FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Real-time scheduling algorithms applicable for embedded systems
Student:	Aykut Sahin
Supervisor:	doc. Ing. Hana Kubåtová, CSC.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021 / 2022

Instructions

1. Study the different types of scheduling algorithms.
2. Select the ones that are applicable to real-time embedded systems.
3. Implement several selected ones (preferably in C language).
4. Discuss their possible include into RTOS operating system.
5. Compare and evaluate algorithms and their implementation with respect to predetermined design constraints (processing speed, determinism, size, maximum operating frequency, deadline fulfillment, etc.)'

Electronically approved by Ing. Michal Valenta, Ph.D. on 14 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Real-Time Scheduling Algorithms Applicable for Embedded Systems

Aykut SAHIN

Department of Digital Design
Supervisor: doc. Ing. Hana Kubátová, CSc.

June 16, 2021

Acknowledgements

I wish to express my sincere thanks to doc. Ing. Hana Kubátová, CSc., head of the Department of Digital Design, for this opportunity of working on a very interesting topic and for providing me the necessary resources and assistance for Completion of my thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on June 16, 2021 _____

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Aykut Sahin. *Real-Time scheduling algorithms applicable for embedded systems.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstract

This thesis work is deeply about the scheduling algorithms in operating system schedulers. The algorithms that are used for implementing a scheduler for a real time operating system is vital to the lifecycle and outcome of any set of real time applications that get executed. When it comes to real time applications, especially with the hard real time applications, the deadlines and response times required by the applications have to be followed as strictly as possible. There are various scheduling algorithms that are used in many different scheduler designs. However, not all of them pursue the above objective for real time applications. Therefore, we discuss and study various scheduling algorithms in a technically detailed manner in order to determine the ones that would be applicable for real time operating systems that would run real time embedded applications. Later, we implement and integrate the scheduling algorithms that are chosen for real time applications based on our study, to an open source real time operating system FreeRTOS in C programming language, of course. Then as a final step to this study, we run a real time embedded application together with our modified FreeRTOS with our custom implementations for the scheduler algorithms are included, on ARM Cortex-M3 microcontroller architecture by using QEMU with respect to predetermined design constraints such as processing speed, determinism, size, maximum operating frequency, deadline fulfillment, etc.. We demonstrate functioning scheduling algorithms running in a real time operating system executing a real time application without having to own a hardware by QEMU, which makes testing and scheduling analysis of real time applications and scheduling algorithms on all software. We also compare and evaluate the performance of the implemented algorithms in order to better understand how each of the algorithms behave under certain circumstances which will help us choose better suitable algorithms for our real time applications.

Keywords: real time operating systems, real time embedded systems, real time scheduling algorithms, QEMU, FreeRTOS, scheduling analysis

Abstrakt

Tato závěrečná práce je podrobně zaměřena/úzce orientovaná na plánovací algoritmy, tedy plánování procesů. Tyto algoritmy se používají pří plánování procesů u operačních systémů, pracujících v reálném čase a jsou zásadní pro životnost, funkci a výstup jakékoliv aplikace, pracující v reálném čase. V případě aplikaci pracujících v reálném čase, zejména takzvaných „hard real time“ aplikací musí být stanovené hraniční termíny dodržovány co nejstriktněji a doba odezvy musí být co nejpřesnější. Existuje široká škála různých plánovacích algoritmů, které jsou využívány při návrhu v celé řadě plánováních procesů. Avšak ne všechny tyto druhy plánovacích algoritmů jsou vhodné pro aplikace, pracující v reálném čase. Z tohoto důvodu je těmto typům algoritmů věnována velká pozornost. Plánovací algoritmy jsou tak v současnosti předmětem podrobných technických testů a analýz, které jsou zaměřeny na posouzení jejich kvalit a vlastností s cílem zjistit jejich způsobilost implementace do aplikací operujících v reálnem čase. Vybrané plánovací algoritmy jsou tak dále v této práci implementovány do prostředí operačního systému FreeRTOS, pracujícím v reálnem čase a založeném na programovacím jazyku C. Konečným krokem této práce je pak spuštění vybraných aplikací v reálném čase společně s vytvořenou modifikací operačního systému FreeRTOS a vlastními implementacemi algoritmů plánovacích aplikací. Dané aplikace byly spuštěný prostřednictvím mikrokontroleru ARM Cortex-M3, a to pomocí QEMU s ohledem na konstrukční a omezení, jako je rychlost zpracovávání, determinismus, velikost, maximální provozní frekvence a hraniční termíny zpracování. Dále byly demonstrovány fungující plánovací algoritmy, běžící na operačním systému, vykonávajícím dané aplikace v reálném čase bet nutnosti vlastního hardwaru, a to díky využití QEMU, který umožnil testování, analýzu a plánování procesů daných aplikací či plánovacích algoritmů na veškerém softwarovém vybavení. Také jsme porovnali a vyhodnotili výkon implementovaných algoritmů za účelem lépe pochopit, jak se jednotlivé algoritmy chovají při daných okolnostech, což nám pomohlo lépe zvolit vhodný algoritmus pro naše aplikace pracující v reálném čase.

Klíčová slova: operační systémy v reálném čase, vestavěné systémy v reálném čase, algoritmy plánování v reálném čase, QEMU, FreeRTOS, analýza plánování.

Contents

Chapter 1: Introduction

1.1. Overview

There are various systems that assist with functioning of many things around us. Call it a computing system, communication system or an information system. Systems are groups of interrelated elements interacting with each other according to a set of rules to serve a specific purpose [6]. A system is recognized as a real time system when it guarantees a response within a specific time frame [5]. The time frame could be in seconds, milliseconds or even microseconds. When it comes to real time systems, especially hard real time systems, the important thing is to be exact to the deadline of any process [5].

What this study is all about is more related to computing systems and more specifically operating systems. Computing systems are systems that make a great use of computers. A computer, always, runs on a set of hardware and software parts. One typical hardware part for anything that has a processing power is a unit called the processor . A processor is an electronic circuit embedded into a circuit board responsible from performing arithmetic, logic controlling and input/output operations determined by the instructions given by the program that runs against to it [7]. Furthermore, there is also a software side to a computer. A computer can not function without at least the smallest bit of software to tell it what to do with it's hardware. This is the exact point where an operating system comes in place. An operating system is responsible from managing a computer's hardware and software resources and provide a set of services to drive the computer through an interface. Operating systems vary a lot with respect to the purpose and hardware for which it will be used for. Every hardware (processor) family has it's own instruction set that is used to communicate with the hardware. Therefore, an operating system must comply with a specific instruction set of an hardware in order to be eligible to operate it. On top of that, the operating system should be designed in a way to support the use cases of the user. There are several different types of operating systems even in the most abstract meaning. Real time operating system is the one that we are closely be dealing with.

1.2. Real Time Operating Systems

The design and thinking behind a real time operating system have a motive of being able to serve real time tasks at a very consistent level in terms of the time spent on accepting and responding them. Another unique attribute of real time operating systems is that since they are serving a way more specific and smaller set of tasks, unlike general purpose operating systems, they tend to be much smaller in terms of size. This allows real time operating systems to operate embedded devices. That's just one of the reasons why most of the time MCUs, MPUs or even some other embedded development devices run on real time operating systems.

Apart from the above, arguably, one of the most significant components of a real time operating system is it's scheduler. The scheduler is quite differentiative when it comes to operating systems and there are many algorithms used to implement them. Since schedulers are the main topic of this study, it's quite appropriate to dive deeper into them.

1.3. Scheduling

Scheduling in operating systems is the process of organizing the way the workload is handled [4]. Simply put, it's thanks to the scheduler the processor knows what to do next. Scheduler is the decision maker part of an operating system and there are mainly three decisions to make.

- Processor assignment:

  Not all the operating systems run on single core CPUs. Therefore, it happens that the operating system needs to decide which processing core the awating task it should assign to. This process is called the processor assignment [4].

- Task ordering:

  It's certainly the case for many operating systems that there are more tasks than the available processing cores. In such a situation, depending on how the operating system handles, it's possible to block tasks for certain amount of times to allow other tasks to run. Deciding in which order these tasks should be is called task ordering [4].

- Execution timing:

  As much as the ordering of the tasks matter, the timing of the task executions can also be quite significant in many ways. Out of a set of tasks to be executed, we might want to give a higher rate of execution time exposure to a certain task just because of it's priority is higher for the application's use case [4]. Therefore, the scheduler allocates specific execution times to tasks in certain cases. This is called execution timing [4].

Designing an operating system has a lot to do with introducing the above mechanisms to the software with both so called design patterns and algorithms. Over the years, people who worked in this field developed many different ways to introduce these features into operating systems. One of the foremost differentiation between schedulers is the possibility to make the decisions it's supposed to make either statically, at compile time or dynamically, at run time. There are also schedulers developed using a little bit from both design patterns at the same time in more hybrid way. That brings the development of operating system schedulers to a few main different subtitle:

- Fully static scheduler:

  How and when to run tasks is statically stated during the design specification of the scheduler [4]. The way tasks will run is known before the execution. This is achieved by using execution timing on tasks. However, fully static schedulers are not so reliable especially with the modern processors since the precise execution time for a task is extremely unpredictable.

- Offline scheduler:

  Task assignment for processors and ordering is done at the design specification of the scheduler [4]. However, the decisions are not yet made until the run time starts. If nothing unusual for the scheduler it will continue execution with what's specified but if there is any circumstance such as a task being blocked on a semaphore or lock then the scheduler may adjust it's decisions during run time [4].

- Static assignment scheduler:

  Only task assignment is done at the design specification [4]. Basically, the processors know what set of tasks they need to execute before the execution starts. However, the scheduling within the set of tasks for each processor is done dynamically at run time.

- Fully dynamic scheduler:

  All the decisions belong to the scheduler are made during run time.

- Preemptive scheduler:

  Preemption is the temporarily halting of the execution of a task with the intention of continuing it at a later time. Preemptive scheduler has the privilige of making a decision during the execution of a task. A preemptive scheduler can block a task somewhere in between it's lifecycle and allow another task to execute. Mostly, general purpose operating systems use such schedulers in order to allow the execution of many tasks simultaneously.

- Non-preemptive scheduler:

  As oppose to the preemptive scheduler, non-preemptive schedulers doesn't interfere with task executions once they are started. It allows the task to run until completion for a dedicated period of time. Real time operating systems tend to be non-preemptive because of their nature. Real time tasks are critical with respect to their deadlines and therefore it's not considered as a good fit to use a preemptive scheduler. On top of that, real time operating systems tend to run relatively smaller and more dedicated set of processes. Therefore, being able to run as many processes as possible simultaneously is generally not an objective of a real time operating system. With that being said, the fact that there are real time operating systems with preemptive schedulers can't be avoided.

1.4. Task Model

Task model is another important topic for schedulers. The set of informations known or assumed about the tasks to be scheduled by the scheduler creates the task model [4]. This is necessary for the scheduler to be able to make the decisions that it's supposed to do. Often for the real time operating systems the tasks to be executed are assumed to terminate at some point however, for general purpose operating systems that's not the case [4]. Moreover, some schedulers makes an assumption on the information regarding the task even before the scheduling takes place, whereas some schedulers support arrival of tasks which corresponds to the fact that the tasks can be introduced to the scheduler during the execution of other tasks. Schedulers needs to take into consideration of the possible scenarios about the lifecycle of the tasks. There are tasks that needs to be executed repeatedly, maybe forever [4]. On the other hand some other taks might require an

execution only periodically [4]. All of these and possible other informations about the tasks help scheduler to determine the timing of the tasks. There are also timing properties for tasks with respect to it's environment. Timing properties help the scheduler correctly time the tasks of which are the responsibility of the scheduler. In order to understand how the interrelated timing of tasks occur, let's discuss the timing properties for the real time operating systems.

- Ready time:

  Task is ready to be executed at ready time.

- Deadline:

  Task execution must be finished by the deadline. However, there are two main real time tasks that differ in terms of execution deadline of the tasks. Those are:

  - Hard real time:

    The execution must be completed by the deadline. Otherwise, the system or the execution session is considered as faulty.

  - Soft real time:

    It is still expected for the task execution to be finished by the deadline. However, in this type of task and system, it's tolerable to be late on execution completion with respect to a certain degrade in the result of the run.

- Minimum delay:

  The time between when the task is ready to be executed and when the task actually gets executed has to be greater than or equal to the minimum delay [4].

- Maximum delay:

  The time between when the task is ready to be executed and when the task actually gets executed has to be lower than or equal to the maximum delay [4].

- Worst case execution time:

  The total amount of execution time of the task starting from the time when the task is ready to be executed, can not be greater than the worst case execution time [4].

- Run time:

  Without any preemption, the total amount of time to complete the execution of the task starting from the time when the task is ready to be executed, is called run time.

- Priority:

  Priority is a relative term. If a task is the only task within the system then it's priority is meaningless. However, when there are multiple tasks to be executed then the priority expresses the urgency of any task compared to the other tasks in queue.

How the scheduler is able to process all the information and make the decisions that it's responsible from? The short answer to that question is the scheduling algorithms. They are one of the most important components of any scheduler. By the scheduling algorithm the scheduler knows about what to do with the information it acquires from it's environment which later on is used for decision making. There are several scheduling algorithms when it comes to operating systems. As per the study, the next chapter discusses the scheduling algorithms.

Chapter 2: Related Study

2.1. Objective

There are several scheduling algorithms already existing and used in many operating systems. The purpose of this chapter is to study several of them and find out the most applicable ones for real time embedded systems with respect to the task model of real time applications. As a result of this work we will know about the studied scheduling algorithms and therefore we will be able to choose the appropriate ones for the further parts of this thesis.

2.2. Scheduling Algorithms

A scheduling algorithm is the core of any scheduler with respect to managing the resource allocation between the processes. The way the scheduler manages resource allocation for entities greatly changes according to the way it's scheduling algorithm is designed. Scheduling algorithms that are designed for general purpose operating systems tend to aim fairness with regards to resource allocation distribution between tasks, whereas a scheduling algorithm for a real time operating system takes more into consideration deadlines and priorities of the tasks.

Another aspect to scheduling algorithms is that they often involves a set of assumptions prior to achieving their claim of functionality. Since there is no optimal scheduling algorithm for all circumstances, scheduling algorithms has to require some set of properties which the tasks or the environment have to have, so that it can perform consistent and optimal.
We will discuss several scheduling algorithms below.

2.2.1. Rate Monotonic

Rate Monotonic (RM) scheduling algorithm is a uniprocessor, static, preemptive, priority assignment algorithm that is preferred to be used in real time operating systems [2, 4, 5]. The static priority assignment is done by taking into consideration the period of the tasks. The longer the period the lower the priority of the task will be during scheduling. Rate Monotonic is a simple algorithm yet it can yield great performance and is pretty popular.

The reason that Rate Monotonic algorithm is a static algorithm is because the tasks are constrained to have a fixed priority that is known by the algorithm prior to the execution which can be difficult sometimes [4]. Therefore, it provides the optimal results when the tasks' behaviour are as predictable as possible.

Rate Monotonic is an optimal, static, preemptive scheduling algorithm since if there is any given static preemptive scheduler algorithm that yields a feasible schedule for a given set of tasks, then Rate Monotonic algorithm also yields a feasible schedule [2, 4, 5, 8]. It has some basic characteristics. One of them is the fact that all tasks that are being scheduled by Rate Monotonic algorithm have to be periodic and independent from each other. Which means that they can not share any kind of resource with another task [4, 8, 11]. Being a uniprocessor scheduling algorithm means that there must be only a single processor that powers up the algorithm. The time it takes Rate Monotonic algorithm to switch between task executions is negligible [2, 4, 5, 8]. Therefore, there is a pretty low system overhead.

Rate Monotonic is a decent algorithm when it comes to feasibility as well. Feasibility stands for being able to meet requirements and constraints (time, resource etc.) by the tasks.Usability factor (utilization) for Rate Monotonic algorithm is calculated as follows [4, 5, 8]:

$$\mu = \sum \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Figure 2.1 [4].

Where C_i is the execution time of the task,
T_i is the period of the task and
n is the number of tasks.

Processor utilization limit according to the Liu & Leyland limit [4, 5, 8]:

$$\lim_{n\to\infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693$$

Figure 2.2 [4].

Where n is the number of tasks approaching infinity.

This calculation demonstrates the fact that for any given set of tasks, the Rate Monotonic algorithm will meet all the deadlines of the set of tasks', unless they use more than or equal to 69.3% of the available processor utilization time [4].

An example to the Rate Monotonic feasibility (utilization factor, utilization bound) calculation is below.

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

Figure 2.3 [12].

The utilization will be 1/8 + 2/5 + 2/10 = 0.725

3*(2^1/3 – 1) = 0.780

Since 0.725 < 0.780 the system is feasible.

2.2.2. Earliest Deadline First

Earliest deadline first (EDF) algorithm is a uniprocessor, dynamic, preemptive, priority assignment scheduling algorithm [8, 9]. As the name suggests, Earliest Deadline First algorithm executes it's tasks in an ascending order with respect to the tasks' deadlines. The sooner the deadline of a task, the sooner it gets executed. Simply put, the task with the earliest deadline compared to others, gets executed first. Earliest Deadline First is considered as a bit more complicated of an algorithm with respect to it's implementation. That's mainly due to fact that Earliest Deadline First being a dynamic priority assignment scheduling algorithm, which is harder to implement [4]. However, Earliest Deadline First makes up to that with a great feasibility. An Earliest Deadline First schedule is feasible if the total task load is under 100% for any task set [9, 10]. Due to that Earliest Deadline First is one of the most important scheduling algorithms that are used for real time operating systems out there.

The reason this algorithm is a dynamic scheduling algorithm is that since the task with the closest deadline might change, the algorithm also modifies the assigned priorities to the tasks accordingly [10].

Earliest Deadline First is an optimal dynamic priority assignment scheduling algorithm since if Earliest Deadline First can not schedule a task then there is not other scheduling algorithm that can [2, 4]. This can be proved by the fact that Earliest Deadline First minimizes the maximum lateness [4]. Even though Earliest Deadline First is theoretically better than Rate Monotonic, Earliest Deadline First has a higher system overhead. Apart from that, overloaded system is unpredictable for Earliest Deadline First [2, 4, 5, 8]. For Earliest Deadline First the tasks do not have to be periodic [2]. Dynamic priority assignment is done generally by using a list or queue that is sorted by their deadlines [2].

Usability factor (utilization) for Earliest Deadline First is calculated as follows [4, 5, 8]:

$$\mu = \sum \frac{C_i}{T_i} \leq 1$$

Figure 2.4 [4].

Where C_i is the execution time of the task,
T_i is the period of the task.

An example to a working Earliest Deadline First is below.

– T_1=50, C_1= 25; T_2= 62.5, C_2= 10; T_3=125, C_3= 25.

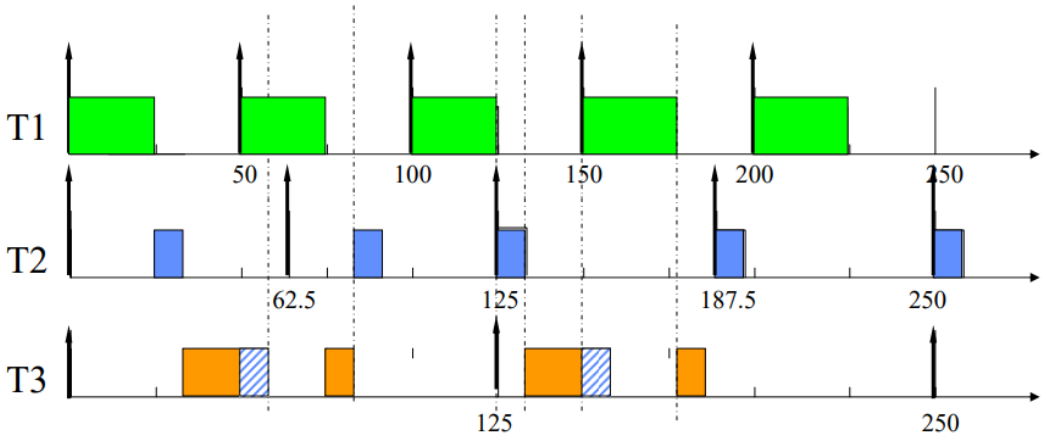

Figure 2.5 [9].

Where C_i is the execution time of the task,
T_i is the period of the task and.

An example to the Earliest Deadline First algorithm feasibility (utilization factor, utilization bound) calculation is below for the above example run.

The utilization factor will be 1/2 + 1/5 + 1/6.25 = 0.86.
Utilization bound for Earliest Deadline First is 1.
Since 0.86 < 1 the system is feasible. [12]

2.2.3. Maximum Urgency First

Maximum Urgency First (MUF) algorithm is designed with inspirations from static and dynamic priority assignment scheduling algorithms [4, 8, 9, 10]. Each task in a set of tasks is assigned an attribute so called urgency. Urgency is a byproduct of two different static priority factors and one

dynamic priority factor. One of them is a static priority that describes the criticality of the task.Second one is also a static priority that describes the user priority of the task. Third one is a dynamic priority that is determined by how low the laxity of the task is [4, 8, 9, 10]. This combination provides two main features to the algorithm; predictability under overloaded conditions and a feasibility bound of 100% for it's critical set (explained below) [10].

The way MFU works is in phases. There are two phases where the priorities are assigned. In the first phase the static priorities are assigned. In the second phase the dynamic ones are assigned [9].

Phase 1:

1) Sorts the tasks by their periods in an ascending order. Defines a critical set of tasks which contains N tasks. Number N is chosen in a way that the processor utilization is not 100% or more. Creating a critical set of N tasks allows us to have a set of tasks that will never fail to be feasible [9].
2) The tasks in the critical set are given high criticality priority. As oppose to that, the tasks that are outside the set of critical tasks are given low criticality priority [9].
3) Every task in the system is given an arbitrary static priority defined by the user [9].

Phase 2:

The below algorithm is executed each time a task enters a ready state.

1) If there is only a single highly critical task execute it [9].
2) If there are multiple highly critical tasks select the one with the least laxity [9]. At this point the laxity of a task is considered to be a dynamic priority calculated during run time.
3) If there are multiple tasks with same criticality and dynamic priority select the one with the highest user priority assigned [9].

As per the dynamic priority of MUF, laxity is measured as the difference between the time to deadline and the remaining computation time to finish. A task with negative laxity won't meet its deadline, so laxity provides early detection of temporal failures [9].

2.2.4. First Come First Serve (non-preemptive)

First come first served (FCFS), is considered as the simplest scheduling algorithm [14]. The reason for that is that the algorithm only executes the tasks that are enqueued to a queue of ready tasks in the order of their arrival without any priority factor. The task which is in the beginning of the queue gets executed until completion and then the same applies to the next one.

Since FCFS is a non-preemptive algorithm the task that gets picked up for execution is kept until completion. An advantage manifested by that is there is minimal system overhead because there is no redundant context switching between tasks. However, on the other hand this creates a so called convoy effect for the system. Simply put, a convoy effect occurs when a process with a very short run time waits for a process with relatively much bigger run time for a long period. Main reason of the occurrence of this disadventageous situation is the lack of priority factors for tasks. Due to that it's very likely for FCFS to miss deadlines of the tasks. Another issue that is manifested by the previous statements is the average waiting time for the tasks being high.

2.2.5. Round Robin

Round robin (RR), is a preemptive, clock driven scheduling algorithm that is named after the round-robin principle [15]. Which means sharing something arbitrary equally and turn based. Therefore, the principle behind RR algorithm is all about multitasking and fairness between tasks.

The most important aspect to RR algorithm is it's time quantum. Time quantum is a time interval for which amount the tasks get executed each time they get picked up for a run. RR executes tasks without taking into consideration any priority. Moreover, the algorithm spends a significant amount of time on context switching depending on the time quantum. Therefore, it's not so reliable for time critical tasks as it tends to miss deadlines.

The performance of RR is heavily dependent on the time quantum for the system [15]. Coming up an efficient time quantum is a challenging task in such systems [15].

2.3. Discussion

As per the above study, we will be implementing Rate Monotonic, Earliest Deadline First and Maximum Urgency First scheduling algorithms in the next chapter of this thesis. First come first served and round robin scheduling algorithms are significantly far from being a real time scheduling algorithms compared to the others studied. One of the most important reasons for that is the fact that netiher of them takes priority factor into consideration for tasks. Which is a vital thing when it comes to real time embedded applications.

Think of a scenario in which there are tasks T1 and T2 are waiting in the ready queue. Then there comes a task T3 with a very critical responsibility of job to the ready queue. What is expected from a real time system is to prioritize the execution of the task T3 and finish it as soon as possible. However, for first come first served and round robin algorithms, task T3 will have to wait until it's task T3's turn to be executed.

Chapter 3: Practical Work

3.1. Problem

We would like to compare and evaluate the scheduling algorithms we have chosen above. In order to achieve that, first we need to implement and include the algorithms into a real time operating system. Moreover, we need an embedded-real time application to execute together with the real time operating system, so we can get some results. Lastly, we need to test the real time operating system with the real time embedded application and for that there needs to be a host for the real time operating system running with the application. We will discuss how to overcome these challenges in the next section.

3.2. Possible Solutions

One of the most naive solutions to the above problem is to simulate an embedded-real time application that is implemented in a high level programming language like C++. Then integrate a scheduling mechanism with the algorithms included into this simulation. This solution would be fairly easy to execute since we would use only a C++ program to handle everything. No actual real time operating system and host would be needed. However, the results acquired from a simulation solution is highly dependent on the quality of the simulation program. That means a lot of time would go into something that is not the main topic of this thesis.

A better solution would be to use an open source real time operating system and modify the scheduling algorithms within. This is for sure not a trivial task to accomplish. Modifying or adding scheduling algorithms in a real time operating system requires the one to have knowledge about the real time operating system, so that the modifications work correctly with the rest of the real time operating system.

When it comes to hosting a real time operating system there are a couple of options available. First off, if the simulator is used there is already no need for a host. However, if an actual real time operating system is used then there is a need for a host.

One way to host a real time operating system is to acquire some kind of a microcontroller hardware that has a processing power and run the real time operating system on it. This method is the one that is the closest to how things would be done if there was an actual embedded application being developed for real life, commercial purposes. However, it's needed to buy such

a hardware and it costs money.
Another way to host a real time operating system is to use an open source emulator. QEMU software is very suitable for this. It's possible to run a real time operating system on any microcontroller architecture that is supported by QEMU. For example, one of the supported architectures is ARM Cortex M-3. One can run their real time operating system on this architecture without having to own a hardware, thanks to QEMU.

As a final step for the solution to the above problem, we need a real time embedded application. A simple real time embedded application with a set of tasks can be implemented, preferably in C or C++ languages and included into either the simulator or the real time operating system.

3.3. Method

As per this thesis work, we will use an open source real time operating system called FreeRTOS. Since FreeRTOS is implemented with **C programming language**, we will implement and integrate our scheduling algorithms into it with also C.

For an embedded real time application we will create a dispatcher simulation program. The dispatcher will have items to dispatch with certain time periods and execution times. For example, let's say there are three dispatchers A, B and C. These three dispatchers are considered as different tasks. Each dispatcher has a specification regarding the time it takes to dispatch an item, the time period for which an item has to be dispatched exactly once and the priority for the dispatcher. Implemented scheduling algorithms will manage the dispatchers with respect to predetermined design and requirements of the dispatchers.

The last step is to host the real time operating system together with our dispatcher program, so we can execute. We will use an open source emulator called QEMU for that. QEMU has support for many architectures but we will target **ARM Cortex M-3** architecture. First, we will compile FreeRTOS version ported for our specific target architecture and **GCC compiler** together with our real time embedded application and create an executable. Then we will run our executable with QEMU and see the results.

3.4. Implementation

In order to better understand the implementation it's necessary to know a couple of important things regarding the infrastructure of FreeRTOS Kernel. One of the main component of FreeRTOS Kernel is the so called **tasks** which takes place in **task.h** and **task.c** files in the source code of the kernel [16, 17, 18]. There are three main parts of the **task** module in FreeRTOS.

- **TaskControlBlock**
  - A data structure that holds the necessary information about the tasks and with respect to their management.
- **xTaskCreate**
  - An API function that allows the user to register their custom functions as tasks. All tasks wanted have to be created before calling the below function.
- **vTaskStartScheduler**
  - Another API function that starts the scheduler. When called, the execution of all the created tasks are started at that point.

A typical usage for FreeRTOS with a custom set of tasks is to first create necessary functions that will act as the tasks within the code, then register those task functions as tasks with **xTaskCreate(…)** and once they are all ready, call **vTaskStartScheduler()** to start execution [16, 17]. As shown:

```
void taskFunction1(void)
{
    // task code
}

void taskFunction2(void)
{
    // task code
}

int main(void)
{
    xTaskCreate(taskFunction1, ...parameters...);
    xTaskCreate(taskFunction2, ...parameters...);

    vTaskStartScheduler();
}
```

Currently, the FreeRTOS configuration that is used for this work only allows a static preemptive priority assignment scheduling. The assignment of the priority of the tasks are made before the scheduler starts and the prioritites are passed in with the function call **xTaskCreate(…)** by the user.

In order to implement arbitrary, custom algorithms into the FreeRTOS Kernel it's important to understand how the already existing scheduling is working. There is not a specific module in which the scheduling is done in FreeRTOS. Instead, since scheduling is the controlling of the tasks, the mechanism that manages that is written all over the place in **task.c** source file. The file contains approximately **5000** lines of code and it's pretty hard to find your way in it. Apart from that, another component that is highly related to scheduling is the **context switching**. Again, there is

not any specific module within FreeRTOS Kernel that does it. Instead, it takes place mostly in the **port.c**, **port.h** and even **port.asm** files that are specific to the ported architectures and compilers [16, 17].

In order to achieve the objective of this work, I decided to create a **scheduler** module that would act as a wrapper to the already existing scheduling module of FreeRTOS and contain the necessary mechanism to schedule tasks in whatever way I would like. That allows me to govern the scheduling of my tasks from within only one, small file. The scheduler module can be considered as a more user space centered version of the actual **task** module of FreeRTOS Kernel.

First off, in order to be able to create custom tasks it's needed to have a structure like **TaskControlBlock** to hold whatever information is needed about the custom tasks. Therefore, I created my own data structure for it named as **SchedulerTaskControlBlock.** It has a couple of fields that are pretty important:

```
typedef struct SchedulerTaskControlBlock {
//Pointer to the user defined task function.
TaskFunction_t taskFunc;

//Pointer to the parameters to pass to the task function.
Void *parameters;

//User priority assigned by the user
UbaseType_t userPriority;

//A priority value of the task for scheduling.
UBaseType_t priority;

//A reference for the task function.
TaskHandle_t *taskHandle;

//A time by when the task is ready for execution.
TickType_t readyTime;

//A time period by which the task has to complete exactly once.
TickType_t period;

//A time by which the task has to complete execution (For periodic tasks
//the period is often the deadline. For example: Rate Monotonic
//scheduling)
TickType_t softDeadline;

//A time by which the task has to complete execution with respect to the
//system time. Calculated as (first ready time + period * i'th
//exetuion).
TickType_t hardDeadline;

//Execution time the task takes to complete
TickType_t executionTime;
```

```
//Expresses whether the task control block is occupied
BaseType_t bEmptyBlock;  // empty == pdTRUE, not empty == pdFALSE

//Expresses whether the task is considered as critical
BaseType_t bCritical;    // yes == pdTRUE, no == pdFALSE
...}
```

Then I created **SchedulerTaskCreate()** as a wrapper to the **xTaskCreate()** so that I can incorporate my custom fields and algorithms to it the original task instances of FreeRTOS. I also created **SchedulerStart()** function which basically is another wrapper around **vTaskSchedulerStart()** and does a couple of things like setting priorities in our custom way and getting our scheduler module ready. Below is a code snippet shows how it's done:

```
BaseType_t SetNewTaskControlBlock(SchedulerTaskControlBlock_t **newTask)
{
    for(BaseType_t i = 0; i < MAX_NUM_TASKS; i++)
    {
        // check if the task control block is free
        if(gTaskControlBlocks[i].bEmptyBlock == pdTRUE)
        {
            ++gTaskCounter;
            *newTask = &gTaskControlBlocks[i];
            return pdTRUE;
        }
    }
    return pdFALSE;
}

void SchedulerTaskCreate(...parameters for TaskControlBlock...)
{
    SchedulerTaskControlBlock_t *newTask;
    taskENTER_CRITICAL();
    configASSERT(SetNewTaskControlBlock(&newTask) == pdTRUE);
    *newTask = (SchedulerTaskControlBlock_t){...parameters set...};
       taskEXIT_CRITICAL();
}
```

Lastly, I created a parent task function that would act as the original task that is registered to the task module of FreeRTOS which contains a pointer to the custom task function and fires it up once it's running. Explanation of what the parent (wrapper) task function does in half-pseudo code:

```
static void SchedulerParentTaskFunc(void *parameters)
{
    get the child task (custom task function)
    for ( ;; )
    {
        ...
        Update priorities if EDF or MUF
        Run child task function
        Update priorities if EDF or MUF
        ...
    }

}
```

When we create our tasks the first thing happening is that a **SchedulerTaskControlBlock** is created for the specific task and stored in an array. Once the task creation is done, it's time to register them into FreeRTOS task module. Below it's shown how the code handles that:

```
static void RegisterSchedulerTasks(void)
{
    SchedulerTaskControlBlock_t *tmpTask;
    for(BaseType_t i = 0; i < gTaskCounter; i++ )
    {
        configASSERT(gTaskControlBlocks[i].bEmptyBlock == pdFALSE);
        tmpTask = &gTaskControlBlocks[i];
        configASSERT(xTaskCreate(SchedulerParentTaskFunc,
                                 tmpTask->taskName,
                                 tmpTask->stackSize,
                                 tmpTask->parameters,
                                 tmpTask->priority,
                                 tmpTask->taskHandle) == pdPASS);
        ...
    }
}
```

The next step is to update the priorities that are set by the user because we do not want user assigned priorities anymore as per the way FreeRTOS scheduler works. Instead, we want our scheduler module to decide what the task priorities are going to be with respect to some certain factors. See below for **UpdatePriorities()** function…

The algorithms are scheduled as follows. For **Rate Monotonic**, the periods are taken into consideration and the tasks are given priorities accordingly once and for all. This is done by setting the priorities of the array of **SchedulerTaskControlBlock(s)** with respect to their periods. However, for **Earliest Deadline First** algorithm, since it's a dynamic priority assignment scheduling algorithm, which means the absolute deadlines of the tasks can vary relative to the system time, it's necessary to handle this priority assignment both in the beginning and also for

the rest of the program, repetitively. This is handled by both initializing and then updating the priorities of the array of **SchedulerTaskControlBlock(s)** with respect to the absolute deadlines of the tasks. This array is checked and re-updated according to the new deadlines after task executions, When it comes to **Maximum Urgency First**, the scheduler does both, statically assigns the criticality and the user assigned priorities of the tasks and dynamically assigns and re-assigns the least laxity priority of the tasks through out the program. The scheduler also handles it by first initializing and then updating the array of **SchedulerTaskControlBlock(s)** with respect to the least laxity of the tasks. Below is a quick visualization:

```
#define MAX_NUM_TASKS 10
#define RM_SCHEDULING 1       // Rate Monotonic
#define EDF_SCHEDULING 2      // Earliest Deadline First
#define MUF_SCHEDULING 3      // Maximum Urgency First
#define SCHEDULING_ALGORITHM EDF_SCHEDULING

static SchedulerTaskControlBlock_t gTaskControlBlocks[MAX_NUM_TASKS] =
{0};
static BaseType_t gTaskCounter = 0;

void SetInitialPriorities(void)
{
    #if(SCHEDULING_ALGORITHM == RM_SCHEDULING)
        // set the static priorities w.r.t. periods
    #elif(SCHEDULING_ALGORITHM == EDF_SCHEDULING)
        // set the initial priorities w.r.t. deadlines
    #elif(SCHEDULING_ALGORITHM == MUF_SCHEDULING)
        // create the critical set, set the initial priorities
        // w.r.t. least laxity and user priorities
    #else
        configASSERT(pdTRUE == pdFALSE)
    #endif
}
void UpdatePriorities(void)
{
    #if(SCHEDULING_ALGORITHM == EDF_SCHEDULING)
        // update priorities by deadlines
    #elif(SCHEDULING_ALGORITHM == MUF_SCHEDULING)
        // update priorities by least laxity or user priority
    #else
        configASSERT(pdTRUE == pdFALSE)
    #endif
}
// For run time assignments of the priorities
void SetPriorities(void)
{
    for(BaseType_t i = 0; i < gTaskCounter; i++ )
    {
        vTaskPrioritySet(*gTaskControlBlocks[i].taskHandle,
                         gTaskControlBlocks[i].priority);
    }
```

```
}
```

Now, it's time to start the scheduler, our wrapper **SchedulerStart()** sets the priorities of the tasks according to the chosen scheduling algorithm for that specific program run and registers them for the actual start of the scheduling. A quick demo below:

```
static void dispatcherTask(void *parameters)
{
    TickType_t *executionTime = (TickType_t*)parameters;
    TickType_t startTime = xTaskGetTickCount();
    printf("dispatching item...\n");
    for( ;; )
    {
        executionTime[0] = xTaskGetTickCount() - startTime;
        if (executionTime[0] >= executionTime[1]) break;
    }
}

int main(void)
{
    SchedulerInit();

    SchedulerTaskCreate(dispatcherTask,
                        /* Any name for the task */,
                        /* Allowed stack size for the task */,
                        /* Optional parameters for the task */,
                        /* User assigned priority for the task */,
                        /* A pointer to be able to refer to the task */,
                        /* Ready time of the task*/,
                        /* Period of the task */,
                        /* Execution time of the task */);

    SchedulerStart();
}
```

As the final part of the implementation, there is a small dispatcher simulation added to the program. The dispatcher acts as task that dispatches items. The time it takes to dispatch an item per dispatcher, which corresponds to the execution time of a task, can be set by the user together with the period of time per which a dispacth has to be made and the ready time of a dispatcher. The scheduler logs the data for dispatcher to the console for observation. Such as:

```
Dispatcher #1 (SysTime:[0]): Period:[6], AbsDeadline:[6], Priority:[9],
ExecTime:[2]

Dispatcher #1 (Job Description): dispatching item...

Dispatcher #1 (SysTime: [2]): COMPLETED!
```

Since the real time operating system with our scheduler and the dispatcher program is compiled and ran on an emulator, QEMU, the system wide hardware configurations are irrelavant. However, there are software configurations introduced to FreeRTOS that allows the operating system to be configured as if it's hosted on an hardware. General system configurations are as follows:

	FreeRTOS ARM Cortex M-3 PORT (QEMU)
Clock Rate (Max)	20 MHz
Process Speed (Tick Rate)	1000/second
Heap Size	274 Kb
Min. Stack Size per Task	2000 Bytes

Figure 3.1

All the **source code** together with a **user manual** can electronically be found in my GitHub repository [19].

3.5. Experiment

Evaluation and comparison of the implemented algorithms among each other was done by running several set of tasks with the program. For this experiment, all the task sets contain only **periodic tasks** and **deterministic task model** is used. Also **context switching overhead** is neglected. Objectives with respect to the experiment are as follows:

- For a specific task set, demonstrate the behaviour of all three scheduling algorithms.
- For a specific task set, observe the **feasibility** and **deadline fulfillment** with all three scheduling algorithms and compare. Calculate **utilization factors** and validate.
- For each specific algorithm, determination of the maximum number of specific tasks that could be schedulable.
- For a specific task set, compare **context switching potential overhead** between all three scheduling algorithms.

3.5.1. Task sets

Task Set #1	Ready Time (ms)	Period(ms)	Execution Time(ms)	User Assigned Priority (not applicable to RM and EDF)
Dispatcher #1	0	5000	1000	2
Dispatcher #2	0	3000	500	1
Dispatcher #3	0	3000	1000	1
Dispatcher #4	0	4000	800	3
Utilization Factor	0.9			

Figure 3.2

Task Set #2	Ready Time (ms)	Period(ms)	Execution Time(ms)	User Assigned Priority (not applicable to RM and EDF)
Dispatcher #1	0	7000	700	1
Dispatcher #2	0	10000	1000	2
Dispatcher #3	0	2500	1000	3
Dispatcher #4	0	5000	500	4
Utilization Factor	0.7			

Figure 3.3.

Task Set #3	Ready Time (ms)	Period(ms)	Execution Time(ms)	User Assigned Priority (not applicable to RM and EDF)
Dispatcher #1	0	6000	2000	1
Dispatcher #2	0	8000	2000	2
Dispatcher #3	0	12000	3000	3
Utilization Factor	0.83			

Figure 3.4

Task Set #4	Ready Time (ms)	Period(ms)	Execution Time(ms)	User Assigned Priority (not applicable to RM and EDF)
Dispatcher #1	0	6000	2000	1
Dispatcher #2	0	8000	5000	2
Dispatcher #3	0	12000	3000	3
Utilization Factor	1.20			

Figure 3.5

Task set #1 utilizes the processor at **90%**. Therefore, it is used for the demonstration of **Rate Monotonic** and **Earliest Deadline First** algorithm behaviours and failure of **Rate Monotonic** algorithm.

Task set #2 utilizes a relatively less portion of the processor. Therefore, it is used for a feasible execution with **Rate Monotonic** algorithm.

Task set #3 is used for measuring context switches of all three algorithms in a closed time interval (0-28 seconds).

Finally, the task set #4 is used for the demonstration of **Maximum Urgency First** algorithm and it's superiorities compared to other algorithms.

3.5.2. Demonstration of Algorithms

Rate Monotonic with Task Set #1:

The **Rate Monotonic** scheduler was able to correctly assign static priorities to the tasks.
Program run (**Time unit is in seconds in console outputs**):

```
taskName: Dispatcher #2, period: 3, deadline: 3, priority: 9
taskName: Dispatcher #3, period: 3, deadline: 3, priority: 8
taskName: Dispatcher #4, period: 4, deadline: 4, priority: 7
taskName: Dispatcher #1, period: 5, deadline: 5, priority: 6
```

The scheduler was able to also execute tasks in the correct order.

```
Dispatcher #2 (SysTime:[1]): Period:[3], Deadline:[3], AbsDeadline:[3],
               Priority:[9], ExecTime:[0.5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [0.5]): COMPLETED!

Dispatcher #3 (SysTime:[0.5]): Period:[3], Deadline:[3],
               AbsDeadline:[3], Priority:[8], ExecTime:[1]
Dispatcher #3 (Job Description): dispatching item...
Dispatcher #3 (SysTime: [1.5]): COMPLETED!

Dispatcher #4 (SysTime:[1.5]): Period:[4], Deadline:[4],
               AbsDeadline:[4], Priority:[7], ExecTime:[0.8]
Dispatcher #4 (Job Description): dispatching item...
Dispatcher #4 (SysTime: [2.3]): COMPLETED!

Dispatcher #1 (SysTime:[2.3]): Period:[5], Deadline:[5],
               AbsDeadline:[5], Priority:[6], ExecTime:[1]
```

```
Dispatcher #1 (Job Description): dispatching item...
```

Correctly preempts according to priorities. Before the **Dispatcher #1** completes it's execution, **Rate Monotonic** algorithm preempts it at **0.7s** of it's execution and switches to **Dispatcher #2**.

```
Dispatcher #2 (SysTime:[3]): Period:[3], Deadline:[3],
               AbsDeadline:[6], Priority:[9], ExecTime:[0.5]
```

Earliest Deadline First with Task Set #1:

The **Earliest Deadline First** scheduler was able to correctly assign priorities to the tasks in the beginning.

Program run (**Time unit is in seconds in console outputs**):

```
taskName: Dispatcher #2, period: 3, deadline: 3, priority: 9
taskName: Dispatcher #3, period: 3, deadline: 3, priority: 8
taskName: Dispatcher #4, period: 4, deadline: 4, priority: 7
taskName: Dispatcher #1, period: 5, deadline: 5, priority: 6
```

The scheduler was able to also execute tasks in the correct order. Apart from that, as it can be seen, **Earliest Deadline First** algorithm repetitively updates the priorities according to the absolute deadlines of the tasks.

```
Dispatcher #2 (SysTime:[0]): Period:[3], Deadline:[3], AbsDeadline:[3],
               Priority:[9], ExecTime:[0.5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [0.5]): COMPLETED!


Dispatcher #3 (SysTime:[0.5]): Period:[3], Deadline:[3],
               AbsDeadline:[3], Priority:[9], ExecTime:[1]
Dispatcher #3 (Job Description): dispatching item...
Dispatcher #3 (SysTime: [1.5]): COMPLETED!


Dispatcher #4 (SysTime:[1.5]): Period:[4], Deadline:[4],
               AbsDeadline:[4], Priority:[9], ExecTime:[0.8],
Dispatcher #4 (Job Description): dispatching item...
Dispatcher #4 (SysTime: [2.3]): COMPLETED!


Dispatcher #1 (SysTime:[2.3]): Period:[5], Deadline:[5],
               AbsDeadline:[5], Priority:[9], ExecTime:[1],
Dispatcher #1 (Job Description): dispatching item...
Dispatcher #1 (SysTime: [3.3]): COMPLETED!...
```

Maximum Urgency First with Task Set #4:

Below we show that **Maximum Urgency First** algorithm correctly assignes the priorities with respect to **laxity** of the tasks together with the specification of the critical task set.

Program run (**Time unit is in seconds in console outputs**):

```
taskName: Dispatcher #2, period: 8, deadline: 8, priority: 9, crit: True
taskName: Dispatcher #1, period: 6, deadline: 6, priority: 8, crit: True
taskName: Dispatcher #3, period: 12, deadline: 12, priority: 7, crit:
False
```

The scheduler was able to also execute tasks in the correct order. Apart from that, as it can be seen, **Maximum Urgency First** algorithm repetitively updates the priorities according to the **laxity** of the tasks. Below we can see that **Maximum Urgency First** algorithm is able to schedule **Dispatcher #1** and **Dispatcher #2** (critical tasks) without missing any deadline of theirs.

```
Dispatcher #2 (SysTime:[0]): Period:[8], Deadline:[8], AbsDeadline:[8],
               Priority:[9], ExecTime:[5]
Dispatcher #2 (Job Description): dispatching item...

Dispatcher #1 (SysTime:[2]): Period:[6], Deadline:[6],
               AbsDeadline:[6], Priority:[9], ExecTime:[2]
Dispatcher #1 (Job Description): dispatching item...
Dispatcher #1 (SysTime: [4]): COMPLETED!

Dispatcher #2 (SysTime: [7]): COMPLETED!

Dispatcher #1 (SysTime:[7]): Period:[6], Deadline:[6],
               AbsDeadline:[12], Priority:[9], ExecTime:[2]
Dispatcher #1 (Job Description): dispatching item...
Dispatcher #1 (SysTime: [9]): COMPLETED!

Dispatcher #2 (SysTime:[9]): Period:[8], Deadline:[8], AbsDeadline:[16],
               Priority:[9], ExecTime:[5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [14]): COMPLETED!
...
```

3.5.3. Deadline Fulfillment and Feasibility

Rate Monotonic with Task Set #1:

We calculate the utilization bound for the number of tasks for **Rate Monotonic** algorithm with the following formula [4]:

$$n\left(2^{1/n}-1\right)$$

Number of tasks for the **task set #1** being 4, gives us the utilization bound of **0.756** for **Rate Monotonic**.
The utilization factor of the task set has to be less than or equal to this value so that **Rate Monotonic** can be feasible. However, as we see it's not the case. Therefore, we expect missed deadline(s).

Program run (**Time unit is in seconds in console outputs**):

The first task to be picked is expected to be the **Dispatcher #2**, and it's picked up and executed.

```
Dispatcher #2 (SysTime:[0]): Period:[3], Deadline:[3], AbsDeadline:[3],
               Priority:[9], ExecTime:[0.5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [0.5]): COMPLETED!
```

..and run continues.

Here at this point when the **Dispatcher #1** is picked up. It's preempted after **0.7** seconds of execution because there are other tasks that are within their new period and are with a higher priority.

```
Dispatcher #1 (SysTime:[2.3]): Period:[5], Deadline:[5],
               AbsDeadline:[5], Priority:[6], ExecTime:[1],
Dispatcher #1 (Job Description): dispatching item...

Dispatcher #2 (SysTime:[3]): Period:[3], Deadline:[3], AbsDeadline:[6]
               Priority:[9], ExecTime:[0.5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [3.5]): COMPLETED!

Dispatcher #3 (SysTime:[3.5]): Period:[3], Deadline:[3],
               AbsDeadline:[6], Priority:[8], ExecTime:[1]
Dispatcher #3 (Job Description): dispatching item...
Dispatcher #3 (SysTime: [4.5]): COMPLETED!

Dispatcher #4 (SysTime:[4.5]): Period:[4], Deadline:[4],
               AbsDeadline:[8], Priority:[7], ExecTime:[0.8]
Dispatcher #4 (Job Description): dispatching item...
```

```
Dispatcher #4 (SysTime: [5.3]): COMPLETED!
```

After the preemption of **Dispatcher #1**, it continues it's execution at **5.3s** system time and finishes at **5.6s** system time. However, it's deadline was until **5s**. Therefore, we can see that the **Rate Monotonic** indeed missed a deadline.

```
Dispatcher #1 (SysTime: [5.6]): COMPLETED!
```

Rate Monotonic with Task Set #2:

The utilization factor for the **task set #2** is **0.7**. Which is less than the utilization bound of **Rate Monotonic** algorithm with respect to the task set. Therefore, it's expected to fulfill all the deadlines. We can observe that by running the program with the given task set. Since all our tasks periodically run forever, the console output is not shared here.

Earliest Deadline First with Task Set #1:

We know that for **Earliest Deadline First** algorithm as long as the utilization factor for the task set that is being executed is less than 1.0 (100%), the system is feasible. Therefore, since the **task set #1** has a utilization factor of 0.9, **Earliest Deadline First** should be able to schedule the tasks without missing any deadline.

Let's take a look at the exact same part of the program run where **Rate Monotonic** algorithm failed to schedule and miss a deadline.

Program run (**Time unit is in seconds in console outputs**):

```
Dispatcher #1 (SysTime:[2.3]): Period:[5], Deadline:[5],
               AbsDeadline:[5], Priority:[9], ExecTime:[1]
Dispatcher #1 (Job Description): dispatching item...
Dispatcher #1 (SysTime: [3.3]): COMPLETED!
```

As it can be seen here, unlike **Rate Monotonic** algorithm, **Earliest Deadline First** didn't preempt **Dispatcher #1** because it has the earliest deadline and so it has the highest priority. The task completed and the run didn't miss a single deadline.

```
Dispatcher #2 (SysTime:[3.3]): Period:[3], Deadline:[3],
               AbsDeadline:[6], Priority:[9], ExecTime:[0.5]
Dispatcher #2 (Job Description): dispatching item...
Dispatcher #2 (SysTime: [3.8]): COMPLETED!


Dispatcher #3 (SysTime:[3.8]): Period:[3], Deadline:[3],
               AbsDeadline:[6], Priority:[9], ExecTime:[1]
```

```
Dispatcher #3 (Job Description): dispatching item...
Dispatcher #3 (SysTime: [4.8]): COMPLETED!


Dispatcher #4 (SysTime:[4.8]): Period:[4], Deadline:[4],
               AbsDeadline:[8], Priority:[9], ExecTime:[0.8]
Dispatcher #4 (Job Description): dispatching item...
Dispatcher #4 (SysTime: [5.6]): COMPLETED!
```

Maximum Urgency First with Task Set #4:

The utilization factor of the task set #4 is 1.20. Which is above 100% of cpu usage by the three tasks. In such a task model, all three (RM, EDF and MUF) algorithms fail to schedule all the tasks. However, **Maximum Urgency First** algorithm does something different. **Maximum Urgency First** algorithm creates a **critical subset** of the task set that it can schedule 100%. Therefore, **Dispatcher #1** and **Dispatcher #2** are considered as the critical tasks according to **Maximum Urgency First** algorithm since they have the lowest period and it will schedule them without missing any deadline of theirs.

On the other hand, **Earliest Deadline First** does it in an unstable way, which means we do not know when and which of these three tasks will miss a deadline.

When it comes to the difference from **Rate Monotonic** which would also guarantee the schedulability of the critical set of tasks, there is a catch. The tasks in **Rate Monotonic** have only static priority and therefore it doesn't take into consideration the factors like **laxity**. **Rate Monotonic** starts the execution of the tasks with **Dispatcher #1** and fails to schedule **Dispatcher #2** whereas **Maximum Urgency First** starts with **Dispatcher #2**, then switch to **Dispatcher #1** before **Dispatcher #1's** deadline and then back to **Dispatcher #2**. Therefore, **Maximum Urgency First** doesn't miss any deadline of the critical set.

The program run can be found in the **Section 3.5.2. Demonstration of Algorithms, Maxium Urgency First with Task Set #4**.

3.5.4. Schedulability Limit

The number of tasks that could be schedulable by a specific algorithm depends on the **utility factor** of the task set and the **utility bound** of the scheduling algorithm. It's possible to execute a set of any tasks as long as the utility factor of the task set doesn't exceed the utility bounds of the scheduling algorithm.

Utilization factor of a task set is shown by the following symbol μ and calculated as follows:

$$\mu = \sum \frac{C_i}{T_i}$$

Figure 3.6 [4].

Where C_i is the execution time of the task,
T_i is the period of the task [4].

For **rate monotonic** algorithm we have the following formula:

$$\mu = \sum \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Figure 3.7 [4]

Where C_i is the execution time of the task,
T_i is the period of the task and
n is the number of tasks.

Any arbitrary number of tasks that holds that inequality true will be schedulable by the **Rate Monotonic** algorithm.

For the **Earliest Deadline First** algorithm, since it's an optimal one, we have the following formula in order to find out whether a task set will be schedulable or not:

$$\mu = \sum \frac{C_i}{T_i} \leq 1$$

Figure 3.8 [4].

Where C_i is the execution time of the task,
T_i is the period of the task.

Again, any arbitrary number of tasks of which has a utilization factor of less than or equal to 1, are going to be feasible by Earliest Deadline First algorithm.

Since **Maximum Urgency First** algorithm has characteristics of the both above other algorithms, the amout of tasks that could be feasibly executed is calculated in a similar way. There is one huge difference and it's the critical set is still guaranteed above 100% of CPU utilization. That's due to the fact that **Maximum Urgency First** algorithm assigns a subset of tasks to a critical set that is guaranteed to be feasible.

3.5.5. Potential Context Switch Overhead

We say potential because in the execution of programs we neglected context switching overhead. However, that's a pretty important aspect when it comes to real time scheduling algorithms. Therefore, we compared all three of our algorithms with **the task set #3**. The results were as follows:

Context switching was measured in amount of occurrence starting from the system time **0s** up **until 28th s**,

- Rate Monotonic algorithm switched context **13 times**.
- Earliest Deadline First algorithm switched context **11 times**.
- Maximum Urgency First algorithm switched context **13 times**.

3.6. Results

	Task Set #1	Task Set #2	Task Set #3	Task Set #4
FreeRTOS Static Preemptive Priority	**Not Feasible**	**Feasible**	**Not Feasible**	**Not Feasible**
Rate Monotonic	**Not Feasible**	**Feasible**	**Not Feasible**	**Not Feasible**
Earliest Deadline First	**Feasible**	**Feasible**	**Feasible**	**Not Feasible**
Maximum Urgency First	**Feasible**	**Feasible**	**Feasible**	**Feasible (only critical set)**

Figure 3.9

Chapter 4: Conclusion

In particular, Rate Monotonic is a straightforward yet effective scheduling approach. Because of the assigned static priorities, if a job is going to miss a deadline, it will be the one with the lowest priority. Utilization bound is not as strong as the other two algorithms, however Rate Monotonic is a stable algorithm. Given the design constraints and task model together with the feasibility analysis, Rate Monotonic will be able to schedule at least a certain subset of the task set.

On the other hand, Earliest Deadline First algorithm is an optimal dynamic priority assignment algorithm with a utilization bound of below 100%. Which means any set of task which have a utilization factor of less than 100% is going to be feasibly executed, which is great. It also features the lowest potential for context switching overhead. However, Earliest Deadline First algorithm is a bit more complicated to implement compared to Rate Monotonic algorithm and it's also not a stable algorithm.

Moreover, Maximum Urgency First algorithm is a mixture of the above two other algorithms. It's definitely way more complicated to implement compared to other algorithms. The most powerful feature of Maximum Urgency First is that it's a stable algorithm which guarantees a 100% feasibility for the critical subset of tasks it creates out of the original set of tasks that are intended to be executed at above 100% utilization. Furthermore, it also yields a way better scheduling than static priority algorithms since it takes into consideration the laxity of the tasks, dynamically.

Then there is the First Come First Served algorithm. According to the studies this algorithm is definitely more centered for general purpose operating systems. That's because the algorithm literally only takes into consideration one factor; how early the task has become ready. Deadlines, periods, execution times of the tasks are not taken into consideration at all. Therefore, this algorithm is not a good choice for real time embedded systems.

As per the final of the scheduling algorithms; Round Robin scheduling approach is for sure a very popular method used in many systems. However, due to it's design principle, it's no good use for real time embedded systems. Again, that's due to the fact that this algorithm only operates with respect to the defined time quantum. If it's a task's turn to be executed, then it will be. Otherwise, it will have to wait even though it might have a higher priority for the system.

Real time scheduler algorithms Rate Monotonic being static, Earliest Deadline First being dynamic and Maximum Urgency First being a mixture of the other two are quite appropriate choices for real time embedded systems. We were able to see that first in our study of the topic, then later by implementing and experimenting with the algorithms. Although each of them has a few advantages and drawbacks.

Choosing the best algorithm for a set of tasks is not a trivial problem. In this thesis work, findings with regards to that was as follows:

As per the formulas see Figure 2.1 and Figure 2.2, we know that Rate Monotonic algorithm's utilization bound changes according to the number of tasks. When the number of tasks is two, the utilization bound is the highest at 0.830, whereas when the Liu Layland limit is used and utilization bound is calculated as the number of tasks are approaching to infinity, the utilization bound is 0.693. That shows us that when there are more tasks for Rate Monotonic algorithm, it's less likely to be feasible. Rate Monotonic algorithm would be the ideal choice when there is a need to have a simple and effective scheduling method and when there is a relatively small set of tasks with the appropriate utilization factor. On the other hand, according to the formula, see Figure 2.4, the utilization bound of Earliest Deadline First is pretty high together with Maximum Urgency First algorithm. When there is a set of tasks with a very high (still below 100%) utilization factor, Earliest Deadline First or Maximum Urgency First would be the ideal solution. However, since the implementation of Earliest Deadline First is much simpler compared to Maximum Urgency First algorithm's implementation, for a task set with a high utilization factor below 100%, Earliest Deadline First will do the job perfectly. When it comes to a set of task with a utilization factor of above 100%, then Maximum Urgency First is by far the best approach to take. That's because it guarantees the schedulability of the critical set at the least even above 100% utilization factor.

When it comes to implementing scheduler algorithms for an arbitrary purpose, we saw that FreeRTOS, an open source real time operating system combined with QEMU, an emulator for microcontroller architectures makes a great combination of tools in order to execute embedded applications on a real time operating system without having to own any hardware.

The evaluation of the algorithms with respect to predetermined design constraints showed us that there is no single algorithm that can be the best at everything, at least yet. Therefore, one of the most important aspects to real time scheduling algorithms that is applicable to any embedded system is to be as much acknowledged as possible about the task model and design constraints of the system. Only after then we are able to tell what algorithm would be better suitable for the job.

Chapter 5: Bibliography

[1] Wikipedia contributors. (2021, May 26). Real-time operating system. In *Wikipedia, The Free Encyclopedia*. Retrieved 22:57, June 17, 2021, Available: https://en.wikipedia.org/w/index.php?title=Realtime_operating_system&oldid=1025214460

[2] Arezou Mohammadi and Selim G. Akl. Scheduling Algorithms for Real-Time Systems. [online]. Kingston, Ontario, Canada. Queen's University, School of Computing. 15 July 2005. [viewed 16 June 2021]. Available from: https://research.cs.queensu.ca/home/akl/techreports/scheduling.pdf

[3] Wikipedia contributors. (2021, June 12). Scheduling (computing). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:20, June 17, 2021, Available: https://en.wikipedia.org/w/index.php?title=Scheduling_(computing)&oldid=1028274434

[4] E. A. Lee and S. A. Seshia, Introduction to Embedded Systems - A Cyber-Physical Systems Approach, Second Edition, MIT Press, 2017

[5] Jane W. S. Liu, Real-Time Systems, First Edition, Prentice Hall, 2000, ISBN: 0130996513

[6] Wikipedia contributors. (2021, June 10). System. In *Wikipedia, The Free Encyclopedia*. Retrieved 22:15, June 21, 2021, Available: https://en.wikipedia.org/w/index.php?title=System&oldid=1027870566

[7] Wikipedia contributors. (2021, June 15). Central processing unit. In *Wikipedia, The Free Encyclopedia*. Retrieved 22:21, June 21, 2021, Available: https://en.wikipedia.org/w/index.php?title=Central_processing_unit&oldid=1028678287

[8] Kopetz. H, Real Time Systems – Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997

[9] Khalil Mustafa Yousef and Nabeel Barakat. Embedded Systems Task Scheduling Algorithms and Deterministic Behaviour [online]. Jordan University of Science and Technology, Faculty of Computer and Information Technology, Dept. of Computer Engineering. 18 November 2006. [viewed 16 June 2021]. Available: https://www.just.edu.jo/~tawalbeh/cpe746/slides/Scheduling_Algorithms.pdf

[10] Alberto Daniel Ferrari, Real Time Scheduling Algorithms, Achieving predictability for critical applications [online]. Laboratorio de Controle e Microinformatica at the Universidade Federal de Santa Catarina in Florianopolis, Brazil. 1994. [viewed 16 June 2021]. Available: http://dns.uls.cl/~ej/daa_08/Algoritmos/books/book10/9412f/9412f.htm#0206_00ae

[11] Wikipedia contributors. (2021, May 12). Rate-monotonic scheduling. In *Wikipedia, The Free Encyclopedia*. Retrieved 18:04, June 22, 2021, Available: https://en.wikipedia.org/w/index.php?title=Ratemonotonic_scheduling&oldid=1022812496

[12] Hana Kubátová, 2017, Real Time Systems, Real Time Task Scheduling, BIE-SRC, Lecture 6, Department of Digital Design Faculty of Information Technology Czech Technical University in Prague

[13] Ed Overton, A Foray into Uniprocessor Real Time Scheduling Algorithms and Intractibility, 1997 December 3, Available: http://www.cs.unc.edu/~anderson/diss/overton.pdf

[14] Wikipedia contributors. (2021, June 17). Scheduling (computing). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:46, June 22, 2021, Available: https://en.wikipedia.org/w/index.php?title=Scheduling_(computing)&oldid=1029097341

[15] Wikipedia contributors. (2021, June 21). Round-robin scheduling. In *Wikipedia, The Free Encyclopedia*. Retrieved 23:43, June 22, 2021, Available: https://en.wikipedia.org/w/index.php?title=Roundrobin_scheduling&oldid=1029654349

[16] FreeRTOS API References, available: https://www.freertos.org/a00106.html

[17] FreeRTOS Documentation: available: https://www.freertos.org/features.html

[18] Robin Kase, Efficient Scheduling Library for FreeRTOS, Master's Thesis at KTH Information and Communication Technology, Stockholm, Sweden, 2016. Available from: http://www.diva-portal.org/smash/get/diva2:1085303/FULLTEXT01.pdf

[19] Aykut Sahin, real time scheduler algorithms, https://github.com/aykut4/real-time-scheduling-algorithms, 2021

Appendix

Appendix A: Structure of the Memory Medium

|Thesis.pdf
|License
|Source
|Readme (Users manual)
|App (Source code)
|---CORTEX_M3_MPS2_QEMU_GCC
|---CMSIS
|---init
|---scripts
|---FreeRTOSConfig.h
|---Makefile
|---Readme
|---main.c
|---main_app.c
|---scheduler.h
|---scheduler.c
|---syscall.c