled. [7].

2.2.2 Context Switch

Context switch is an action performed by operating system, in which CPU is switched from one task to another. Context switch is managed often by a dispatcher, that is a part of task scheduler. Context switching is considered a crucial feature of multitasking operating systems. By using context switching at rapid phase, the operating system creates an illusion of concurrent task execution on a single CPU. Each task has context that is stored in task's TCB. CPU's registers and program counter at any given time is what defines a context for a task. Task's context can be thought of as a snapshot of a CPU at any given time. The names and amount of registers for the context varies across processor architectures. [8]

Context switch consists of two sequences: storing task's context and restoring another task's context. When context switch occurs task's current context is stored by pushing all CPU's current values of registers to a task's stack. By doing so task's state of execution can be stored into TCB and thus task will be able to continue where it left off next time when it is it's time to execute. After that another task's TCB is retrieved and context is restored by popping all register values from task's stack to the CPU's registers. This restoring action will give means to continue task to execute from the same place it was before context switch. After theses sequences of actions that is context switch a new restored task's context would be executing on a CPU. [9]

Context switching however requires a substantial amount of processor's time and thus is considered one of the most expensive action(in terms of processor time) in operating systems. [8] That is because of during a context switch processor does not do anything else. When designing a operating system extra care must be taken when deciding where and when context switch should occur.

2.2.3 Preemtive and Cooperative Scheduler

Most task schedulers can be characterized into two types: cooperative and preemptive. In cooperative scheduling CPU is kept by the task until it voluntarily releases it. The CPU is released for instance, by task terminating or yielding it's execution trough a

special function call. [10] This approach performs less context switches, which utilizes the CPU more. This approach is also platform independent since it does not need special hardware such as system clock timer that is often used in preemptive scheduling on real-time operating systems. However this approach increases responsibility for a programmer to maintain systems proper defined behaviour. This is because now the programmer is responsible for task to release the CPU at correct times. In the worst case scenario other task would never be able to execute, because the current task would never release the CPU.

In preemptive scheduling the context switch is interrupt driven. That interrupt is often the system clock timer interrupt or other special hardware that would take care of that. However, in preemptive scheduling there is a possibility of race conditions. Typical race condition occurs when two tasks share the same resource. When one task is writing to that resource it might be preempted unexpectedly leaving the resource unfinished, and so when the next task tries to read that resource, the resource would be in inconsistent state. In the worst case that would lead to unexpected behaviour of the whole computer system. In order to prevent that additional synchronization means should be added to operating system.

2.2.4 Semaphore

One of the synchronization primitives that operating systems provide are semaphores. A semaphore can be described as a integer that works as a counter that is modified using two operations P and V. Originally P was denoted for a Dutch word *proberen* that meant “to test” and letter V derived from Dutch word *verhogen* that meant “to increment”. Names for the these operations varies across operating systems [11].

To guarantee correct function of the semaphore operations that modify the semaphore need to be executed indivisibly [11]. This means that no other task shall modify given semaphore when it currently is modified by other task. In preemptive scheduling this also means that semaphore being modified should be modified without interruption, also known as atomically. This is common in preemptive scheduling where interrupt can happen asynchronously at any moment of execution.

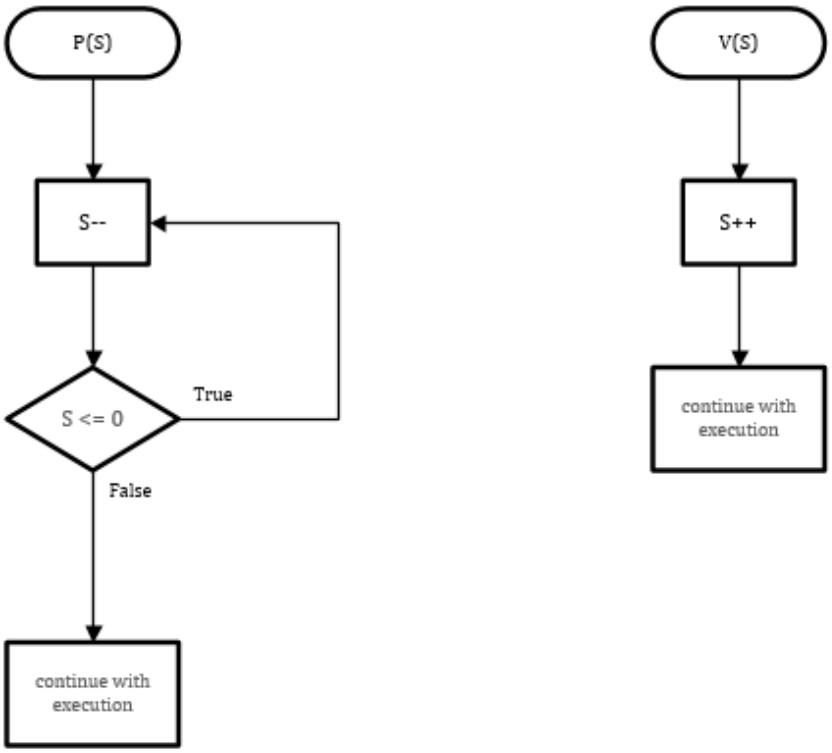

Figure 5: Semaphore P and V operations, where semaphore is denoted as S

Semaphore can be used to control access to a certain resource or to synchronize execution of tasks. To control access to a given resource a **counting** semaphore is used. Initially the counting semaphore is initialized to the amount of given resource is available. Whenever task needs to access given resource the semaphore is decremented using operation P. As seen in the figure 5 the P operation first decrements the semaphore and after that it checks or tests the value. When task is done accessing the resource, it gives the resource back by incrementing the semaphore with operation V [11]. As name suggests the counting semaphore counts how many instances are available. Whenever there are no resources left of given resource the semaphore for that resource the semaphore will be equal or less than zero. Subsequently when there is a resource available the semaphore will be equal to a positive value.

Other type of semaphore called **binary** semaphore is can be used for task synchronization. As the name suggests, the binary semaphore can have only two values 0 or 1. Thus it can be also described as counting semaphore which is initialized with

value of 1. The binary semaphore can be used in for task synchronization for example in scenario where one task needs to wait until other task has performed some action (for example execute some block of code that prepares data for the other task) before it can be executed.

3 Implementation

The scheduler is thread based, this means that each task is actually a thread and not a separate program as it would be for general purpose operating systems. In most real-time operating systems each task is implemented as a thread. The scheduler itself is the main program and each task is just a thread of execution in that program. Thus the scheduler actually schedules its own threads rather than separate programs. Thus, each task is implemented as standard function.

3.1 Implementation of Task

ARM Cortex-M3 processor implements a full descending stack, meaning that processor's stack pointer will always point to the memory address of the last item placed on the stack. The stack grows downwards so from the higher memory address to the lower memory address. When an item is pushed onto the stack, the stack pointer is first decremented and then the item is written to the memory location the stack pointer points to. [12] That way the stack pointer will always point to a valid memory address and thus also to a valid data.

As discussed in Chapter 2.1.2, by definition each task needs it's own stack. Task's stack was implemented as a 32-bit unsigned integer array. By using array as the stack allows task to implicitly define it's address range for own stack. That is because compiler takes care of correct addressing for each variable in the program and thus also for the array. This allows the program to access the stack using an implicit memory address such as the index of the array instead of explicit raw address. Since Cortex-M3 defines stack's width as 32 bits or 4 bytes, the array is also defined as 32 bits wide. Because the array is represented as contiguous memory block in the system's memory, it will behave much like the stack. By being contiguous memory block no random data or undefined data will appear in the middle of the array.

Each stack for each task is allocated at compile time and is provided by the user. The array is allocated from low to high memory address. This means that array will start

from the lower memory address and will end at the higher memory address. This also means that the stack will end at the same memory address as where the array starts. So the initial memory address the stack pointer will point to is the same address as the array's last index memory address. The relationship between the array and the stack pointer is illustrated in the Figure 6. Figure illustrates basic relationship between the array that is used as a stack and the processor's stack pointer. The array is denoted as character a in the figure. As seen in that figure, the compiler has allocated 24 bytes (each slot in the array is 32 bits wide, so 6*4 bytes = 24 bytes) of memory for the stack. Now the first and the last indices of the array **a** define the address range for the stack. Because the stack and the array grow in opposite directions from each other, the stack starts at the same memory address as where the array **a** ends (last index a[5]).

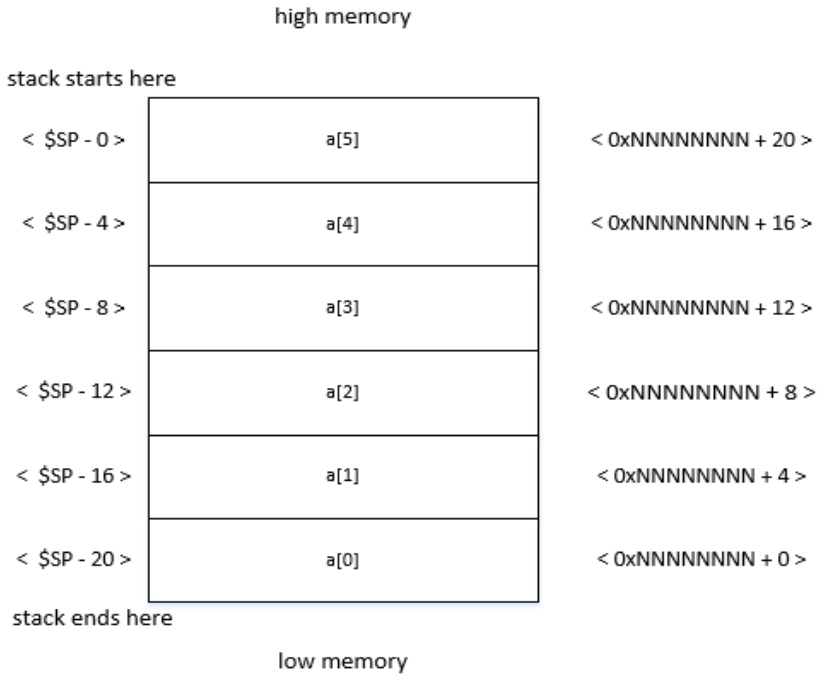

Figure 6: Relationship between stack pointer and array when using array as a task's stack

Since each task is just a thread in one program that is called task scheduler, the global data is shared across the whole program. This means that each task can access that global data. However the scheduler does not provide memory protection between tasks. That means that each task can accidentally overwrite other tasks contents. For instance that might happen when task leaves it's scope. Thus, extra care must taken by the user so that the task does not leave it's own scope.

3.1.1 Stack operations

Cortex-M3 provides standard `push` and `pop` assembly instructions for accessing and manipulating the stack. These instructions are used mostly in context switching which is discussed in Chapter 3.2.4. Using these instructions processor can store to and restore values from the stack. These instructions are illustrated in Figure 7 and Figure 8.

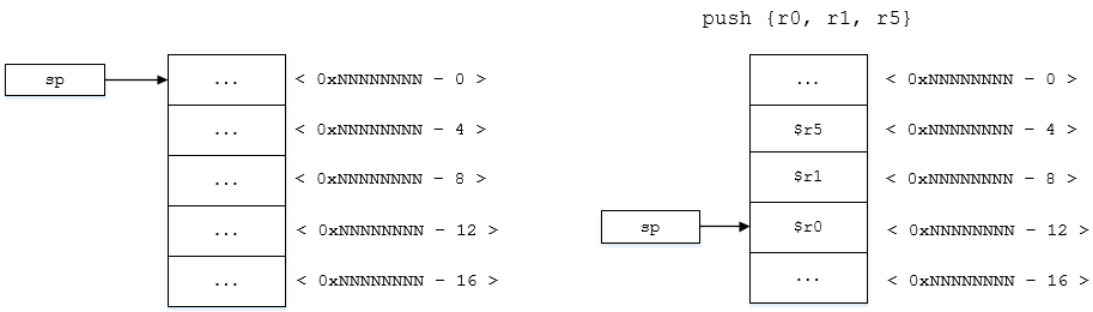

Figure 7: Stack's manipulation using push assembly instruction

When `push` instruction is executed the processor will store specified registers on the stack. The stack pointer is first decremented after which the value contained in the register will be written to the memory location specified by the stack pointer. Processor arranges registers so the lowest numbered register will be stored in the lowest memory address and highest numbered register will be stored in the highest memory address. When the instruction is complete, the stack pointer will be updated to hold the memory address of the last value stored on the stack. As shown in the Figure 7, registers `r0`, `r1` and `r5` are stored by using `push {r0, r1, r5}` instruction. After this the stack pointer will point to the same memory address where the value of the register `r0` was stored. When values stored in the stack need to be retrieved, `pop` instruction is used. The pop instruction loads the specified registers with the values from the stack. Unlike the push instruction the value from the stack is first loaded to a register after which the stack pointer is incremented. This means that in the end of the instruction the stack pointer will contain the memory address of value that is not guaranteed to be valid. In Figure 8 `pop {r1, r2}` instruction loads `r1` and `r2` registers from the stack. In the end of the instruction the stack pointer points to

the memory address that is after the value that was loaded into register `r2`. In this case the memory address is the address in which value of register `r8` was stored. As with the push instruction after completion the stack pointer will hold the updated memory address.

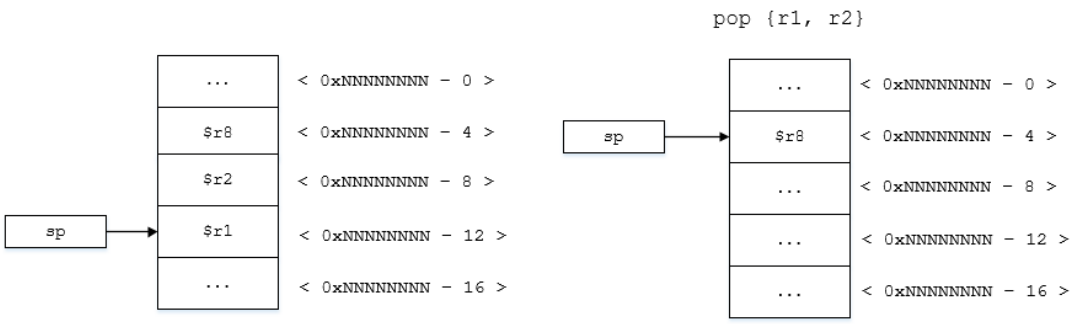

Figure 8: Stack's manipulation using pop assembly instruction

3.1.2 Implementation of Task Control Block

To represent each task in the computer system TCB is defined. TCB is implemented as a struct data type. The structure of task control block as struct is shown below in Listing 1.

```
1       typedef struct tcb{
2
3           uint32_t        *sp;
4           task_state    state;
5           uint8_t        prio;
6           struct tcb    *next;
7       }tcb;
```

Listing 1: TCB implemented as struct

As seen above, the TCB consists of four data entries. First entry in the TCB is task's stack pointer denoted as `uint32_t *sp`. TCB's stack pointer points to the memory address of item placed on top of the task's stack. However TCB's stack pointer points to valid top of the task's stack only during context switching. TCB's stack pointer can be thought of as a task's temporary stack pointer that is only valid and used during context switch. That stack pointer is used only for locating, storing and restoring task's context as seen in Chapter 3.2.4. This means that task's stack pointer read from the TCB will point to the same memory address that processor's stack pointer pointed to right after the

context switch. Thus during execution of task's normal code task's stack pointer should not be accessed trough task's TCB.

Stack pointer is defined as pointer to a unsigned 32-bit integer. It holds a 32-bit memory address because Cortex-M3 defines each memory address as 32 bits wide, meaning that the highest possible memory address is `0xFFFFFFFF` and the lowest possible memory address is `0x00000000`. Each of those addresses are 32-bit unsigned values. The stack pointer also points to a 32-bit unsigned integer value, because the stack may contain memory address, which is by definition unsigned 32-bit value. This is needed in order to read and store the correct memory address for instance the memory address of a function as seen in Chapter 3.2.1

The second entry denoted as `task_state state` is task's current state. Task can be only in one of the three defined sates: Running, Ready and Blocked. The data type `task_state` is implemented as enum as seen in Listing 2.

```
1       typedef enum{
2
3           TASK_RUNNING,
4           TASK_READY,
5           TASK_BLOCKED
6       }task_state;
```

Listing 2: Implementation of task states by using enum data type

Since task states are implemented as enum, each state is just an integer. This means that when task is running it's task state will read as 0, when task is in Ready sate the state will read as 1 and when task is in Blocked state state will read as 2. However as seen in Listing 3 instead of enum as task's state, a simple macro expansions can be used to achieve same result.

```
1       #define TASK_RUNNING    0
2       #define TASK_READY      1
3       #define TASK_BLOCKED    2
```

Listing 3: Alternative way to implement task states

However using enum is more flexible, because whenever a new state needs to be added it can be directly added to a enum structure instead of creating a new macro definition for new task state. Also the enum will number new task state with non-used integer automat-

ically. That makes code less error prone since there would not be task states with same numbers.

Third entry is task's scheduling priority `uint8_t prio`. Scheduling priorities allow user to decide which tasks are more important than others and enable them to be scheduled more frequently. Task's priority is configured by the user at compile time. The highest priority is given for the task that needs to be executed as frequently as possible. Scheduling priority is implemented as unsigned 8-bit integer variable. The lowest possible priority is priority of value 0 and the highest possible priority of value 7. This means that task can be configured with eight different priorities. However the lowest priority, priority 0 is reserved for special task called Idle task. The amount of priority levels can be configured through configuration file. The Last entry in the TCB is a pointer to the next TCB, meaning it holds the address of the next TCB structure. This is used for creating and maintaining a task queue for scheduling purposes.

3.2 Cooperative Scheduler

3.2.1 Task Creation

In order for a task to be scheduled it first must be created and initialized. A new task is created with `task_create` function. This function creates the context for the new task and initializes all of it's components that are required for the task to be scheduled by the scheduler and executed by the CPU.

```
1  void task_create(void (*task_function)(void),
2                   tcb *tcb,
3                   uint32_t *stack,
4                   uint32_t stack_size,
5                   uint8_t priority)
6  {
7      /* create stack frame as it would be created by context switch*/
8
9      tcb->sp = &stack[ stack_size - (uint32_t)1 ];       // stack start
           address
10     *(tcb->sp) = (uint32_t) task_function;              // LR
11     (tcb->sp) -= 8;    // R11, R10, R9, R8, R7, R6, R5, R4
12
13     tcb->prio = priority;
14     tcb->state = TASK_READY;
```

```
15
16      // push newly created task to ready queue
17      ready_queue_push(priority, tcb);
18
19      // set ready queue with priority as runnable
20      set_runnable_prio(priority);
21  }
```

Listing 4: Task creation function

As seen in Listing 4, the function takes five parameters. The first parameter `void (*task_function(void))` is a pointer to a task's function. That function is the main entry point for the task when it is first time executed. The function itself also contains the main code for the task itself. Using function pointer passed to the function, the CPU will know from which memory address it should start to execute the task once it is scheduled by the scheduler. The second parameter `tcb *tcb` is a pointer to a task's TCB as seen in Chapter 3.1.2. Task's TCB is allocated statically at compile time and is provided by the user. The TCB passed to the function does not need to be initialized with correct values before function call. The function will initialize the TCB with correct data as seen through out Listing 4. Next two parameters to the function are stack's address and stack's size.

As discussed in Chapter 3.1 task's stack is implemented as an array and thus the memory address of the array's first index is passed to the function. The `uint32_t stack_size` parameter as name suggests, is the size of the task's stack. Size of the stack defines how much data it can hold. Since task's stack is implemented as array the value passed to the function and the actual size of array should match. Since Cortex M3 defines the stack as 32-bit wide the size of the stack can be calculated using following formula: $S = 32bits * N$, where N is the amount of elements in the array or the maximum amount of items that the stack can hold. The minimum stack size the scheduler defines is 100. This means that minimum stack size is $32bits * 100 = 3200bits$, which is 400 bytes.

Task creation starts with task's stack and context initialization. As discussed previously in Chapters 3.1 and 3.2.1, task's stack is indeed just an array and the address of array's first index is passed to the `task_create` function. As discussed in Chapter 3.1 array's last index is used to locate the stack. It is done through following operation: `tcb->sp = &stack[ stack_size - (uint32_t)1 ];` In this operation the task's stack pointer is accessed through task's TCB. After that task's stack pointer is assigned to

array's last index. After this simple operation task's stack pointer will point to the stack's start address which is indeed the memory address of the array's last index. Once the task's stack pointer has been initialized to the correct memory address, the initial context for the task is created.

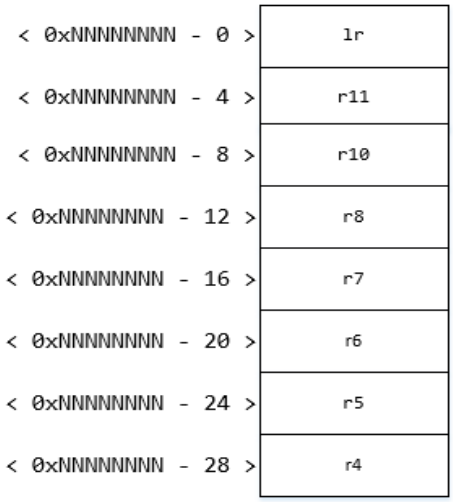

Figure 9: Task's context inside memory

The task's context is created in the same way as it would have been created normally by the context switch.

The context holds nine register values as illustrated in Figure 9. In the initial context only the register `lr` holds known data. Rest of the context will hold unknown data. The initial context starts with assigning the top of the stack that TCB's stack pointer points to with the address of task's function. This is achieved in following way: `*(tcb->sp) = (uint32_t) task_function;`. As discussed previously the task's function is the entry point of the task when it is executed the first time. It is reserved for `lr` register also known as link register. Rest of the context is created simply by incrementing TCB's stack pointer eight times. This way rest of the context and thus values for the registers are created. Values for these registers are unknown but task will not utilize these register values since, when task is started to execute it's own code it will overwrite these register values.

Rest of the task initialization is assigning task with priority provided as parameter to the function and adding it to the Ready queue of corresponding priority. When task

is pushed to the Ready queue the corresponding priority bit is set for the corresponding task queue.

3.2.2 Implementation of Task Queue

The scheduler maintains task queues that it uses for task scheduling. Task queues consists of TCBs of each task in the computer system formed into a linked list. Task queue is implemented as struct. As seen in the Listing 5, Task queue itself is just a simple struct that encapsulates the linked list. Task Queue keeps track of first and last task in the queue using head and tail pointers. Head points to the first task and tail points to the last task in the queue. Since task's are removed and added to queue, the size of the queue is also maintained. The scheduler updates theses queues using push and pop functions. As discussed in Chapter 2.2.1, push pushes the task's TCB to the back of the queue an pop pops TCB from the font of the queue.

```
1 typedef struct{
2
3     uint8_t      size;              // size of the queue
4     tcb         *head;              // head of the queue
5     tcb         *tail;              // tail of the queue
6 }task_queue;
```

Listing 5: Task queue implementation

One of the queues that the scheduler maintains is Ready queue. The Ready queue contains tasks that are ready to run, thus in Ready state. The scheduler picks next task to run only from the Ready queue. When the scheduler schedules the task, the scheduler always chooses the task with highest scheduling priority. In order for the searching for a task with the highest priority to be easier and faster, the scheduler implements multiple Ready queues instead of one Ready queue. Each Ready queue has it's own priority in the same way task's have priorities. Ready queues and tasks share the same priorities. Each Ready queue with priority contains only tasks that have the same corresponding priority. This means that each Ready queue contains tasks with same priority as task queue. For example Ready queue with priority 3 will hold tasks only with the same priority 3. As seen in Listing 6 for easier access to each Ready queue, they are formed into array, where each array index corresponds to one scheduling priority. The `TASK_PRIORITY_CNT` corresponds to the amount of priority levels. For example if `TASK_PRIORITY_CNT` equals to

three, then only three priority levels will be configured, so the scheduling priorities from 0 to 2. Thus the array will also contain only three queue entries. Since each index N in the array corresponds to a one Ready queue entry with priority N, that Ready queue with priority N can be accessed and manipulated through array's index N. For example `ready_queues[2]` will access the Ready queue with priority 2, where each task is with scheduling priority of 2.

```
1   static task_queue ready_queues[TASK_PRIORITY_CNT];
```

Listing 6: Task queue implementation

The scheduler maintains a priority bitmap `static uint8_t runnable_queue_prio_bmap` to keep track of empty and runnable Ready queues. The Ready queue is said to be *runnable*, when that Ready queue is not empty and thus contains tasks that are ready to be run. The bitmap is implemented as an unsigned 8-bit integer variable in which each bit position corresponds to one scheduling priority. As mentioned in Chapter 3.1.2 there are at most eight priority levels from 0 to 7, and same priority levels correspond to the same bit positions in the bitmap. This means that priority 0 corresponds to bit 0 in bitmap, priority 1 corresponds to bit 1 and so forth. When one of the Ready queues with priority of N is or becomes empty, the corresponding bit N is toggled off to 0. When Ready queue fills up with at least one task that bit is then set back to 1. Toggling on and of the bit in the bitmap is done with bit-shift operations
`runnable_queue_prio_bmap = runnable_queue_prio_bmap | (1 << prio)` and
`runnable_queue_prio_bmap = runnable_queue_prio_bmap ^ (1 << prio)`.
Where `prio` corresponds to the priority that needs to be toggled on or off. So in brief when Nth bit is set in the bitmap then Ready queue with priority N is ready to run.

The bitmap variable is particularly useful when searching for Ready queue which is with highest runnable priority. This means the Ready queue that is runnable but also is with the highest priority among other runnable Ready queues. The search is accomplished using the bitmap variable. Since each bit in the bitmap corresponds to one runnable priority, only the most significant bit is checked from the bitmap to reveal the highest runnable priority.

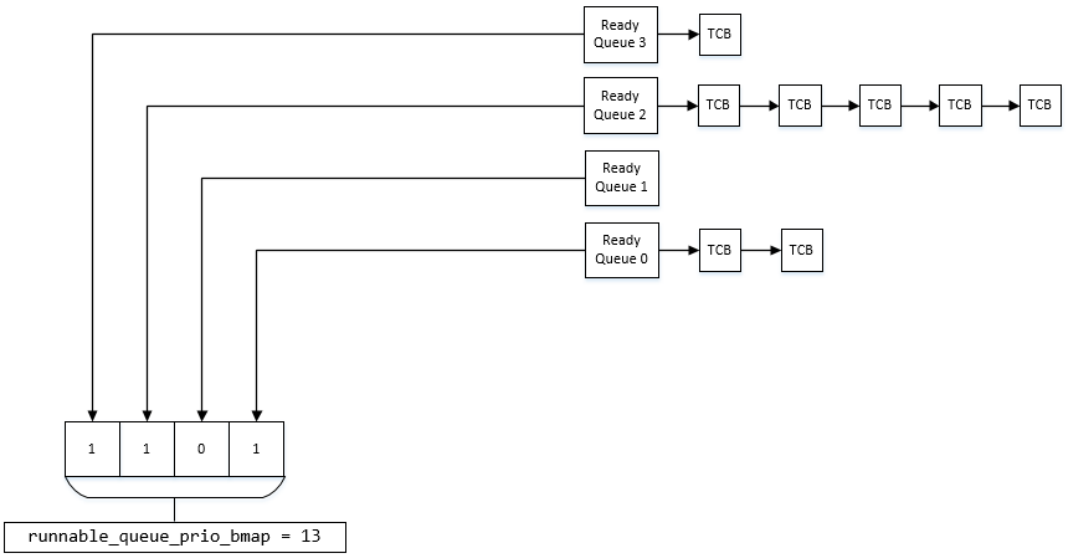

Figure 10: Relationship between Ready queue and Bitmap

The Figure 10 illustrates the relationship between Ready queues and the bitmap. As seen in the figure there is four Ready queues. Only Ready Queue 1 is the empty and thus it is not runnable and rest of queues are runnable. Since each Ready queue is corresponds to a one bit in the bitmap, it is seen that the bitmap is equal to the binary value 1011, which is number 13. From this binary representation it is easy to see that most significant bit is bit 3. This means that Ready queue with the highest runnable priority is Ready Queue 3.

3.2.3 Semaphore and Wait Queue

The task scheduler maintains also Wait queues that are exclusively used by semaphores. Implementation of the semaphore is illustrated in Listing 7. As seen in Listing 7, the semaphore contains three variables.

```
1 typedef struct{
2
3     int32_t      count;  // semaphore count value
4     int32_t      limit;  // semaphore limit value
5     task_queue    wait;  // semaphore wait queue
6 }semaphore;
```

Listing 7: Implementation of a semaphore

The `int32_t count` variable is the counter, that is modified when semaphore is ei-

ther incremented or decremented. Semaphore's counter variable is modified using `semaphore_take` and `semaphore_give` functions. `semaphore_take` function decrements `count` variable and thus it corresponds to the P operation and latter one corresponds to operation V as discussed in Chapter 2.2.4. The `int32_t limit` variable defines the limit for the counter variable. As discussed in Chapter 2.2.4 for counting semaphore the number can be theoretically any positive value, while for binary semaphore the limit value is 1.

Wait queue is used to hold tasks that are waiting for the semaphore. Each created semaphore has own unique wait queue and thus they are created only when semaphores themselves are created. If semaphore is not currently available, task decrementing or in this case "taking" the semaphore is pushed to a semaphore's wait queue. That is done through `task_block_self` function, that simply changes task's state to `BLOCKED` and pushes it to specified task queue and initiates context switch by calling `task_yield` function. In this case it is semaphore's wait queue. This means if the semaphore is currently available then it is not pushed to semaphore's wait queue. When one task "gives" the semaphore back, then one task from the corresponding semaphore's wait queue is popped and pushed to a back of the Ready queue based on popped task's scheduling priority. That is done through `task_ublock_one` which pops one task from specified task queue, changes it's state to `READY` and pushes it to the Ready queue.

3.2.4 Context Switch

Context switch is the main action taken by the task scheduler. By using context switch the scheduler is able to switch tasks. As discussed in Chapter 3.2.1. The scheduler is implemented as cooperative scheduler, meaning that task itself needs to tell the scheduler when context switch should occur. Task scheduler was implemented as cooperative because it is easier to design and implement. Also it is easier to build preemptive scheduler on top of the cooperative one. The cooperative scheduler works as a base for possible additions.

The scheduler is started by invoking `task_start_scheduler` function. Before the function call the scheduler has been already executing for some time. That time is used

for the scheduler to initialize tasks' task control blocks, stack pointers, all necessary task queues and other components. For all of that initialization the scheduler uses it's own stack pointer called `uint32_t *sp_kernel`. That stack pointer is also used for starting the first task. The first task is started by switching context between the scheduler and the first task. When the scheduler is started one special task is created called Idle task. The Idle task is used when there are no task ready to be executed, this might be for example if all tasks are in Blocked state. Idle task is no different from other task. As any-other task Idle task has own TCB, stack and scheduling priority. It is however configured with the lowest possible scheduling priority, so it does not execute when there are actually tasks that are ready to execute. The Idle task is there just so CPU would not sit idle doing nothing. The Idle task does not do anything useful itself, it just calls `task_yield` function.

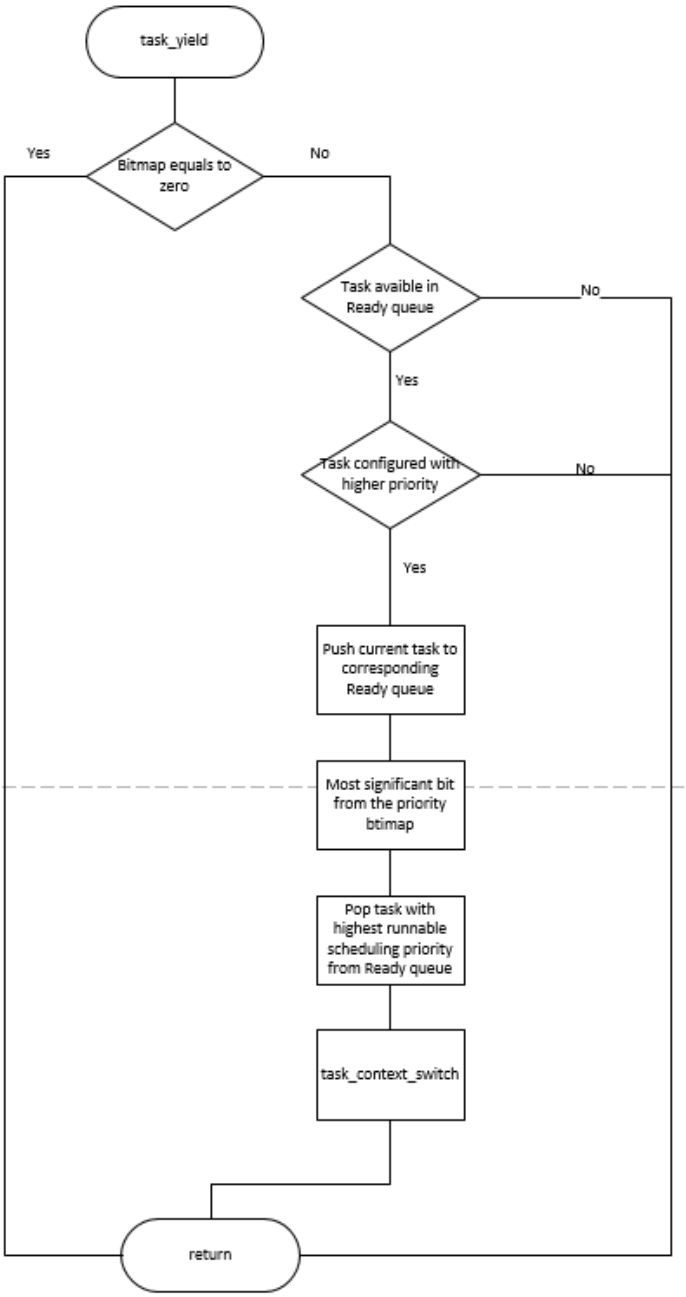

Figure 11: Task yield flowchart

The flow of the context switch is illustrated in Figure 11. As seen in the Figure XX, the context switch is invoked by calling `task_yield` function. That function is called by the task itself to invoke the scheduler and tell it to switch the current running task to a new one. The scheduler maintains current running task with `static tcb *current_running_task` variable. That variable is just a pointer that points to a currently executing task on the CPU. When the function is invoked first thing that is checked is the priority bitmap. The priority bitmap value is zero, when there are no runnable Ready queues. This means that

current running task is the only task that is currently able to execute. Thus as seen in the Figure 11 when that holds the function will immediately return and no context switch will occur. However if there is at least one task in the one of the Ready queues the current task's state and scheduling priority is checked. If current task is in Ready state and it is configured with highest runnable scheduling priority the function returns immediately as previously discussed. However if the current task is blocked and thus in Blocked state or it has lower scheduling priority than task that is in Ready queue, the task with highest runnable priority must be chosen. Before the next task is chosen, the current running task is pushed back to a Ready queue with corresponding scheduling priority. As discussed in Chapter 2.2.1 task with the highest runnable priority is chosen by extracting priority bitmap's most significant bit. When scheduling priority is known the Ready queue is accessed and the task's TCB is popped from that queue. After that the pointer to the current task `current_running_task` is changed so it points to a task with highest runnable scheduling priority that was popped from the Ready queue. After that the actual context switch is invoked through `task_context_switch` function.

```
1   __attribute__((naked))void task_context_switch(uint32_t **sp_st,
        uint32_t **sp_ld){
2     __asm__ __volatile__(
3
4         "   push {r4-r11, lr}          \n" // save context
5         "   str sp, [r0]               \n" // store old stack pointer
6         "   ldr sp, [r1]               \n" // context switch
7         "   pop {r4-r11, lr}           \n" // restore context
8         "   bx lr                      \n"
9     );
10  }
```

Listing 8: Implementation of context switch

As discussed in Chapter 3.2.4 the context switch consists of two parts: saving the context of one task and restoring context of the other. As seen in the Listing 8, `task_context_switch` function takes two parameters `uint32_t **sp_st` and `uint32_t **sp_ld`. First parameter is the stack pointer of task whose context will be stored and latter one is a stack pointer of task, whose context will be restored and run on the CPU. Context switch starts by pushing current task's general purpose registers r4-r11 and special link register `lr` onto the stack using `push {r4-r11, lr}` instruction. Only these registers are saved by the scheduler, because *ARM Architecture Procedure Calling Standard* [13] informs that only these registers need to be preserved by the subroutine. In this case sub-

routine is the `task_context_switch` function. As seen in figure 12, after task's context has been successfully pushed, the task's stack pointer is updated to a top of the stack, which is done through `str sp, [r0]` instruction. This instruction copies CPU's stack pointer `sp` register's contents into the memory address stored in register `r0`. The memory address held in `r0` is the memory address of the current task's stack pointer. Thus current task's stack pointer is updated with top of the saved stack. The context of the task can be said to be saved.

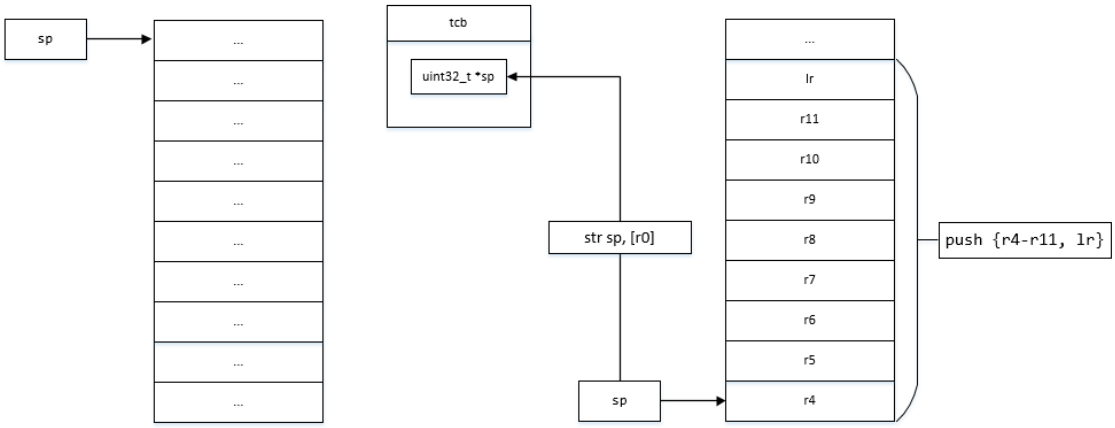

Figure 12: Illustration of saving task's context

After this other's task context can be restored. This is done by first loading CPU's stack pointer register with the address pointed by task's stack pointer `uint32_t **sp_ld`. This is done with `ldr sp, [r1]` instruction as seen in figure 13. After this operation the CPU's stack pointer points to a top of the other task's stack. Since CPU's stack pointer points to task's context it can be then restored with `pop {r4-r11}, lr` instruction. To continue from the same place the task left off right before the previous context switch `bx lr` instruction is executed. The link register `lr` holds the address of the next instruction right after `task_context_switch` function call. This is because when context switch happens task whose context is stored does not return from the `task_context_switch` function until it's context is again restored on the next `task_context_switch` function call.

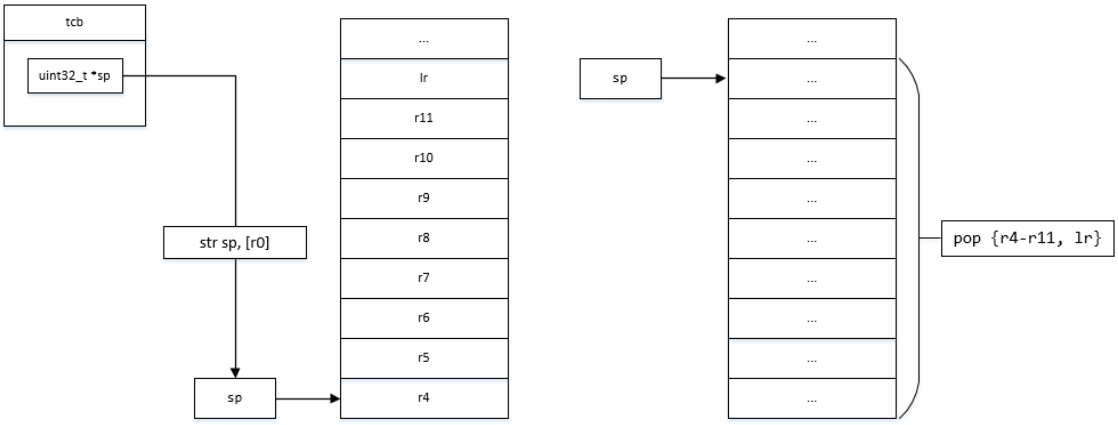

Figure 13: Illustration of restoring task's context

4 Conclusions

The minimum requirements for the task scheduler were achieved and the scheduler works as intended. With current features, it is possible to create tasks, cooperatively schedule them, as well as providing synchronization through the use of semaphores. This might be already enough for smaller scale project where tasks need to be scheduled cooperatively.

Current state of the project was achieved with a research about the topic and using that researched information to implement the project. The main resource for the research was gathered from literature about the operating system theory. In order to properly implement the cooperative task scheduler, comprehensive understanding of operating system theory as well as Cortex-M3 processor specific architecture was essential.

This task scheduler as is can be already be used for explaining one of the main functions of any operating system in a simple manner. This task scheduler is also suitable base for implementing a preemptive task scheduler which could be expanded into a to a full fledged real-time operating system. To expand the current scheduling method from cooperative to preemptive, several features need to be added. One of those features would be use of exceptions that Cortex-M3 processor core provides. ARM's Cortex-M3 documentation explicitly informs which exceptions can be used and configured to be used in potential operating system [14]. For instance a *SysTick* exception can be used as system tick that would generate exception on configured interval that would interrupt CPU asynchronously [15]. That tick can then be used for initiating context switch based on timing, instead of manually doing it as with cooperative scheduling. In that case CPU will be responsible for invoking context switch instead of the tasks themselves. To make the scheduler more complete two different stack pointers could be used instead of one. ARM Cortex-M3 processor core provides two stack pointers: Main stack pointer (MSP) and Process stack pointer (PSP) [16]. ARM recommends to use MSP as a operating system kernel stack and PSP to be used to run task's own program code [17]. The addition of two stack pointers will add a layer of security, because the scheduler and tasks will be separated from each other. By that time the task scheduler

would start to look like more like an operating system.

To make the preemptive scheduler more like an operating system a few attributes that are essential characteristics of an operating system should be added. One of the obvious features would be the addition of mutex also called mutually exclusive semaphore. Since the current task scheduler already contains semaphore it can be easily be modified to make a mutex and other nuances that come with it such as priority inversion. Another feature that would define inter process communication between tasks are message queues. With message queues tasks could easily send information between each other instead of depending on primitive variables with fixed lengths and semaphores. To add run-time security for each task a simple stack overflow checking mechanism sould also be added. This would prevent for one task to accidentally interfere with other task's stack and other data.

Bibliography

1 Stallings W. In: Operating Systems: Internals and Desing Principles, Seventh Edition. Pearson; 2012. p. 76.

2 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 103.

3 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 104.

4 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 105.

5 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 105–106.

6 Philip A Laplante SJO. In: Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition. Wiley; 2012. p. 99.

7 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 109.

8 LINFO. Context Switch Defenition; 2004. Accessed on 28.10.2019. http://www.linfo.org/context_switch.html.

9 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 112.

10 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 204.

11 Abraham Silberschatz GG Peter B Galvin. In: Operating System Concepts, Ninth Edition. Wiley; 2014. p. 255–256.

12 ARM. Stacks; 2010. Accessed on 15.11.2019. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABDGADF.html.

13 ARM. The Base Procedure Call Standard; 2018. Accessed on 21.11.2019. https://developer.arm.com/docs/ihi0042/g/procedure-call-standard-for-the-arm-architecture-abi-2018q4-documentation.

14 Exception types. ARM; 2010. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABBGBEC.html.

15 System Timer (SysTick). ARM; 2007. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0179b/ar01s02s08.html.

16 Stacks. ARM; 2010. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABDGADF.html.

17 Core registers. ARM; 2010. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/CHDBIBGJ.html.