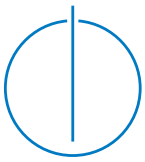
DEPARTMENT OF INFORMATICS

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

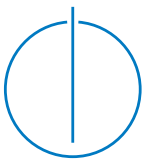
TECHNISCHE UNIVERSITÄT MÜNCHEN

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, Gurusiddesha Chandrasekhara

Acknowledgments

Writing this master's thesis would not have been possible without the help of these people. First, I would like to thank my advisors Daniel Krefft and Sebastian Eckl, who immediately accepted my request to do the thesis at the Chair and guided me to select the research topic. Special thanks to Daniel Krefft for his continuous support and brilliant advice throughout the thesis phases.

I would like to thank my supervisor, Prof.Baumgarten for allowing me to do my research and for providing all the facilities at the Chair.

Thanks to Paul Nieleck, who worked on his master thesis at the department, for his excellent collaboration and sharing his knowledge on the project. I would also like to thank Alexander Reisner for sharing his invaluable experience on the project.

Thank you to my family and friends for providing me the required freedom and support.

Abstract

This thesis introduces a method for updating the task ready queues in the Genode Operating System Framework and the L4/Fiasco.OC microkernel. The presented method will be used in Organic Computing paradigm's Observer-Controller architecture. The Observer monitors the system and gathers the data and passes it to the Controller, which processes the data and takes a decision to schedule the tasks. This requires a method in place to provide the synchronized access to ready queue and update the tasks.

The implementation of the ready queue update mechanism consists of a high-level Genode component and a low-level kernel module. The Genode component communicates with the Controller via a shared dataspace and receives the tasks to be updated and then passes them to the kernel module. The kernel module identifies the corresponding kernel threads and updates them to the ready queue. Various synchronization methods are presented in this thesis with special importance to lock-free algorithms. RCU and STM synchronization methods are suggested for synchronizing the kernel ready queue access.

Tests of the task-ready queue update mechanism showed that the threads can be updated successfully to the ready queue and executed. On the other hand, complete ready queue swapping leaves the system in an unstable state. The high-level component is able to communicate successfully with the Controller via Genode's shared dataspace. The proposed design and implementation can be successfully used in the Observer-Controller architecture and it serves as a good starting point for the KIA4SM project's goal of having the ready queue update mechanism as a fully high-level component.

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

MMUF Modified Maximum Urgency First

1 Introduction

The implemented synchronization component in this Master's thesis is part of the KIA4SM project of the Chair of Operating Systems. The work aims to implement a method for dynamic update of task ready queues in L4 Fiasco.OC/Genode while providing a synchronized access to them.

1.1 Overview of KIA4SM project

KIA4SM (Cooperative Integration Architecture for Future Smart Mobility Solutions) is a research project at the Chair 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 a homogeneous software platform for heterogeneous hardware systems. The project focuses on developing systems for the interaction and coordination between partially or fully autonomously functioning computer-assisted vehicles. It also aims to improve the ad-hoc networking between vehicles.

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

- Providing a device independent platform. The different devices include vehicles, mobile devices of users and the supported devices for traffic and transport management.
- Providing a mechanism for online dynamic reconfiguration, based on migration of software functionality, having a adaptive routing policy and flexible scheduling of tasks on the ECUs.

In order to achieve the goals of the project, a number of different methods have been applied such as application of the Organic Computing (OC) paradigm. The OC addresses the challenges of complex distributed systems by making them more life-like (organic) by endowing them with abilities such as self-organization, self-configuration, self-repair or adaptation [Bra+06]. In order to realize this, universally applicable Electronic Control Units (ECU) and a common run-time environment are used, which provide Hardware/Software Plug-and-Play properties.

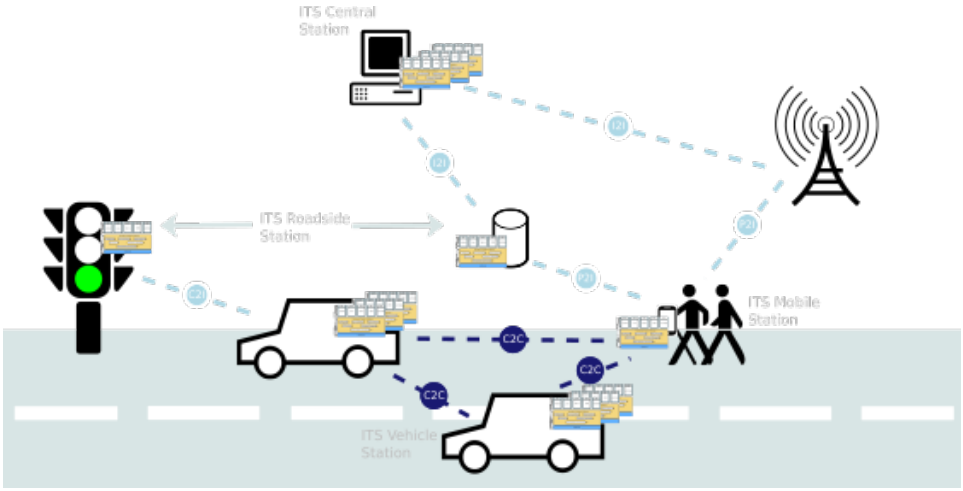


Figure 1.1: KIA4SM vision - homogeneous platform for heterogeneous be it factors andT devices [EKB15]

1.2 Motivation

There are a number of micro controllers that are used for different calculations in a modern vehicle. The KIA4SM project aims to replace them with more powerful and standardized hardware, such as universally applicable ECUs.

OC approach proposes a Observer-Controller architecture shown in figure 1.2, which is similar to the MAPE architecture (monitor, analyze, plan, execute). An observer collects the data from all the ECUs, computes and generates indicators. The controller then takes a decision based on the indicators and generates an action.

One such action of the controller is to decide the tasks that should be executed at a certain time in order to maintain the system in safe state. It is essential to be able to add threads and modify the execution order during operation time. Also, if an ECU is malfunctioning, thread migration is required. This involves swapping the threads from the malfunctioning one to the working ones. So, it is important to generate new ready-queues based on the information we receive from the other ECUs in the grid, and then exchange them with the actual ready-queue that the scheduler uses. 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.

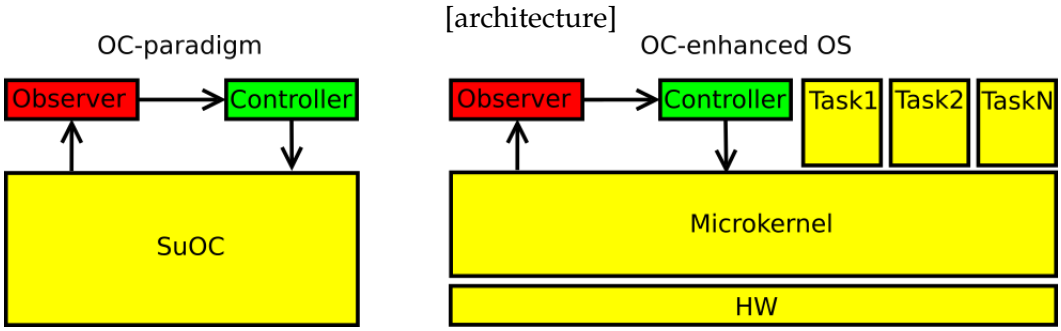


Figure 1.2: Organic Computing: Applying the Observer/Controller pattern to existing microkernel architecture [EKB15]

1.3 Thesis structure

The thesis is structured in a inverted triangle model, the surrounding concepts are explained, before delving in to the specifics.

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 details of synchronization module.

The fifth chapter explains the implementation of the design presented in the fourth chapter along with the code examples.

The sixth chapter is dedicated to explain the 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 the synchronization component in the KIA4SM project. It explains the synchronization algorithms available and provides the results of a comparative study of these algorithms.

2.1 Synchronization of L4 Fiasco.OC 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], describes the design of a scheduler best suited for the KIA4SM project. He also gives a comparison study of the different schedulers and the synchronization methods suited for updating the task ready queue.

He suggests the Modified-Maximum-Urgency-First(MMUF) algorithm as the best choice for the KIA4SM project due to the importance of safety and security in embedded systems. After comparing the synchronization algorithms, the sequential lock technique was chosen to be the best since it offers better control. Another option that he suggested is the Read-Copy Update(RCU) algorithm. However, both these methods are untested.

This work is an extension of Häcker's findings. However, the focus of the thesis is to develop a good synchronization method without the implementation of a scheduler.

Some of the ideas and code knowledge were taken from 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 his thesis, he added the EDF scheduling strategy. Though his thesis concentrated on using it in the L4RE environment, it provided a good starting point for this thesis.

2.2 Synchronization Methods

This section describes the synchronization methods that were studied in this thesis. The synchronization methods are categorized based on the techniques that they use to provide concurrent access. The figure 2.1 shows the different types of synchronization methods along with examples.

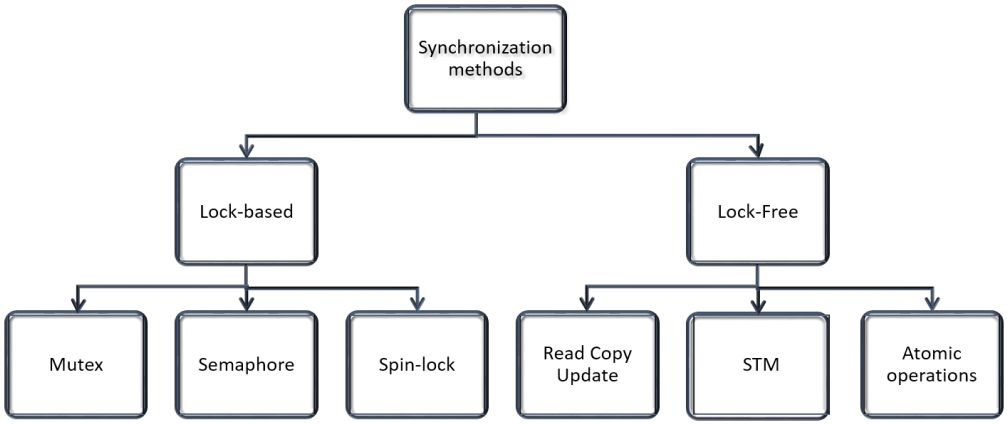


Figure 2.1: Lock-free and lock-based synchronization methods

2.2.1 Lock Based Algorithms

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

Some of the lock-based synchronization methods are,

Mutex Mutex is a synchronization primitive that stands for mutual exclusion, which prevents the simultaneous access to the shared resource. The thread, which wants to execute in a critical section has to acquire the mutex. Once finishes the execution in the critical section, it has to release the mutex.

Semaphore It is a variable that is used to control the common resource access between multiple processes/threads. A typical semaphore is initialized with an initial value equal to the number of resources 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 event management, where the initial value is 0 and the 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 a mutex.

Spin-lock It is a type of lock, where the thread that 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 a very good advantage if the threads are blocked for shorter periods of time.

The lock-based synchronization primitives are simple and easy to implement. However, there exist a lot of disadvantages. If the programmer is not careful, deadlocks can occur, which are difficult to debug. One interesting problem that might arrive is priority inversion, which is defined for two threads i and j as, if J is in a critical section, assign J the highest priority and when it exits the critical section, assign its original priority. The problem with priority inversion is that a higher priority thread has to wait for the lower priority thread. This is not desirable for real time systems. Another problem is thread starvation, where a thread doesn't get CPU time due to long waits.

2.2.2 Lock-Free Algorithms

Lock-Free algorithms refer to a synchronization method where the access to critical section for all the threads is guaranteed without using the 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 is a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort and 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 ensure 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 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,

Publish-Subscribe Mechanism (for insertion): Publish-subscribe mechanism is a way of enforcing the compiler to execute the instructions in the correct order. If a pointer is getting updated, an RCU call(rcu_assign_pointer()) is used to change the pointer. This can be thought of as publishing the data. This requires that the readers wait for the update to happen to read the correct data and 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 2.2 shows RCU's way of waiting for pre-existing reader threads to finish. RCU

introduces a grace period where it waits on a read-side critical section. The simple way to find out when the reader threads have finished executing the critical section is to check for the context switch of the CPU.

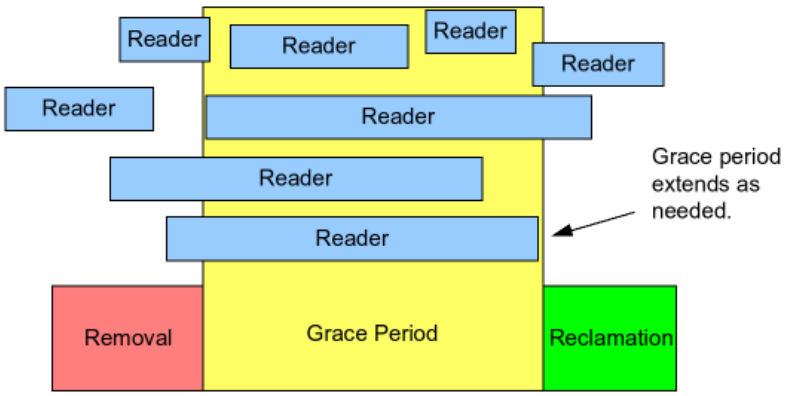


Figure 2.2: RCU showing the usage of grace period [MW07]

Maintain Multiple Versions of Recently Updated Objects (for readers) : RCU maintains multiple versions of the data in between the removal and the reclamation phases. The figure 2.3 shows how RCU maintains these multiple versions of data. In the linked list, node 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 are 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).

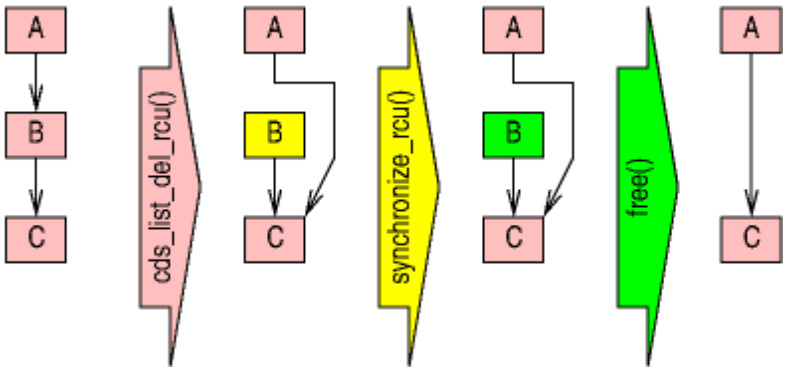


Figure 2.3: Ready copy update mechanism, showing deferred destruction [MW07]

Since the introduction of RCU, there have been a lot of improvements. The research

from McKenney *et al.* shows that RCU can provide order-of-magnitude speedups for read-mostly data structures [MW07]. RCU is optimal when less than 10% of accesses are updates over a wide range of CPUs. The work by Sarma *et al.* shows that making RCU suitable for real-time systems relies on improving RCU callbacks and suggests three methods for doing so. The first is to provide per-CPU kernel daemons to process RCU callbacks. Second is to directly invoke the RCU callback and lastly, to throttle the RCU callbacks so that the limited number of callbacks are invoked at a given time [SM04].

Software Transactional Memory: Software Transactional Memory (STM) is a concurrency control mechanism that 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 that the other threads will not see any changes that this thread is making. At the end, either the transaction is committed (if no other threads have modified the data), or aborted (if other threads have modified the data) and the transaction is restarted.

STM requires language extensions and the compiler takes care of data versioning 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. Code listing 2.1 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,

Wait-free algorithms: A wait-free implementation of a concurrent data object guarantees that a thread executes in a finite number of steps, regardless of execution times of other threads [Her91]. For example, when a higher priority thread *A* detects that a lower priority thread *B* is in critical section that it wants to enter, it lends *B* its priority to let *B* finish the execution. When *B* has finished, *A* executes in its own critical section.

Lock-free synchronization: This method works completely without locks and uses an atomic update operation like Compare-and-Swap(CAS). *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 [HH01]. The CAS works in two steps. The *compare* step detects the conflict between two threads that are updating the memory location. In case of failure, the whole operation is restarted. This gives deadlock a free code, but requires that primitives for atomic memory update operations are available.

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 based on 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 this mechanism. STM is better than locks because with using many locks, the code can become unreadable, whereas the 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 and list operations.

RCU is better to use when there are multiple readers, so that the read speed is very good. RCU also has a better write-speed when compared to locks. Locks perform the worst in read and write speed since only one thread can access the data at a time. STM's read and write speed is also good, but the only drawback is that it might have to read again if the data changes in the middle of a transaction.

There is no deadlock problem in RCU, while deadlocks are common in locks. STM also suffers from data races. RCU and locks have less overhead compared to STM. This is because 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 logs to keep the old data.

The recent research on STM is increasing. Ferad *et al.*'s research suggests that implementing STM from scratch is better than trying to convert the programs, which use locks to an STM implementation [Zyu+09]. The research by Victor *et al.* [PA14]

showed that STM is a very good tool but needs C++ language refinements and better debugging support, and large transactions can hurt the performance.

RCU seems to be a better choice for providing synchronized access to the ready queue but it adds considerable code, given the fact that it is an embedded system. Testing the lock-based and a lock free algorithm on a real system will provide a better heuristic.

3 Foundations

This project uses the Genode operating system framework with the L4/Fiasco kernel. 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 related to both Genode operating system and L4 Fiasco.OC kernel. Many concepts explained here are based on ideas from the Genode book [Fes15], so they are not explained in detail.

3.1 Genode Operating System Framework

The Genode operating system framework is a tool kit for building secure and special purpose operating systems. Genode is maintained by the 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.

The Genode operating system uses a recursive tree structure, where each node in the tree represents a component. Each node is owned by its parent, which controls every aspect of its child like resource control and execution environment. The root of the tree is a minimalistic microkernel, 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 the device drivers, network stacks, file systems, virtual machines are the nodes of the operating system. Each component gets a share of the physical resources available. The components can grant the resources to their children. If a components wants additional resources, it can request its parent, and the parent can grant or deny it.

The above features make Genode operating system have a smaller trusted computing base(TCB) and give better security features. If a parent component 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 different operating systems. For example, the OS has to provide an assurance that threads get fair share of execution time while accommodating rich and dynamic

workloads. A similar mechanism is employed for providing security features and user friendliness.

Genode supports both x86 and ARM CPU architectures, and can be used with most members of the L4 family(NOVA, Fiasco.OC, OKL4 v2.1, L4ka::Pistachio, Codezero and L4/Fiasco) of micro kernels. On the Fiasco.OC, Genode supports paravirtualization, which is 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 mixed-criticality systems such as the one presented in the KIA4SM project. Hence, it makes a very good choice of operating system to use for the KIA4SM project.

3.1.1 Source Tree Structure

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

Inside the `repos/` folder, `base/` repository contains the basic framework related code. The `base-<platform>/` folders contain the platform specific code, where <platform> refers to the microkernel name. For example, *foc* contains the code for the Fiasco.OC.

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 that gives access to this RPC object is called `Capability`.

When an RPC object is created, Genode creates a so-called `object-identity`, which represents the RPC object in the kernel space. The kernel maintains a capability space, which holds reference for the object identities. This capability space is explained in detail in 3.2. The capabilities can be passed to different components, and 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 that 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 when the client calls the method.

Clients generally have to trust the server since they are granted from the parent through session invocation. Servers that do not trust their clients can cancel the service anytime if they detect a security threat.

3.1.4 Component Creation

Genode component is made up of five basic parts,

RAM session Allocates memory for a program's BSS and heap

ROM session Contains an executable binary

CPU session Creates an 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, which 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 also has the ability to delegate the capabilities and has the system-wide authority. RPC mechanism is similar to a way a function calls work, where control transfer between caller and callee happens. Synchronous RPC mechanism uses kernel's IPC to transfer messages between a client and the server. The asynchronous notification is helpful when the caller doesn't want to wait for the control to come back. The Synchronous RPC mechanism was considered for use in this work, but this is not useful in cases of bulk transfer of data between Controller and Synchronization component. So, the idea of Shared memory concept came as a replacement.

The steps taken in allocating and using shared dataspace are,

1. The first step is to allocate the memory. The server does this by interacting with the core's RAM service to allocate a new RAM dataspace. This space is owned by the server.

2. The server attaches the dataspace to its own RM session. This makes the dataspace contents available in its virtual address space.

3. Server delegates the authority to a client whenever a request for the dataspace is made.

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 were used in combination.

3.1.6 Interaction with the Kernel

The interaction of user-level threads with the kernel can be seen as a state machine with state transitions. The user-level threads in Genode enter the kernel via a system call, either by device interrupts or by a CPU exception. Once entered, the kernel takes the corresponding action for the event that caused the kernel entry, and the thread leaves the kernel to the user-space.

One such kernel interaction is scheduling a thread. 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 the L4 family of microkernels. Fiasco provides multi-processor support, 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 the ideal choice for using in this project. Fiasco.OC can used with the L4 Runtime Environment (L4Re), which provides the 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 the objects interact with each other with a kernel

provided IPC mechanism. Each object provides services that other objects can use. The capabilities in the 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. The kernel also provides a factory for object management.

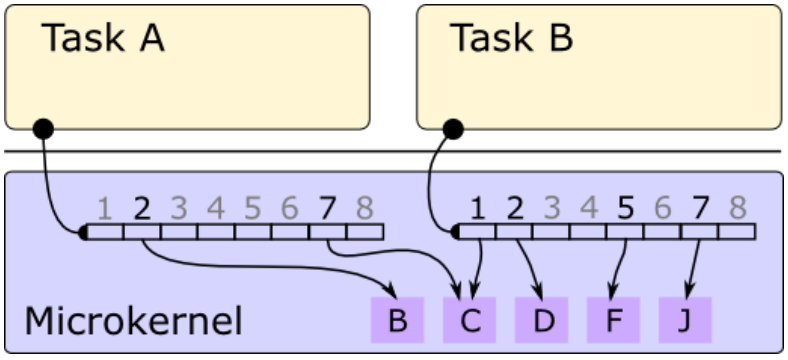


Figure 3.1: Per process capability table in Fiasco.OC [OS b]

The table 3.1 shows the kernel provided objects.

Kernel Object	Description
Task	Represents a protection domain in which the threads can execute
Thread	Unit of execution in a task
Factory	Used to create 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 [OS b]

The kernel has an object space, which contains the capabilities to objects. This can be considered as a pool of objects and a lookup can be done with the given id to find out the object. Fiasco.OC uses the 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, namely the `ready`, `running` and `blocked` states.

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

- MR message registers - contain 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 the kernel acts as a driver class, which controls most of the functionality. The execution of thread has a scheduling context and an execution context. which are explained 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 the Fiasco microkernel. A new class is introduced, which is called Sched_Context and contains all the scheduling and accounting parameters. 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 a SLAB allocator. This separation of the execution and scheduling context allows the system to do fast IPC (the sender of an IPC donates its time quantum to the 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 another thread, the kernel has to only switch the execution context of the destination thread while 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 of a 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. A 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, represents the type of the scheduler (Fixed priority or EDF)
Deadline	In case of EDF scheduler, this represents 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 that are ready to be executed next. The Fiasco.OC microkernel supports 256 priorities ranging from 0 to 255. In case of a fixed priority scheduler, there is a list that exists for each of the priority levels. The scheduler works in a round-robin fashion, where it picks the head of the highest priority thread to execute, runs until a certain time quantum and chooses 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. A note to be taken for the Fiasco.OC kernel is that the thread that is on the CPU will not be in the ready queue. Once it finishes its allocated CPU time for the run, it is re-queued back to the ready queue list(provided its 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 is a critical section operation. The uniprocessor implementation uses a simple CPU lock to disable the interrupts and SMP processors use CPU specific ready queue operations. In order to use per-CPU ready queue list, the kernel

has to ensure that preemption is disabled. Enqueue operation takes a scheduler context object to be enqueued and checks if it is 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 in a Genode application is done by creating a class and inheriting the Genode `Thread` class. This class takes the stack size as a template parameter. Figure 3.2 shows the sequence of calls that take place between the application and the kernel. 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 inheritance of 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 following sequence of calls takes 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 and selecting the affinity space and priorities for the thread. Further, there are two types of platform_thread constructors available. One for the core threads, which takes just the name of

the argument and the 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 a thread's local ID. This call creates an L4 thread in the kernel.

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

4. `l4_scheduler_run_thread` makes an IPC call to the 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.

3.4 Creating a Kernel Object in Fiasco.OC

In this section, creating a new kernel object is explained. This object can be utilized to make dedicated work in the kernel. The following changes are required to create a 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 a kernel object using a macro (FIASCO_DECLARE_KOBJ()) provided by the Fiasco.OC. 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 it and an interrupt request(Irq_base) object defined. In the constructor of the class, it is important to register this object to the initial kernel objects.

The class definition and the registration of the kernel object can be seen in Listing 3.2. 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 3.2: Creating new kernel object

This class implements methods from the inherited classes, such as `icu_bind_irq` and `icu_set_mode` and a `kinvoke` call. The `kinvoke` call is then used by the Fiasco.OC objects for interprocess communication.

This created kernel object needs to be associated with a protocol ID. These protocol IDs are used for kernel implemented objects and are assigned in the protocol section of the l4_msg_tag in `l4_types.h`.

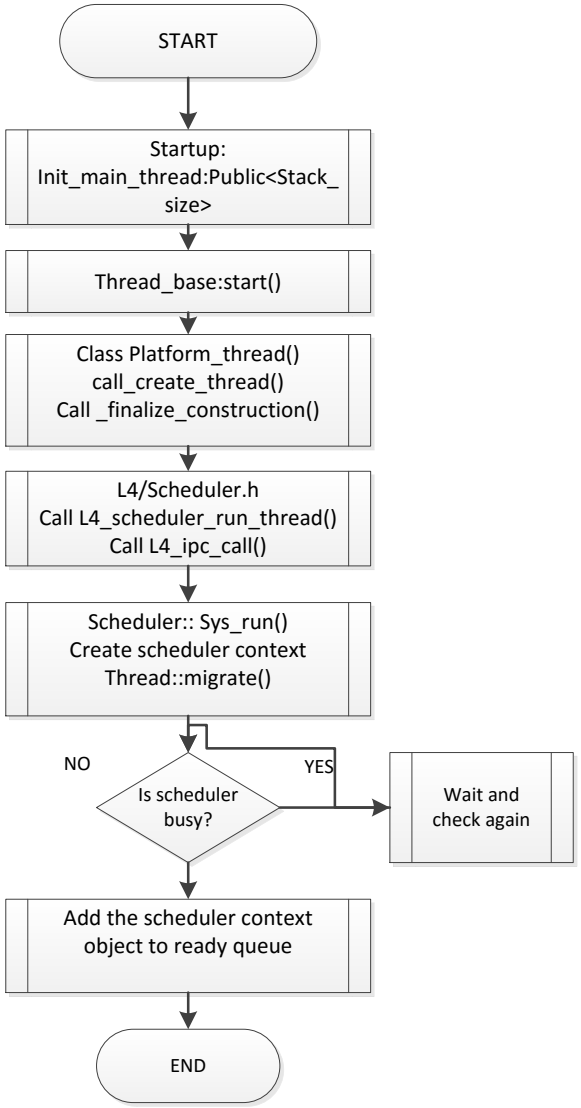


Figure 3.2: Thread creation calls in Genode operating system

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

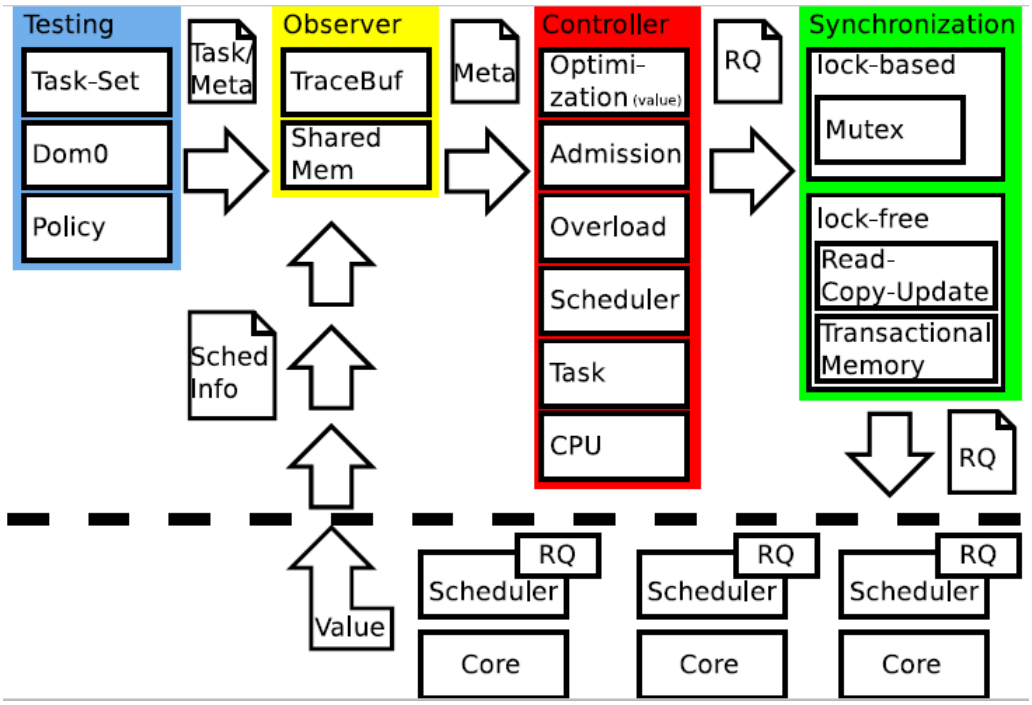


Figure 4.1: The Observer- Controller architecture

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 it with the ready queue given by the Controller. This requires synchronized access to the kernel ready queue. However, further research showed that this design required modification due to the following reasons:

- Kernel ready queue cannot be directly accessed from a high-level component such as the one proposed, since the kernel protection domain prevents direct access.
- 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 differ 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). Hence, a user-level component such as the 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 the kernel and needs a re-implementation in the user-level component.

The high-level view of the new design is depicted in figure 4.2. The Synchronization module is 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 a 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 (see 4.2).

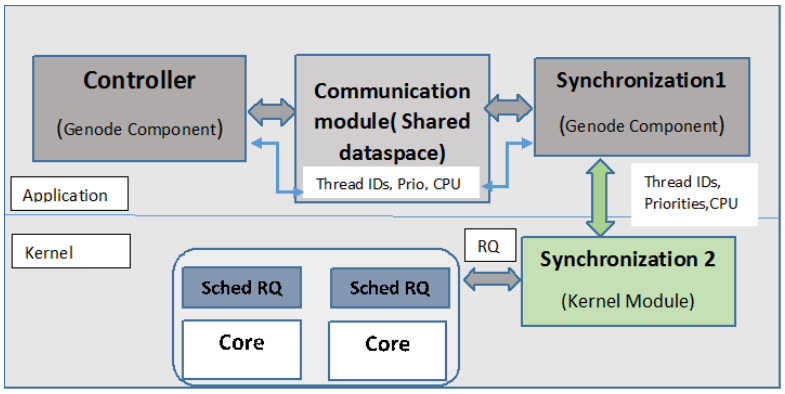


Figure 4.2: The synchronization modules design with communication module and Controller

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.

Figure 4.3 shows the data and control flow, and the interfaces between the modules of the new design.

The `Rq_manager` is a Genode component and is responsible for providing the communication interface between `Controller` and `Synchronization`. 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 to create a 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 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. The clients which access the server's services can directly communicate with each other once they obtain the shared dataspace.

The `Controller` and the `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 consists of the Genode OS framework thread ids, processor id on to which the thread is to be scheduled, and the priorities of the threads. However, the `Controller` doesn't enforce any policies to the `Synch_client`.

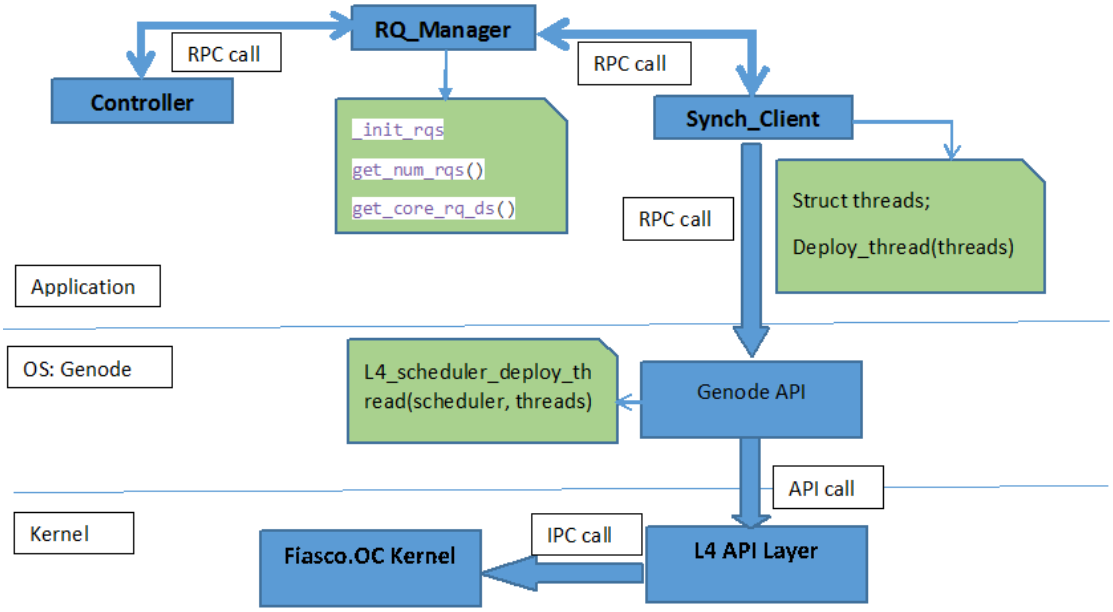


Figure 4.3: The data flow between Synch component, Rq manager and Controller

So, both these components work independently from each other but are 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 and 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, 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 the L4 layer. The L4 API communicates the information to Fiasco.OC kernel object by using the IPC service provided by the kernel. The kernel threads are identified by the kernel and updated 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.

The `Synch_client` also 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. The `Synch_client` then updates this information to the shared dataspace so that the `Controller` can take a scheduling decision.

4.4 Ready Queue Update Mechanism

R eady 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 4.4 shows the high level overview of the involved classes in this process.

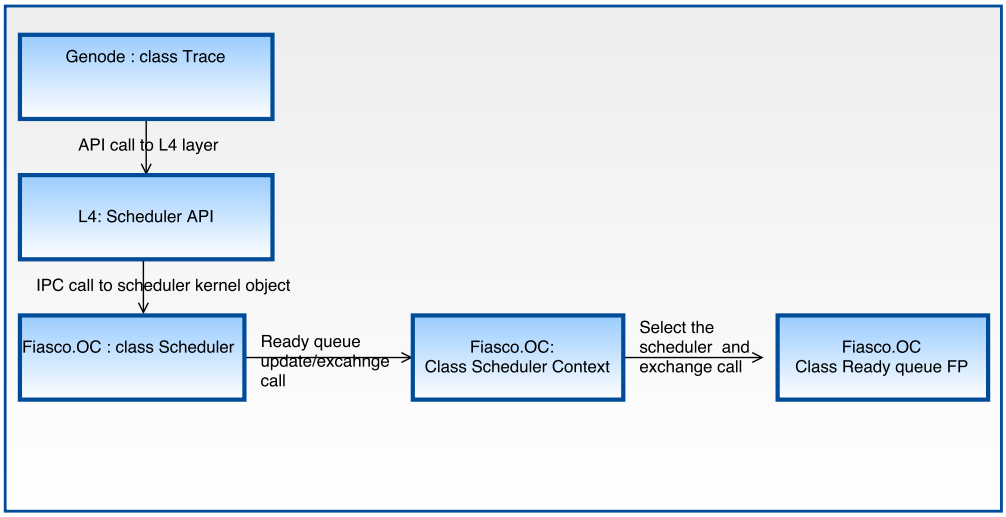


Figure 4.4: Involved classes in ready queue update meachsnism

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 involved in ready queue update mechanism.

4.4.1 Genode Module

Genode module is a component which provides an API that can be called from the `Synch_client` to communicate with L4/Fiasco.OC kernel. 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, the Genode module returns 1 (ready queue update successful) or 0 (ready queue update failure) to the `Synch_client`.

4.4.2 L4 Module

The L4 module represents the L4 API layer, which governs the communication between Genode and Fiasco.OC. It provides an API written in L4sys code, which is called from the Genode module. It 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 and L4 system calls (L4sys). Once the thread ids are obtained from the Genode API, it extracts and fills them in the 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 to the Genode module.

4.4.3 Fiasco.OC Kernel Module

The Fiasco.OC kernel module is responsible for receiving the threads from the L4 API and safely updating them to the ready queue list of the scheduler. 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 3.4). 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 figure 4.5 shows the involved classes, data and control flow between these classes in the kernel module.

The Scheduler class is in *foc/kernel/fiasco/kern/src* and has a static variable `scheduler`, which represents the kernel object. The Scheduler context and Ready queue classes implementation is quite complicated. The class `Sched_context` is the central class. The `Ready_queue` and the `Ready_queue_base` classes are nested inside the `Sched_context` class. Also the `Ready_queue_base` class is inherited by `Ready_queue` and has both FP and EDF ready queue objects.

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, which 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 makes `switch_ready_queue` function call of the `Sched_context` class to switch or update the ready queue.

The call is then transferred to the `Ready_queue` class. The type of the ready queue is

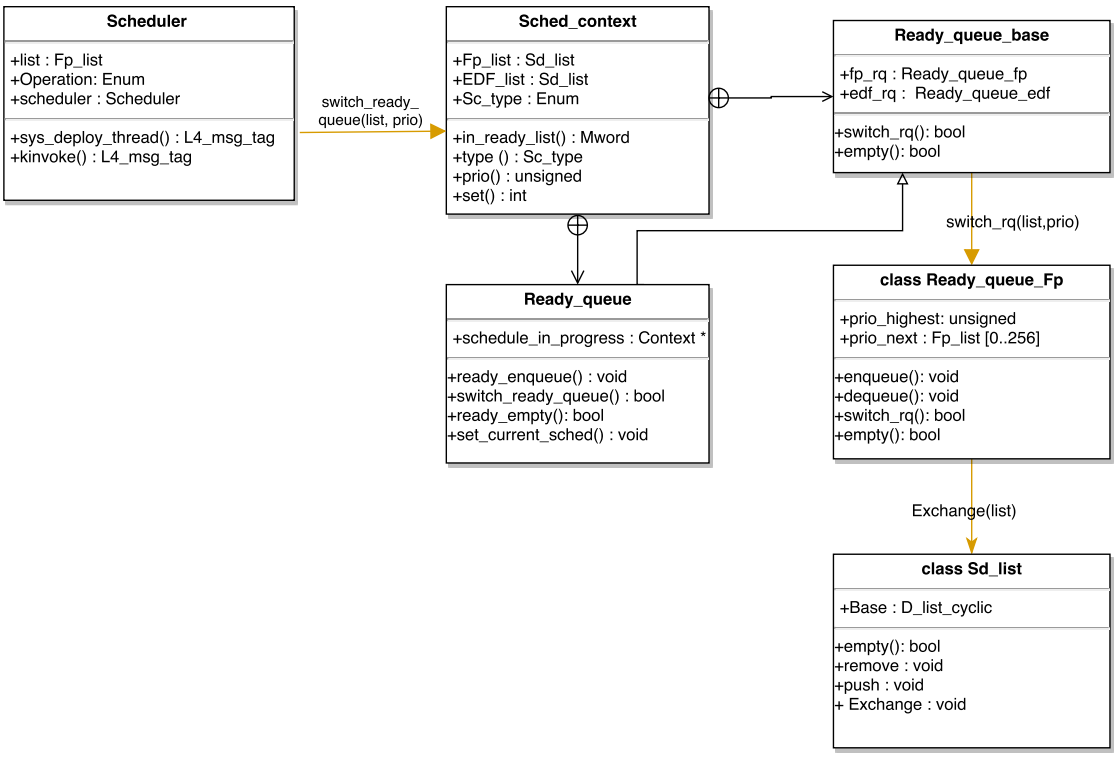


Figure 4.5: Class diagram of the involved classes (yello arrows show the function calls)

selected depending on the type of the scheduler in use. If it is a Fixed priority scheduler, the `switch_rq` call is made to the `Ready_queue_fp` class. The `switch_rq` call receives the new ready queue list and the priority. A call is made to the `sd_list` class 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 Ready Queue 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 ready-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. The controller can either wait or preempt the currently executing thread.
- Variable point-in-time: In this method, the synchronization mechanism is used is such way that it takes a decision to pick the best to update the ready queue. If RCU is used, then the update happens after the grace period. If STM is used the transaction is committed if there is no thread is accessing the ready queue (see 2.2).

5 Implementation

This chapter gives the implementation details for the design 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].

This implementation used the Genode OS framework 15.11 and was written in C++ to be compatible with the Genode operating system and the Fiasco.OC, which are both developed in C++. The following sections describe the module implementation.

The first aim was to find access to the ready queue of the scheduler from the Genode. The task was to find an L4 API, which has access to the ready queue and unfortunately there existed none. According to the l4-hackers, on Fiasco's API level, there is no interface to directly access the run queues of processors. The current APIs allow to set only the scheduling and CPU affinity parameters [Lac]. This forced the creation of a new API that gives access to read/write the kernel ready queue to the Genode or to a high-level application.

The new API should be able to access the ready queue directly. However, this was also not possible due to the way the ready queue is implemented. The ready queue list is a private list inside the ready queue class. There exists a ready queue for each of the three types of schedulers (FP, WFQ and EDF). Moreover, each processor has its own ready queue, which is maintained in a per-CPU variable to synchronize between multiple processors. In order to access the ready queue, methods should be provided from the ready queue class to the Genode applications. So, instead of taking the ready queue list all the way upto the Genode/high-level application, the thread to be updated is propagated via a series of calls to the ready queue class.

The sequence diagram 5.1 shows the order of the calls taking place from the Genode component synch_cient to the kernel. As shown, the `Rq_manager` returns the dataspace capability for the `Synch_client`'s request and follows the call to the Trace service from the `Synch_client` for deploying the thread. The L4 deploy API is called from the Trace service with the thread information. Then, the scheduler kernel object is called from the L4 API, takes the threads and identifies the scheduler context objects to update to the kernel ready queue.

The following sections explain the coding details of the modules. At first, the `Rq_manager` module is explained, which provides the communication mechanism. Then the Synch_client class is described, followed by the kernel design of the ready queue

update mechanism.

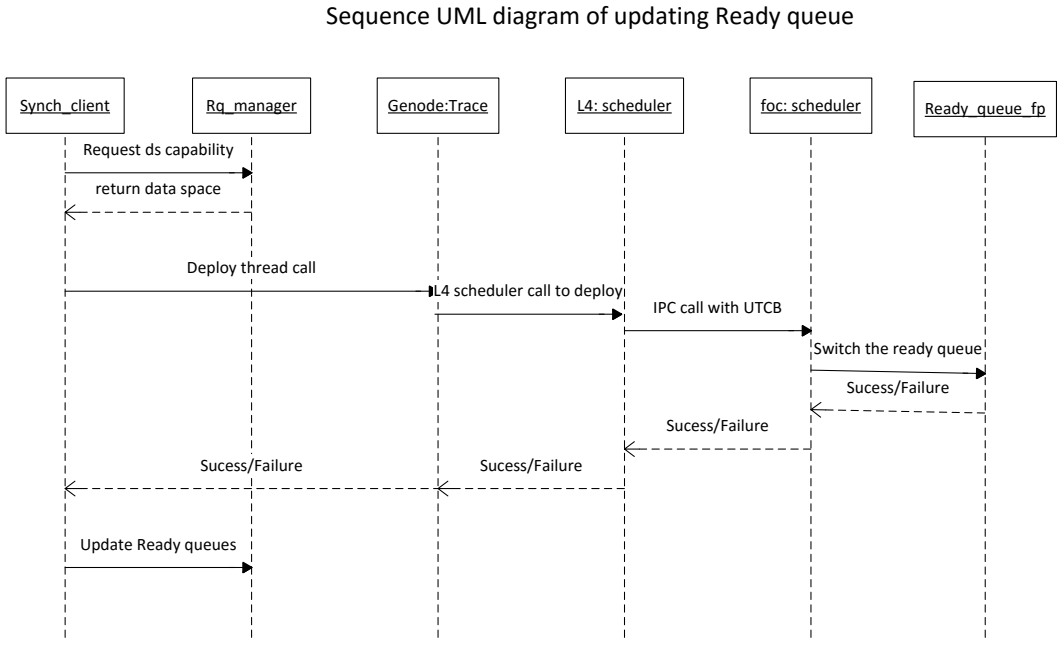


Figure 5.1: Sequence diagram of updating ready queue from the Genode component

5.1 Rq_manager Module

The class `Rq_manager` is the driver class of this component. It creates and maintains multiple ready queues 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 the Genode dataspace. Each entry in the Genode dataspace is a circular array of type Rq_task struct, which represents a Genode thread.

The code listing 5.1 shows the implementation of the `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 the ready queue.

```
class Rq_manager
{

    private:

      int _num_cores = 0;
      Rq_buffer<Rq_task> *_rqs; /* array of ring buffers (Rq_buffer
          with fixed size) */

      int _set_ncores(int);

    public:

      int enq(int, Rq_task);
      int deq(int, Rq_task**);
      int get_num_rqs();
      Genode::Dataspace_capability get_core_rq_ds(int);

      Rq_manager()
      {
        PINF("Value of available system cores not provided -> set to 2.
            ");

        _set_ncores(2);
        _init_rqs(100);
      }

      _init_rqs(int rq_size)
      {

        _rqs = new Rq_buffer<Rq_task>[_num_cores];

        for (int i = 0; i < _num_cores; i++) {
          _rqs[i].init_w_shared_ds(rq_size);
        }
        Genode::printf("New Rq_buffer created. Starting address is: %p
            .\n", _rqs);

        return 0;

      }
}
```

Listing 5.1: Rq_manager class

The Rq_buffer is a template class, which represents the ready queue on the application

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

Rq_buffer has enqueue and dequeue functionalities, which can be called by the `Rq_manager` to insert or remove an element from the ready queue. It also provides a functionality to initialize and create a shared dataspace. The code listing 5.2 shows the process of allocating memory 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 dataspace

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 that are used 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 dataspace must be synchronized, since both the `Controller` and the `Synch_client` use it. This is done using the locks provided by the Genode.

5.2 Synch_client Module

The `Synch_client` class represents a Genode component that controls the process of the scheduler ready queue update. It acts as a client in the Genode by using the services of the `Rq_manager` and accesses the shared dataspace. The `Controller` and the `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 a consumer(`Synch_client`) picks the threads to the kernel ready queue.

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

_rqbufp Hold the pointer to the Rq_buffer type

_lock Lock needed to ensure mutual exclusion

_head Head pointer, points to the start of the Rq_buffer

_tail Tail pointer, points to end of the Rq_buffer

_window The window of free spaces available in between the head and the 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, since the available number of ready queues is unknown at the time of creation. The declaration of all the data structures are shown in 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 the`Rq_manager`. The first RPC call is to obtain the number of ready queues available. For each ready queue available, the Connection object has to make an RPC call to get the dataspace capability, attach to the local RM session and initialize all the pointers shown in Listing 5.4.

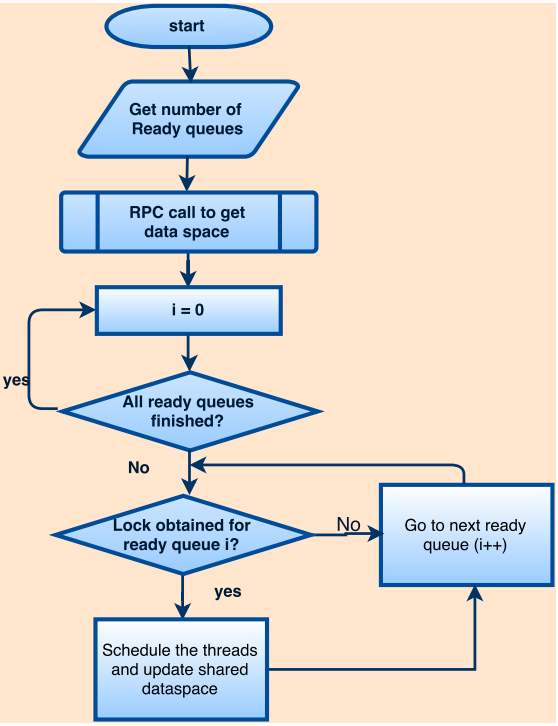


Figure 5.2: The execution of Synch_client module

The working flow of the the `Synch_client` module can be seen in Figure 5.2. 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, 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 get scheduled. Second, it has to reduce the amount of time spent in the critical section as to little as possible. In order to achieve the above mentioned goals, the ids of the threads/tasks are copied to a local buffer, the head and tail pointers updated and the lock is released. In this way, the scheduling of threads by calling the 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. If each thread works on a different ready queue, there is no shared data between these threads, and this 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 Figure 5.3. The following subsections explain the implementation of the modules described in Figure 4.4.

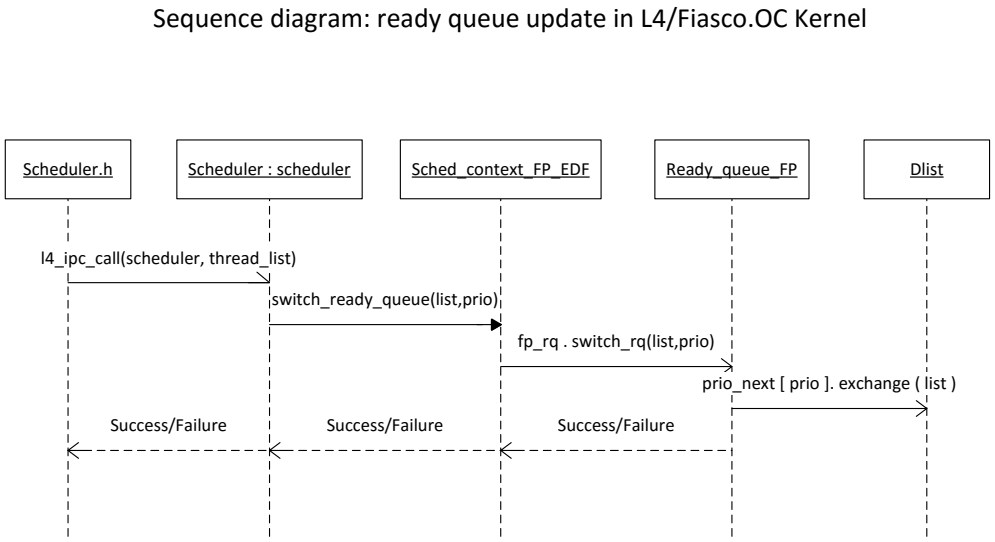


Figure 5.3

5.3.1 Genode Code Changes

The 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 the `Trace` service. The `Trace` service is 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 from the `Synch_client` component to access the L4 APIs.

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

5.3.2 L4 API Calls

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

Listing 5.5 shows the l4_sched_thread_list structure and the L4 API call, which is called from Genode. This code is present 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 perform with the information. In this project, 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 is filled, 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 project, L4 flex represents a thread object. For each thread id that 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.

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. The following sections explain the code changes in each file.

5.4.1 Scheduler Class

The `l4_ipc_call` from scheduler.h invokes the `kinvoke` function. The first parameter of the UTCB is decoded to find out the operation that the thread is involved in. The operation in this case was set to `deploy_thread`, which calls the function `sys_deploy_thread` and can be seen in Listing 5.7. A for loop to get the number of available threads is executed. The associated flex page is derived from the sent items and a lookup is performed to obtain the thread objects. The thread objects can be used to obtain the scheduler context that they are associated with. The scheduler context objects are used

to create a 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 thread's home CPU can be used. The current implementation creates and works with a 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 Ready_queue_base Class

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 either of the ready queues 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 is excluded and the FP ready queue is used by default. However, it can be extended easily to work with both the lists. 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 that are part of the threads and can be run next. Modifying the ready queue list is a critical section operation and mutual exclusive access must be guaranteed. In a uniprocessor implementation, this can work with a simple CPU lock operation, which disables the interrupts on the local CPU so that no other threads can get CPU time in the middle of a ready queue manipulation. For SMP systems, the kernel must use a ready list local to the CPU. The Fiasco.OC implements a per-CPU variable that is used for synchronizing the ready queue access. When using per-CPU variable, the exclusive access to this variable is guaranteed by disabling the CPU preemption. Once this access is finished, the CPU preemption is enabled. By this way, if any CPU is in the middle of a critical section, no other thread is allowed to replace the current thread's execution on the CPU.

The present method uses the per-CPU variable method in which it locks the CPU before exchanging the ready queue list. The Software Transactional Memory(STM) method was implemented, but testing it was a problem because using STM forces the GCC compiler to employ the pthreads library, which was not possible to include for the Fiasco.OC. The STM method will eliminate the need to preempt the CPU since it executes all the exchange 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 contains the idle thread, then the immediate call is made to exchange the list. The *static point-in-time* method checks if a thread from this ready list is getting executed. If the thread is running, the exchange call waits till the scheduler takes the control back and exchanges the thread list. This method uses the `schedule_in_progress` flag to check if the scheduler is busy.

5.5.1 Smart-Sync Method

Finding the suitable time to update the ready queue is explained in section 4.4.4. The implementation of it is termed as `Smart-Sync` method. The *Empty ready-queue* method is implemented by checking the running queue in the Scheduler class. The second method, *Static point-in-time* is implemented by checking the scheduler business and this can be seen in Listing 5.5.1.

```
1 /*
2  * Empty ready-queue method
3  */
4 if(ready_empty(prio)) {
5   rq.switch_ready_queue(&list, prio);
6 }
7
```

```
8  /*
9   * Static point-in-time method
10  */
11 while(scheduler_in_use) { //Do nothing}
12 rq.switch_ready_queue(&list, prio);
```

5.5.2 Ready_queue_fp Class

From here on, 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 are sent to the `switch_rq` function from the sched_context-fp_EDF class. It is assumed at this point that the new list contains the system threads, such as the idle and pager threads that are necessary to execute other threads. The Fiasco.OC uses an idle thread, which keeps the CPU busy when there are no threads to be executed. The idle thread needs to be kept in the new list if it is not existing already. The old list can be used to identify the idle thread, which sits in the end of the old list.

The exchange of the list is a simple operation of changing the head pointers. The cyclic list is implemented in a file called `dlist`. The `exchange` function was implemented to take the list to be exchanged with the present list and set the head of the old list to the new list.

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 Implementation Challenges

This section describes the challenges that were faced during the implementation phase and the how these challenges were overcome.

5.6.1 Mapping the Threads from the Genode to the Kernel

The threads are represented differently in the Genode OS framework and in the Fiasco.OC kernel. This was a problem while mapping the threads from the user-space to the kernel-space. A thread object can be identified by the integer ids of type *unsigned long* that they carry. However, these differ between the user-space and the kernel-space. The `thread_base` class in Genode has a native thread id, but it is not the same as the one represented in the kernel.

To identify the kernel id, the thread creation model was followed to check the mapping process. This happens in the L4 layer and the Scheduler class as explained in Section 5.3.2.

5.6.2 Sending Threads to the Kernel Module

The thread ids and their priorities have to be sent to the kernel module from the user-space application in order to execute them. the thread ids can be sent one at a time, but this slows down the process because the calls have to be made for each thread id. The dynamic array creation also doesn't work since the user-space memory cannot be accessed in the kernel-space.

The solution was to use a structure which has an array for thread ids and an array for the corresponding priorities. This structure is defined in the *L4/scheduler.h* file (see 5.3.2).

5.6.3 Exchanging the Ready Queue

The ready queue list is sent from the `Scheduler` object to the `Ready_queue_fp` class for to be exchanged. The swapping is not possible by just assigning the new list address to the old list address because it doesn't ensure the validity of the list since the head pointer of the list is not assigned. The solution is to assign the head pointer of the old list to the new list (see 5.5.2).

6 Testing and Results

In this chapter testing of the modules is explained. The complete Observer-Controller architecture is still in development stage, so the integration testing of the complete architecture described previously was not possible. So the user-level Synchronization module(Synchronization 1) is tested separately from the kernel Synchronization module (Synchronization 2). The user-level component is tested to make sure the dataspace access is possible and it can communicate with the `Controller` via the dataspace.

To test the kernel ready queue update mechanism as a standalone component, the Genode thread ids need to be sent from a user-level component. Therefore, a new Genode component called `gehello` was developed as a utility to test the kernel module. It creates a set of threads and passes them to the kernel. In order to obtain the thread ids required Genode's trace facility is extended to get the thread ids from platform thread class of Genode OS framework which is explained in the next section.

6.1 Test Utility Using Trace: gehello

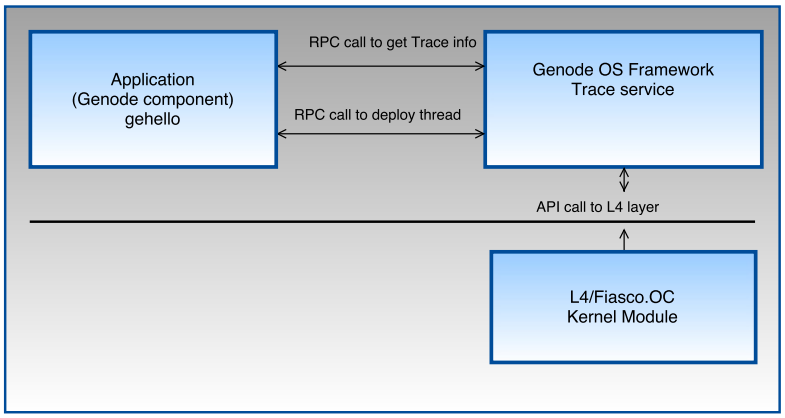


Figure 6.1: Test utility design using Trace service

The gehello test utility is implemented as a Genode component, it creates a set of threads and passes them to the kernel module to update them to the ready queue. In

order to create the threads, it has to inherit the Genode's thread class and specify the stack size as a template parameter.

The Listing 6.1 shows this thread creation. The class `Mythread` inherits the Genode's Thread class. It has a constructor, which in turn calls the base class's constructor with a string (name of the thread) as a parameter. This call goes to platform thread class to create the thread as explained in section 3.3. The entry function of this thread just does a print on the command line. The `Thread_creator` class creates the `Mythread` class object(myt). It has a run_thread function which calls the start method associated with the thread. The start method invokes the entry function defined in the `Mythread` class and it has to call join method of the thread to end specify the end of thread's execution.

```
1  class Mythread : public Genode::Thread<2*4096>
2  {
3    public:
4      Mythread() : Thread("MyThread") { }
5      void entry(){
6        printf("I'm a thread\n");
7        // Some useful work
8      }
9  };
10 class Thread_creator
11 {
12   Mythread myt;
13   public:
14     int run_thread(){
15         myt.start();
16         Genode::Thread_capability mycap = myt.cap();
17         PINF("Got Thread capability information. %lx\n", myt.tid());
18         myt.join();
19     }
20 };
```

Listing 6.1: gehello trace utility

6.1.1 Trace Extension

The information about the Genode's tasks/thread can be obtained from using Core's Trace service. Trace service was extended to get the thread information from Genode's platform thread class.

The `source_registry.h` has a information struct called `Info`, which is returned by Trace service to an application. A new element has been added to the Info struct for thread id.

```
1   struct Info
2     {
3       Session_label      label;
4       Thread_name        name;
5       Execution_time     execution_time;
6       Affinity::Location affinity;
7       unsigned       prio;
8       unsigned long    thread_id;
9     };
```

The `info()` method of the `subject_registry` calls the trace_source_info() function of the `Cpu_session_component` class to obtain the thread information. The class `Cpu_session_component` has the `platform thread` object and returns the thread id.

```
1   Trace::Source::Info trace_source_info() const
2       {
3         return { _session_label, _name,
4                  _platform_thread.execution_time(),
5                  _platform_thread.affinity(),
6               _platform_thread.prio(),
7               _platform_thread.thread_id()};
8       }
```

Platform thread class is extended with a method to return the thread's local id as shown below.

```
1 unsigned long Platform_thread::thread_id() const
2 {
3   return _thread.local.dst();
4 }
```

6.2 Building the System

The Synchronization module is tested on Ubuntu 14.04 LTS using QEMU to virtualize the hardware (PBX-A9) board. The following subsections explain the installing dependencies and compiling Genode along with the Fiasco.OC.

The changed Fiasco.OC and Genode source can be downloaded from [Cha16].

6.2.1 Installing Dependencies

Genode and Fiasco.OC require these following packages to be installed.

- GNU Make version 3.81 or newer

- libSDL-dev
- tclsh and expect
- byacc
- QEMU: Required fir virtualizing the hardware.

Another option is to install the pre-compiled Genode tool chain for Linux available in Genode website.

6.2.2 Compiling the gehello Application

After installing the dependencies, clone the Genode operating system branch `focnados` from GitHub repository.

Once the Genode is downloaded prepare the Fiasco.OC kernel by issuing *make-prepare* command in *repos/base-focnados* folder. But this would use the default Fiasco.OC code. In order to use the modified code for the thesis the *port/focnados.port* file should be updated to access the code from the above mentioned GitHub repository.

Create a build directory using Genode tool for pbxa9 board using the below command

```
$ ./tool/create_builddir focnados_pbxa9
```

The `gehello` application exists in the repos folder. By issuing the below make command in the build directory, the `gehello` application can be built and tested. The expected output is given in the section 6.3.

```
$ make -j4 run/gehello
```

6.3 Results

The `gehello` application creates five threads and the Trace service returns the thread information to the user-space application. Listing 6.2 shows the information returned from the Trace. The names are compared to find out the threads created by the `gehello` application and these thread ids are selected and sent to the kernel using the Genode API.

```
1 ID:11 0 prio:128  thread_id: 293000 init
2 ID:9 0 prio:128  thread_id: 2c0000 init -> gehello name:gehello
3 ID:8 0 prio:128  thread_id: 2e1000 init -> gehello name:MyThread
4 ID:7 0 prio:128  thread_id: 2e6000 init -> timer name:timer_drv_ep
5 ID:6 0 prio:128  thread_id: 2ec000 init -> gehello name:MyThread
6 ID:5 0 prio:128  thread_id: 2f6000 init -> gehello name:MyThread
```

```
7  ID:4 0 prio:128  thread_id: 2fd000 init -> timer name:timeout_scheduler
8  ID:3 0 prio:128  thread_id: 301000 init -> gehello name:MyThread
9  ID:2 0 prio:128  thread_id: 30b000 init -> gehello name:MyThread
10 ID:1 0 prio:128  thread_id: 317000 init -> timer name:signal handler
11 ID:0 0 prio:128  thread_id: 32c000 init -> gehello name:signal handler
```

Listing 6.2: The returned information from the Trace service

The kernel threads are then identified in the kernel and added to the run queue. Listing 6.3 shows the kernel thread ids and the updated ready queue, when updating single thread at a time and the updated threads start execution when the `gehello` application starts the threads.

```
1
2  [Scheduler:sys_deploy] Thread to be scheduled: cd
3  fp_rq: cd => d => e9 => end
4
5  [Scheduler:sys_deploy] Thread to be scheduled: d8
6  fp_rq: d8 => cd => d => e9 => end
7
8  [Scheduler:sys_deploy] Thread to be scheduled: e2
9  fp_rq: e2 => d8 => cd => d => e9 => end
10
11 The thread execution:
12 [Platform_thread: start] 2e1000 started!
13 [init -> gehello] Got Thread capability information. c000
14
15 [init -> gehello] I am a thread!
16
17 [Platform_thread: start] 2ec000 started!
18 [init -> gehello] Got Thread capability information. f000
19
20 [init -> gehello] I am a thread!
```

Listing 6.3: The kernel thread ids and ready queue update

7 Summary, Future Work and Conclusion

This chapter is divided into four sections. Section 7.1 presents a summary of the thesis. The contributions and importance of the work along with the limitations are discussed in 7.2. Section 7.3 outlines the possibilities for future work, and Section 7.4 presents the conclusions drawn from the thesis.

7.1 Summary of the Thesis

The dynamic task update method introduced in this thesis is required for the Observer-Controller architecture used in the KIA4SM project. The implementation is done on the Genode Operating System Framework and the L4/Fiasco.OC microkernel. This thesis presented the lock-based and lock-free synchronization methods and their benefits in terms of *implementation, read and write speed, security, deadlocks*. The lock-free methods, Read-Copy Update(RCU) and Software Transactional Memory (STM) were found to be better suited for this thesis work since they provide good read/write speed and prevent deadlocks.

The Genode OS framework has a recursive tree structure, and is used for building safe, secure and robust operating systems. It can be used with different microkernels. L4/Fiasco.OC was the preferred choice of microkernel for the KIA4Sm project, since it offers many features such as, paravirtualization and multi processor support. The L4/Fiasco.OC is a object capability system, where everything in the kernel is represented as an object and objects interact with each other through a kernel provided IPC mechanism. The thread execution in the kernel is divided between execution context, which takes care of execution parameters and scheduling context object, which holds scheduling parameters.

The design and implementation details of the Synchronization module are described in Chapter 4 and Chapter 5. The communication between the Controller and the Synchronization module is that of a producer-consumer relationship and the communication between these two modules is handled by a shared dataspace, which is synchronized by using the locks provided from Genode. The limitations in accessing the ready queue from the user-level application, led to the development of kernel module. The application interacts with the Genode API, which in turn calls the L4 layer. The L4 layer makes the flex pages for the threads that are sent from the Genode

and sends them to the scheduler kernel object with an IPC call. The scheduler decides the right time for updating the ready queue and updates the threads. One option is to create a new ready queue list with the threads and exchange it with the actual ready queue. the other option is to update the existing ready queue.

Since the integration testing was not possible, only the individual components were tested. A Genode component was created to test the kernel module by extending the Genode's trace service.

7.2 Discussion

7.2.1 Thesis Contribution

This thesis has developed a method for updating the kernel ready queue. A user-level component was developed to handle the communication with the Controller and the kernel module handles the ready queue update mechanism. The thesis gives insights to the working environment of the L4/Fiasco.OC kernel and its interaction with the Genode OS framework. The thread creation model in the Genode OS is explored and the thread migration model of the L4/Fiasco.OC is explained to scheduling method of the L4/Fiasco.OC scheduler.

7.2.2 Limitations

Though proposed solution updates the task ready queue successfully, it has the following limitations. First, the user-level application handles each ready queue one at a time and has to wait till it acquires the lock to access the shared dataspace, which may lead to starvation of the threads. Second, the user-level Genode component cannot access the kernel ready queue list directly or make calls to the L4 API layer of the kernel. This limits the scope of the user-level component. Though creating a new ready queue list to exchange with the actual ready queue takes place, the ready queue exchange mechanism, however doesn't ensure the thread execution. The current implementation of the kernel module is limited to work with fixed priority scheduler.

7.3 Future Work

This section describes the ideas for future research to be done on this concept. Further investigations should be done to identify the best approach to integrate the Controller and the Synchronization module. The proposed idea for integration is that the Observer gathers data including the Genode thread ids to the Controller and the Controller has to take decisions on scheduling and make the corresponding thread ids available in

the shared dataspace. The user-level Synchronization module can read this dataspace and update them to the ready queue using the kernel module. Furthermore, the kernel module must be extended to make it work with different types of schedulers.

The initial aim of the thesis was to develop a high-level component for the ready queue update. However, this had to be changed since the ready queue was not accessible from a high-level component. Further work should be carried out to find out if a user-level component can replace the kernel module's work. The communication between the user-level module and the kernel module involves many API calls. More research could be done on this regard to reduce the number of API calls.

7.4 Conclusion

The Synchronization module developed in this thesis provides a working method for updating the threads to kernel ready queue. It serves as a good starting point for the `Observer-Controller` architecture's goal of having a user-level component for ready queue update.

List of Figures

List of Tables

Bibliography

[Bra+06] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck. "Organic Computing–Addressing complexity by controlled self-organization." In: *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*. IEEE. 2006, pp. 185–191.

[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.

[Her91] M. Herlihy. "Wait-free synchronization." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), pp. 124–149.

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

[Lac] A. Lackorzynski. *Genode + Fiasco.OC: How to access multiple run queues?* [Accessed: 22-October-2016]. URL: `http://os.inf.tu-dresden.de/pipermail/l4-hackers/2016/007897.html`.

[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.