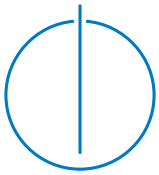
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design of an OC-based Method for efficient Synchronization of L4 Fiasco.OC Mircrokernel Tasks

Robert Häcker



DEPARTMENT OF INFORMATICS

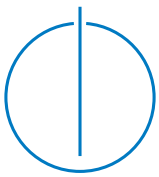
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design of an OC-based Method for efficient Synchronization of L4 Fiasco.OC Mircrokernel Tasks

Konzeption eines OC-basierten Verfahrens zur effizienten Synchronisation von Tasks im L4 Fiasco.OC Mikrokern

Authors:	Robert Häcker
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Daniel Krefft, M. Sc.
	Sebastian Eckl, M.Sc.
Submission Date:	6/15/2015



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 6/15/2015 Robert Häcker

Acknowledgments

I want to thank my advisor, Daniel Krefft, for his extensive support. Also I like to thank my sister Barbara and my friend Bernhard for proofreading of this thesis.

Abstract

This bachelor thesis shows, how a possible implementation of a scheduler for the L4 Fiasco.OC micro-kernel could look like. This scheduler will be integrated into a grid which supports the organic-computing paradigm. At the same time, a modification of the ready-queue, based on the decisions made in the grid, should be possible. Scheduling algorithms and synchronisation-methods, that fit this propose will subsequently be presented, explained and compared to each other. As result the Modified-Maximum-Urgency-First Scheduler in combination with the sequential-lock is recommended and it is shown, how a possible implementation of both, into the L4 Fiasco.OC, can look like.

Diese Bachelorarbeit zeigt auf, wie eine mögliche Umsetzung eines Schedulers für den L4 Fiasco.OC Mikrokernel aussehen könnte. Dieser Scheduler wird in ein, das Organic-Computing Paradigma unterstützendes, Netzwerk intergriert. Dabei soll auch eine Modifikation der Ready-Queue, auf Grundlage der im Netzwerk eigenständig getroffenen Entscheidungen ermöglicht werden. Für diesen Zweck geeignete Scheduling-Algorithmen und Synchronisations-Mechanismen werden nacheinander vorgestellt, erklärt und miteinander verglichen. Als Ergebnis wird der Modified-Maximum-Urgency-First Scheduler in Kombination mit dem Sequential-Lock empfohlen und aufgezeigt, wie beides im L4 Fiasco.OC umgesetzt werden kann.

Contents

1 Introduction

This thesis is part of the KIA4SM project. Its aim is to concept a mechanism to add and schedule threads on a L4 Fiasco.OC micro-kernel. The whole system is part in an automotive environment. The aim of this thesis is to concept a thread-scheduler for the L4 Fiasco.OC kernel based on the needs of the KIA4SM project and to synchronized the access to the ready-queue of the system.

The KIA4SM Project KIA4SM is a research project of the department of operating systems "F13" of the Technical University of Munich (TUM). The acronym KIA4SM stands for cooperative integration architecture for future smart mobility solutions ("Kooperative Integrationsarchitektur für zukünftige Smart Mobility Lösungen"). It was started to develop a hybrid simulation model for vehicles based on virtual and real ones.

In normal vehicles a large number of different micro-controllers and different sorts of control-units compute all necessary data, which the vehicle needs to work. The aim of the KIA4SM project is to replace them and use more power-full and standardized hardware. The calculation inside these vehicles is done by multiple distributed electronic-control-units (ECUs) to achieve greater redundancy, reliability and flexibility than a single central system is able to attain. The most promising way to do so is by using methods of organic-computing.

Organic-Computing According to H. Schmeck's paper [Sch05] it is important how to design large and self-organizing systems and the organic-computing is the best approach so far.

The idea is to teach the system a life-like behavior. Part of that are the self-x properties: self-organisation, self-configuration, self-optimization, self-healing, self-explaining and self-protection [Sch05]. To achieve that behavior an observer-controller architecture can be used [Kam09]. Also these controllers in our vehicle have to communicate with each other to decide what to do.

The Environment Of The System The core of the system is the L4 Fiasco.OC kernel which is written in C++ and developed by the Technical University of Dresden [Döb13].

It is based on the L4 which was developed in 1993 by Jochen Liedtke who already invented the predecessors L3, L2 and L1 [Wik15].

In our automotive system, hard real-time-conditions are consisting. This means it is critical that tasks and threads will reach their deadlines. If a task or thread in a soft real-time-system is not completed before its deadline, nothing terrible happens. In a hard real-time-system (like a vehicle) missing a deadline can have a huge impact on the whole system. According to R. Brause [Bra01] the distinction between different real-time-systems is based on the urgency of reaching the deadline of the threads. If the result of a thread is useless when the deadline is expired, it is called a hard real-time-system. In soft real-time-systems the result of the thread which expired its deadline may still be a little bit useful or at least nothing terrible happens because of the delay. All real-time-systems in this thesis are hard real-time-systems.

Scheduling Threads Figure 1.1 shows the relation between programs, processes, tasks and threads. A thread is the smallest part of a running program called process. On embedded systems, like the ECUs of the KIA4SM project, the process is also called task. A task can have multiple threads, which can be executed in parallel on the different CPU-cores of the ECUs [Gla06].

Because the L4 Fiasco.OC is a very minimalistic Kernel and the whole systems is kind of an embedded system, the threads are only able to have three different conditions: ready, computing or waiting. Ready threads are waiting for computing-time on the CPU. Computing threads are computing on the CPU and waiting threads are waiting for resources, locks/semaphores or computing threads. After a thread is created it will be added to the ready queue.

The function of the scheduler is, to decide which of these threads are allowed to get computing time on the CPU. The chosen thread is marked as running while it is running on the CPU.

If the thread is finished its status is set to terminated and it will be removed. If the thread is blocked because a resource is missing, an event did not occur or the thread was cancelled, it will be added to the ready queue again. An overview of the changes between these states is given in figure 1.2.

1.1 Current State

There is a small model car which is equipped with multiple ECUs. Each of them handles an operating system which is based on the L4 Fiasco micro-kernel. On this kernel all required threads will be performed. Until now, the L4 Fiasco is able to define the execution order of threads at development time. The attempt is to extend its

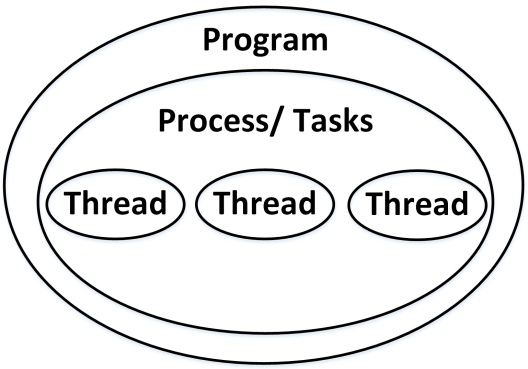


Figure 1.1: Overview of the relation between program, process, task and thread.

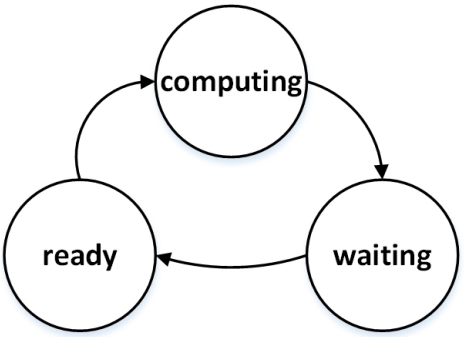


Figure 1.2: The different conditions threads can have on the L4 Fiasco.OC micro-kernel

capability to handle a dynamic order of threads at run-time.

A modern vehicle needs the additional flexibility at executing and scheduling threads, to achieve a dynamic system, which is able to meet the requirements of modern systems like autonomous driving or intelligent transport networks.

Therefore it is essential to be able to add threads and modify the execution order during operation time. A flexible thread handling is also required for example in case a ECU is malfunctioning. In this case it would be possible to swap the threads from the malfunctioning one to working ones. So it is important to generate new ready-queues based on the information we are receiving from the other ECUs in the grid, and then exchange them with the actual ready-queue the scheduler uses.

1.2 Requirements

Real-Time-Compatibility Because the system is used in an embedded real-time environment, every part of it has to be real-time-compatible. The scheduler has to handle threads with deadlines in a way, that important threads reach their deadlines [LZ10]. Furthermore it is important, that the system can not reach dangerous systems-states. Therefore the scheduler must be predictable, even in overload-situations when the theoretical system-load of all the threads would be over 100%.

Not only the scheduler has to fulfill certain conditions. Also the OC.controller should guarantee not to cause unstable system-states. To be sure the whole system is safe enough to be used for a real-time-system in an automotive environment all access to critical parts of the system has to be protected, for example the access to the ready-queue.

Organic-Computing-Compatibility To achieve the properties of organic-computing, it is important to use the right observer-controller architecture. In our case we use the one shown in figure 1.3. It is also called distributed structure, because each ECU in the grid has its own observer and its own controller. The exact communication between observer and controller, or between different controllers in the grid is not relevant for this thesis. That will possibly be implemented in a later state of the project. For now it is just important to know, it is part of the observers job to collect data and sent it to the controller and to the other controllers in the grid. This is visualized in figure 3.1.

The controller receives additional information from other observers in the grid. Based on all these information the controller can intervene into the system. In particular, it can change the ready-queue and add or remove threads. It is important, that the controller is not able to block other important tasks, that use the ready-queue (e.g. the scheduler) and thereby cause unstable or unpredictable system-states. To guarantee

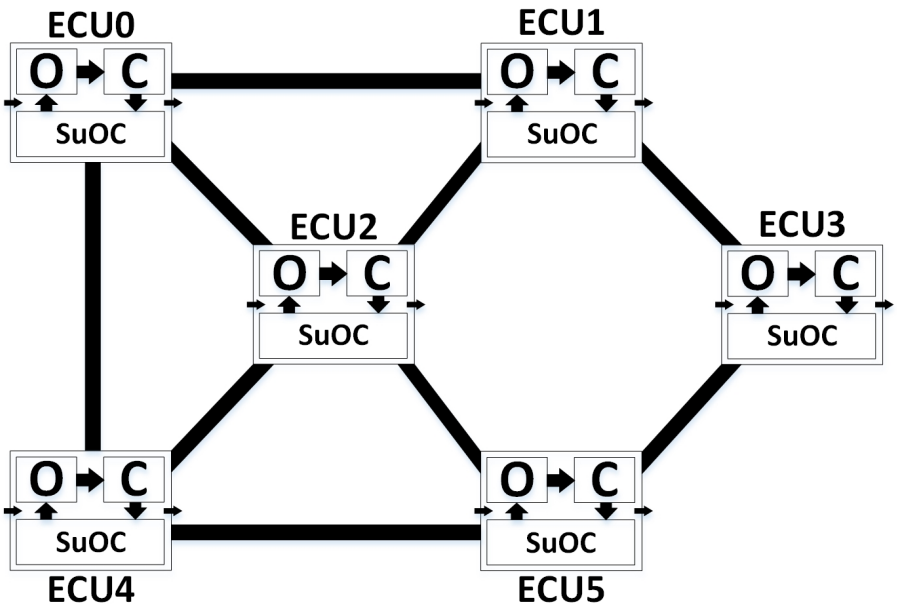


Figure 1.3: Overview of the observer-controller architecture in a distributed organic-computing paradigm. O stands for observer, C for controller and SuOC for System under Observervation and Control

this, the access to the ready-queue has to be synchronized and supervised.

2 Analysis

Before the wanted mechanism can be analyzed, a short overview about the different components is needed.

To be able to concept a method for efficient synchronization of threads (to achieve the requirements defined in 1.2),first the different techniques and methods, that could be used to achieve this aim are analysed. Scheduler are categorized in three main groups: static, dynamic and hybrid. We will have a closer look on two exemplary chosen algorithms in each category. In chapter 2.2 we take a look on the different synchronisation methods that can be used to guarantee a safe access to the ready-queue. The discussed schedulers are a preselection of the most appropriate ones.

2.1 Scheduler Variations

Schedulers can be categorized in a bunch of types. A preemptive scheduler is able to interrupt a running thread and switch it with another. Non preemptive scheduler have to wait till the thread is finished before it is able to run the next thread [Dod06]. Both kinds have its advantages and disadvantages.

In order to decide, which thread gets computing time on the CPU, scheduler use various properties of these threads. E.g. the duration of the period, the priority, the deadline or even the computing time of the thread. Some algorithms are even going one step ahead and calculate the time the thread should start computing on the CPU to reach its deadline, which is called laxity-time. Based on the use of these information, it is possible to categorize the scheduler in even more types than just preemptive or non preemptive. The chosen selection is categorized in static-, dynamic- and hybrid scheduler.

For better comparability a notation quite similar to the one from G.C.Buttazzo from his book Hard Read-Time Computing Systems [But11] is used.

τ_i	a generic periodic thread
$\tau_{j,i}$	the j-th. instance of the thread τ_i
$r_{i,j}$	the release time of the j-th. instance of the thread τ_i
$d_{i,j}$	deadline of the j-th. instance of the thread τ_i
$s_{i,j}$	start time of the j-th. instance of the thread τ_i
$f_{i,j}$	finishing time of the j-th. instance of the thread τ_i
C_i	computing time of the thread τ_i
T_i	time of the interval of thread τ_i
p_i	priority of the thread τ_i
Δt	Time Quantum
$U_i = \frac{C_i}{T_i}$	Utilisation Factor

A few scheduler algorithms are based on periodical threads. Due to the fact, that not every thread is a periodical thread, we describe non-periodical threads as a periodical threads with just one period. Also we say for all threads $d_{i,j} = r_{i,j+1}$.

2.1.1 Static Scheduler

Static scheduler do not change any properties of threads at run-time. Based on the thread information, threads are ordered once, and will always keep their order [AR98]. This means, if a thread gets the highest priority, it will keep it forever or at least until the scheduler has to schedule all threads again.

2.1.1.1 Fixed Priority Round Robin (FPRR)

The Fixed Priority Round Robin is a modified version of the Round Robin Algorithm which is one of the simplest scheduling algorithm. It is also a preemptive scheduler.

Function The round robin algorithm works off the running queue and lets each of these threads compute on the CPU for a certain amount of time $\triangle t$. If the thread is finished, the next thread gets computing time on the CPU. When the time is expired, the thread is interrupted and the next thread gets computing time. This kind of scheduler is also called O(1) scheduler, because the overhead, the scheduler needs to select, which thread is getting computing time, is O(1). This efficiency is the reason why this O(1) scheduler is also used in some Linux kernels.

The disadvantage of the normal Round-Robin algorithm is, that it is not possible to distribute priorities to the threads. It treats every thread just in the same way as every other thread. So the Fixed Priority Round Robin (FPRR) is much better for the use case we have to handle. The FPRR adds the opportunity to set a priority to a thread. The

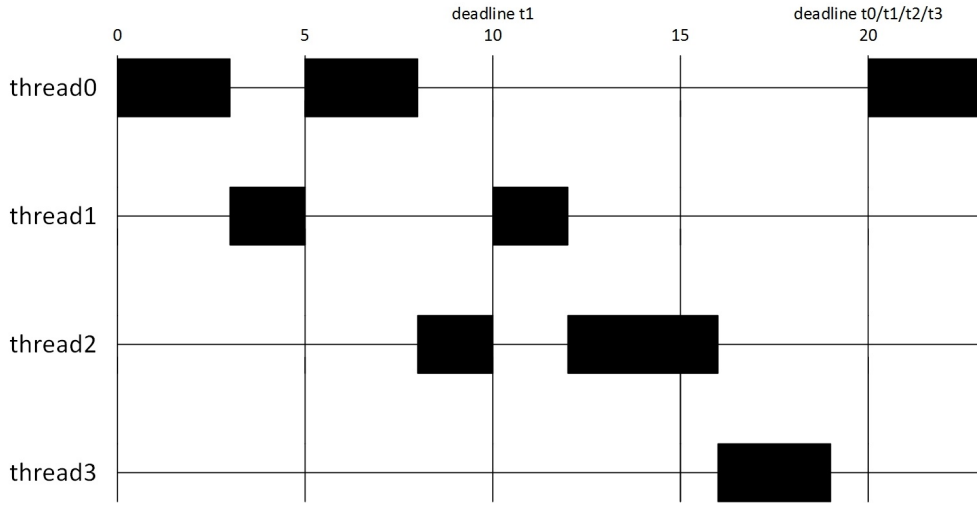


Figure 2.1: A schedule scheduled by the fixed priority round robin algorithm.

algorithm has to sort the threads in the running queue by their priorities. Afterwards it uses the normal round robin algorithm to execute the highest prioritized threads. Every thread with a lower priority does not get any execution time on the CPU.

After the threads with the highest priority are finished, the scheduler selects the threads with the highest remaining priority. If a thread with a higher priority than the currently executed thread arrives, the executing thread will be interrupted and the new and higher prioritized thread is executed. So it is ensured that important threads get really quick execution time on the CPU and do not have to wait very long.

If $p_i > p_{i+1}$ then $s_{i,j}$ - $r_{i,j} < s_{i+1,j}$ - $r_{i+1,j}$

Example Figure 2.1 shows a schedule for a bunch of example threads with a total Utilisation Factor of 75% :

τ_0: $T_0 = 20$ $C_0 = 2$ $p_0 = 1$ $U_0 = 0.1$
τ_1: $T_1 = 10$ $C_1 = 2$ $p_1 = 1$ $U_1 = 0.2$
τ_2: $T_2 = 20$ $C_2 = 6$ $p_2 = 2$ $U_2 = 0.3$
τ_3: $T_3 = 20$ $C_3 = 3$ $p_3 = 3$ $U_3 = 0.15$

First the two prioritized threads τ_0 and τ_1 were executed in turns. After both are completed, thread τ_2 has the highest priority and is executed until τ_1 is ready again. When this happens, τ_2 gets stopped and τ_1 is executed again. After τ_1 is finished, τ_2 is finished and finally τ_3 with the lowest priority gets execution time on the CPU.

Advantages and disadvantages The drawback of the FPRR is that it does not ensure any deadline. High prioritized threads may often reach their deadlines, but it is impossible to say which of the threads with a low priority do reach theirs. Also there are a lot of context switches because the algorithm changes after every Δt the executing thread which leads to much time where the CPU is switching.

On the other side the implementation of the FPRR is not as complicated as other algorithms discussed later. Also multiple threads with the same priority are able to get computing time nearly at the same time because the scheduler is switching between them. So they get fast computing time and do not have to wait as long as they have to if other algorithms are used.

2.1.1.2 Rate-Monotonic-Algorithm (RMA)

The RMA is primary developed to schedule periodical threads.

Function The algorithm sorts the threads by the duration of their period. Afterwards it allocates the thread with the shortest period the highest priority and the thread with the longest period the lowest priority. The execution order is now defined by the priority of the ready threads in the ready queue. The thread with the highest priority gets computing time on the CPU until it is finished or a thread with a higher priority arrives at the ready queue.

If the thread is completed, the next highest prioritized thread gets computing time on the CPU. If two or more threads have the same priority they are executed alternately just like in the RR algorithm.

Example The example threads with a system-load of 90% are scheduled in figure 2.2.

τ_0: $T_0 = 10$ $C_0 = 2$ $p_0 = 2$ $U_0 = 0.2$
τ_1: $T_1 = 5$ $C_1 = 2$ $p_1 = 1$ $U_1 = 0.4$
τ_2: $T_2 = 10$ $C_2 = 2$ $p_2 = 2$ $U_2 = 0.2$
τ_3: $T_3 = 20$ $C_3 = 2$ $p_3 = 3$ $U_3 = 0.10$

As described the priority of the threads are based on the period length. Thread τ_1 has the shortest period and gets instantly computing time. If τ_1 is finished, τ_0 and τ_2 are the remaining threads with the highest priority, so they share the computing time. At the end, when the other threads are all finished, τ_3 is able to get executed.

Advantages and disadvantages The crux is, that RMA only guarantees threads to reach their deadlines at a CPU utilization up to 69%. If the Utilization of the System is beyond 69% it could be possible that deadlines will be reached, but there is no guarantee. If threads did not reach their deadline, it is still possible to predict which

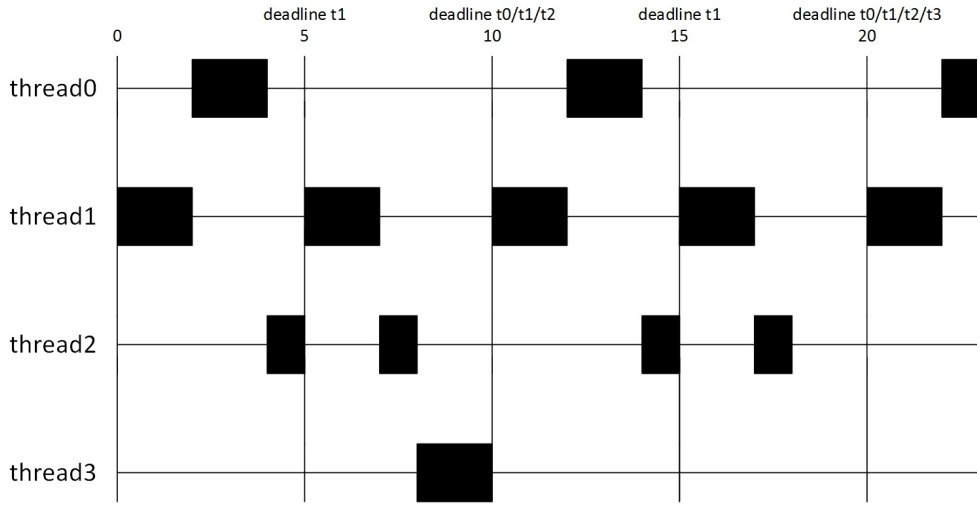


Figure 2.2: Threads scheduled by the Rate-Monotonic-Algorithm.

ones. Inherent in the nature of the Algorithm, the threads with the longest periods and thus the lowest priority will not reach their deadlines in overload situations.

Also threads with long periods often get interrupted by threads with shorter periods and thus higher priorities. But still it is an advantages the RMA guarantees that all threads reach their deadlines up to a CPU utilisation of 69%. Beyond 69% short periodical threads still reach their deadlines. But if there are multiple threads with the same period duration at the overload area, it is not possible to say which one will reach its deadline and which one not.

Also the overhead of the RMA is much lower than the overhead of the dynamic or hybrid Scheduler, because all the threads get scheduled once and after this, the schedule is used until the scheduler has to reschedule everything. Every time a single thread is added to the system, the algorithm has to redistribute all the priorities and has to reschedule all threads. With a high occurrence of arriving threads this leads to an inefficient behavior.

2.1.2 Dynamic Scheduler

The big advantage of dynamic Scheduler in comparison to static Scheduler is their property to adapt to the situation. Dynamic schedulers often use more information to decide, which thread is next on the CPU than static scheduler. This enables them to make more flexible decisions at run-time. As a consequence a thread is able to get multiple different priorities from a dynamic Scheduler, and will still reach its deadline.

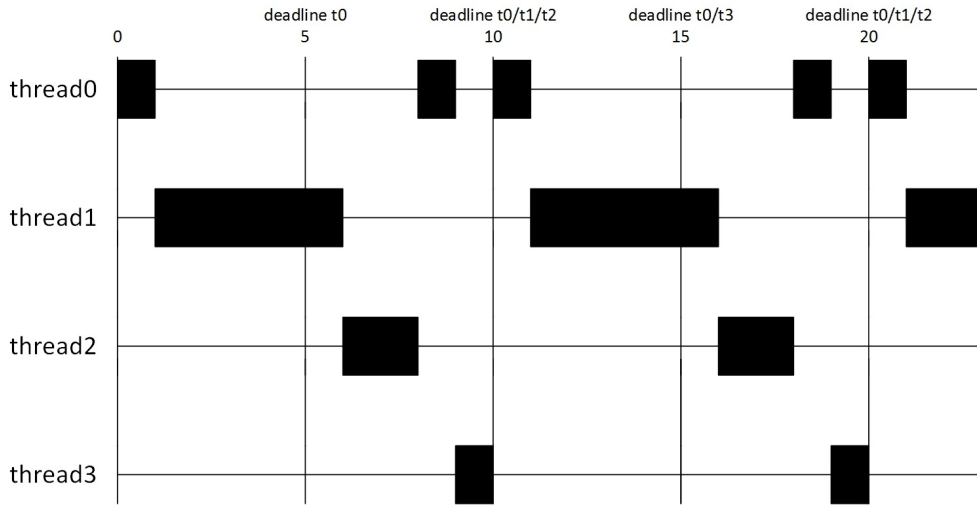


Figure 2.3: Earliest-Deadline-First algorithm used to schedule some threads.

2.1.2.1 Earliest-Deadline-First (EDF)

EDF is a development of the Earliest-Due-Date algorithm. EDF guarantees that threads will normally reach their deadline.

Function The algorithm of this Scheduler is based on the deadlines of the threads. It compares every thread in the ready-queue to find the one with the earliest deadline. This thread gets the highest priority and instant computing time on the CPU. The other threads get priorities according to their deadlines. The longer away the deadline is, the lower is the priority.

Example To visualize the EDF algorithm figure 2.3 shows a part of the schedule of these example threads with a total Utilisation of over 95%.

τ_0: $T_0 = 10$ $C_0 = 3$ $U_0 = 0.3$
τ_1: $T_1 = 15$ $C_1 = 4$ $U_1 = 0.267$
τ_2: $T_2 = 10$ $C_2 = 2$ $U_2 = 0.2$
τ_3: $T_3 = 20$ $C_3 = 4$ $U_3 = 0.2$

As it is shown, the schedule is not as evenly than the schedules of the static scheduler. Because the priority of the threads are calculated on the fly, the order of threads is not always the same.

Advantages and disadvantages Like the RMA the EDF is able to guarantee, that threads will normally reach their deadlines. Also it is able to schedule threads, that RMA can schedule as well and even some more (an example is given in [But11] on page 102). Normally threads run on the CPU until they are completed. Only sometimes threads get interrupted and replaced, to let another thread reach its deadline.

Also compared with the RMA it is much easier to add additional threads to the scheduler. On the other side threads often only get computing time at the end of their period, because earlier their priority is too low to get any computing time (just like τ_3 in figure 2.3). So threads often have to wait long until they will be completed.

2.1.2.2 Least-Laxity-First (LLF)

This algorithm is sometimes also called Minimum-Laxity-First or Next.

Function First of all, the scheduler has to calculate the Laxity-Time for each thread. A Thread has to start not later than its Laxity-Time to reach its deadline.

$$\text{Laxity-Time} = d_{i,j} - C_i$$

The thread with the least Laxity-Time gets the highest priority and the thread with the highest Laxity-Time gets the lowest priority.
If $d_{i,j} - C_i < d_{i+1,j} - C_{i+1}$ then $p_i > p_{i+1}$

If a thread got computing time on the CPU, it runs until it is completed or a new thread arrives. When a new thread arrives on the ready-queue, the scheduler has to compute the missing laxity-times and then compare all of them again to distribute the right priorities to the threads.

Example Figure 2.4 shows a part of the following four threads scheduled by the LLF algorithm. It represents a system-load of over 90%. In this example we take the thread with the lower index variable if two or more threads got the same laxity-time.

τ_0: $T_0 = 5$ $C_0 = 1$ $U_0 = 0.2$
τ_1: $T_1 = 10$ $C_1 = 4$ $U_1 = 0.4$
τ_2: $T_2 = 8$ $C_2 = 2$ $U_2 = 0.25$
τ_3: $T_3 = 15$ $C_3 = 1$ $U_3 = 0.067$

Every time a thread is ready, the running thread is paused and the scheduler calculates the laxity-time for every thread again. That is also the reason why τ_1 gets paused at time 15.

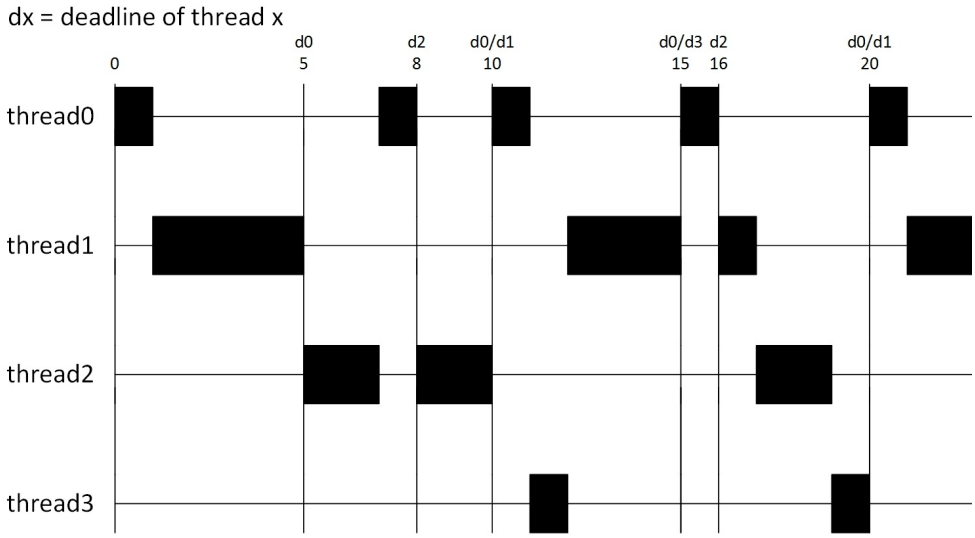


Figure 2.4: Threads scheduled by the Least-Laxity-First algorithm.

Advantages and disadvantages Like shown in figure 2.4, threads like τ_3 may wait long until they get computing-time on the CPU. Also the LLF algorithm needs a lot more overhead then for example the RMA algorithm. If just a single thread was added the LLF needs to calculate all the laxity-times again. That results in lots of calculation if a new thread arrives at every time-unit. Figure 2.5 represent an overload of the system of 20% with the following threads:

τ_0: $T_0 = 10$ $C_0 = 5$ $U_0 = 0.5$
τ_1: $T_1 = 10$ $C_1 = 5$ $U_1 = 0.5$
τ_2: $T_2 = 5$ $C_2 = 1$ $U_2 = 0.2$

Just one thread out of three reaches it deadline because in overload-situations the LLF algorithm works quite unstable and insecure.

On the other side, in non overload-situations the LLF algorithm guarantees, that threads will normally reach their deadlines.

2.1.3 Hybrid Scheduler

Hybrid scheduler are a mix of static and dynamic components.

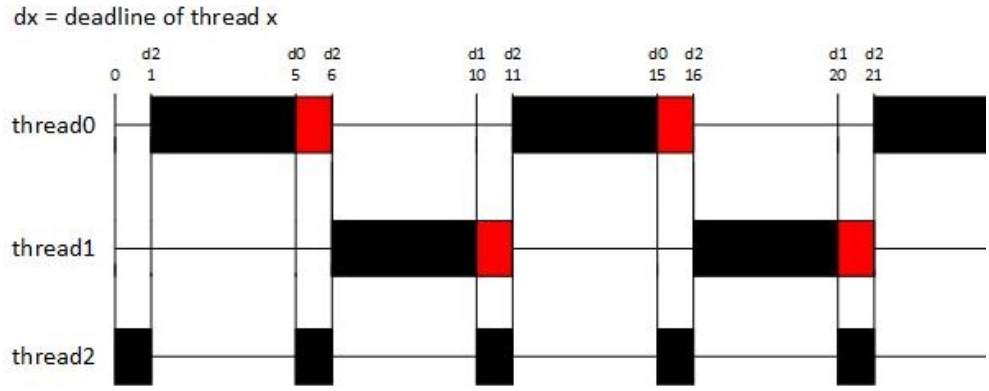


Figure 2.5: partial excerpt of the overload-situation in a system scheduled by the LLF algorithm.

2.1.3.1 Maximum-Urgency-First Algorithm (MUF)

The MUF algorithm is a development of different existing scheduler algorithms to achieve the best possible results in overload situations.

Function The Maximum-Urgency-First algorithm uses not only one, but multiple priorities per thread to make the best decision. It has fixed priorities like the user-priority and the critical-condition. Both values are set once and then never changed, just like with a static scheduler. But the MUF also uses a dynamic priority which is based on the Laxity-Time of the thread.

First the algorithm sorts all threads by their period. Afterwards it defines the first n threads as critical threads so that the Utilisation Factor $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is below 100%. These threads get the property "critical", the remaining threads are "not critical". The critical threads will always proceed. Even during overload situations.

In the next step, each thread gets a user-priority. If the scheduler has to decide, which thread gets computing time on the CPU, it sorts the available threads in the ready-queue by critical and not critical. Then the threads in these two groups get sorted by their dynamic priority, which means the threads with a lower laxity-time get a higher dynamic priority and thus earlier computing-time.

If more than one thread gets the same dynamic priority, the user-priority decides, which thread gets first. Only if more than two threads got the same dynamic priority, the user-priority decides which one is more important and gets computing time first. Only if every critical thread is completed, the non critical threads are handled in the same way as the critical.

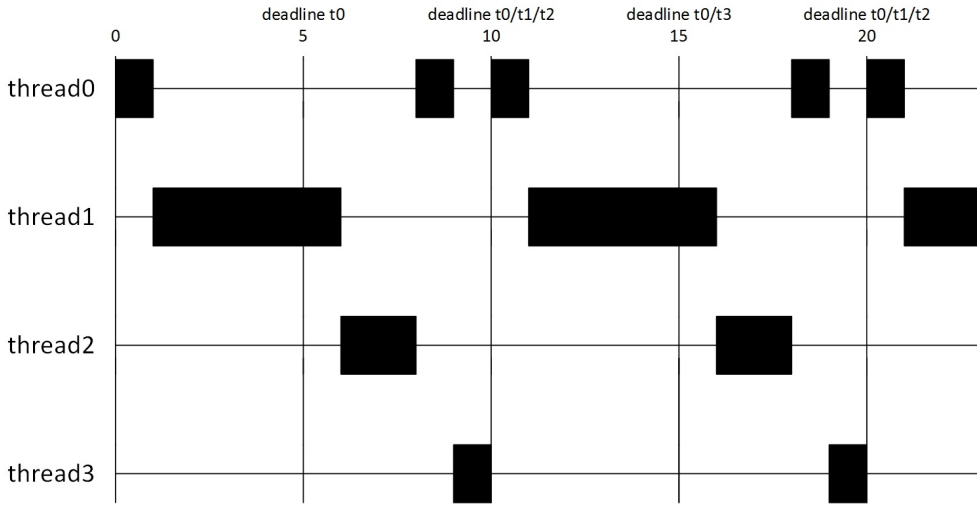


Figure 2.6: Part of the schedule-plan of a scheduler using the Maximum-Urgency-First algorithm.

Example Figure 2.6 shows a partial excerpt of the following threads scheduled by the MUF algorithm. In this case the priority p_i is the user-priority handed out by the user.

τ_0: $T_0 = 10$ $C_0 = 5$ $p_0 = 2$ $U_0 = 0.5$
τ_1: $T_1 = 10$ $C_1 = 1$ $p_1 = 1$ $U_1 = 0.1$
τ_2: $T_2 = 15$ $C_2 = 3$ $p_2 = 2$ $U_2 = 0.2$
τ_3: $T_3 = 20$ $C_3 = 4$ $p_3 = 3$ $U_3 = 0.2$

The Utilisation Factor U_i is 100%. The threads τ_0 and τ_1 are critical. So both of them get executed before the other threads. Because the laxity-time of τ_0 and τ_1 is that different, τ_0 gets always executed first. The remaining execution-time is splitted between τ_2 and τ_3 according to their laxity-time. The user-priority does not matter in this example.

Advantages and disadvantages The Maximum-Urgency-First scheduler supports deadlines just like the RMA, the EDF and the LLF. But unlike these scheduler, the MUF is a scheduler that guarantees the execution of the critical threads even on overload situations. That is a major safety aspect of the system of the project. Also it is possible to prioritize the threads by the user-priority.

On the other side the MUF algorithm needs a lot operations to choose the next thread. First it has to initialize the critical threads and check, that the sum of their U_i is below 100%, and if new critical threads from an other ECU arrive all this must be redone.

Also the algorithm has to store and handle three different priorities. This results in more needed storage and comparisons.

2.1.3.2 Modified-Maximum-Urgency-First Algorithm (MMUF)

The MMUF is a variation of the MUF. It uses an other algorithm to calculate the dynamic-priority [SN06]. In the referenced paper they use the EDF or a modified version of the LLF algorithm [SN06]. In this thesis the version with the modified LLF (MLLF) algorithm is used. The MLLF algorithm was originally introduced as alternative to the EDF algorithm [VN07]. Basically the MLLF works like the LLF with fewer context-switches.

Function MLLF The MLLF works like the LLF. It selects the thread with the earliest laxity-time to run on the CPU. Just in case of a context-switch it calculates if the new thread will also reach its deadline if it starts later [SN06]. If it is possible to start the thread later, the current executing one continues its execution to reduce context-switches [VN07]. The other thread is executing if the CPU is free or its laxity-time is reached.

Function MMUF Unlike the normal MUF first the user-priority is assigned to the thread. The threads get sorted by their value then the first n threads with an Utilisation Factor $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ below 100% are marked as "critical" threads.

If the scheduler has to select a thread, it first sorts the threads in the ready-queue into critical and not critical threads. If there is just one critical thread, the scheduler selects it and the thread gets execution time on the CPU.

If there is more than one critical thread, the thread with the earliest laxity-time (based on the MLLF algorithm) gets computing time. If two or more threads have the same laxity-time, there are two options: If one of these threads is already running on the CPU, select it and let it run. If none of these threads is executing at the moment, select the thread with the highest user-priority.

In case no critical thread is in the ready-queue, the scheduler takes the non critical thread with the earliest laxity-time (also based on the MLLF algorithm). If two or more non-critical threads have the same laxity-time, there are the same two options as for critical thread: If one of these threads is already running on the CPU, select it and let it run. If none of these threads is executing at the moment, select the thread with the highest user-priority.

Advantages and disadvantages Because the thread with the shortest period is not always the most important one, it is a big advantage, that the user has the possibility

to decide which thread is critical and which not. Even in overload-situations of the system, the MMUF guarantees a certain behavior (to schedule the critical threads) and it has fewer context-changes than the MUF. Further advantages of the MMUF against the MUF are shown in "MMUF: An Optimized Scheduling Algorithm for Dynamically Reconfigurable Real-Time Systems" [SN06].

A disadvantage of the MMUF is the huge amount of comparisons the algorithm needs to work. These produce a huge overhead. If an important thread was added, which has to be marked as critical thread, the whole algorithm has to start over from the beginning, or at least it has to be checked if the Utilisation Factor is still under 100%.

2.2 Synchronisation Methods

Synchronisation methods are processes, which allow a secure access from multiple threads to the same resource. It is important to use these methods because if they would not be used, a lot of problems can occur.

Race-Condition If the access to a shared resource is not synchronized, a race-condition can happen. It is very important to find and solve such race-conditions because they can lead to inconsistent operations and wrong data. Just like the name says, time is a critical aspect at race-conditions. Two or more processes "race" to change the value of the shared resource. Often the problem is, that the operations the processes are going to execute are not atomic. This means if it is wanted to increase an integer for example, it is first necessary to read it, then increase the value and then rewrite it. If there is just one thread that increases the integer, there is no problem. But if there are more than one, the race-condition starts.

Figure 2.7 shows the sequence of two threads that both accesses the same integer. Thread 0 wants to increase it and Thread1 wants to decrease it. The integer is initialized with 5. Normally after each thread will be executed once, the integer should has the value 5. Due to the fact, that the access to the integer is not synchronized and the commands of the threads are not atomic, a race-condition occurs and Thread1 overwrites the result of Thread0 without reading it.

Producer-Consumer-Problem Already in 1965 E.W. Dijkstra described the basics of the Producer-Consumer-Problem [Dij65]. There are two periodic processes. One is the producer, which produces every period certain information and write it into a buffer. The other process is the consumer, which consumes every period information from the buffer. Both processes are not able to communicate with each other.

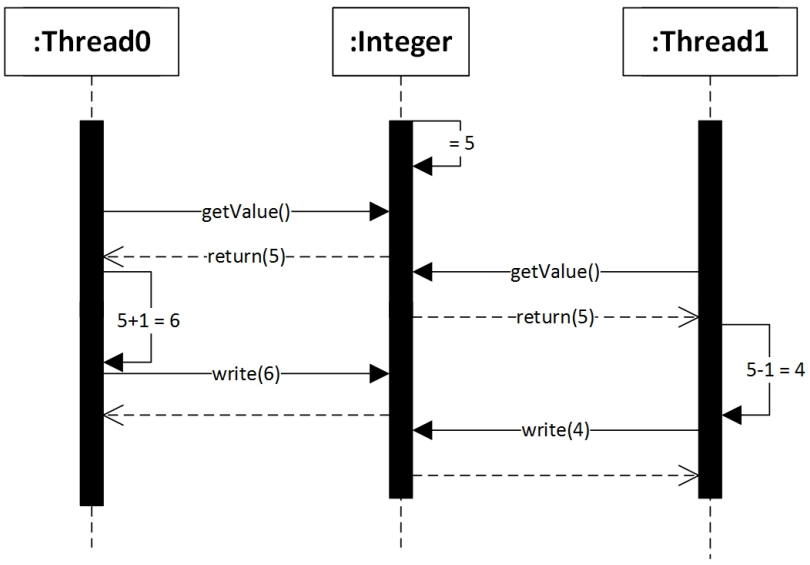


Figure 2.7: Two threads have unsynchronized access to an integer, which results in an race-condition

In our use-case the consumer is the scheduler, which works off the threads in the ready-queue and the controller is the producer, which adds new threads to the queue. The Ready-queue is in this case the buffer on the shared memory. The challenge is, to be able to guarantee that no element in the shared memory gets overwritten or lost.

Also some consumer can not handle the access to an empty buffer. So it is important to make sure that the producer and the consumer both get access to the buffer and no unwanted behavior occurs.

2.2.1 Semaphore/Mutex

Mutex A mutex is a very simple and still very efficient way to synchronize the access to a shared memory. The mutex works like a key. Just one process at a time can have the key and only the process with the key can access the shared data. To get a mutex the process has to request it. If the mutex is already used by an other process, the requesting one has to wait until the mutex is free again. While having the mutex the process is able to access and change all data on the shared memory the mutex is used for. If the process is finished using the shared data, it releases the mutex, so other processes can request it.

Semaphore A semaphore works just like a mutex. If a semaphore is initialized it is important to define the number of processes you want to allow to get the semaphore at the same time [Ral92]. This value is saved in the semaphore. Each time a process request the semaphore, the value is decreased by one. If the value reaches 0 the next requesting process has to wait until another process releases the semaphore again. Then the value is increased by one. This mechanism guarantees that just the defined amount of processes are allowed to access the shared memory. If a mutex be can compared with a key to a room, people can only enter if they have the key. A semaphore would be a counter to a room, which says how many people are allowed to enter to the room at the same time.

Initializing a semaphore with the value 1 result in a semaphore with the exact same behavior as a mutex: semaphore(1) = mutex()

Each time the scheduler wants to read or update the ready-queue it has to request the semaphore and afterwards it has to release it again [Gla06]. Also the controller has to request the semaphore. Each time the controller wants to modify the queue, it also has to request the semaphore.

Advantages and disadvantages If multiple thread needs the same semaphores/mutexes it can easily happen, that each thread already got the semaphore/mutex the other one needs to start. This is called a deadlock. Deadlocks have to be solved by an intelligent logic. If there are multiple readers on the shared resources, every reader needs to get the semaphore or mutex to start reading. Even though multiple reader can read the same resource without bother each other, this results in a great performance loss.

On the other side, using a semaphore or a mutex is really safe. If a reader once red a value, the value is right. No further checking or calculating is needed.

2.2.2 Read-/Write-Lock (r-/w-lock)

The read-/write-lock is a variation of the semaphore. Its behavior is quite similar but there is a distinction bestween reading and writing. If a thread wants to access a shares resource, it first has to request a lock. If the thread is going to read the resource it is requesting a read-lock. If it is going to write on the resource it is requesting a write-lock. When the thread has finished its access to the resource, it releases the lock it requested before.

The main difference to semaphores or mutexes is that multiple threads are able to receive a read-lock at the same time. Any number of threads is able to read the shared resource at the same time. But a write lock can only be attain, if no other write-lock or

any read-lock is used at the time. This means if a thread wants to write on the shared resource, it has to wait until all other threads are finished reading or writing.

Advantages and disadvantages If there are many readers and just a few or only one writer, read-/write-locks are much better than semaphores or mutexes because multiple readers can have parallel access on the shared resource. Also the red value is always correct.

On the other side writers can have to wait a long time until all active readers finished their jobs. Also even with read-/write-locks deadlocks are possible. If there are two threads: thread0 requires read-lock(A) and write-lock(B) and thread1 requires read-lock(B) and write-lock(A). If both threads gets their read-lock simultaneously, they both get their write-lock only, if each of the other returns their read-lock.

2.2.3 Sequential-Lock (seqlock)

The seqlock was added to the linux kernel in version 2.5.60 and originally it was developed to update time variables [Cor03].

There is a sequence-number for the shared resource. If a reader wants to read data in the critical section, first it has to read the sequence-number. After it is finished reading it has to read the sequence-number again. If those two values of the sequence-number are not the same, the reader has to read again [Jon05].

A writer has to request a write-lock to avoid interaction with other writers. After receiving the lock, the sequence-number is increased before and after it writes [Cor03] so the reader recognizes if the sequence-number is odd and a writer is writing. If the writer finishes its job, it returns the write-lock. If there is just one writer on the shared resource, the write-lock is unnecessary and can be saved.

Advantages and disadvantages The main advantage of a seqlock is that writers do not have to wait for readers. They just have to wait for other writers. So the performance of writers is massively increased compared to read-/write-locks. Also based on the sequence-number a reader can recognize if the shared resource was modified or not. Another advantage is that it is not possible to run into deadlocks.

On the other side readers might have to read multiple times until every writer is finished and the sequence-number stops changing.

2.2.4 Read-Copy-Update (RCU)

Read-copy-update is a synchronisation mechanism, which works without locks. It can be splitted in two phases: the first one hides any change from other threads and the

```
public void readsth(){
    int red = 0;
    boolean again = true;
    while(again){
        red = seq;
        read(sth);
        if(red == seq){
            again = false;
        }
    }
}

public void writesth(){
    lock(access);
    increase(seq);
    write(sth);
    increase(seq);
    unlock(access);
}
```

Figure 2.8: Pseudo-code of a reader and a writer using a seqlock.

second releases the changes controlled [Lin05]. According to [MS] it is designed for read heavy use cases. The different to the other synchronisation mechanism is, that RCU works best with resources, that are accessed via pointers. The access to these pointer must happen via atomic operations [Jon05].

Because reading threads do not interact with each other, they do not need special restrictions. If a writer wants to change a value, first it has to read the whole structure from the referred pointer and copy it to a new space [MW07]. This copy will be modified.

After the change is finished the writer just has to change the old pointer with the pointer of the modified version. To guarantee, that two writers do not change the pointer immediately after each other and one of the writers changes gets lost, the change of the pointers also has to secured.

Figure 2.9 shows a short example of using RCU to delete element B of a linked-list containing A,B and C. In step one all three elements of the list are marked red. This means old (readers that already exists) readers can have references and new readers

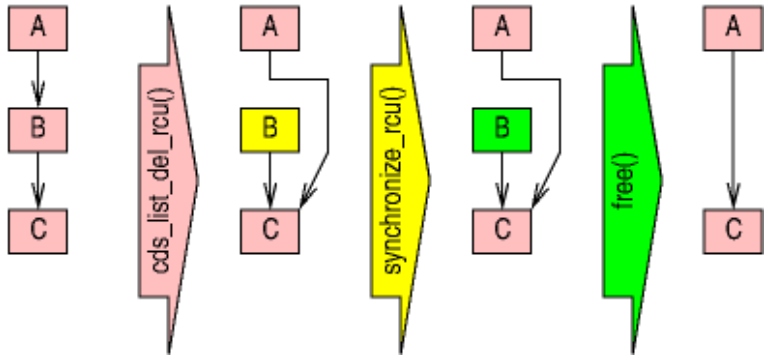


Figure 2.9: Short example that shows the different phases in RCU on modifying a linked-list from [PJ13]

will get references to them. In the second step element B will be removed from the list. New readers will not see element B anymore, but old readers might still have a reference to B that is why B is marked yellow and the link from B to C still works. Step three is waiting for the end of all readers that already exists when B was removed (all the readers that might have a reference to B). If all these readers are finished, it is possible to delete B in step four.

Advantages and disadvantages Forced to work with pointers and structures that need pointers does unnecessarily complicate the structure of the system. Also it complicates the rcu mechanism, since it has to wait for a certain time before it can delete the structure behind the old reference because some running threads still might use the old pointer. RCU can not force readers to reread the shared resource after a write occurs.

2.3 Evaluation

It is important to choose the right components for the OC-based Method for efficient Synchronization. To decide which components are more suitable than others, it is necessary to compare them.

2.3.1 Comparison of the Scheduler

For the comparison of the scheduler algorithm, the requirements defined in 1.2 are used.

Real-time-compatibility: The scheduler has to handle with deadlines. Every introduced algorithm except the FPRR can deal with deadlines. According to [Sch00] LLF is even better than EDF because it can recognize, when a thread will not reach its deadline before it even missed it. Because MUF uses the LLF algorithm to calculate the dynamic priorities and the MMUF can also use a modified version of the LLF, they are both able to do the same.

The scheduler has to guarantee that important threads reach their deadlines, even in overload-situation when some less important threads will miss their deadlines. In the FPRR it is possible to guarantee the execution of important threads by increasing their priority. But it is necessary to pay attention, that the utilisation of the high prioritized threads is not over 100% (like the critical attribute in the MUF and MMUF algorithm). Also this is just an unofficially way to reach the needed behavior, FPRR is still better than EDF or LLF, that can not guarantee anything. With RMA it is kind of the same situation.

Officially it does not guarantee any specific behavior in overload situation, but the threads with the shortest period will still reach their deadlines. The MUF algorithm guarantees that the critical threads will reach their deadlines, but the critical threads are chosen by the length of the period of the thread.

However, important threads do not always have the shortest period time. That is why RMA is even worse than FPRR, and MUF is not as useful than expected. Only MMUF can guarantee, that important threads will always reach their deadlines. And these important threads are freely selectable.

Resources: To keep in mind, that the system is an embedded system, we have also to take a look on the resources the different algorithms need: The overhead the algorithm causes at work and the memory the scheduler needs.

The FPRR just has to handle the different priorities of the threads. To store them in lists, one for each priority, the algorithm does not need any additional memory or comparisons beside the normal ready-queue. Its overhead is the lowest of all scheduler. But the FPRR causes the most context-switches. The RMA needs to reschedule every thread in the ready-queue if threads are added. EDF and LLF have more overhead during the system-run, but they do not have additional load when adding threads. The overhead is the result of the many calculations of the laxity-times and the comparisons of it (LLF) or the comparison of the deadlines (EDF).

Also it is necessary to store the laxity-times. This results in greater need of memory for the LLF. But the most overhead is caused by the MUF and MMUF. Both need the most comparisons of all scheduler algorithms. Additionally they need to store priority values like the user-priority. But the additional memory the algorithms need is negligible in total.

Implementation: The FPRR is the easiest one to implement followed by the RMA, EDF and LLF. The MUF and MMUF are again come off badly. This comparison is strongly related with the overhead the algorithm cause, because a complexer coder produce more overhead.

This results are summarized in the following table:

Name	Deadlines	Safety	Overhead	Memory	Imple.
FPRR	0	2	5	5	5
RMA	3	1	2	4	4
EDF	4	0	3	3	3
LLF	5	0	2	2	2
MUF	5	3	1	1	1
MMUF	5	5	1	1	1

0 = no ; 1-5 = yes, higher is better.

Deadlines:	The ability to handle deadlines and to guarantee their compliance.
Safety:	The ability to work as intended and guarantee at least important deadlines even in overload situations (higher number is more safety).
Overhead:	The load the scheduler is causeing during its work (lower number is more overhead).
Memory:	The memory (not storage) the scheduler needs (lower number is more memory).
Imple.:	The difficulty of the implementation (higher is easier to implement).
Result:	The weighted result (higher is better).

Result of the Comparison Because the safety of the system is the most important aspect and things like overhead, memory demand and implementation expense are relevant for such a project but not even close important to the security of the vehicle, the algorithm with the most points in the table is not the best one. Because it guarantees important threads to reach their deadlines, even in overload-situations, and its pretty good results in other discussed aspects the Modified-Maximum-Urgency-First algorithm is the best choice.

2.3.2 Comparison of the Synchronisation-Methods

The most important requirement is the avoidance of deadlocks and unsafe states, when multiple threads get non synchronized access to a shared resource. The speed of the

write and read operations on the resource is also important.

Deadlocks are an important topic. To avoid Deadlocks, completely lock-free synchronization methods are needed. Semaphores, mutexes and read-/write-locks are not lock-free. Only sequential-locks and read-copy-update are lock-free. But a read-/write-lock is still better than a semaphore or a mutex, because often a reader only needs read-locks and a writer only needs write-locks, so deadlocks appear much less often.

The reading-speed depends on the amount of threads that can simultaneously read the shared resource. While a mutex only allows one reader at a time, semaphores and r-/w-locks can allow multiple.

In semaphores the amount of readers is the same as writers and we can allow only one writer. So a semaphore is as bad as a mutex. A r-/w-lock allows multiple readers to read at the same time, but if a write-lock is assigned, every reader has to wait until the writer is finished. If a seqlock or RCU is used, a reader can still read the shared resource even during a write, but on a seqlock the reader might have to reread the resource.

The write-speed is pretty similar to the read-speed. Seqlock and RCU perform the best, because in both mechanisms the writer does not need any lock. In other techniques writers need to wait until all other writers and all readers are finished before they can receive a (write-) lock.

A further aspect is how correct the shared resource is. This means: can the system be sure that if a value is changed, no thread will read or use the old one? Using a semaphore/mutex or a r-/w-lock only readers or one writer has access to the shared resource. So it is ensured, that every thread reads the correct values. On RCU and seqlock a read can happen while a shared resource is modified.

On seqlock the reader finds out about the modification because the sequential-number has changed. On RCU the reader keeps the old value. Because RCU changes the old version with the modified version of the resource when all threads, which know the old value, are finished and the access on the resource is low. So it is not possible to know exactly when the change happens.

Finally it is important how much overhead the synchronisation produces. In our use-case there are a lot more readers than writers. So r-/w-lock, seqlock and RCU are much better than semaphores or mutexes.

Name	Deadlocks	Read-Speed	Write-Speed	Security	Overhead
Semaphore/Mutex	1	1	1	5	1
Read-/Write-Lock	3	3	1	5	3
Sequential-Lock	5	4	5	5	3
Read-Copy-Update	5	5	5	3	3

Results of the comparison This leads us to the evaluation. The best score of the different mechanisms achieves the sequential-lock. It is better than RCU, because it is possible to know exactly when the change of shared resource happens and readers get notified when a change occurs. The system has to store an additional value (the sequence-number) and has to compare different versions of it, but in some cases it might be irrelevant if the red data is 100% correct and the reader does not have to read again. A semaphore or a mutex does not allow multiple readers to slow down the whole system, the r-/w-lock has tremendous waiting-times for writers, which would also slow down important updates of the ready-queue.

These are the reasons why sequential-lock is the best choice as synchronisation mechanism. But read-copy-update is also good and maybe worth to implement it too, to compare their run-time behavior.

In figure 2.10 the different interactions between the components for an update of the ready-queue by the OC.controller is shown.

The example shows a rather bad situation, because while the scheduler is reading the ready-queue gets updated. Thereby the read takes more time, because the scheduler has to read again, but it is guaranteed that it uses the latest version of the ready-queue.

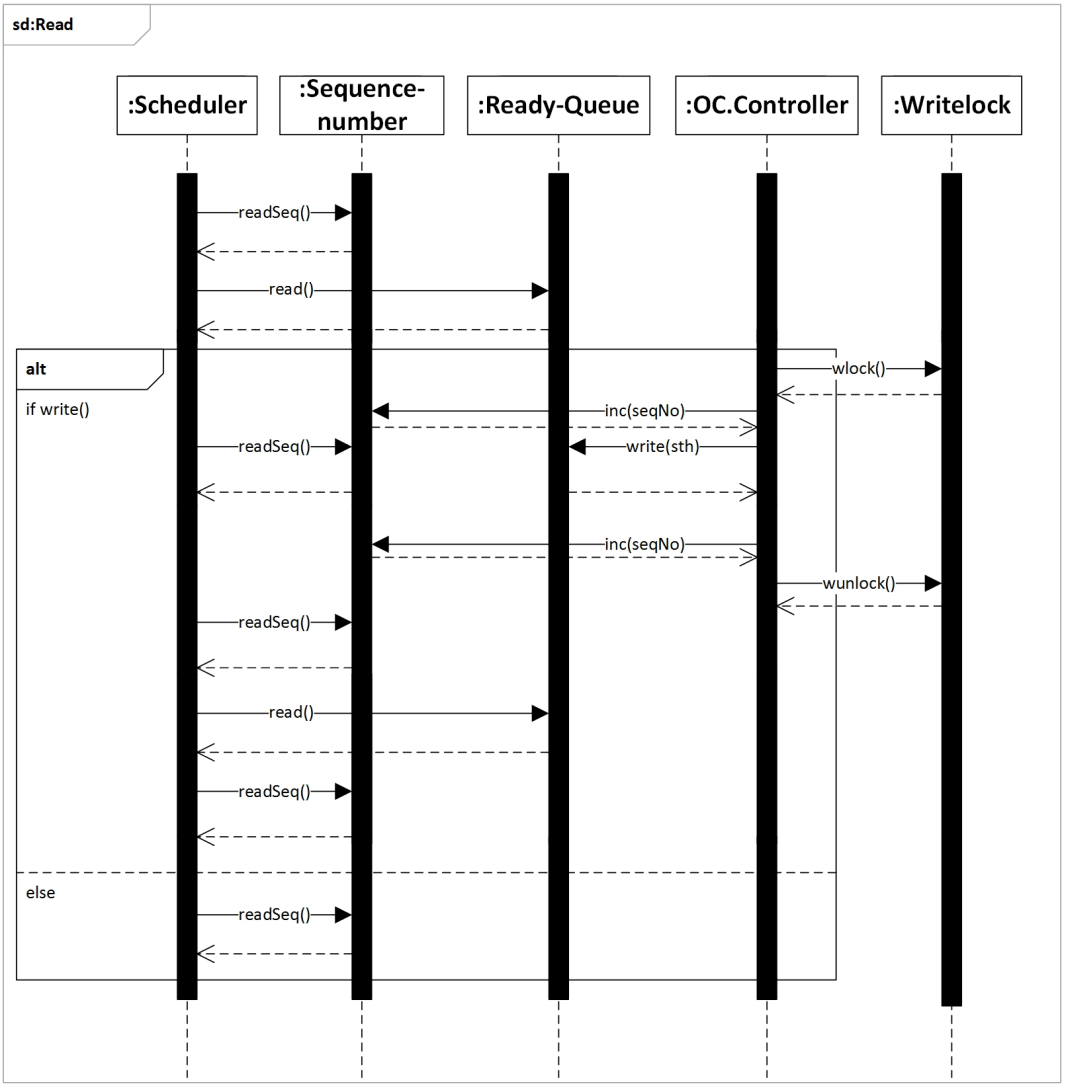


Figure 2.10: Read-access of the scheduler to the ready-queue while the OC.controller writes to it

3 Design

This chapter shows how a design of the chosen scheduler and synchronisation mechanism can look like. It is not an implementation into the L4 Fiasco.OC kernel, but it points out which parts of the kernel have to be modified for an implementation.

Figure 3.1 shows an overview of the relationships between the important components. OC labels the components as part of the organic-computing structure of the system. The task of the OC.observer is to collect data from the scheduler and about the ECU to send it to the OC.controller and the OC.grid. The OC.controller receives additional system-information from the other ECUs in the grid and makes decisions, based on all the data it gets, to add or remove threads from the ready-queue. The scheduler accesses the ready-queue and schedules the threads on the CPU. After a periodic thread is completed it is added again to the ready-queue.

3.1 Integration into the L4 Fiasco.OC

To be able to modify the current scheduler of the L4 Fiasco.OC kernel, we have to take a closer look on it. The current scheduler is the Fixed-Priority-Round-Robin described in 2.1.1.1.

Current Scheduler (FPRR): Every CPU core gets its own ready-queue, therefore the scheduler has to handle all of them. A thread is represented as an instance of the class thread which is a child of the context class which is bound to the sched_context class, storing all the information of the thread, like priority. A thread always belongs to a task. The relationship between these classes is shown in figure 3.2.

The shown class sched_context is the version of the FPRR scheduler (labeled as "Fixed_prio" in the figure). While using the FPRR, the ready-queue consists of a list of lists. Each list represents a priority level of the threads. The function next_to_run() of the class ready_queue_fp returns the next thread that gets computing time (in this case, the first element of the highest prioritized list).

Enqueue() inserts a thread into the end of the list of its priority. The function requeue() is called every time the time-quantum of the thread is finished. It returns the thread back in its list and sets the beginning of the list on the element beyond.

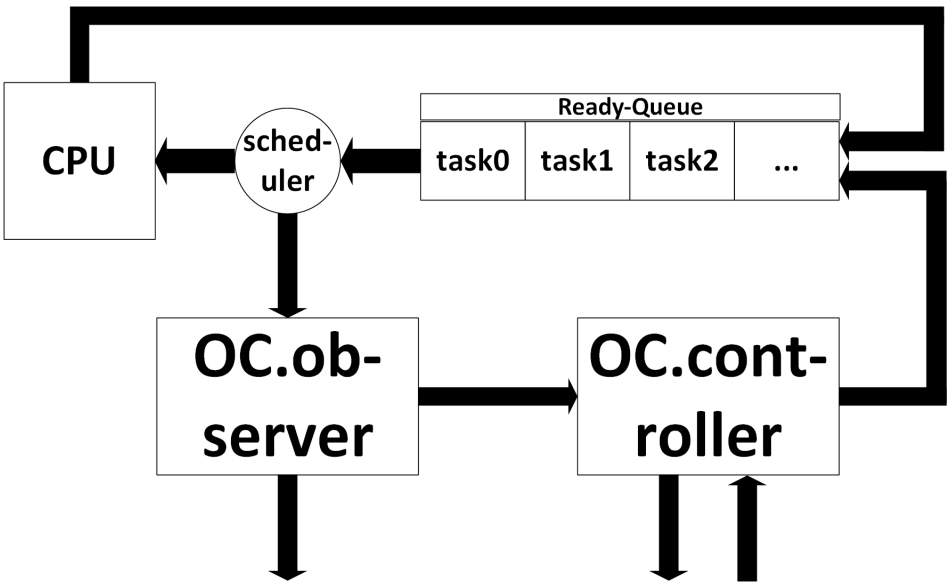


Figure 3.1: Concept of the behavior of scheduler, ready-queue and organic computer paradigm.

Needed Modifications: To store the additional information the MMUF algorithm requires to make its decisions, a modification of the sched_context class is needed. Additional attributes like "user-priority", "critical" and "laxity-time" are necessary. These changes can be seen in figure 3.3.

The MMUF algorithm has to be implemented in the ready_queue_fp class, especially in the function next_to_run(). According to [Hau14] it is very difficult to disable the time-slice mechanism of the Fiasco.OC. This is the mechanism, which is responsible for the exchange of the running thread after Δt.

The easiest way to implement the MMUF is, to use the time-slice structure of the kernel but disable the mechanism, that changes the threads after Δt. To do so, the requeue() function has to re-queue the thread at the exact same position and do not change the beginning of the list (according to the implementation of the edf algorithm in [Hau14]).

It is advisable to keep the structure of the FPRR and store the threads in different lists, one for the critical threads and one for the non critical threads. First the threads in the "critical" list will be scheduled. Later, if this list is empty, the threads in the "non critical" list will be scheduled. For an implementation of the RCU synchronisation mechanism the use of lists is also advantageous. Seqlock does not require a specific data structure for the ready-queue to work.

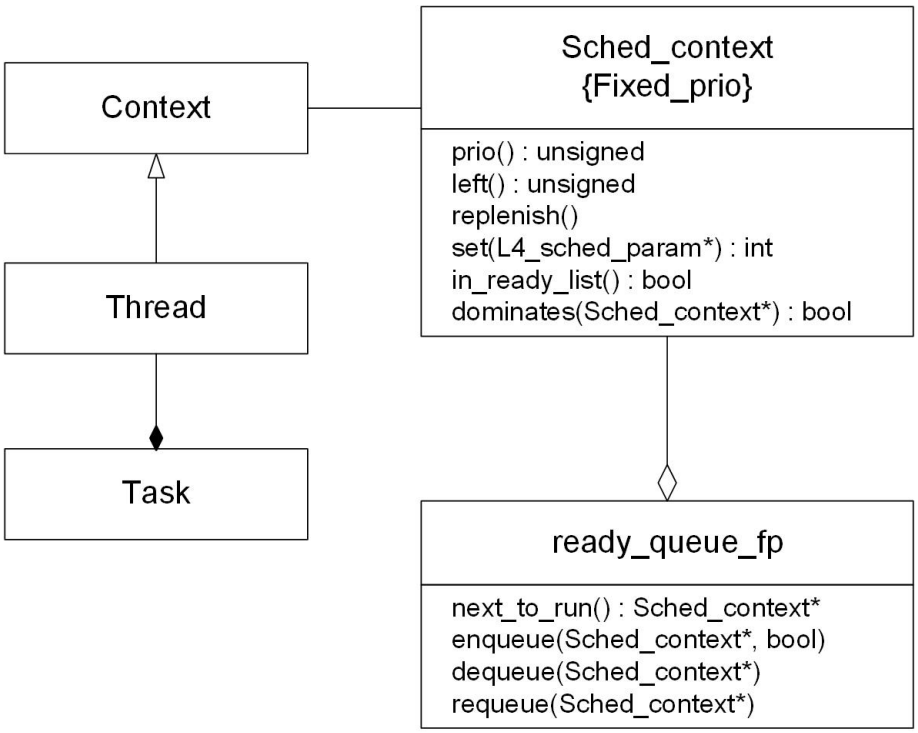


Figure 3.2: The relevant classes in the L4 Fiasco.OC before the modification occurs (related to [Hau14]).

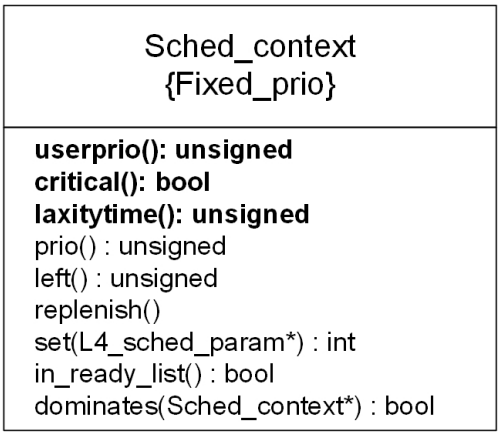


Figure 3.3: Possible design of the new sched_context class for an MMUF scheduler (new attributes are marked bold).

The needed modifications on the run-time environment to support threads with deadlines are already done by Valentin Hauner, who developed it for the KIA4SM project [Hau14].

3.2 Integration into Organic Computing

The integration into the organic-computing paradigm can be done like displayed in figure 3.1 with an observer-controller architecture like shown in figure 1.3. In this case we use the MMUF scheduler, but every other scheduler can also be used.

The job of the observer (shown in figure 3.1) is to just collect information and spread them to the other observers. It does not have to do any calculating. Based on the information they get, the observers in the grid have to calculate, whether their ECU can handle additional threads, or needs to outsource threads to an other ECU. After the results for the own ECU are calculated, the ECUs "talk" to each other to find out, if and how they have to change their own behavior to fulfill all needs.

Figure 3.4 shows the different operations which are needed between the OC.Controller, OC.Observer, the scheduler and the grid, to decide and to be able to insert a thread into the ready-queue. An observer collects information from the other observers in the grid (displayed in figure 3.1 as grid) and the scheduler of its ECU. System-load, free memory and if every deadline will be reached, maybe even a list of the threads that are running on the corresponding ECU can be part of these information, that are forwarded

to the controller, which consults the other controllers in the grid (also displayed as grid) what to do. When it finally gets a thread to insert it to the ready-queue, it has to pay attention if the thread is really able to run on this particular ECU. If everything (like system load and available memory) is fine, the thread gets added to the ready-queue. A much more detailed run of the adding-procedure is shown in figure 2.10.

The calls, which are needed by the synchronisation mechanism, are not included in this figure because the focus is set on the organic computing components and their behavior. The plsupdate() call of the controller to the scheduler can maybe be realised with the sequence-number of the sequence-lock. This means an actual communication between controller and scheduler is not needed.

The task of these controllers in the grid is not only load balancing, they also have to check if all important threads are up, running and will reach their deadlines. If something does not work in the right direction, the controllers have to find solutions on their own. Only then the self-x properties are integrated.

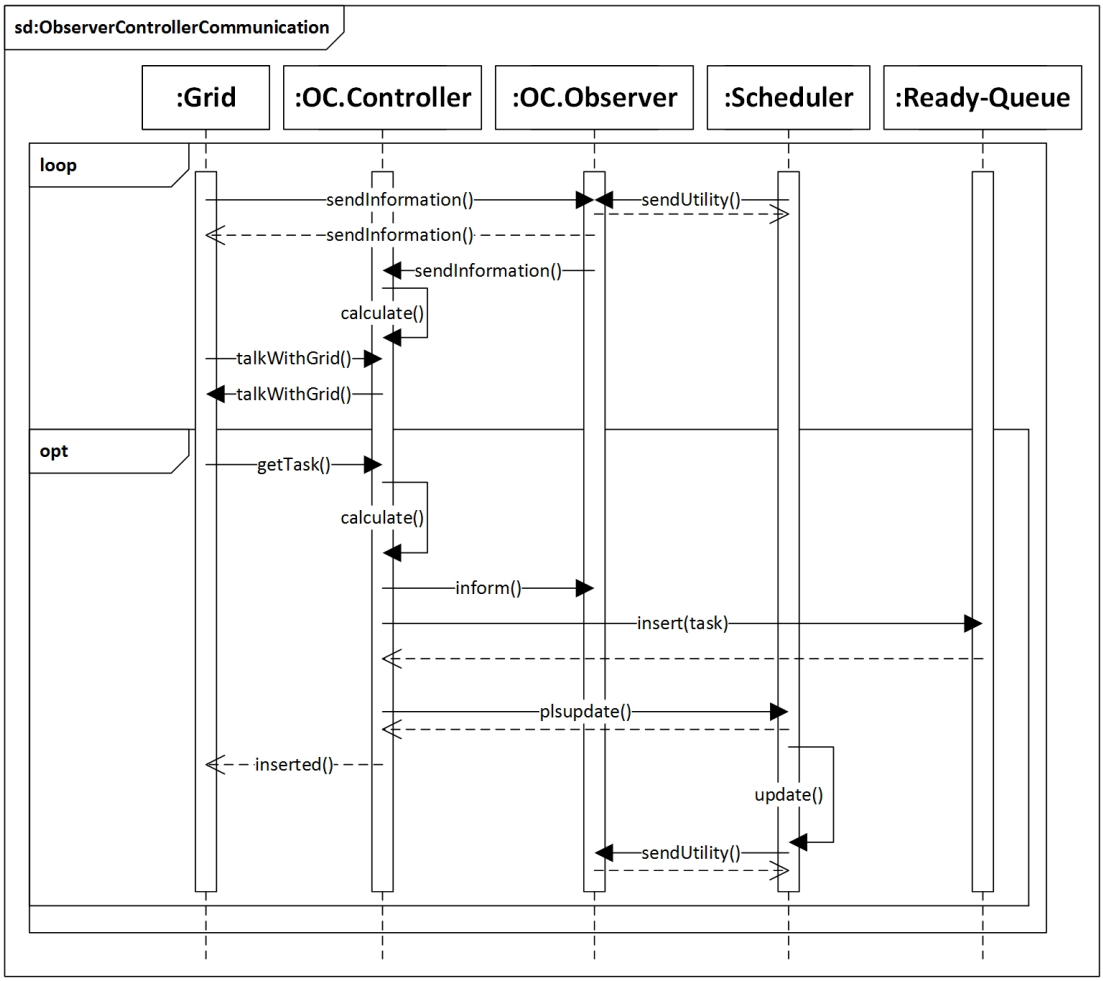


Figure 3.4: Overview of the communication between the different components of the organic computing architecture and their conversation to realise the self-x properties.

4 Conclusion and Future Work

4.1 Conclusion

In this thesis, six different scheduler algorithms and four different synchronisation mechanisms are presented to show which algorithm and which method is the most promising way to use it in an use-case in an automotive real-time-system. The schedulers are compared regarding the real-time and security aspects of their further environment. The current scheduler of the L4 Fiasco.OC (Fixed-Priority-Round-Robin) is compared to the other scheduler, the worst one for the use-case.

The dynamic scheduler Earliest-Deadline-First and Least-Laxity-First are theoretical pretty good scheduler for real-time-systems. But in overload-situations, when the threads need more resources than the ECU can bring up, both of them can not guarantee a minimum of security to the system. For this reason hybrid scheduler, especially the Modified-Maximum-Urgency-First algorithm, are the best choice for a security relevant embedded system.

The synchronisation mechanisms are compared by their performance, safety and deadlock avoidance. While standard locking mechanisms like mutexes, semaphores and read-/write-locks perform quiet solid, sequential-lock and read-copy-update perform even better because they can not cause deadlocks. Thanks to the better control, when the write actually occurs, sequential-lock theoretically is the better choice for our system.

In the end of this thesis a possible design is shown. Also it is explained, which classes of the L4 Fiasco.OC have to be modified to fulfill the new concept and how the organic computing paradigm can be realised by using a distributed observer-controller architecture.

4.2 Future Work

The result of this thesis helps the KIA4SM project to further develop the Kernel of their ECU-grid in the automotive environment. The next steps will be to implement the MMUF scheduler and the sequential-lock and to quantify if the performance of the system still fulfills all the requirements and needs. Maybe a comparison of the measured data from an implementation with sequential-lock and one with read-copy-

update should be done to see, if the theoretical advantage of the sequence-lock or rather the disadvantage of the read-copy-update mechanism really matters that much.

List of Figures

Bibliography

[AR98] J. L. Alex Gantman Pei-Ning Guo and F. Rashid. "Scheduling Real-Time Tasks in Distributed Systems: A Survey." In: *University of California, San Diego, Department of Computer Science* (1998).

[Bra01] R. Brause. *Betirebssysteme: Grundlagen und Konzepte*. Springer, 2001.

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[Cor03] J. Corbet. *Driver porting: mutual exclusion with seqlocks*. last checked: 05.06.2015. Feb. 2003. URL: https://lwn.net/Articles/22818/.

[Dij65] E. Dijkstra. *Cooperating sequential processes, technical report ewd-123. Technical report*. 1965.

[Döb13] B. Döbel. *Microkernel-based Operating Systems - Introduction*. last checked: 05.06.2015. Oct. 2013. URL: http://os.inf.tu-dresden.de/Studium/KMB/WS2013/01-Introduction.pdf.

[Dod06] R. B. Dodd. "An Analysis of Task Scheduling for a Generic Avionics Mission Computer." In: *DSTO Defence Science and Technology Organisation* (2006).

[Gla06] E. Glatz. *Betirebssysteme: Grundlagen, Konzepte, Systemprogrammierung*. dpunkt.verlag, 2006.

[Hau14] V. Hauner. "Extension of the Fiasco.OC microkernel with context-sensitive scheduling abilities for safety-critical applications in embedded systems." In: *FAKULTÄT FÜR INFORMATIK DER TECHNISCHEN UNIVERSITÄT MÜNCHEN* (2014).

[Jon05] A. R. Jonathan Corbet Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition: 5.7. Alternatives to Locking*. last checked: 05.06.2015. Feb. 2005. URL: http://www.makelinux.net/ldd3/chp-5-sect-7.

[Kam09] A. Kamper. *Dezentrales Lastmanagement zum Ausgleich kurzfristiger Abweichungen im Stromnetz*. Karlsruhe KIT Scientific Publ, 2009.

[Lin05] I. Linux Kernel Organization. *RCU Concepts*. last checked: 05.06.2015. Dec. 2005. URL: https://www.kernel.org/doc/Documentation/RCU/rcu.txt.

[LZ10] G. R. Li Jie and S. Zhixiang. "The Research of Scheduling Algorithms in Real-time System." In: *International Conference on Computer and Communication Technologies in Agriculture Engineering* (2010).

[MS] P. E. McKenney and J. Slingwine. *Read-Copy Update: Using Execution History to Solve Concurrency Problems*. last checked: 05.06.2015. URL: `http://www2.rdrop.com/users/paulmck/rclock/rclockjrnl_tpds_mathtype.pdf`.

[MW07] P. E. McKenney and J. Walpole. *Linux info from the source: What is RCU, Fundamentally?* last checked: 05.06.2015. Dec. 2007. URL: `https://lwn.net/Articles/262464/`.

[PJ13] M. D. Paul E. McKenney and L. Jiangshan. *LWN.net: User-space RCU*. last checked: 13.06.2015. Nov. 2013. URL: `https://lwn.net/Articles/573424/`.

[Ral92] J. Ralph C. Hilzer. "Synchronization of the Producer/Consumer Problem using Semaphores, Monitors, and the Ada Rendezvous." In: *ACM SIGOPS Operating Systems Review Volume 26 Issue 3* (1992).

[Sch00] A. Schulz. *Entwurf und Evaluierung von Echtzeit-Scheduling-Strategien in Hardware für einen mehrfädigen Java-Mikrocontroller*. last checked: 08.06.2015. June 2000. URL: `http://www.shark-linux.de/diplomarbeit/diplomarbeit.html`.

[Sch05] H. Schmeck. "Organic Computing – A New Vision for Distributed Embedded Systems." In: *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (2005).

[SN06] V. S. Saman Taghavi Zargar and M. Naghibzadeh. "MMUF: An Optimized Scheduling Algorithm for Dynamically Reconfigurable Real-Time Systems." In: *Information and Communication Technologies, 2006. ICTTA '06. 2nd* (2006).

[VN07] S. T. Z. Vahid Salmani and M. Naghibzadeh. "A Modified Maximum Urgency First Scheduling Algorithm for Real-Time Tasks." In: *International Journal of Computer, Control, Quantum and Information Engineering Vol:1, No:9* (2007).

[Wik15] Wikipedia. *Wikipedia: the free Encyclopedia*. last checked: 05.06.2015. June 2015. URL: `http://en.wikipedia.org/wiki/L4_microkernel_family`.