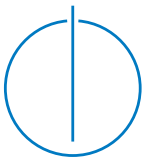
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Prototypical Implementation of a High-Level Synchronization Component for Dynamic Updates of Task Run Queues in L4 Fiasco.OC/Genode

Gurusiddesha Chandrasekhara



DEPARTMENT OF INFORMATICS

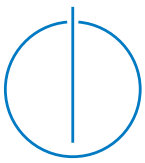
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Prototypical Implementation of a High-Level Synchronization Component for Dynamic Updates of Task Run Queues in L4 Fiasco.OC/Genode

Entwurf und Prototypische Implementierung einer High-Level Synchronisationskomponente fuer dynamische Updates von Task-Warteschlangen in L4 Fiasco.OC/Genode

Author:	Gurusiddesha Chandrasekhara
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Daniel Krefft,M.Sc.; Sebastian Eckl,M.Sc.
Submission Date:	November 15, 2016



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

Munich, November 15, 2016 Gurusiddesha Chandrasekhara

Acknowledgments

Abstract

Contents

Acronyms

ECU Electronic Control Unit

IPC Inter Process Communication

ITS Intelligent Transportation System

KIA4SM Cooperative Integration Architecture for Future Smart Mobility Solutions

EDF Earliest Deadline First

FP Fixed Priority

IPC Inter-Process Communication

SMP Symmetric Multi Processing

RPC Remote Procedure Calls

OC Organic Computing

OC Object Capability System

PD Protection Domain

API Application Programming Interface

RCU Read Copy Update

STM Software Transactional Memory

CPU Central Processing Unit

OS Operating System

L4Re L4 Runtime Environment

QEMU Quick Emulator

RAM Random Access Memory

RM Region Manager

RQ Ready Queue

UTCB User-level Thread Control Block

GNU GNU's Not Unix

GCC GNU Compiler Collection

GMBH full name

GPL full name

1 Introduction

The implemented low-level synchronization component in this master thesis is part of the KIA4SM project of the chair of operating systems. The work aims to implement a method for dynamic update of tasks ready queues in L4 Fiasco.OC/Genode while providing a synchronized access to them.

1.1 Overview of KIA4SM project

KIA4SM (stands for Cooperative Integration Architecture for Future Smart Mobility Solutions) is a research project at the department of operating systems [EKB15]. Traditionally Cooperative Intelligent Transport Systems(C-ITS) have been built on heterogeneous systems. The KIA4SM project aims to provide an architecture of having homogeneous software platform for heterogeneous hardware systems. The project focuses on developing systems for the interaction and coordination between computer-assisted vehicles, be it partially or fully autonomously functioning actors and also aims to improve on the ad-hoc networking between vehicles

The final vision of the project is illustrate in figure 1.1. The goals of the project are,

- A common platform as foundation for device independent (vehicles, mobile devices, traffic and transport architecture) provision and execution of software-based functionality
- Mechanisms that allow for online dynamic reconfiguration, based on
  - en-/disabling and relocation/migration of software-based functionality
  - adaptive (data-centric) routing policy
  - flexible scheduling of tasks per ECU

In order to achieve the goals of the project a number of different methods have been applied. This has led to the application of Organic computing paradigm. Organic Computing (OC) has the vision to address the challenges of complex distributed systems by making them more life-like (organic), i.e. endowing them with abilities such as self- organization, self-configuration, self-repair, or adaptation. In order to realize this, universally applicable Electronic Control Units (ECU) and a common run-time environment are used which provides Hardware/Software Plug-and-Play properties.

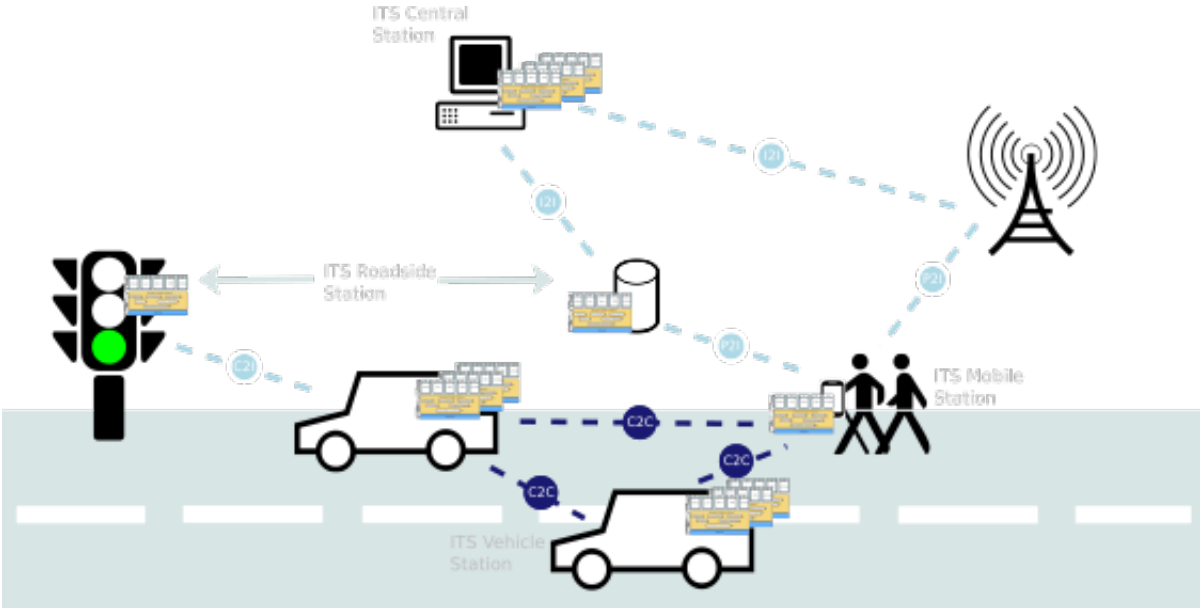


Figure 1.1: KIA4SM vision - homogeneous platform for heterogeneous devices [EKB15]

1.2 Motivation

There are number of micro controllers used for different calculations in a modern vehicle, KIA4SM project aims to replace them with use of more power-full and standardized hardware, universally applicable ECUs.

OC approach proposes a Observer-Controller architecture similar to MAPE architecture(monitor, analyze, plan, execute) . An observer collects the data from the all the ECUs and computes and generates indicators where a controller takes a decision based on the indicator and generates an action.

One such action of the controller is to decide what tasks should be executed at what time in order to meet the aforementioned requirements of the safety critical systems. 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.

The controller decides and produces a run/ready queue (RQ). There needs to be a method which allows to safely update the scheduler ready queue of the system. The work in this thesis concentrates on the scheduler ready queue update mechanism.

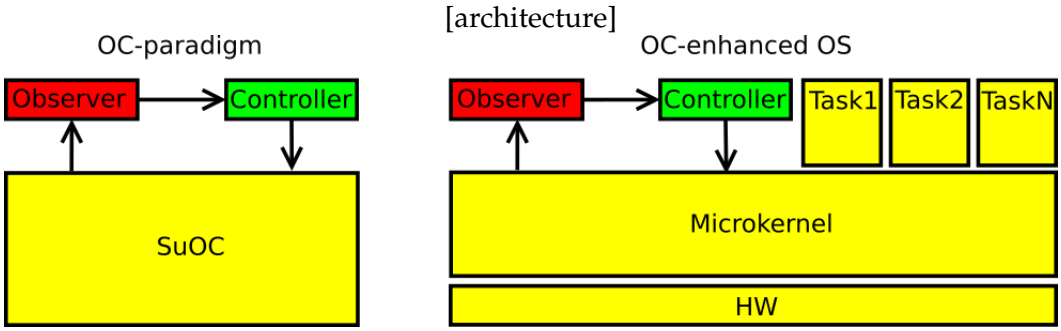


Figure 1.2: Organic Computing: Applying the Observer/Controller pattern to existing microkernel architecture [EKB15]

1.3 Thesis structure

The thesis is structured in a way that the reader understands the importance of the work carried out here and also the surrounding concepts before delving in to the specifics (inverted triangle).

The second chapter summarizes the related work on the state of the art algorithms for synchronization and different types of schedulers in use and at the end of the section an evaluation of the synchronization methods is provided in order to chose the best possible approach for the existing project.

The third chapter explains the Genode and L4 Fiasco.OC details in brief, in order for the user to have an overview of the system.

The fourth chapter deals with the design considerations along with implementation details.

The fifth chapter is dedicated to explains testing method and the results obtained. And the final chapter concludes the thesis with the limitations and future work to be done.

2 Related Work

This chapter explains the previous work and concepts which led to the development of synchronization component to the KIA4SM project and it explains the synchronization algorithms available and makes a comparative study of these algorithms. The comparative study is based on the research work of many papers which are referred here. An attempt is made to pick the best choice algorithm for the work.

2.1 Synchronization of L4 fiasco tasks

The thesis is largely based on the work of Robert Häcker, who in his bachelor thesis *"Design of an OC-based Method for efficient Synchronization of L4 Fiasco.OC Mircrokernel Tasks"* [Hae15], explains the design of a scheduler best suited scheduler for KIA4SM project and also gives comparison study of the different schedulers and synchronization methods suited for updating the task ready queue.

He suggests Modified-Maximum-Urgency-First algorithm as the best choice for KIA4SM project due to the importance of safety and security in embedded systems. After comparing the synchronization algorithms the sequential lock technique has been chosen for the better control it gives. He has suggested to verify the practical implication of Read-Copy Update(RCU). At the end he proposes a design for the existing system including aforementioned scheduler(MMUF) and sequential locks.

This work is an extension of Häcker's findings. However, the focus of the thesis it to develop a good synchronization method, the implementation of the scheduler is not carried out. As go forward I make reference to Robert Häcker's work.

Some of the ideas and code knowledge is taken from the Valentin Hauner's bachelor's thesis *"Extension of the Fiasco.OC microkernel with context-sensitive scheduling abilities for safety-critical applications in embedded systems"* [Hau14]. In this thesis he added EDF scheduling strategy. Though his thesis concentrated on using it in Genode and L4RE environment, it helped in understanding the scheduler.

2.2 Synchronization Methods

To do the justification to many synchronization methods are studied and are explained in this section. The synchronization categories are divided based on the methods they apply. The figure shows the different types of synchronization methods along with examples.

2.2.1 Lock Based Algorithms

Lock based algorithms is a simple way of enforcing the limits on access to shared resource, where there are multiple threads of execution. A lock enforces a mutual exclusion concurrency control. The area where the read or write is happening to a shared resource is called critical section. The thread has to acquire a lock before entering the critical section, only one thread can acquire this lock and whenever it leaves the critical section the thread should release the lock, so that the other threads can enter critical section.

There are many different types of lock based synchronization methods exists.

Mutex Mutex is a synchronization primitive stands for mutual exclusion, which prevents the simultaneous access to shared resource. The thread which wants to execute in critical section has to acquire the mutex and after it leaves it has to give the mutex

Semaphore It's a variable used to control the common resource access between multiple processes/thread. A typical semaphore is initialized with an initial value which is equaled to the number of resources are available and the value is decremented whenever a process takes the resource. If the value of the semaphore is 0 then all the resources are empty and the thread/process has to wait. This works in the opposite way for the event management, where initial value is 0 and semaphore is increased every time an event occurs and decremented when the corresponding event is processed. A binary semaphore has only two values(0 and 1) and works similar to mutex.

Spin-lock It's a type of lock, where the thread wants to access the critical section waits in a loop (spin) while trying to acquire the lock. The thread remains active on the CPU while no useful work is being done. Spin locks have very good advantage if the threads are blocked for shorter periods of time.

The lock based synchronization primitives are simple, easy to implement, however, there are a of disadvantages. If the programmer is not careful, deadlocks can occur and

are difficult to debug. One interesting problem might arrive is priority inversion, which is, if J is in critical section, assign J to highest priority and when it exits critical section assign its original priority. This posses a problem when J is executing in critical section, an higher thread has to wait which is not desirable for real time systems.

Another problem includes thread starvation, where a thread doesn't get CPU time due to long waits, which is again problem for safety critical systems, since threads should reach their deadline.

2.2.2 Lock-Free Algorithms

Lock Free algorithms refer to a synchronization methods where the access to critical section for all the threads is guaranteed without the using of locks. Two major algorithms are discussed in the category of lock free algorithms.

Read-Copy Update: Read-Copy Update(RCU) guarantees concurrent read and write operations to the same data. It's a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort that is optimized for read-mostly situations. RCU achieves scalability improvements by allowing reads to occur concurrently with updates [MW07]. Compared to conventional locking primitives which ensures mutual exclusion among concurrent threads regardless of read/write operation, RCU supports concurrency between single updater and multiple readers. This is achieved in RCU by keeping multiple versions of variables and ensuring that they are not freed up until all the pre-existing readers have finished using the old copies of the variable. RCU uses three main mechanisms to achieve lock-free synchronization method which are,

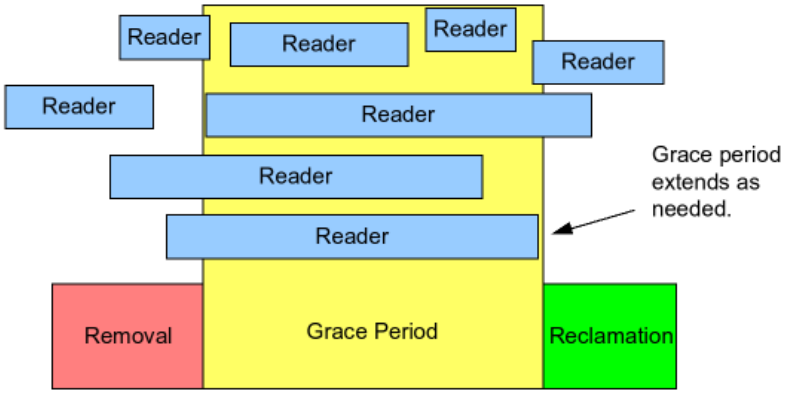


Figure 2.1: RCU showing the usage of grace period [MW07]

Publish-Subscribe Mechanism (for insertion): Publish subscribe mechanism is a way of enforcing the ciompiler to execute the instructions in correct order. If

a pointer is getting updated, an rcu call(rcu_assign_pointer()) is used to change the pointer. This can be thought as publishing the data. This requires the readers also should wait for the update to happen to read the correct data, the dereferencing of the pointer is done by using an rcu call(rcu_dereference()), which is further guarded by rcu read lock.

Wait For Pre-Existing RCU Readers to Complete (for deletion): The figure 2.1, shows this mechanism in picture, where RCU's way of waiting for existing reader threads to finish. RCU introduces a grace period, where it waits on read-side critical section. The simple way to find out when the reader threads have finished the critical section is to check for the context switch of the CPU.

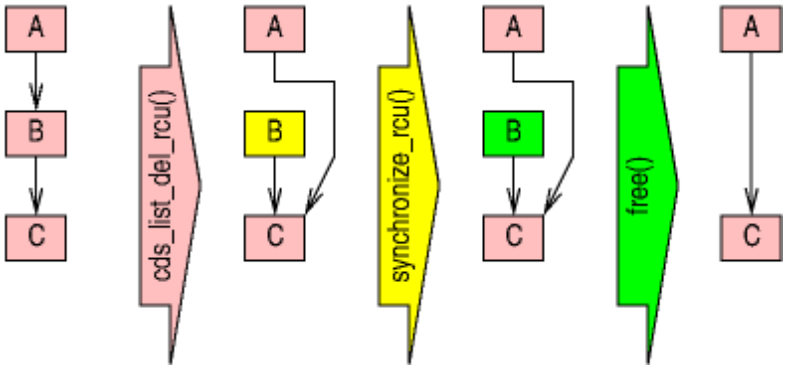


Figure 2.2: Ready copy update mechanism, showing deferred destruction [MW07]

Maintain Multiple Versions of Recently Updated Objects (for readers) : RCU maintains the multiple versions of the data in between removal and reclamation phases. The figure 2.2, shows how RCU maintains the multiple versions of data. In the linked list, B has to be removed. The first phase is changing the link from A to C, but the link from B to C still exists(removal phase). This way, if any readers were at B, they can reach C and any new readers will see only A to C. When the grace period ends, memory for B is freed(reclamation phase).

Since the introduction of RCU, there is been lot improvements made. The research from McKenney et al. shows that RCU can provide order-of-magnitude speedups for read-mostly data structures. RCU is optimal when less than 10% of accesses are updates over wide range of CPUs. Sarma et al. [SM04] work on making the RCU suitable for real-time systems improves on carefully RCU callbacks. They suggest three

methods such as, providing per-CPU kernel daemons to process RCU callbacks, directly invoking the RCU callback and throttling the RCU callbacks so that the limited number are invoked at a given time.

Software Transactional Memory: Software Transactional Memory(STM) is concurrency control mechanism works in a similar way to database transactions by providing atomic and isolated execution for regions of code. The instructions to access shared memory are executed in a transaction, so the other threads will not see any changes when a thread is executing instructions in a transaction. At the end either the transaction is committed(if no other threads have modified the data) or aborted and the transaction is restarted.

STM requires language extensions and compiler takes care of data virsioning and conflict detection mechanisms and makes sure that the global state of the program is consistent. From GCC 4.7, STM support has been added, which makes it ideal to use in this project. The code listing shows the example transaction execution in GCC.

```
1 void testfunc(int *x, int *y){
2 _transaction_atomic{
3 *x += *y;
4 }
5 }
```

Listing 2.1: STM example code in GCC

Other nonblocking data structures which are being researched in embedded systems and other operating system groups are [HH01], Wait-Free algorithms (uses priority exchange from a higher priority thread to lower one which is in critical section) and Lock free sync using compare-and-swap (critical code sections are designed such that they prepare their results out of line and then try to commit them to the pool of shared data using an atomic memory update instruction).

2.3 Evaluation of Synchronization techniques

Criteria	**Mutex**	**RCU**	**STM**
Implementation	+	- -	++
Read-speed	- -	++	+
Write-speed	- -	+	+
Deadlocks	- -	++	+
Overhead	+	+	- -
Security/Consistency	++	-	+

Table 2.1: Evaluation of synchronization techniques

The evaluation of these methods are according to the study of the above mentioned papers and the work of heaker's analysis. Lock based methods and STM are easy to implement since the language constructs provide the mechanism, STM serves better than locks because with using many locks code can become unreadable while STM code has better readability [PA14]. RCU is hard to implement, since the developer has to take care of providing all the read side locks, grace period handling, list operations etc.

RCU is better to use when there are multiple readers, so the read speed is very good while write-speed is still better than locks, which perform the worst in read and write speed since only one thread can access and other threads need to wait. STM's read and write is good, but only drawback it might have to read again, if the data changes in the middle of the transaction.

There is no deadlock problem in RCU, while the deadlocks are common in locks. STM also suffers from data races. RCU and locks have less overhead compared to STM. STM has to keep the logs of the transaction and has to take care to roll-back if something goes wrong. The data is consistent at all times while using locks but this is not same for RCU and STM. RCU has multiple versions of the data and STM has the logs and keeps the old data.

The recent research in STM is increasing, Ferad et al. [Zyu+09], research suggests that implementing STM from scratch is better than trying to convert the programs from locks to STM. The research from Victor et al. [PA14] showed that, TM is very good tool but needs, C++ language refinements and better debugging support and large transactions can hurt the performance.

RCU seems a better choice for using in updating the ready queues but also it adds considerable code, given the fact that it's embedded system. The practical approach of implementing a lock based and a lock free algorithm makes a better case to find out the preferred method.

2.4 System Requirements

Since the system in use is a hard-real time system, there are certain factors to be considered in proposing a solution to the system.

1. Since the system is a safety-critical real time system, the safety becomes major criteria. The system should be in a predictable state always and is able to gracefully handle any error state.

2. Any solution that is proposed should be efficient, since the used system is a embedded system. Since the system contains the less memory and processing power, the code should be simple and efficient.

3. The utilization of the CPU should be intelligent to ensure that the all threads reach their deadline, since the computed value of no use once the deadline has passed. This requires a effective scheduling strategy to be used.

3 Foundations

Genode operating system framework and L4/Fiasco is used as the preferred kernel in this project. Since the kernel itself acts a type-I Hypervisor which allows for partitioning of different software components via separated container, making them work on user mode level [EKB15].

To understand the design and implementation of this project, the reader has to be aware of few important concepts in both Genode operating system and L4 Fiasco.OC kernel. Many concepts here are the relevant parts explained from the Genode book [Fes15] according to author's understanding, but the reader can always refer to the book for detailed explanation.

3.1 Genode Operating System Framework

Genode operating system framework is a tool kit for building secure, special purpose operating systems. Genode is maintained by German company Genode Labs GMBH, which offers both commercial licenses and open-source license under GPLv2. The operating system can be used as an embedded operating system or a fully sophisticated general purpose operating system.

Genode operating system uses a recursive tree structure, where each node in the tree represents a component. Each node is owned by its parent and it controls every aspect of its child like resource control, execution environment etc. The root of the tree is a minimalistic micro-kernel, which is responsible for providing protection domain, threads of execution and communication mechanism between the protection domains. The rest of the operating system's features such as, device drivers, network stacks, file systems, virtual machines are the nodes of the operating systems. Each component gets a share of the physical resources available. The components can grant the resources to its children. If the components wants additional resources it can request its parent, where the parent can grant or deny it.

This makes Genode operating system to have less trusted computing base(TCB) and gives better security features. If a parent thinks that a child is compromised, it can destroy the child while keeping the rest of the system safe. Genode operating system has been developed to strike a balance between various aspects of operating systems. For example, OS has to provide an assurance that threads get fair share of execution

time while accommodating rich and dynamic workloads and similarly for security features and user friendliness.

Genode supports both x86 and ARM CPU architectures and it can be used with most members of L4 family(NOVA, Fiasco.OC, OKL4 v2.1, L4ka::Pistachio, Codezero, L4/Fiasco) of micro kernels. And on Fiasco.OC it supports paravirtualization, a virtualization technique that provides an interface to virtual machines that are similar to their underlying hardware. This method can be effectively used to solve safety related issues of the mixed-criticality system such as the one presented in the KIA4SM project. This makes a very good choice of operating system to use for KIA4SM project.

3.1.1 Source Tree Structure

At the root of the directory there are 3 folders, `doc/` which contains the documentation in and the release notes of all versions. `tool/` folder for build systems and tools used in the system and finally `repos/` which contains the source-code repositories.

Inside the `repos/` folder `base/` repository contains the basic framework related code and `base-<platform>/` folders contain the platform specific code, where <platform> refers to foc, which contains related code for Fiasco.OC etc.

3.1.2 Capabilities, RPC Objects, Protection Domain

Genode consists of many components and each component lives in a `protection domain`, which provides an isolated execution environment. The resource is abstracted to an `RPC object` and a token which gives access to this RPC object is called `capability`.

When RPC object is created, Genode creates a so called `object-identity` which represents the RPC object in kernel space. The kernel maintains a capability space, which holds reference for the object identities. This capability space is explained in detail in The capabilities can be passed to different components, this operation of transferring capability from one protection domain to another is called `delegation`

3.1.3 Client-Server Relationship

Capabilities are used to call methods of RPC objects which are from different protection domains. The component that uses the RPC object is called client and the owner of the RPC object plays the role of the server. The server should at-least have one thread called `entrypoint`, which gets activated whenever when the client calls the method.

Clients generally have to trust the server since they are granted from the parent through session invocation. Servers do not trust their clients.

3.1.4 Component Creation

Genode component is made up of five basic ingredients,

RAM session Allocates memory for program's BSS and heap

ROM session Contains executable binary

CPU session creates initial thread of the component

RM session manages the component's address space

PD session represents the protection domain

As mentioned earlier each Genode component is created out of a parent and the parent is responsible for granting these sessions to child.

3.1.5 Inter-component Communication

Genode provides three principle mechanisms for inter-component communication namely synchronous remote procedure calls (RPC), asynchronous notifications, and shared memory. Synchronous RPC is the prominently used communication mechanism in the Genode world, since this not only able to transfer the information but has the ability to delegate the capabilities and has the authority throughout the system. RPC mechanism is similar to a way the function calls work, where control transfer between caller and callee happens. Synchronous RPC mechanism uses kernel's IPC to transfer messages between client and server. The asynchronous notification is helpful when the caller doesn't want to wait for the control to come back. The Synchronous RPC mechanism can be used in this work but there might be a bulk transfer of data between Controller and Synch component. So the idea of Shared memory concept came as a replacement and is very well suited.

The steps taken in allocating and using shared dataspace are,

1. The first step is allocating the memory, server does this by interacting with core's RAM service to allocate new RAM dataspace. This space is owned by server
2. The server attaches dataspace to its own RM session. This makes the dataspace contents available in it's virtual address space.
3. Server delegates the authority to client whenever a request for the dataspace.
4. The client can attach the obtained dataspace from the server to its own RM session to access the contents

All of these methods are rarely used in isolation and most of the communication methods are used in combination of these methods. In this particular work RPC and shared memory are used in combination.

3.1.6 Scheduling

Scheduler maintains a list of so called scheduling contexts and each of these refers to a thread. Each time the kernel is entered, the scheduler is updated with the passed duration. When updated, it takes a scheduling decision by making the next to-be-executed thread the head of the list. At kernel exit, the control is passed to the user-level thread that corresponds to the head of the scheduler list.

3.1.7 Trace Service

Genode has a Trace service, which can be used by the user level components for light weight event-tracing. This can be used for obtaining the thread related information for all the threads available in the system. This service is used in the testing of the thread update mechanism which will be explained later in chapter 6.

3.2 Overview of L4/Fiacso.OC

Fiasco.OC is a 3rd generation capability-based microkernel which belongs to L4 family of microkernel. Fiasco provides multi processor support and hardware assisted virtualization and paravirtualization. It is capable of real-time scheduling and is scalable from embedded to HPC systems [OS a]. These features made Fiasco.OC has the ideal choice for using in this project. Fiasco.OC can used with L4 Runtime Environment (L4Re), which provides necessary support to develop applications or with an operating system such as Genode.

The OC in Fiasco.OC stands for Object capability system. In this kernel everything is represented as an object and they interact with each other with kernel provided IPC mechanism. Since the system is built around objects, where each object provides a service which other objects can use. The capabilities in Fiasco.OC represents references to the kernel objects and are stored in a per task capability mapping tables which increases the security of the kernel. The figure 3.1 [OS b], shows the per task capability table. Kernel provides a factory for object management.

The table 3.1 shows the kernel provided objects.

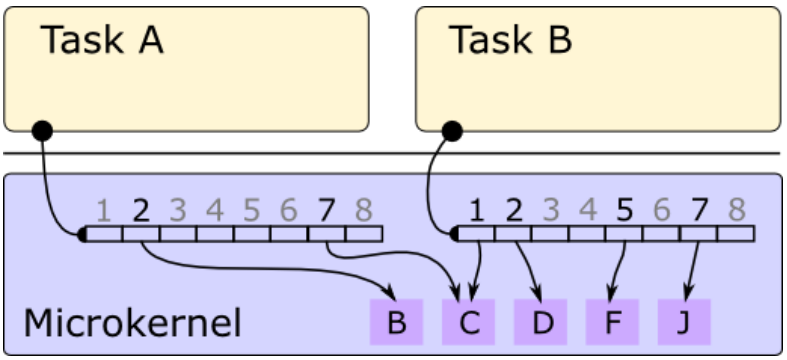


Figure 3.1: Per process capability table in Fiasco.OC [OS b]

Kernel Object	Description
Task	Represents a protection domain in which the threads can execute
Thread	Unit of execution in a task
Factory	Used for creation of kernel objects
IPC Gate	Kernel provided IPC channel
IRQ	Interrupt request signal object used for asynchronous signaling
ICU	Hardware interrupt request controller
Scheduler	The scheduler object managing the CPUs

Table 3.1: The Fiasco.OC kernel objects

The kernel has a object space which contains the capabilities to objects. This can be considered as a pool of objects and a look can be done with the given id to find out the object. Fiasco.OC uses so called flex pages which are passed via IPC and are used to identify the kernel objects. A thread object in Fiasco.OC represents a execution unit and belongs to a task which provides a protection domain for the thread to execute. The thread has 3 states in Fiasco.OC, `ready, running, blocked` states.

The thread holds a User-level Thread Control Block(UTCB), mainly used for system call parameters and has following contents,

- MR message registers, contains untyped message data and items
- BR buffer registers, receive buffers for capabilities
- TCR thread control registers used for error codes and user values

The `Thread class` implemented in kernel acts as a driver class which controls most of the functionality. The execution of thread has scheduling context and an execution context which are explains in the next section.

3.2.1 Scheduler Context

Steinberg - who developed Quality assuring scheduling in the Fiasco Microkernel explains some of the concepts of the Fiasco in his thesis [Ste04]. Steinberg decouples the execution and scheduling in order to help with the IPC and thread time donation in Fiasco microkernel. In which a new class is introduced called Sched_Context containing all the scheduling and accounting parameters. And the class that implements execution contexts is called Context

The regular scheduling context becomes part of the TCB and additional scheduling contexts for each thread via SLAB allocator. This separation of execution and scheduling context allows the system to do fast IPC (The sender of an IPC donates its time quantum to receiver, which gets activated and avoids the invocation of the scheduler). During the execution of a thread, kernel switches to the execution context and the scheduling context of the thread to be executed. When a situation arrives where the thread is donating its time and priority to other thread, the kernel has to only switch the execution context of the destination thread and the scheduling context remains the same.

The scheduling context object is implemented in a `sched_context` class and the `Context` class represents the execution context or thread.

The table 3.2 shows the scheduling context's attributes and their meaning.

Attribute	Description
Owner	This is a pointer, which points to the owner thread of the scheduling context
ID	A positive number to identify the scheduling context, the regular scheduling context is assigned 0
Prio	Priority of the scheduling context
Quantum	The total time quantum associated with the scheduling context
Left	The remaining time of the original time quantum
Prev, Next	Pointers pointing to the next and previous scheduling contexts
Per_cpu<Ready_queue> rq	Processor specific ready queue
Sc_type	Enum object, the type of the scheduler (Fixed priority or EDF)
Deadline	In case of EDF scheduler, the deadline of a thread

Table 3.2: The attributes of scheduling context

3.2.2 Ready Queue

The ready queue holds a list of threads which are ready to be executed next. The Fiasco.OC microkernel supports 256 priorities ranging from 0 to 255. In case of fixed priority scheduler, there is a list exists for each of the priority levels. The scheduler works in round robin fashion, where it picks the head of the highest priority thread to execute, it runs until a certain time quantum and choses the next thread in the list. In case there are no threads to be executed in the highest priority list, the scheduler picks the thread from next highest priority list. Note to be taken in Fiasco.OC kernel the thread that is on the CPU will not be in the ready queue, once the thread finishes it's allocated CPU time for the run, it is re-queued back to the ready queue list(provided it's time quantum still exists).

In Fiasco.OC `prio_next[256]` represents the ready queue list and uses a variable called `prio_highest` to determine the highest priority available.

3.2.3 Enqueue and Dequeue operation

Modification of ready queue operation is a critical section. The uniprocessor implementation uses a simple CPU lock to disable the interrupts and SMP processors uses CPU specific ready queue operation. In order to use per CPU ready queue list, the kernel has to ensure the preemption is disabled. Enqueue operation takes a scheduler context

object to be enqueued and checks if it's already in the list. This is checked against the corresponding priority list. If the thread is not in the ready queue list, it is enqueued and prio_highest variable is set accordingly.

The dequeue operation is similar to enqueue, after disabling the CPU preemption, it deletes if the thread exists in the list. Once the operation is completed the CPU preemption is enabled.

3.3 Thread Creation calls in Genode and Fiasco.OC

Thread creation Genode application is done by creating a class and inhering the Genode `Thread` class. This class takes stack size as template parameter. The diagram shows the sequence of calls taken from application to all the way down the kernel level. The derived class needs to have a constructor and an entry function. The entry function implements the work done by the thread. When the thread objects are created and the start method of the Genode Thread class is called, the entry method starts executing. The code listing 3.1 shows the inhering the Thread class to create user threads.

```
1  class Mythread : public Genode::Thread<2*4096>
2  {
3    public:
4
5      Mythread() : Thread("MyThread") { }
6
7      void entry()
8      {
9        printf("I'm a thread\n");
10       // Some useful work
11     }
12 };
```

Listing 3.1: Thread creation class

Once the Genode application creates the thread the sequence of calls take place.

1. The kernel specific code takes over in the Genode, In case of Fiasco.OC, the call goes to `base-foc/src/core/Platform_thread.cc`. Platform_thread.cc implements the major functionalities to interact with the kernel, such as creating the thread, setting the thread related values, selecting the affinity space and priorities for the thread etc. Further there are two types of platform_thread constructors available. One for the core threads which takes just the name of the argument and other for user level threads which takes the name and priority of the thread. The

platform thread constructor calls `_create_thread` and `_finalize_construction` .

2. The _create_thread calls l4_factory_create_thread API with the L4_BASE_FACTORY_CAP and threads local ID. This call creates L4 thread in kernel.

3. The _finalize_construction calls L4 APIs to create IRQ, set the name of a thread in kernel and sets the scheduling parameters (l4_scheduler_run_thread).

4. `l4_scheduler_run_thread` makes an IPC call to scheduler kernel object, by which kinvoke call of the scheduler object is invoked.

5. The sys_run call identifies the thread and scheduling parameters associated with it. The scheduler_context object is created with the defined scheduling strategy(FP or EDF). The thread is then migrated to the corresponding CPU and added to the ready queue.

4 Design

This chapter gives a high-level overview of the design of `Observer-Controller` architecture used in the KIA4SM project. It also describes the design details of the `Synchronization` module and the communication architecture between the `Synchronization` module and `Controller` module.

4.1 Overview

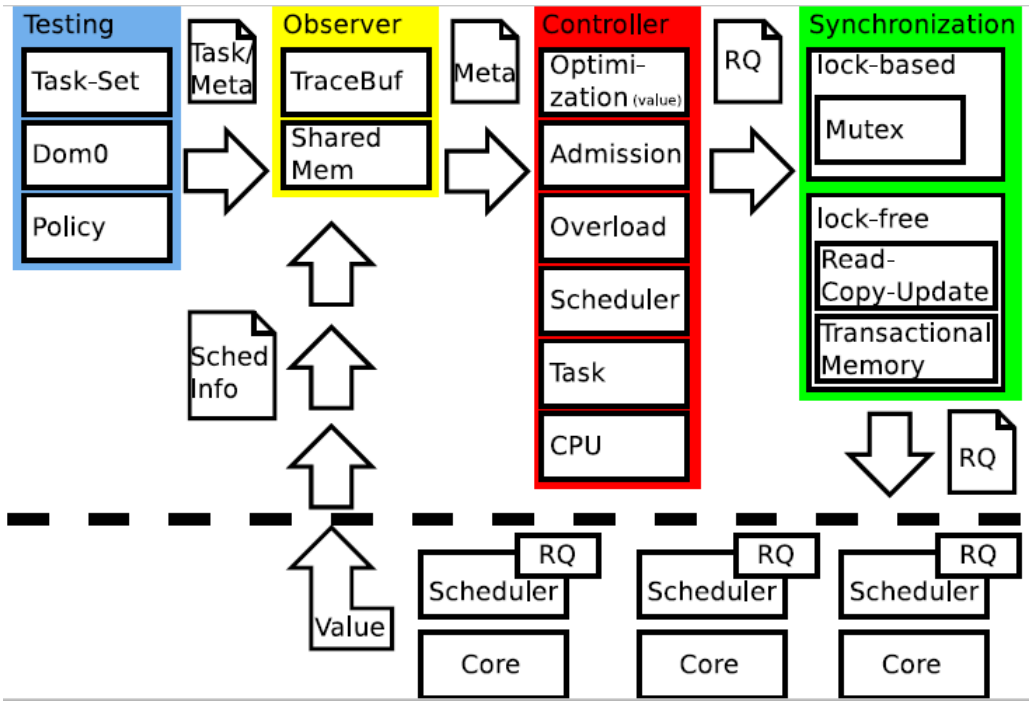


Figure 4.1: The Observer- Controller architecture

The figure 4.1 shows the `Observer-Controller` architecture. Its goal is to have a system that can automatically take the decisions of scheduling tasks/threads by observing the system. There are three main modules involved, namely, `Observer`,

`Controller` and `Synchronization`. All these modules are implemented as Genode OS applications (or components).

The `Observer`'s (also called Monitor/Monitoring agent) job is to collect all the information from the system, analyse it and send it to the `Controller`. The `Observer` collects the information by using the `Trace` facility available in Genode. The data collected from the `Observer` includes *number of tasks*, *process ID*, *execution time of each task*, *RAM info*, *CPU quota*, etc.

The `Controller` is responsible for maintaining the system in a safe and optimized state. It analyses the data collected from the `Observer` and takes a decision for system reconfiguration[EKB15]. The system reconfiguration involves updating a single thread to the ready queue or generating a new ready queue. It then communicates the system reconfiguration data to the `Synchronization` module. The `Controller` also applies load balancing or energy saving techniques to keep the system in an optimized state.

The design of `Synchronization` component is quite complicated. The initial design consisted of having a high-level component, which can directly access the kernel ready queue and exchange with the ready queue given by the `Controller`. This requires synchronized access to the kernel ready queue. However, further research showed that this is not possible due to the following reasons:

- Kernel ready queue cannot be directly accessed from a high-level component such as the one proposed due to the kernel protection domain.
- The `Controller` cannot create the ready queue that can be directly used by the kernel. This is because the thread ids represented in Genode OS framework differs from the thread ids in the kernel. Moreover, the ready queue list in the kernel has scheduler context objects in it (explained in 3.2.2 and 3.2.1). So a user-level component such as `Controller` cannot access scheduler context objects.
- The list type for ready queue is implemented in a template class called `Dlist`, which is accessible only from kernel and needs a re-implementation in the user-level component.

The above mentioned reasons forced a change in design, which is shown in figure The Synchronization module was split into two major parts. The first part is implemented as a high-level Genode component which is responsible for communicating with the `Controller` and passing the data to the second part. The `Controller` and the `Synchronization` module interaction is that of a producer-consumer relationship. The producer (`Controller`) produces a thread (or ready queue) and consumer (`Synchronization` module) consumes the thread by updating it to the kernel ready queue. This needs a synchronization mechanism to be implemented between these two components, which is explained in subsection 4.2.

The second part of the `Synchronization` module is responsible for taking the thread data from the user-level component and updating the threads to the ready queue. It is implemented in the Genode operating system, with most of the work done in L4/Fiasco.OC microkernel. It has to ensure that the access to the ready queue is synchronized. It should also makes sure that the updated threads execute and the system is in safe state.

The design has been broken down into submodules and are shown in figure 4.2.

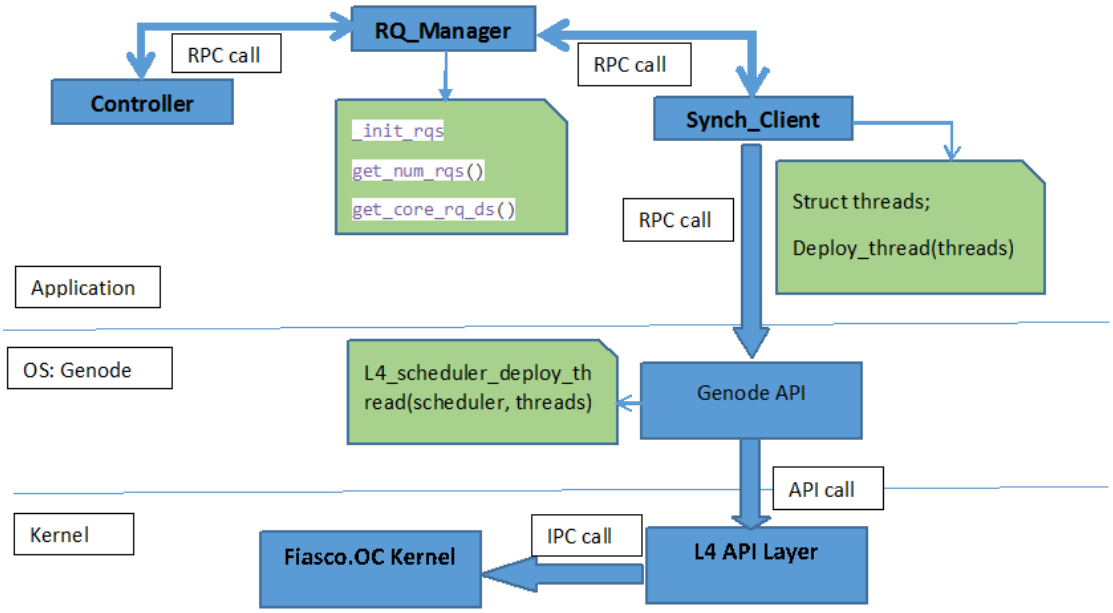


Figure 4.2: The data flow between Synch component, Rq manager and Controller

The `Rq_manager` is a Genode component and it's responsible for providing the communication interface between `Controller` and `Synchronization` component. The main aim is to have a fast communication between the components, in order to reduce latency. The best choice for providing a communication interface in Genode is creating shared dataspace as explained in section 3.1.5. The `Rq_manager` acts as a server in Genode by providing its services to the clients. This is the first component created and in the execution sequence and it provides RPC interface to its clients. This component is only required to provide the initial communication interface and dataspace management and the clients which accesses the server's services will directly communicate once they obtain the shared dataspace.

The `Controller` and he `Synch_client` are Genode components and act as clients to

the Rq_manager. The Controller uses the Genode RPC calls to obtain the dataspace capability from Rq_manager. Once it has the access, it updates the dataspace with thread information. This information contains the thread ids as represented in the Genode OS framework, processor id on to which the thread to be scheduled and priorities of the threads. However, the Controller doesn't enforce any policies to the Synch_client. So both these components work independently from each other but it is synchronized via dataspace.

The Synch_client also uses the Genode RPC calls to obtain the dataspace capability from Rq_manager. Once it obtains the access to shared dataspace, it checks for the thread information and transfers the thread ids, priorities, CPU information to the kernel. The communication between Genode and Fiasco.OC kernel is governed by L4 API calls. However, the Synch_client being a high-level Genode component, it cannot make use of L4 API calls by itself. In order to communicate with the kernel, the Synch_client makes use of Genode OS framework's service. It first creates Genode's RPC object, then using this it makes a call to the API of Genode's RPC object. The Genode's RPC object makes an API call to L4 layer. The L4 API communicates the information to Fiasco.OC kernel object by using IPC service provided by the kernel. The kernel is responsible for identifying the kernel threads and updating them to the ready queue.

The following sections give detailed design of the individual modules.

4.2 Rq_manager module

The communication part of the work was collaborated with Paul Nieleck, who worked on the Controller part of the Observer-Controller architecture[Nie16]. As Nieleck designed and implemented this module, only the details that help to understand the Synchronization component are described below.

The Rq_manager has two main functionalities. First, creating a shared dataspace using Genode's APIs and providing a mechanism to access and modify the shared dataspaces to its clients. The Rq_manager acts as a server in Genode's client-server relationship. It interacts with the Core's RAM service and obtains a RAM dataspace. The Core's RAM service returns a dataspace capability after allocating memory. Then, the Rq_manager attaches the dataspace to its RM session and whenever a client requests this dataspace capability, the server delegates it. Once the clients obtain the dataspace capability, they can attach it to their own RM session and access the contents inside the dataspace. Now, both the server and clients have access to shared memory via their respective virtual addresses.

4.3 Synch_client module

The Synch_client module is a Genode component that is responsible for updating the threads to the ready queue. It communicates with the Controller using a shared dataspace created by the Rq_manager. This acts as a client to the Rq_manager and obtains the dataspace capability from the Rq_manager via an RPC call. It does so by creating a Connection object provided by the Rq_manager. This is also responsible for transferring the thread update information to the kernel.

Once the dataspace is accessible, the Synch_clinet runs in an infinite loop to continuously check the dataspace for any thread that needs to be scheduled. Once it finds such a thread, it updates the thread to the ready queue and then, informs the Controller about the scheduling by updating the shared dataspace. This way, the Controller knows whether the thread scheduling was successful.

Furthermore, the Synch_client communicates with the kernel. However, this is done by using the Genode's Trace service. It creates a Connection object provided from the Genode's Trace service to make an RPC call to the Trace object. It makes RPC call with the thread information and waits for the call to return. The return value indicates the success/failure about updating the threads to the kernel ready queue. It updates the same information to shared dataspace. The Controller takes a decision on the updated information from the Synch_client.

4.4 Ready queue update Mechanism

Ready queue update mechanism is the process of reading the ready to execute threads and making sure that they are updated in the respective ready queues of the processors. Figure (class diagram, which involves, Synch_client to scheduler class) shows the high level overview of the involved classes in this process. A Genode module provides an API, which can be called from the Synch_clinet. The L4 module provides API calls, which can be called from Genode. The Fiasco.OC kernel module accesses the incoming threads and updates them to the ready queue.

The following subsections give an overview of the modules shown in figure (class diagram)

4.4.1 Genode module

Genode module is a component which provides an API that can be called from the Synch_client. The Synch_client communicates with L4/Fiasco.OC kernel by using this module. Since L4 calls cannot be made directly from the application, a Genode helper component is necessary to make these calls. The Genode API acts as a wrapper

to the L4 calls. It takes the information from the `Synch_client` and copies it to the data structure used by the kernel and calls L4 API. The call to L4 is a blocking one. It waits for the L4 call to return and checks for the success or failure. Based on the return value of the L4 API, this returns 1(ready queue update successful) or 0(ready queue update failure) to the `Synch_client`.

4.4.2 L4 module

L4 module represents the L4 API layer, which governs the communication between Genode and Fiasco.OC. Genode API makes a call to L4 using an API provided by L4 module. L4 module is responsible for obtaining the thread ids from Genode and pass them to the Fiasco.OC module using an L4 IPC call. The L4 provides many facilities to the kernel, such as utilities for different platforms, GCC libraries, L4 system calls(L4sys). This L4 API written is in L4sys code. Once the thread ids are obtained from the Genode API, it extracts and fills them in kernel's UTCB(used for transferring function call parameters) to make the IPC call.

Additionally, the L4 module is also responsible for converting the thread ids to kernel understandable flex pages (explained in 5.3.2). Once the IPC call returns, this module returns the same value back to the Genode module.

4.4.3 Fiasco.OC Kernel module

Fiasco.OC kernel module responsible for receiving the threads from the L4 API and safely updating them to the ready queue list of the scheduler. The figure shows the involved classes, data and control flow between the classes.

There were two design choices available to receive the incoming threads from L4 API. The first one was to use an existing kernel object (scheduler) and the second was to create a new kernel object (see 5.6). The second method ensures that the legacy code is clean and provides a dedicated object. However, it uses extra memory and adds the complexity of maintaining one more kernel object. So, the idea of using the scheduler object was used in designing this module.

The scheduler kernel object is used to get the thread data from the L4 API. The obtained thread information is in the form of L4 flex pages. These pages are converted to get the thread ids. The thread ids can be used to obtain the scheduler context objects. A list is created similar to the ready queue list using the scheduler context objects. This module then makes a decision about when the ready queue should be updated according to the current state of the system. After the decision, it calls the methods in the ready queue class to update/exchange the list.

The call is then transferred to Sched_context_Fp_EDF class. The type of the ready

queue is selected depending on the type of the scheduler in use. If it's Fixed priority scheduler, an exchange call is made to FP ready queue class.

The FP exchange call receives the new ready queue list and the priority. A call is made to the list data type to switch the ready queue. A boolean value is returned to the scheduler to indicate the success.

Additionally, this module has to provide a synchronized access to the ready queue. The synchronization method used is explained in 5.5.

4.4.4 Finding the right time for RQ update

Finding the right point for synchronization is a big challenge. Since this toolchain is used on a safety critical system, care should be taken to ensure that the system remains in a safe and predictable state. Theoretically, the points for safe update of the ready queue are:

- Empty Run Queue: If the run queue is empty, exchanging the list is easy since no threads exist in the ready queue. In case there is a single thread in the queue, the thread must be an idle thread. So, in the idle thread and empty queue cases,it is safe to exchange the entire ready queue list with a new list. A CPU lock is used to disable the thread preemption and the ready queue list is exchanged.
- Static point in time: In this method, the `Controller` decides which thread should be allowed to complete its execution before exchanging the ready queue list.
- Variable point in time:

5 Implementation

This chapter gives the detailed implementation details for the toolchain presented in chapter 4. Code blocks are used at necessary places to give extensive details. The code written as part of the thesis can be accessed in the Github repository [Cha16].

The implementation is done in C++ language to be compatible with Genode operating system and Fiasco.OC which are developed in C++. The following sections describes the module implementation and the last section explains about creating a new Fisaco.OC kernel object.

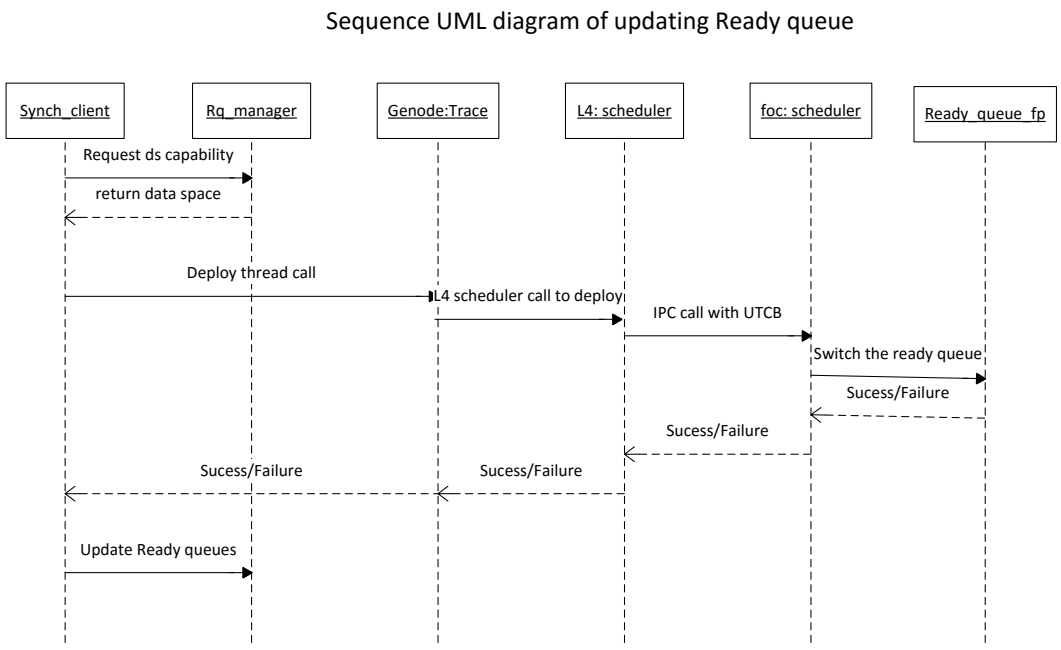


Figure 5.1: Sequence diagram of updating ready queue from the Genode component

5.1 Rq_manager module

The sequence diagram 5.1, shows the order of the calls taking place from the Genode component synch_cient to the kernel. The class Rq_manager is the driver class of this component. It creates and maintains multiple ready queues at the at the user-level component, where each queue is of type Rq_buffer. The Rq_buffer is a template class, which implements a circular array by using Genode dataspace, each entry in the circular array is of type Rq_task struct, which represents a Genode thread.

The code listing 5.1 shows the implementation of Rq_manager class. The constructor of Rq_manager class first gets the number of available cores from Monitoring agent, then the `_init_rq()` function initializes the ready queues by using `init_w_shared_ds` method from the Rq_buffer class. The Rq_manager provides enqueue and dequeue operations for the manipulation of ready queue.

```
1  class Rq_manager
2  {
3
4      private:
5
6        int _num_cores = 0;
7        Rq_buffer<Rq_task> *_rqs; /* array of ring buffers (Rq_buffer
             with fixed size) */
8
9        int _set_ncores(int);
10
11     public:
12
13       int enq(int, Rq_task);
14       int deq(int, Rq_task**);
15       int get_num_rqs();
16       Genode::Dataspace_capability get_core_rq_ds(int);
17
18       Rq_manager()
19       {
20         PINF("Value of available system cores not provided -> set to 2.
               ");
21
22         _set_ncores(2);
23         _init_rqs(100);
24       }
25
26       _init_rqs(int rq_size)
27       {
28
29         _rqs = new Rq_buffer<Rq_task>[_num_cores];
30
```

```
31          for (int i = 0; i < _num_cores; i++) {
32            _rqs[i].init_w_shared_ds(rq_size);
33          }
34          Genode::printf("New Rq_buffer created. Starting address is: %p
                .\n", _rqs);
35
36          return 0;
37
38        }
39  }
```

Listing 5.1: Rq_manager class

The _buffer is a template class which represents the ready queue in the application side. It implements a circular array with fixed size. It uses `head` pointer, which points to the beginning of the array and `tail` pointer, which points to the end of the array. The elements are always inserted at tail pointer and removed from head. The pointers are wrapped around, whenever they reach the end of the array. The free space available between head and tail is represented by using a pointer called `window`.

Rq_buffer has enqueue and dequeue functionalities which can be called from Rq_manager to insert or remove an element from ready queue. It also provides a functionality to initialize and create a shared data space. The code listing 5.2 shows, memory allocation for the shared dataspace and attaching it to the RM session of the component.

```
1
2   Genode::Dataspace_capability _ds; /* dataspace capability of the
        shared object */
3   char *_ds_begin = nullptr;        /* pointer to the beginning of the
        shared dataspace */
4
5     /*
6       * create dataspace capability, i.e. mem is allocated,
7       * and attach the dataspace (the first address of the
8       * allocated mem) to _ds_begin. Then all the variables
9       * are set to the respective pointers in memory.
10      */
11  _ds = Genode::env()->ram_session()->alloc(ds_size);
12  _ds_begin = Genode::env()->rm_session()->attach(_ds);
```

Listing 5.2: Allocating data space

The code listing 5.3 shows the Rq_task structure which represents an entry in the ready queue buffer. It contains the parameters such as task_id to identify a Genode

task.

```
1  struct Rq_task
2   {
3
4       unsigned long task_id;
5       int wcet;
6       bool valid;
7
8   };
```

Listing 5.3: Rq_task structure

The access to this shared data space must be synchronized since both `Controller` and `Synch_client` use it. This is done by locks provided from Genode.

5.2 Synch_client module

The `Synch_client` class represents a Genode component which controls the process of scheduler ready queue update. It acts as a client in Genode by using the services of `Rq_manager` and accesses the shared data space. The `Controller` and `Synch_client` interaction is that of a producer-consumer relationship. The producer (`Controller`) decides and produces(puts it in a buffer) a thread that needs to be executed and consumer(`Synch_client`) picks the threads to the kernel ready queue.

The initial communication with `Rq_manager` is governed by RPC call. In order to make an RPC call, it creates a Connection object. As explained before there are a number of pointers used for manipulating the circular buffer. The different pointers used are:

_rqbufp Hold the pointer to the Rq_buffer type

_lock Lock needed to ensure mutual exclusion

_head Head pointer, points to the start of Rq_buffer

_tail Tail pointer, points to end of Rq_buffer

_window The window of free spaces available in between head and tail

There can be more than one ready queue. So, we need multiple pointers of each type. The data type `vector` is used to represent the pointers as the available number of ready queues is unknown at the creation time. The declaration of all the data structures can be seen in the listing 5.4.

```
1  Rq_manager::Connection rqm;
2
3  /* using vectors since we dont know the size initially */
4  std::vector<int *> _rqbufp;
5  std::vector<int *> _lock;
6  std::vector<int *> head;
7  std::vector<int *> tail;
8  std::vector<int *> window;
9
10 std::vector<Dataspace_capability> dsc;
11 std::vector<Rq_manager::Rq_task*> buf;
```

Listing 5.4: Data structures used in Synch_client

We can use the Connection object created to make RPC calls to Rq_manager. The first RPC call is to obtain the number of ready queues available. For each ready queue available, it has to make an RPC call to get the data space capability and attach to the local RM session and initialize all the pointers shown in the listing 5.4.

After everything has been initialized the `Synch_client` runs in an infinite loop. For each ready queue it tries to obtain the lock, if the locking is successful then it schedules the threads available in the ready queue. The infinite loop has two goals. First, it has to ensure that it moves on to the next ready queue so that the threads in other queues also gets scheduled. Second, is to reduce the amount of time spent in the critical section as little as possible. In order to achieve the above mentioned goals, the ids of the threads/tasks are copied to a local buffer, head and tail pointers updated and the lock is released. By this way the scheduling of threads by calling of Genode API is kept outside the critical section.

One more option to reduce the critical section is to have separate threads working on each of the ready queues. Each thread will be working on a different ready queue and there is no shared data between these thread, which guarantees the second goal mentioned for the infinite loop.

5.3 Ready queue update Mechanism

The ready queue update mechanism follows the series of calls which are shown in sequence diagram 5.1. The following subsections explain the implementation of the modules described in 4.4.

5.3.1 Genode code changes

Genode should provide a way to the application for calling the L4 API calls. In the present Genode code the L4 APIs calls for scheduling threads are done from `Platform_thread.cpp`, where the thread gets created. But the `Platform_thread` is not accessible from the application, as it doesn't provide any RPC interface for applications to make use of its services. The only way that the components communicate in Genode is by using RPC calls. So an RPC call needs to be provided for the components to use it. As of now this call is kept in `Trace` service. The trace service is being used by the monitoring agent already to obtain the data from the Genode. The Trace service has been extended to provide an API, which can be used to from `Synch_client` component.

The `base/src/trace/trace_session_component.cc` contain the API definition, in which it takes a structure argument of threads and calls the L4 scheduler call.

5.3.2 L4 API calls

I introduced a new L4 API which can be called from the Genode. The APi takes the following arguments:

- l4_cap_idx_t: It takes a kernel object which is scheduler object.
- l4_sched_thread_list: This is a structure which can be seen in 5.5, which has an array to hold the list of threads and the priorities for the same threads.

The code listing 5.5 shows the l4_sched_thread_list structure and the L4 API call which is called from Genode and this code is in `foc/l4/l4sys/scheduler.h`.

```
1
2  typedef struct l4_sched_thread_list
3  {
4    l4_cap_idx_t list[10];
5    unsigned prio[10];
6    int n;
7  }l4_sched_thread_list;
8
9  L4_INLINE l4_msgtag_t
10 l4_scheduler_deploy_thread(l4_cap_idx_t scheduler,
11     l4_sched_thread_list thread) L4_NOTHROW
12 {
13   return l4_scheduler_deploy_thread_u(scheduler, thread, l4_utcb());
14 }
```

Listing 5.5: L4 scheduler API in scheduler.h

The deploy thread function call has to fill the UTCB message registers and make an IPC call to the scheduler kernel object. The first register(mr[0]) is populated with the type of operation that the scheduler should do to with the information, in this case it is L4_SCHEDULER_DEPLOY_THREAD_OP. The second register is populated using a call to l4_map_obj_control function, which returns a word. This word identifies the kernel objects, which come after this word in the message registers. After the map object control word, thread objects are filled in the message registers in the form of L4 flex pages. An L4 flex page represents a naturally aligned area of mappable space, such as memory, I/O-ports, and capabilities (kernel objects), in this case L4 flex represents thread object. For each thread id which is sent an L4 flex page is created and stored in the UTCB message registers. The L4 API makes an IPC call to the kernel scheduler object.

The code listing 5.6 shows the L4 scheduler API implementation.

```
1  L4_INLINE l4_msgtag_t
2  l4_scheduler_deploy_thread_u(l4_cap_idx_t scheduler,
       l4_sched_thread_list thread,
3          l4_utcb_t *utcb) L4_NOTHROW
4  {
5    l4_msg_regs_t *m = l4_utcb_mr_u(utcb);
6    m->mr[0] = L4_SCHEDULER_DEPLOY_THREAD_OP;
7    m->mr[1] = l4_map_obj_control(0, 0);
8
9    for(int i = 0; i < thread.n; i++){
10     m->mr[i+1] = l4_obj_fpage(thread.list[i], 0, L4_FPAGE_RWX).raw;
11   }
12
13   return l4_ipc_call(scheduler, utcb, l4_msgtag(L4_PROTO_SCHEDULER,
         thread.n+1, 1, 0), L4_IPC_NEVER);
14   }
```

Listing 5.6: L4 scheduler API implementation

5.4 Fiasco.OC code changes

Fiasco.OC code changes are made in several files to incorporate the ready queue update mechanism in the kernel and following sections explain the code changes in each file.

5.4.1 Scheduler.cpp

The `l4_ipc_call` from the scheduler.h invokes `kinvoke` function. The first parameter of the UTCB is decoded to find out the operation that is involved in. The operation in this

case was set to `deploy_thread` and which calls the function `sys_deploy_thread`, which can be seen in code listing 5.7. It runs a for loop to get the number of available threads. The associated flex page is derived from the sent items and a look up is performed to obtain the thread objects. The thread objects can be used to obtain the scheduler context which they are associated with. The scheduler context objects are used to create ready queue list(Fp_list). This list is used to switch the actual ready queue of the scheduler.

The specified ready queue of the CPU can be obtained if the CPU was specified from the `Controller` otherwise the ready queue of the threads home CPU can be used. The current implementation creates and works with fixed priority list, however, this can be extended easily to create the corresponding ready queue list. If there is only one thread the `ready_enqueue` function can be called instead of exchanging the complete list.

```
1  Scheduler::sys_deploy_thread(L4_fpage::Rights, Syscall_frame *f, Utcb
       const *utcb)
2  {
3    printf("[Scheduler: sys_deploy_thread] 1\n");
4    L4_msg_tag const tag = f->tag();
5    Cpu_number const curr_cpu = current_cpu();
6
7    Obj_space *s = current()->space();
8    assert(s);
9
10   typedef Sched_context::Fp_list List;
11
12   List list;
13
14   for(int i = 6 ; i <= tag.words(); i++){
15
16         /*
17          *  Get the messages in an iterator
18          */
19         L4_snd_item_iter snd_items(utcb, i);
20
21         /*
22          * Check if the items exist
23          */
24         if (EXPECT_FALSE(!tag.items() || !snd_items.next()))
25           return commit_result(-L4_err::EInval);
26
27         L4_fpage _thread(snd_items.get()->d);
28
29         if (EXPECT_FALSE(!_thread.is_objpage()))
30           return commit_result(-L4_err::EInval);
31
32         /*
33          * Do a look up to get the corresponding thread and cast
```

```
34            * it to Thread_object type
35            */
36          Thread *thread = Kobject::dcast<Thread_object*>(s->lookup_local
                (_thread.obj_index()));
37          if (!thread)
38            return commit_result(-L4_err::EInval);
39
40          printf("[Scheduler:sys_deploy] Thread to be scheduled: %lx\n",
                thread->dbg_id());
41
42          list.push(thread->sched_context(), List::Front);
43
44          Sched_context::Ready_queue &rq = Sched_context::rq.cpu(thread->
                home_cpu());
45
46          if(i==tag.words()){
47           rq.switch_ready_queue(&list, 100);
48          }
49      }
50  }
```

Listing 5.7: Thread extraction and ready list creation

5.4.2 sched_context-fp_EDF.cpp

This class implements the major functionalities to handle the ready queues. This class contains the lists of both FP ready queue and EDF ready queue and serves as a wrapper class to both type of lists. Any call made to access the either of the ready queue, goes through this class. The decision to call the specific ready queue function is made according to the type of the scheduler in use.

Since this work was involved with fixed priority lists, the check to find out the type of scheduler in use is not made and default is to use the FP ready queue. However, it can be extended easily to work with both the lists. Code listing 5.8 shows the calling of the function to fixed priority ready queue.

```
1  IMPLEMENT
2  bool
3  Sched_context::Ready_queue_base::switch_rq(Fp_list *list, unsigned prio
       )
4  {
5    return fp_rq.switch_rq(list, prio);
6  }
```

Listing 5.8: Exchanging the ready queue

5.5 Synchronization method

The ready queue contains the scheduler contexts which belongs to the threads and which can run next. Modifying the ready queue list is a critical section and mutual exclusive access must be guaranteed. In a uniprocessor implementation this can work with the CPU lock, which disables the interrupts on the local CPU and no other threads will get CPU time in the middle of ready queue manipulation. For SMP systems the kernel must use a ready list local to the CPU and advantages of using the per CPU data is explained in the section 3. Fiasco.OC implements a per CPU variable which is used towards synchronization. When using per CPU variable the exclusive access to this variable is guaranteed by disabling the CPU preemption and once the access is finished the CPU preemption is enabled. By this way, if any CPU is in the middle of critical section, no thread is allowed to replace the current operation.

The present method uses the per CPU variable method where it locks the CPU before exchanging the CPU list. The software transaction memory method was implemented but testing that method was a problem since using of SMT forces the GCC compiler required pthreads library to be present which was not possible to include to Fiasco.OC. The SMT method would eliminate the necessary to preempt the CPU since it executes all the exchange the instructions in one transaction.

A check is made in scheduler.cpp to find out if the ready queue is empty. If the ready queue is empty or it contains the idle thread then the immediate call is made to exchange the list. The static point in time method is to check if a thread from this ready list getting executed, if the thread is running the exchange call waits till the scheduler takes the control back and thread list is exchanged. This is checked in the call from `schedule_in_progress` flag.

5.5.1 Ready_queue_fp.cpp

Hereinafter the list that is sent is referred as new list and the list to be exchanged is referred as old list. Once the fixed priority ready queue is decided to be exchanged the new list and the priority is sent to the `switch_rq` function from sched_context-fp_EDF class. It is assumed at this point that the new list contains the threads that are necessary to execute the threads, which include the idle thread, pager threads. The Fiasco.OC uses an idle thread which keeps the CPU busy when there are no threads are to be executed. The idle thread needs to be kept in the new list if it's not existing already. The old list can be used to identify the idle thread, which sits in the end of the old list. The pager thread is nece

The exchange of the list is a simple operation of changing the head pointers. The cyclic list is implemented in a file called `dlist`. I implemented a a function `exchange`

which takes the list to be exchanged with the present list and sets the head of the old list to the new list.

The listing 5.9 shows the `switch_rq` function.

```
1  bool switch_rq(List *list, unsigned prio) {
2       assert_kdb(cpu_lock.test());
3
4       prio_next[prio].exchange(list);
5
6       //prio_next[prio].rotate_to(*++List::iter(list->front()));
7
8       typename List::BaseIterator it = List::iter(prio_next[prio].front
            ());
9       dbgprintf("After exchange fp_rq: ");
10      do
11      {
12        dbgprintf("%lx => ",Kobject_dbg::obj_to_id(it->context()));
13      }while (++it != List::iter(prio_next[prio].front()));
14      dbgprintf("end\n");
15
16      return true;
17    }
```

Listing 5.9: Exchanging the ready queue

5.6 Creating a new kernel object in Fiasco.OC

In this section creating a new kernel object is explained, that can be utilized to make a dedicated work in the kernel. The following changes are required to create kernel object and compile it. A new class should be created, which inherits `Icu_h<object name>` and `Irq_chip_soft` classes. The class has to declare this as kernel object using a macro (FIASCO_DECLARE_KOBJ()) provided from Fiasco.OC and the memory is allocated outside the class by defining the kernel object and sending the class name as a parameter. The class has to have a static object created inside the class and an interrupt request(Irq_base) object defined in it. In the constructor of the class it is important to register this object to initial kernel objects.

The class definition and the registering of kernel object can be seen in listing 5.10. The new object is called `RQ_manager`, which should take care of the ready queue handling.

```
1 class RQ_manager : public Icu_h<RQ_manager>, public Irq_chip_soft
2 {
3   FIASCO_DECLARE_KOBJ();
4   typedef Icu_h<RQ_manager> Icu;
```

```
5
6  public:
7    enum Operation
8    {
9    RQ_info = 0,
10   Schedule_thread = 1,
11   };
12
13   static RQ_manager rq_manager;
14 private:
15   Irq_base *_irq;
16 };
17
18 FIASCO_DEFINE_KOBJ(RQ_manager);
19
20 PUBLIC inline
21 RQ_manager::RQ_manager() : _irq(0)
22 {
23   initial_kobjects.register_obj(this, 8);
24 }
25
26 PUBLIC
27 L4_msg_tag
28 RQ_manager::kinvoke(L4_obj_ref ref, L4_fpage::Rights rights,
       Syscall_frame *f,
29                     Utcb const *iutcb, Utcb *outcb)
30 {
31 }
32
33 enum Protocol{
34   Label_rq_manager = -22L,    // Protocol ID for rq manager objects<
       l4_types.cpp>
35 }
36 enum l4_msgtag_protocol{
37 L4_PROTO_RQ_MANAGER = -22L, //Protocol for messages to a rq_manager
       object<types.h>
38 }
39
40 static Cap_index const C_rq_manager = Cap_index(8); //kernel-thread-std
       .cpp
```

Listing 5.10: Creating new kernel object

This class should implement few of the functions from the inherited classes such as `icu_bind_irq` and `icu_set_mode` and a `kinvoke` call which the Fiasco uses it to do interprocess communication.

This created kernel object needs to be associate with protocol ID. These protocol IDs are used for either kernel implemented objects and this is assigned in the `l4_types.h`, protocol section of the l4_msg_tag and also it needs to be assigned with l4_msgtag_protocol. This kernel object is assigned a `Cap_index` which it was registered against.

6 Testing and Results

6.1 Test utility using Trace

6.2 Building the system

6.3 Results

–> Explain testing framework –> Draw a sequence diagram –> Extension of trace service

7 Future Work and Conclusion

7.1 Limitations

7.2 Future work

7.3 Summary

List of Figures

List of Tables

Bibliography

[Cha16] G. Chandrasekhara. *genode-Synchronization*. [Source code of thesis]. 2016. URL: https://github.com/702nADOS/genode-Synchronization.

[EKB15] S. Eckl, D. Krefft, and U. Baumgarten. "COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions." In: *Conference on Future Automotive Technology*. 2015.

[Fes15] N. Feske. *GENODE Operating System Framework 15.05*. CC-BY-SA, 2015.

[Hae15] R. Haecker. "Design of an OC-based Method for efficient Synchronization of L4 Fiasco.OC Mircrokernel Tasks." Bacherlor Thesis. Technische Universität München, 2015.

[Hau14] V. Hauner. "Extension of the Fiasco.OC microkernel with context-sensitive scheduling abilities for safety-critical applications in embedded systems." Bacherlor Thesis. Technische Universität München, 2014.

[HH01] M. Hohmuth and H. Härtig. "Pragmatic Nonblocking Synchronization for Real-Time Systems." In: *USENIX Annual Technical Conference, General Track*. 2001, pp. 217–230.

[MW07] P. E. McKenney and J. Walpole. *What is RCU, Fundamentally?* [Accessed: 22-October-2016]. 2007. URL: https://lwn.net/Articles/262464/.

[Nie16] P. Nieleck. "Design and Prototypical Implementation of an OC-based Controller-Stack for Optimizing Mixed-Critical Thread Scheduling in L4 Fiasco.OC/Genode." Master's Thesis. Technische Universität München, 2016.

[OS a] T. OS group. *Fiasco features*. [[Accessed: 22-October-2016]]. URL: https://os.inf.tu-dresden.de/fiasco/features.html.

[OS b] T. OS group. *Fiasco.OC*. [Accessed: 22-October-2016]. URL: http://os.inf.tu-dresden.de/Studium/MkK/SS2012/08_fiasco.oc.pdf.

[PA14] V. Pankratius and A.-R. Adl-Tabatabai. "Software engineering with transactional memory versus locks in practice." In: *Theory of Computing Systems* 55.3 (2014), pp. 555–590.

[SM04] D. Sarma and P. E. McKenney. "Making RCU safe for deep sub-millisecond response realtime applications." In: *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*. 2004, pp. 182–191.

[Ste04] U. Steinberg. "Quality-Assuring Scheduling in the Fiasco Microkernel." Diploma Thesis. Technische Universität Dresden, 2004.

[Zyu+09] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. "Atomic quake: using transactional memory in an interactive multiplayer game server." In: *ACM Sigplan Notices*. Vol. 44. 4. ACM. 2009, pp. 25–34.