SEMINAR REPORT
Seminar: Embedded Systems in Summer term 2022

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

A Comparison of Real-time Operating Systems for Embedded Computing

Yoshua Neuhard
Technische Universität Kaiserslautern, Department of Computer Science

Many different comparisons of real-time operating systems (RTOS) have been done. More often than not, these comparisons are specific to a small set of operating systems as well as the experimental setup. The choice of parameters is directly influenced by the systems being compared and the experimental setup. Therefore different comparisons use vastly different parameters. The same applies to the findings of different comparisons. Some microcontrollers can handle a specific RTOS better than other controllers or other RTOS. It is therefore difficult for a developer to choose a specific RTOS for the intended use in an embedded system. This work gives a general introduction to RTOS and details the main points of three selected RTOS µC/OS-II, RTLinux and FreeRTOS. It continues by outlining a common parameter basis to compare different RTOS by and how different comparisons of RTOS can be used to compare their capabilities for a different setup. The paper concludes by giving a brief outlook on future technology, which try to improve RTOS to be able to handle multicore processors.

1 Introduction

Embedded systems have become ever more important in the last decades and are widely available for a range of different purposes. Most cellphones, cars or even planes use different kinds of embedded systems, for example for managing media like radio or as a navigation system in a vehicle. These systems often need to operate on very limited hardware in terms of memory and computation power. Classic general purpose operating systems (GPOS) are not useable on this kind of systems due to their need of memory and lack of real-time computation capabilities. A more basic, less memory heavy system has to be used.

Real-time Operating Systems (RTOS) are such systems. They often only consist of a basic task management and a priority based scheduler. RTOS not only have an incredibly small memory footprint, but also have real-time computation capabilities. Their specific implementation is mostly specialised to the hardware used for the embedded system. Hence these operating systems are used in small embedded system, which are found in cellphones, cars or even in planes and power plants. Especially in aviation and power plants the systems need to be safety critical. A RTOS needs to take special care when scheduling tasks, to not miss a safety critical deadline. For most RTOS it is therefore imperative to be able to handle and also recover from certain failures or unexpected inputs example given from sensors.

Due to their highly specialised nature, a RTOS needs to be compared in either the special use case, which the developer intends to use the systems for, or in different application situation to find a good average.

Recent work [1], [4], [5], [8], [9] and many more compared different kinds of RTOS against each other in terms of varying parameters. More often than not a very specific setup, hardware or parameters to compare the operating systems are used by these papers. This work attempts to outline a common parameter baseline based on recent work in [1], [4], [5], [8] and [9] to compare different RTOS. Therefore this paper is structured as follows:

Section 2 gives a general background by explaining what RTOS are and how they work. Especially scheduling and the problem of priority inversion are explained. Furthermore the main part of the RTOS the kernel is briefly mentioned.

Section 3 introduces a few selected RTOS, which are compared in [1], [4], [5], [8] and [9]. Each of the Operating Systems is briefly described in their general structure. Additionally their solutions to the proposed problems in Section 2 are described and their specific scheduler and kernel analysed.

Section 4 tries to give a common ground between comparisons of the different papers. It starts with an analysis of the parameters and gives some selected parameters to compare the presented RTOS. Subsequently a comparison with the selected parameters of the proposed RTOS is done with the data of the respective papers.

Section 5 gives an overview over future development and improvements to RTOS. It also briefly describes a version of FreeRTOS, which has been adapted for multicore computation [5].

Section 6 concludes the paper and summarizes the proposed common ground for comparing different RTOS with data from already existent studies.

2 Technical Background

While GPOS not only have a significant larger memory footprint, they also lack real-time computation capabilities. Contrary to RTOS, where the scheduling is based around priorities and critical deadlines, a GPOS takes a fair approach to scheduling. It tries to give each task the same amount of resources and splits the computation power equally among tasks. Due to the fair scheduling approach a GPOS won't be able to guarantee a deadline for a certain task. RTOS with their, often preemptive, priority based scheduling are able to guarantee that a task is able to meet its deadline.

In general most RTOS differentiate between a hard and a soft deadline, also called hard or soft real-time. A RTOS, which tries to fulfill a hard deadline, won't be able to fully recover from failure, if it misses the deadline. These hard deadlines are mostly used for safety critical systems, where failure can have catastrophical outcomes. If a RTOS misses a soft deadline, it can recover from certain failures. These deadlines are mostly used for example given media streams or similar systems, that are not safety critical. Missing a deadline in these systems will often only be a minor inconvenience, but doesn't have the catastrophic outcomes a safety critical system has.

Some RTOS also use a kernel, similar to GPOS. However they are vastly different, as they need to be much smaller in size and computation overhead. A Kernel in a RTOS is often very modular and can be adjusted to be specialised for the exact hardware and purpose of the system. There are a few different kind of kernels, that find use in RTOS, which I will describe in Section 2.3

In the following section I will give a brief overview of RTOS and their used technologies and problems, that may arise whilst using them. I will start by explaining how RTOS work in general and the different scheduling methods, which are typically used to schedule tasks in a RTOS. Afterwards I explain the problem of priority inversion, how it arises and what strategies are to solve it. Lastly I explain the purpose of the kernel in a RTOS.

2.1 Real-time Operating Systems

A RTOS is generally a minimal operating system, which runs on small embedded devices with very limited memory and computation power. Such operating systems therefore have to be small in size and have a manageable overhead for execution on the embedded system. They can be used example given for processing sensor data, managing media streams or even be responsible for the airbag system in a car [4].

Small programs or functions, which are executed by a RTOS, are called tasks. Because these operating systems use priority based scheduling, each task gets a priority assigned. The priority of a task determines the execution order since a RTOS selects the task with the highest priority and executes it. Therefore developer should take special care when assigning priorities to tasks, since the performance of an RTOS directly depends on them.

Some RTOS support so-called interrupt service routines (ISR), which allow to interrupt the current task and execute a interrupt handler. These handler are most of the time some other function or task, which gets executed immediately. To be able to switch tasks and not loose all data on the computations of the suspended task, the RTOS needs to save the current state of the task. After the interrupt handler finishes, the RTOS switches back to the previously suspended task, restores the data of it and continues its execution. This is also called context switching, as the system switches from one tasks context (processor state, data in memory) to another.

Tasks can also hold so-called semaphores for resources to execute a critical section, where it shouldn't be interrupted. These prevent any other task, apart from the task holding the semaphore, from accessing a specific resource. They also prevent another task from executing a section, if this section of the task needs the resource the semaphore is blocking. Unless the task, which is holding the semaphore, releases it, the task requesting the semaphore cannot execute. However blocking resources, while useful at first glance, leads to the problem of priority inversion. This will be explained in detail in Section 2.2.1.

RTOS have to have a good scheduling approach to achieve a high performance and also need to solve problems with priority based scheduling. The next two subsections give a brief overview over scheduling techniques, as well as the problem of priority inversion and solutions to it.

2.2 Scheduling

Contrary to GPOS, that schedule in a more "fair" way, where each task gets similar resources, RTOS mostly use preemptive priority based scheduling. As mentioned in the previous section, each tasks gets a priority assigned by the developer. The RTOS then selects the task with the highest priority, when choosing which task to execute. A new task to execute gets chosen, when:

- system is in idle and a new task gets scheduled
- a task finished a critical section
- a task finished executing

In some cases RTOS also utilize other scheduling algorithms, example given when deciding between tasks with equal priority. Round robin scheduling is a technique, which is the closest to GPOS. Here the tasks get an equal share of cpu time by assigning a specific time slot of the cpu to a specific task. This leads to a very high number of context switches in a RTOS and will likely result in a drop of performance, when a high number of tasks have to be scheduled.

First in first out(FIFO) scheduling utilizes the concept of a FIFO queue. It simply schedules the task, which was added first to the queue of tasks, and executes it. This tasks runs until either it finishes execution or a task with a higher priority gets scheduled.

Earliest deadline first (EDF) scheduling takes the task with the earliest deadline and schedules it. Figure 1 shows an example of EDF scheduling. Here you can also see how tasks are interrupted by one another, when a new task with an earlier deadline is released.

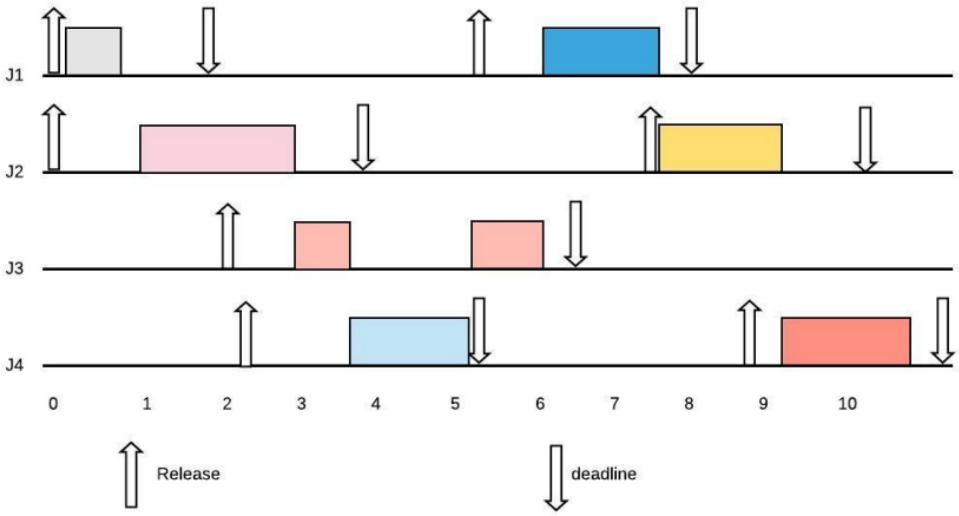

Figure 1: EDF Scheduling. Adapted from [8]

Scheduling algorithms used in RTOS are often preemptive. This means, that a task with a higher priority can claim resources, which are currently used by a lower priority task, and execute its code. Resources blocked by semaphores can't be claimed by a higher priority task, even when the task holding the semaphore has a lower priority. This can lead to the problem of priority inversion. The next section explains how priority inversion can occur and what different strategies exist to solve it.

2.2.1 Priority Inversion

Priority Inversion occurs when a lower priority task prevents a higher priority task from executing. This occurs, when a higher priority task tries to take a semaphore, which is held by a lower priority task. Figure 2 shows a basic example of a easily solvable priority inversion. In this figure task 1 has the higher priority and delays its execution, which enables task 2 with a lower priority to execute. During its execution task 2 takes the semaphore *S1*. After the delay of task 1 has expired, it is executed again and tries to take semaphore *S1*. *S1* is however held by task 2, which in turn blocks the execution of task 1.

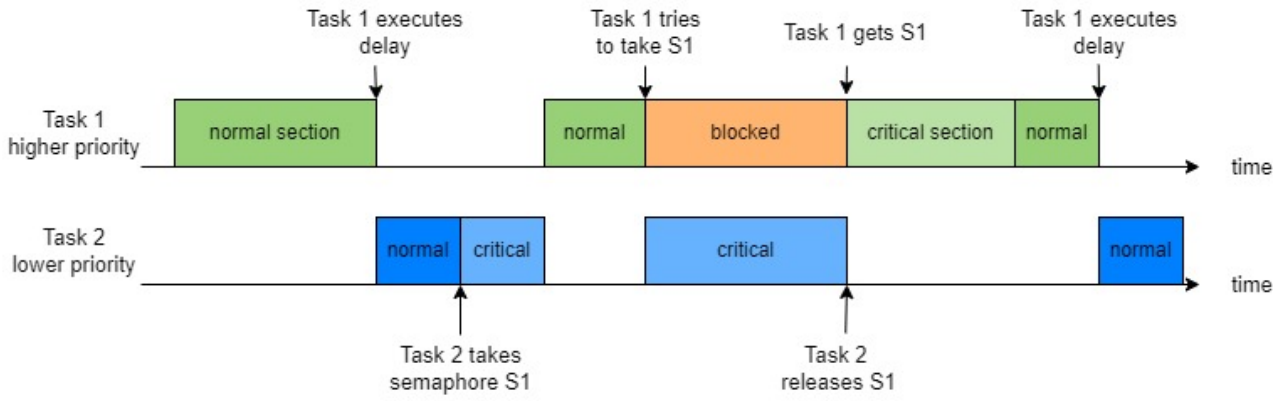

Figure 2: Priority Inversion Example 1

Another example, which is harder to solve, is shown in Figure 3. After task 1, which has the highest priority, and task 2, which has a lower priority, executed a delay, task 3 with the lowest priority executes. Task 3 takes semaphore *S1* and tries to execute its critical section. After the delay of task 1 expired, it gets executed again and tries to take semaphore *S1*. Task 3 currently holds *S1* and therefore blocks the execution of task 1. However, task 3 can't execute as the delay of task 2 also expired and task 2 has a higher priority than task 3. In this case a priority inversion occurs, because task 2 indirectly blocks the execution of task 1 as task 3 can't complete its critical section and release semaphore *S1*.

Therefore the blocked section, marked as a red block on the time of task 1 in Figure 3, can be arbitrary long. This occurs because task 2 or even other tasks with a higher priority than task 3 will execute and the system can't know exactly when task 3 can be executed again. Therefore task 1 is blocked for an unknown amount of time.

There are many different strategies to solve or prevent priority inversion [2]. In this section, I will explain two selected ones of them in detail:

1. Disabling Preemption
2. Priority Inheritance

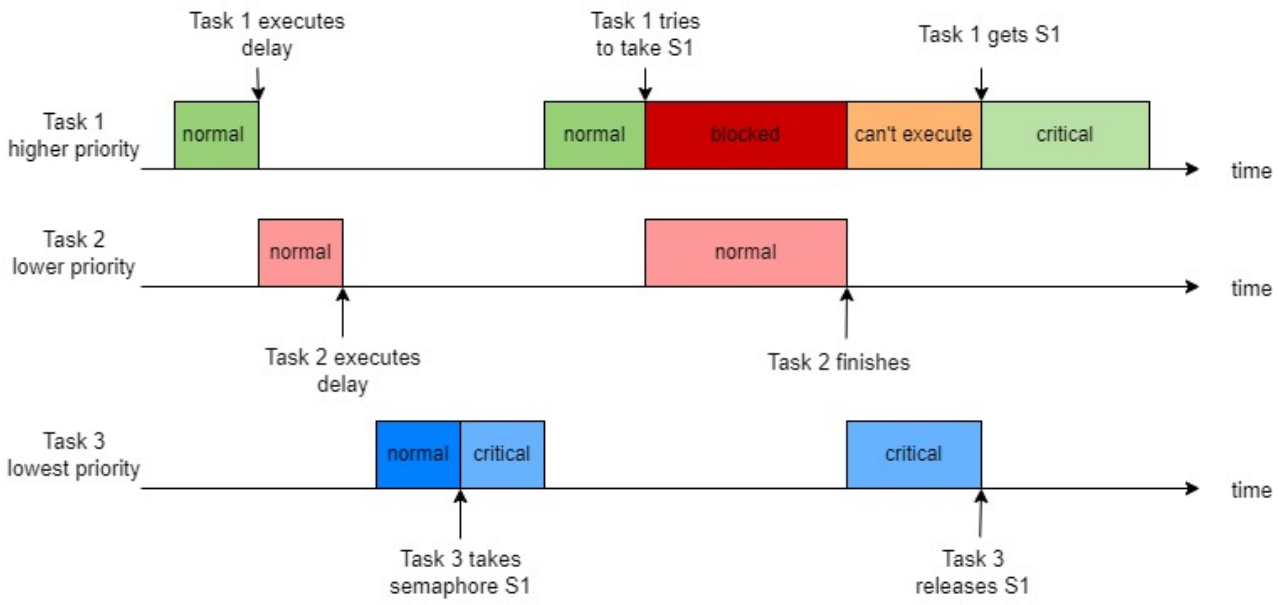

Figure 3: Priority Inversion Example 2

The first of the two strategies is to disable preemption during the execution of a critical section of a task. Figure 4 shows the example, which is displayed in Figure 2, when disabling preemption. After the delay of task 1 expires, it gets blocked from execution as task 2 is currently executing its critical section. When task 2 finishes its critical section, it releases semaphore *S1* and task 1 executes again, because it has the higher priority.

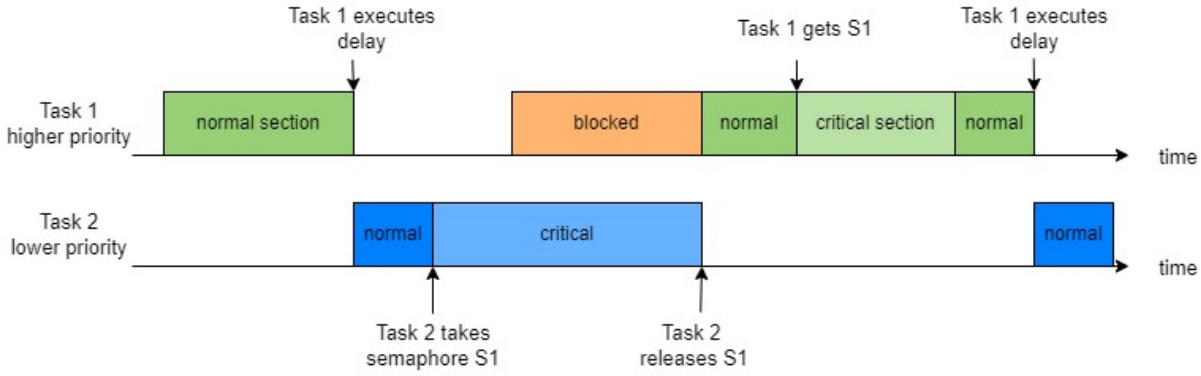

Figure 4: Priority Inversion Example 1 using Disable Preemption

Example 2, which is displayed in Figure 3, changes in a similar way when disabling preemption. Task 3 won't stop the execution of its critical section. After task 3 finishes its critical section, task 1 will execute and will not be blocked by task 2. Task 2 will execute after task 1 finishes, because task 1 has the highest priority.

Priority inheritance describes the second of the two strategies to solve priority inversion. When a task A blocks the execution of one or more higher priority tasks, A inherits the highest priority among the tasks, that A blocks [2]. In case of the example in Figure 2, task 2 would inherit the higher priority of task 1 during the execution of its critical section. However task 2 would only inherit the higher priority as soon as task 1 tries to take semaphore *S1*. This applies to example 2 in Figure 3 in a similar way. Here task 3 would inherit the priority of task 1 as soon as task 1 tries to take semaphore *S1*. Therefore task 3 would execute its critical section, instead of task 2 blocking it from executing. Afterwards task 1 would resume executing as it is the task with the highest priority.

However priority inheritance comes with its own drawbacks. For example cyclic dependencies would result in a deadlock. Figure5 shows an example, where the priority inheritance results in a deadlock. The lower priority task takes semaphore *S1* and starts its critical section. Then task 1, which has higher priority, executes and starts its critical section by taking semaphore *S2*. During the critical section of task 1 it tries and take semaphore *S1*, which is held by task 2. Task 2 therefore inherits the priority of task 1 and resumes execution. Now task 2 tries and take *S2*, which is held by task 1, which in turn waits for task 2 to finish and release *S1*. Therefore the system now entered a deadlock state and can't resume execution.

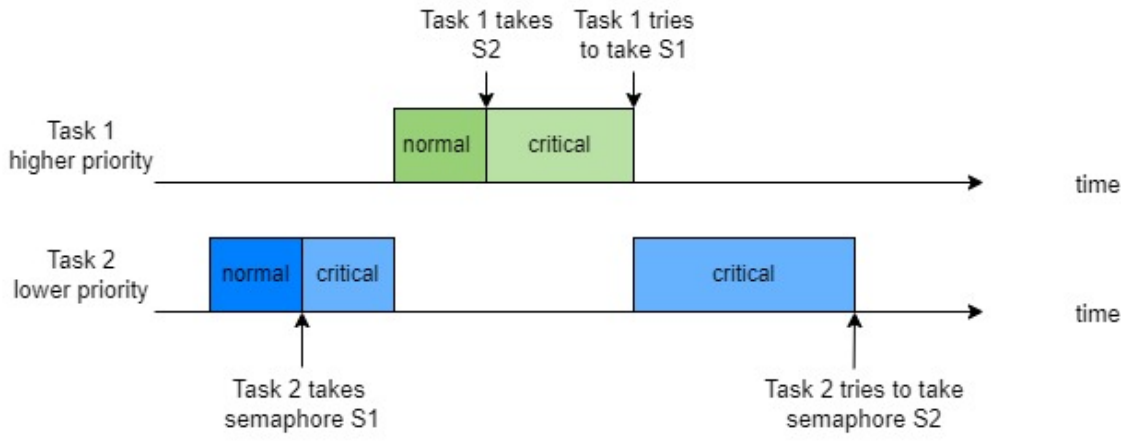

Figure 5: Priority Inheritance Drawback Example

2.3 Kernel

Contrary to GPOS, where a task can't be interrupted by preemption and therefore cause an unpredictable delay, most RTOS support preemption [9] [8]. Generally kernels allow a RTOS to support different application programming interfaces (API), like preemption or ISR. This enables the kernel of a RTOS to, example given, disable interrupts, when using preemption. This prevents an unpredictable delay like with a GPOS, if interrupts occur during preemption.

Kernels in a RTOS are often very modular. This allows a developer to choose, which APIs should be supported by the RTOS on a specific embedded system. Thereby the system can be kept small in terms of memory, computation and even energy needs, which are imperative for

small embedded systems [1]. Often the kernel of a RTOS itself tries to be as efficient as possible with the resources available [8].

In a GPOS a system call sometimes can take an unknown amount of time causing the system to be stuck in a waiting state. Other tasks and services therefore can't execute and may miss their deadline, which can lead to corruption or malfunction within the operating system. The Kernel of a RTOS is specifically designed to have certain barriers in place, which prevent one task or service to corrupt other services [8]. If a task or service in a RTOS malfunctions or fails, the kernel itself as well as other tasks won't get corrupted due to the architecture of the system [8].

There are many different approaches to a kernel in a operating system. The two architectures, that are used the most in RTOS, are either micro kernels or a monolithic kernel. For example μC/OS-II and FreeRTOS use one or more micro kernels, while RTLinux partly uses the Linux kernel, which is a monolithic kernel [8] [9]. Their variance in architecture can be attributed to one main difference:

While a micro kernel is split into many different subsystems, which all run independently, a monolithic kernel has all its functionality implemented in the kernel. Both approaches have their advantages and drawbacks. For example, the monolithic design has a major drawback in its fault-tolerance, due to everything being implemented in the kernel itself, if one service fails, the whole system could fail. A monolithic kernel is also very difficult to maintain as everything is implemented in one kernel. A micro kernel has its functionality split into different modules, which reduces the overall complexity and increases the fault-tolerance. If a service fails in a micro kernel, the module responsible for it can be shut down without the system or kernel itself failing. However a micro kernel needs an efficient way for the different modules to transfer data between them, which makes inter task communication more complex than in a monolithic kernel. Therefore a micro kernel needs an efficient communication mechanism as well as fast task context switching.

3 Operating Systems

A range of different RTOS are compared against each other in terms of selected parameters in [1], [4], [5], [8] and [9]. We will talk about the selection of parameters later in Section 4.1. For now we will focus on the different proposed RTOS.

[8] compares VxWorks, RTLinux and FreeRTOS. While VxWorks is a commercially available RTOS, FreeRTOS and RTLinux are free RTOS. VxWorks and FreeRTOS were developed as RTOS, while RTLinux is a version of the Linux operating system that was extended with real-time capabilities by a patch of the kernel.

The three RTOS compared in [4] are Xenomai, RT Patch Linux (also called RTLinux in [8]) and eCos. Xenomai follows a similar approach to real-time capabilities as RTLinux with a micro kernel. eCos is a open source RTOS, that is highly configurable to fit an applications needs [4]. However, it works only on certain embedded devices for which the application was developed or the RTOS was configured.

Most of the time comparisons done on RTOS are only including parameters on performance and responsiveness of the system. [1] however includes parameters to measure the energy con-

sumption of RTOS. Specifically they compared the energy consumption of μC/OS-II, Echidna and a bare bone RTOS called NOS [1]. While both μC/OS-II and Echidna follow a micro kernel based approach, NOS is a very minimalistic RTOS with only a task scheduler. Each of these operating systems follow a different scheduling approach, which further interests a comparison also in terms of power consumption.

RTOS predominantly run on 32 bit embedded systems. But RTOS have also gained popularity on 16 bit microcontrollers and microprocessors. Therefore [9] tests 4 different RTOS, that can run on smaller 16 bit embedded systems, and compare them in terms of their performance. The four RTOS they chose are μITRON, μTKernel, μC-OS/II and EmbOS. Each of the operating systems support various kinds of APIs and the authors of [9] indicated that the number and type of APIs supported, as well as other supported features, were critical in selecting these RTOSs. The different APIs and a comparison of which RTOS supports which APIs can be seen in Figure 6.

[5] takes a different approach. In [5] FreeRTOS gets modified to be able to do multicore, respectively multitask computation. Therefore only FreeRTOS and their modified multicore version of FreeRTOS are compared to each other. FreeRTOS in itself is a open-source RTOS, which is available for a wide range of embedded systems [5], and employs a micro kernel based approach. The modified multicore version of FreeRTOS keeps the basic approaches of FreeRTOS and extends its capabilities of the kernel and scheduler to a multicore based approach.

In the following sections we take a look at a few selected RTOS of the different comparisons in [1], [4], [5], [8] and [9]. These RTOS are μC/OS-II, RT Patch Linux and FreeRTOS. Each of the selected operating systems is shortly introduced and their general purpose and structure explained. Furthermore we take a look at how these operating systems approach scheduling and solve priority inversion. Lastly we analyse the kernel of the selected RTOS and describe their respective architecture.

3.1 μC/OS-II

μC/OS-II is a small public domain RTOS. It can easily be ported on different microcontrollers or digital signal processors (DSP) [1]. μC/OS-II currently runs on 50 different types of architectures.

Despite its small size of about 1700 lines of code, it supports a wide range of different APIs [1] [9]. Figure 6 shows a range of different APIs and their supported APIs. One can see, that μC/OS-II supports most of the APIs, which are listed below the diagram, only missing APIs for eventflag, data queue, mutex, variable size memory pool and trace API.

μC/OS-II is split in different modules, which handle different tasks or APIs. To keep the memory footprint of μC/OS-II small, only modules, which are used, are compiled into the resulting application [1].

However, μC/OS-II does not support periodically repeating tasks. This can be circumvented by using a user-level task, which is attached to the clock interrupt [1]. Tasks, that handle the periodically repeating actions, are waiting on a unique semaphore. The user-level task then can wake up each task, when their respective period is met. This is possibly, because μC/OS-II provides the possibility to attach user-level tasks to any events.

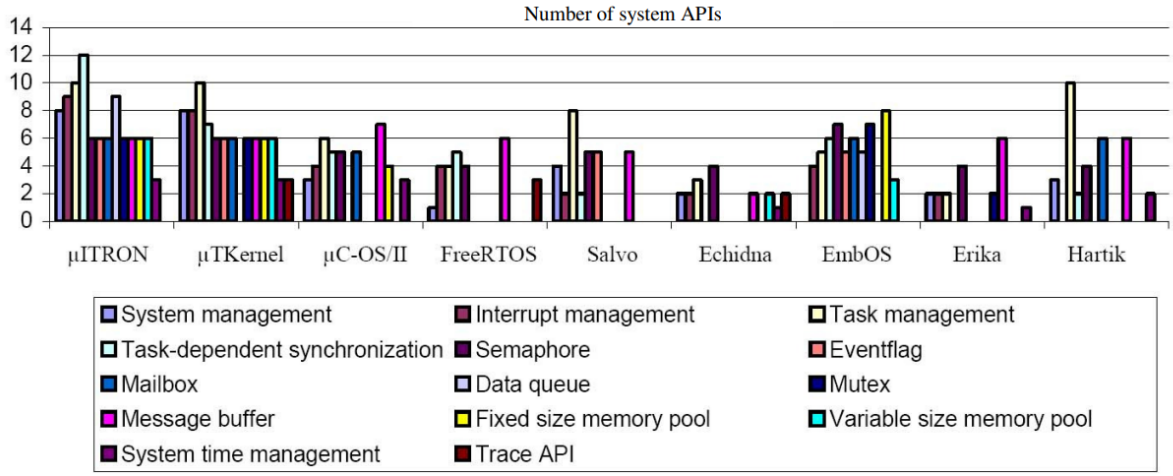

Figure 6: Number of system APIs for various RTOSes. Adapted from [9]

Tasks, which are waiting for a semaphore, in μC/OS-II are put into a FIFO buffer. The order of this buffer can't be changed by the developer [9]. A task waiting for a semaphore must wait its turn, even if it has a higher priority than the task in the buffer before it.

3.1.1 Scheduler

A preemptive priority based scheduler is used in μC/OS-II [1] [9]. This scheduler is vulnerable to the problem of priority inversion. As mentioned in Section 2.2.1, there are many different solutions to priority inversion. While μC/OS-II itself doesn't implement one of these strategies, it is easy to implement different ones into it [10].

On the one hand, the proposed solution of priority inheritance of section 2.2.1 is only possible with a change to the kernel of μC/OS-II. On the other hand, disabling interrupts in a critical section is easily done by calling an enter and an exit function, which are already present in μC/OS-II [10].

3.1.2 Kernel

μC/OS-II uses a micro kernel, where only the used modules are compiled into the application [1]. The operating system supports up to 64 tasks, but 8 of them are reserved for the kernel itself to execute on. Several services like semaphores, memory management and inter process communication (IPC) are supported by the kernel [1]. An overview of all supported APIs was already given in Figure 6.

3.2 RT Patch Linux

RT Patch Linux (also called RTLinux) is a modified version of Linux, which has real-time computation capabilities. It has a special architecture, where applications or tasks with time constraints are running on a specialised real-time kernel. Tasks and applications without time constraints run on the standard Linux kernel [8]. Figure 7 shows the general architecture of RT Patch Linux. As mentioned the time constrained tasks, marked in orange in Figure 7, run on the real-time kernel. However, the real-time kernel controls some functions, which would be controlled by the standard Linux kernel in a non real-time capable version of Linux. More on that later in Section 3.2.2.

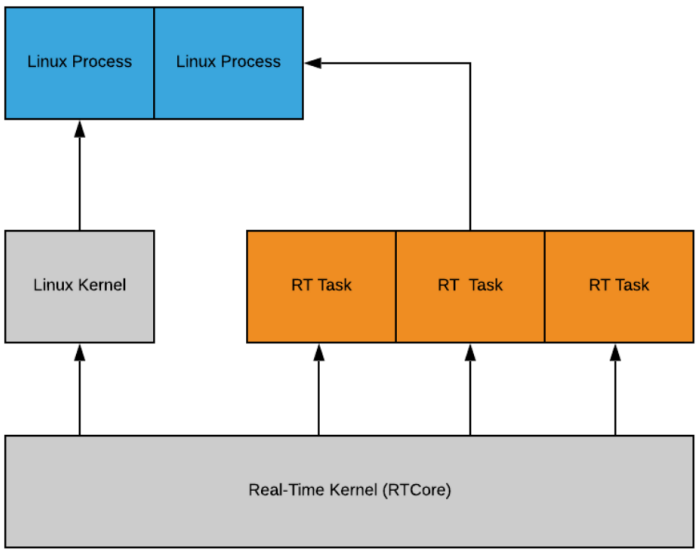

Figure 7: RT Patch Linux Architecture Overview. Adapted from [8]

However, due to its architecture using 2 kernels and the Linux kernel itself being a few megabytes in size, RT Patch Linux has a significant higher memory footprint. This hinders it from being used on smaller embedded systems.

3.2.1 Scheduler

RT Patch Linux supports a flexible scheduler. This scheduler allows for FIFO, EDF and rate-monotonic scheduling, of which the first two are detailed in Section 2.2. While having a larger memory footprint than a RTOS, that only supports preemptive priority based scheduling, this scheduler allows the developer to modify the scheduling to the applications needs. For example in a system with only periodically repeating tasks, it may be beneficial to use rate-monotonic scheduling. This scheduling method schedules the task by their period, which means, that a task with a low period will be scheduled more often and have a higher priority than a task with a long period [8].

While standard Linux doesn't allow preemption, RT Patch Linux makes in-kernel locking-primitives preemptible. So-called rtMutexes are introduced, which make critical sections of tasks preempitble, which leads to the problem of Priority Inversion. This is solved by using priority inheritance for in-kernel spinlocks and semaphores [4].

3.2.2 Kernel

As already mentioned in Section 3.2, RT Patch Linux uses 2 separate kernels. A standard Linux kernel and a real-time kernel. The Linux standard kernel is a monolithic kernel, while the real-time kernel is a micro kernel [7].

The real-time kernel restricts the standard Linux kernel, so it doesn't interfere with scheduling of tasks. Additionally the standard Linux kernel can't disable interrupts [8]. Interrupts, scheduling as well as thread handling is done by the real-time kernel.

Tasks can communicate with the kernel with a FIFO pipe. With both a standard Linux kernel and a real-time capable kernel in place, RT Patch Linux is a fully functional Linux system with real-time computation capabilities [8].

3.3 FreeRTOS

FreeRTOS is a open-source RTOS, which is written in the C programming language [5]. This operating system has been ported to more than 27 different architectures. It is very simple to port FreeRTOS, because of its split into a hardware independent and a portable layer, which can be seen in Figure 8. Therefore not all components of FreeRTOS have to be rewritten for it to be ported to a new architecture. The hardware independent layer performs most of the operating system services and functions. This makes porting to another architecture easier as only hardware specific action, for example context switching or memory allocation and de-allocation, needs to be rewritten [5].

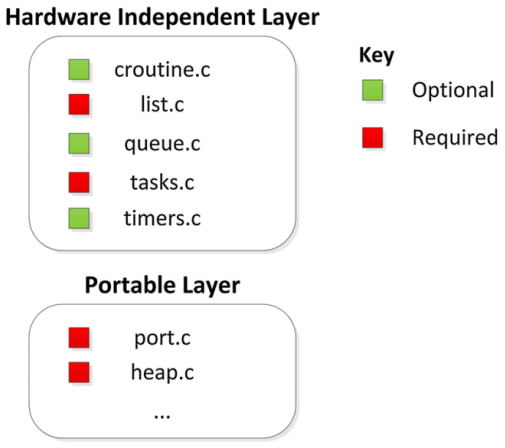

Figure 8: FreeRTOS source structure. Adapted from [5]

Figure 8 shows the general code structure of FreeRTOS and separation into the two layers. It also shows, that not all components of FreeRTOS are required for the operating system to run. This can be used to reduce the already small memory footprint of FreeRTOS even further.

Generally the architecture of FreeRTOS is similar to other RTOS. As Figure 9 shows, tasks communicate through kernel and drivers with the hardware. Inter task communication is implemented via queues, where the task with the highest priority will get access to the queue first [5].

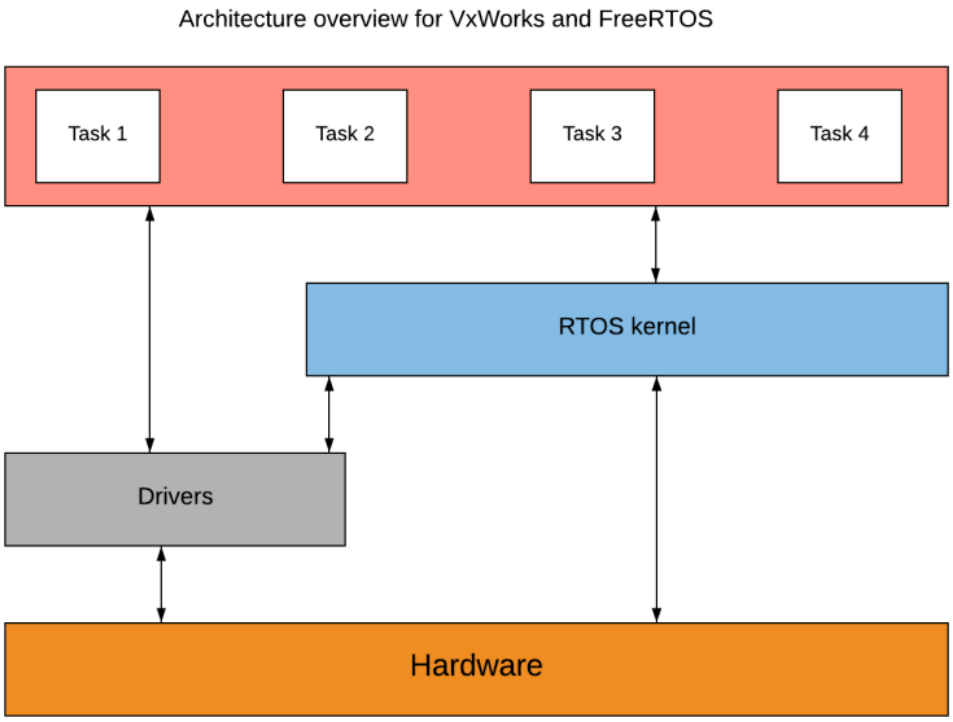

Figure 9: Architecture Overview for VxWorks and FreeRTOS. Adapted from [8]

Interrupts are dependent on the hardware and sometimes need a special interrupt controller on the hardware to be able to support them [5]. This is done in the portable layer of FreeRTOS, which is depicted in Figure 8.

However, the portable layer has to configure and start the interrupt controller before any tasks can be started. Otherwise, if the processor doesn't have sufficient interrupt lanes, interrupts could get lost and no handler would be notified by the operating system. This can be done by the developer by calling a function before creating any task in the *main* function of their application [5].

Task management is not only part of the hardware independent layer as shown in Figure 8, but also part of the portable layer [5]. Task creation is located in the portable layer and therefore the developer doesn't need to care about memory allocation when implementing tasks.

Tasks in FreeRTOS have a so-called task control block (TCB) attached to them. This TCB

contains all information about the respective task, such as the priority or a pointer to the location of the code [5] [3]. However, the kernel restricts developers, when creating tasks, by enforcing, that the memory size given at creation of the task can not be exceeded.

3.3.1 Scheduler

FreeRTOS uses a dynamic preemptive priority based scheduling algorithm [3]. The developer can choose whether a preemption policy is enforced or the tasks run in a cooperative mode. Preemption will always run the highest priority task and only split CPU-time, when there are multiple tasks with the highest priority. However, cooperative mode will not preempt a task, as long as it is not blocked or the specific function to switch context to another task is not called [8].

This scheduling is vulnerable to priority inversion. While it does not provide any strategies to prevent or solve priority inversion directly, it does however enforce non-blocking tasks, when dealing with deadlocks, and blocks other tasks for a fixed amount of time.

3.3.2 Kernel

FreeRTOS uses, contrary to RT Patch Linux, one single micro kernel. It supports a few different APIs, but not as many as μC/OS-II. This can be seen in Figure 6. However, the kernel supports blocking and deadlock avoidance as described in the previous section, as well as scheduler suspension [8].

4 Comparison

Many different comparisons of RTOS with varying parameters have been done in [1], [4], [5], [8] and [9]. While some comparisons like [1] focus more on energy efficiency, others focus on the amount of supported APIs and the overall performance on a, in terms of memory capacity and computation power, strictly restricted system [9]. Their experiment setups as well as choice of parameters are understandably different. However even when generally comparing RTOS in terms of performance, all of the presented comparisons in this paper vary vastly, when it comes to selecting parameters [1] [4] [5] [8] [9].

The results found in different comparisons often highlight the difference of the selected RTOS. However these comparisons are usually restricted to 3 or 4 selected operating systems. Often times the results are also specific to the experimental setup of the paper. For example [1] use a simulation software called *SimBed*. In this case the results are mainly dependent on the accuracy of the simulation and may differ from a real world experiment like [9] and [4].

The following subsections try to give a common parameter baseline to compare different RTOS by and also discuss how the results of different comparisons can be compared. I will start by comparing the different parameters, which were used in the different comparisons, and analyse them to state a common base of parameters. Afterwards I will compare the different findings of some comparisons and analyze how these results can be used to compare RTOS against each other, even if they were not specifically analysed in the same comparison.

4.1 Parameter

Different comparisons of RTOS use varying parameters. They are mostly dependent on the aim of the study and their experimental setup. [8] has no real experimental setup. The aim of this work is to give a general overview of RTOS and compare some in terms of their architecture. Therefore [8] analyzes RTOS in terms of the following parameters:

- Kernel
- Scheduler

[4] aims to analyze the general performance of a RTOS and compare them in terms of parameters based, that describe the performance of a RTOS. They have different benchmark applications for each of the parameters, that create their experimental setup. The following list shows the parameters to measure performance used in [4]:

(a) Latency

(b) Context Switching Time

(c) Jitter

(d) Preemption Time

(e) Priority Functionality

(f) Deadlock Breaktime

(g) Semaphore Shuffle Breaktime

However, not all parameters are measured for all of the selected RTOS in [4]. This is due to restrictions in certain RTOS, that are being compared.

While the last to comparisons compared general architecture and performance, [1] tries to analyze RTOS in terms of performance compared to energy consumption. They therefore selected two performance dependent parameters and describe the energy consumption of the systems. The following list contains their exact parameters:

- Real-Time Jitter
- Response-Time Delay
- Energy Consumption

[9] compares different RTOS in terms of their APIs supported. Their special use case is using RTOS on very limited 16 bit embedded systems. Therefore they split their comparison in two main points. The first main point is code (ROM) and data size(RAM) [9]. They chose the following parameters to compare the RTOS by:

- Task switch time
- Get & release Semaphore (one task)
- Pass semaphore (from one task to another)
- Pass & retrieve message to queue

- Pass message (from one task to another)
- Acquire & release fixed-size memory block
- Task activation from interrupt

Their second main point for comparing RTOS for 16 bit embedded systems is the general execution time. The parameters are similar to the ones used for comparing code and data size, however they split some of the parameters to get a more accurate representation of execution times. The following parameters were used in [9] for measuring execution time:

- Task switch time
- Get semaphore
- Release semaphore
- Pass semaphore (from one task to another)
- Pass message to queue
- Retrieve message from queue
- Pass message (from one task to another)
- Acquire fixed-size memory block
- Release fixed-size memory block
- Task activation from interrupt
- Average execution time

The last comparison we will take a look at is described in [5]. This comparison is different to the previous ones as it only compares one RTOS, which was modified to run on another type of architecture. They compare the normal version of FreeRTOS to their modified version, which tries to utilize a multi core system. The main goal of their study is the porting of FreeRTOS to a multi core platform, they therefore compared the two version only on basic parameters describing the workflow of both systems:

- Run time
- Development time

Comparing the different comparisons, one can see some overlap of parameters especially for performance. These parameters give the first few parameters for a base line to compare RTOS by. Specifically the execution time on the same hardware of different systems, the context switching time and jitter seem to be emphasised by many papers. Jitter describes the variation of execution time, which can be obtained by execution a task or application over and over again and comparing the execution times. Another informative parameter is the amount and kind of supported APIs. These can be important to select a RTOS, when the system needs specific functionality, for example interrupts.

Other parameters for a parameter basis to compare RTOS can be very platform or hardware specific. For example, if a application needs semaphores, it may be beneficial to know the acquire and release times for semaphore, like they are compared in [9]. As these parameters are specific to the application and platform, such parameters will not be taken into account for the parameter basis.

However a basic parameter, that is limited by hardware restrictions, can also be added, as it need to be considered for every embedded system. This parameter is the memory footprint of the RTOS. The memory footprint of different RTOS is mostly not described in the presented comparison, because they select their RTOS to be able to fit onto the limited memory their hardware has to offer. For example [9] did not consider RT Patch Linux, as it would not be able to fit on the memory of a small 16 bit embedded system.

After comparing the parameter selection of different comparisons of RTOS, an overall informative parameter basis seems to be the following list:

- Supported APIs
- Memory footprint
- General execution time
- Context switching time
- Jitter

4.2 Results

In this section I will discuss the results found by different comparisons in terms of the parameter baseline presented in the previous section. Starting with the different APIs supported by different RTOS. Figure 6 show a variety of RTOS and their supported APIs. Here we can see, that μC/OS-II supports more APIs as FreeRTOS, but FreeRTOS supports APIs, for example the trace API, which are not supported by μC/OS-II. RT Patch Linux is not present within this comparison, but, considering it being a fully functional Linux system with real time capabilities on top, it is safe to say, that it supports most if not all of the listed APIs [8] [7].

Figure 10 shows μC/OS-II being relatively small in terms of code size, memory needed in ROM, while using more RAM than similar small systems for 16 bit embedded systems. While we don't have a quantative comparison for RT Patch Linux nor FreeRTOS, we still can make qualitative comparisons based on information given in other comparisons. Looking at RT Patch Linux, [8] and [4] describe this RTOS as memory heavy. A closer study on the kernels for RT Patch Linux itself reveals, that it is more memory heavy than other RTOS due to its dual kernel architecture [8] [7]. FreeRTOS with its more bare bones approach, as can be seen in Figure 6 with its low number of supported APIs or described by [5], is likely to be similar to μC/OS-II in terms of its memory footprint.

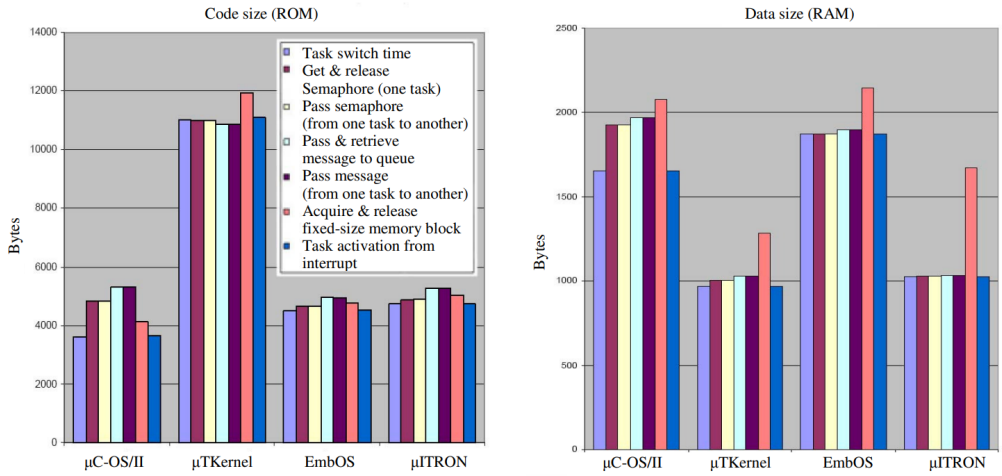

Figure 10: Code and data size comparison for 4 RTOSes. Adapted from [9]

Figure 11 shows the results for execution time in terms of the different parameters used in [9]. Here we can see the different strength of weaknesses of different RTOS. Data is only available for μC/OS-II. However, when comparing μC/OS-II to the other four selected RTOS in [9], one can clearly see, that each RTOS has its own strengths and weaknesses. This is confirmed especially by [4], which shows, that their three selected RTOS are very different in terms of different performance related parameters and execution times. Therefore to compare RTOS in terms of their execution time and performance, which is covered in the common parameter baseline by general execution time, context switching time, a significant part of an application has to be run as a test on the different RTOS.

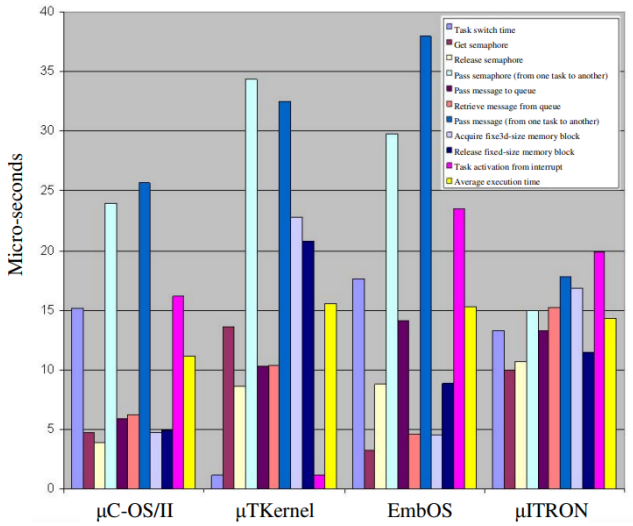

Figure 11: Execution time benchmark for 4 RTOSes. Adapted from [9]

Lastly jitter is very dependent on the RTOS itself, but also the hardware. In order to compare jitter for different RTOS, the same approach as for general execution time and context switching time has to be applied. However it is safe to say, that RT Patch Linux has a greater jitter than μC/OS-II or FreeRTOS, due to the compleity of the system. This is confirmed by a closer analysis of the kernels [8] [3] [7].

5 Future Development

With increasing demands in terms of memory and computation power for embedded systems, the need for new approaches and technologies for RTOS is increasing, too. In current times multi core processor are already present in small embedded systems. However RTOS can't use them to their full potential. This is due to their restriction, that they don't support concurrency of tasks. Recent work explored the possibility of porting existing RTOS to a multi core platform [5] or new RTOS, that are built with multi core execution in mind [8] [6]. It has been shown, that a multi core based approach can increase the performance and even energy efficiency of a RTOS [6]. With an ever increasing amount of tasks to handle and larger data to process for RTOS, [5] state, that the switch to a multi core based operating system is inevitable.

[5] tries to modify FreeRTOS, so it can run on a multi core system. In their work the authors detail their implementation process, different challenges, when porting an existing RTOS to multi core platforms, as well as evaluating their implementation and comparing it to the non modified version of FreeRTOS. Specifically they want to port FreeRTOS to work on field programmable gate arrays (FPGA). They currently have only the required components of FreeRTOS, which are displayed in Figure 8, ported.

Interrupt functionality needed to be modified. In a single core system disabling interrupts is as simple as the following snippet shows:

```
disableInterrupts()
// Critical section of Code
enableInterrupts()
```

Listing 1: Disable Interrupts on a Single Core System. Adapted from [5]

However on multi core systems it isn't as simple as that. To ensure that no two different cores work on a shared resource, a mutual exclusion, that works across all processing cores of the embedded system, is needed. [5] modified the existing mutexes in FreeRTOS to work on such systems.

As already stated in Section 3.3.1 FreeRTOS has a fixed priority preemptive scheduler. They therefore discuss a different solution for a scheduler on a multi core system using core affinity [5]. A task in their multi core version of FreeRTOS executes, if it is not being executed on another core or either no core affinity or a core affinity indicating, that it should execute on this specific core has been set. Additionally the task has to have the highest priority with these conditions [5].

A testing application indicated a speedup of up to 1.87 times faster than the execution on a single core, while using the same hardware (FPGAs). However they state in their future work, that their architecture remains to be proven. Additionally support for platforms other than their used FPGA and optional components of FreeRTOS has to be developed.

[6] tries a different approach, where the authors developed a multi core RTOS called HIPPEROS from the ground up. It tries to combine hard real time capabilities with multi core design principles [8]. Their kernel is implemented in a way, that allows it to run on a arbitrary amount of cores. To achieve this, they choose a asymmetric micro kernel approach, which allows them to have a main kernel, that handles services like system calls or scheduling, run on one core. Other tasks and parts of the kernel can then run on other available cores. This however proposes a problem, that the specific core running the scheduler has to be notified whenever a task is ready to execute. The core running the scheduler needs to then interrupt another core, that it switches its context to run the task. This is done by a master slave architecture, where slave processors can only switch their context, when being notified by the master core [8] [6].

HIPPEROS uses a preemptive priority based scheduler [8]. A configurable policy lets the developer decide how the scheduler and operating system reacts to a task missing a deadline [6].

6 Conclusion

In this work I gave a general overview of what RTOS are in general. How they are structured, what problems occur with the preemptive priority based scheduling approach, which is used by most RTOS, and how these problems are solved. I continued by describing three selected RTOS in detail, what their architecture looks like, their exact approach to scheduling and solving the problem of priority inversion and what features are supported by their kernels. Afterwards I gave a brief overview of different comparisons done in recent work, that their parameter look like and where overlaps of different works are. This work gave a common parameter baseline, based on common parameters in different comparisons and on how RTOS work. The common parameter baseline is then used to compare the three detailed RTOS by the results found of different comparisons. This work concluded by giving a outlook on future development.

References

[1] K. Baynes, C. Collins, E. Fiterman, Brinda Ganesh, P. Kohout, C. Smit, T. Zhang & B. Jacob (2003): *The performance and energy consumption of embedded real-time operating systems. IEEE Transactions on Computers* 52(11), pp. 1454–1469, doi:10.1109/TC.2003.1244943.

[2] Sadegh Davari & Lui Sha (1992): *Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions. SIGOPS Oper. Syst. Rev.* 26(2), p. 110–120, doi:10.1145/142111.142126. Available at `https://doi.org/10.1145/142111.142126`.

[3] Rich Goyette (2007): *An analysis and description of the inner workings of the freertos kernel. Carleton University* 5.

[4] Mastura D. Marieska, Paul G. Hariyanto, M. Firda Fauzan, Achmad Imam Kistijantoro & Afwarman Manaf (2011): *On performance of kernel based and embedded Real-Time Operating System: Benchmarking and analysis.* In: *2011 International Conference on Advanced Computer Science and Information Systems*, pp. 401–406.

[5] James Mistry, Matthew Naylor & Jim Woodcock (2014): *Adapting FreeRTOS for multicores: an experience report. Software: Practice and Experience* 44(9), pp. 1129–1154, doi:https://doi.org/10.1002/spe.2188. Available at `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2188`.

[6] Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens & Ben Rodriguez (2015): *A new configurable and parallel embedded real-time micro-kernel for multi-core platforms. OSPERT 2015*, p. 25.

[7] Federico Reghenzani, Giuseppe Massari & William Fornaciari (2019): *The Real-Time Linux Kernel: A Survey on PREEMPT_RT. ACM Comput. Surv.* 52(1), doi:10.1145/3297714. Available at `https://doi.org/10.1145/3297714`.

[8] Anthony J. Serino & Liang Cheng (2019): *A Survey of Real-Time Operating Systems.*

[9] Su-Lim Tan & Tran Nguyen Bao Anh (2009): *Real-time operating system (RTOS) for small (16-bit) microcontroller.* In: *2009 IEEE 13th International Symposium on Consumer Electronics*, pp. 1007–1011, doi:10.1109/ISCE.2009.5156833.

[10] Xibo Wang, Shan Jin & Zhongling Yang (2013): *Several Methods of Design and Implementation to Solve Priority Inversion Problem in uC/OS-II.* In: *2013 6th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pp. 223–226, doi:10.1109/ICINIS.2013.64.