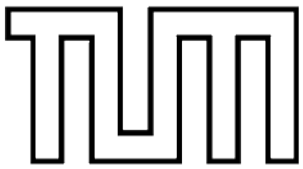


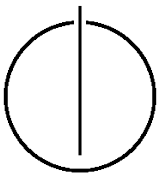
TECHNISCHE UNIVERSITÄT MÜNCHEN

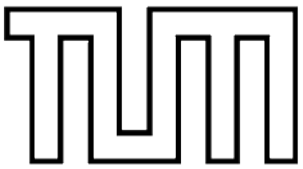
DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Real-time container clusters for seamless computing

Fariz Huseynli





TECHNISCHE UNIVERSITÄT MÜNCHEN

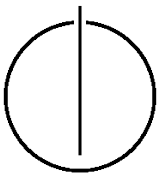
DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Real-time container clusters for seamless computing

Implementierung von geclusterten Echtzeit-Containern zur nahtlosen Datenverarbeitung

Author:	Fariz Huseynli
Examiner:	Professor Dr. Michael Gerndt
Supervisors:	M.Sc. Sreenath Premnadh
	Dipl.-Eng. Vladimir Podolskiy
Submission Date:	September 17th, 2018



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

September 17th, 2018 Fariz Huseynli

Abstract

We are living in the era of containerization, while container technologies are the main actors in the development scene nowadays. Moreover, cloud computing is not an exception. However, there are 23 billion IoT devices in the world, and their computational characteristics are growing from year to year. This fact makes previously fragile ideas to connect the cloud and edge to one sound system to run industrial applications on it, more realistic.

One of the obstacles in this way is the fact that most of those devices mainly run the real-time software. The idea behind this work is to tackle this problem and to understand the limitations of container technologies in the area of real-time applications.

The paper describes the experiments done to clarify the gaps in current solutions that block them from being used in real-time scenarios. Those experiments deliver positive results, and the reasons are explained in detail. It turned out that container technologies are mature enough to be used for real-time applications due to the way they are implemented.

Apart from that we also provided some solutions for enabling the orchestration layer to be aware of real-time containers and make wise scheduling decisions.

Contents

Part I.

Introduction and Background Theory

1. Introduction

1.1. Background

The objective of seamless computing is to provide an application deployment and runtime environment that can be used to seamlessly run industrial applications in a cloud to edge continuum. The constant improvement of resource and computational capabilities of the edge devices makes it possible for cloud-to-edge systems to become more homogeneous. The ability to exploit this scenario to provide the seamless flow of the applications through the system will allow a better resource and financial management. The current implementation uses container orchestration Kubernetes. One of the limitations is currently, is lack of research for the container runtime (Docker) and the orchestration framework (Kubernetes) regarding the support for real-time requirements of industrial applications.

1.2. Previous Work

The concept, requirements, and possible implementation approaches of seamless computing are described in the Siemenss most recent paper on this topic at HPCS 2017 [28]. One of the concerns is to support real-time requirements of industrial systems and applications.

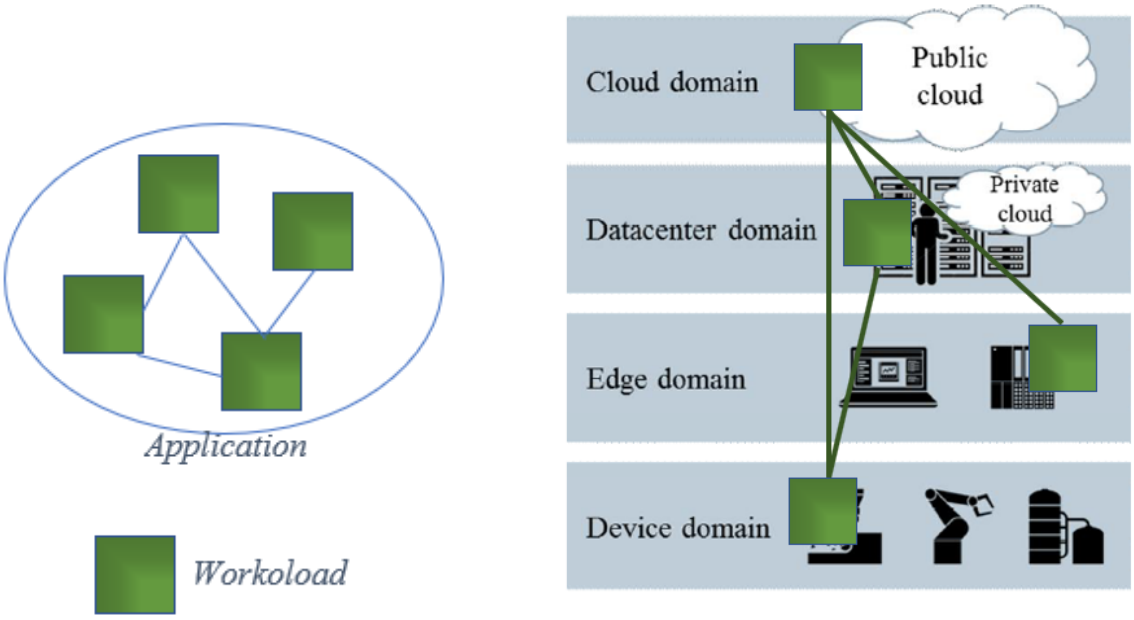
Apart from that paper, there is an ongoing implementation process of seamless computing based on heterogeneous clusters managed by Kubernetes [10]. This implementation relies on the container technology named Docker [17]. The containers are orchestrated by an open-source system for automating deployment, scaling, and management of containerized applications Kubernetes.

1.3. Research Questions

The real-time processing is of enormous importance in the context of seamless computing. Lets consider the model used in [28]. As a base for re-locating applications, we get into modular applications that are composed of self-contained components, so-called workloads. We define workload mobility as the ability to move application components within and across compute domains, using the same code, application lifecycle tools, deployment artifacts, etc.

Figure (b) shows an example scenario of a multi-domain application

Workload mobility comes with two different qualities:



(a) Workloads in application

(b) Multi-domain application

- Static workload mobility: Application components are allocated at deploy time. This allows mapping the requirements for the components to the static characteristics of the compute domains.
- Dynamic workload mobility: Application components can be re-located at run time. This model allows reacting to the dynamic behavior of the compute environments and the application components. Using this model, actual system load or error situations could lead to improvements of the system by dynamic reconfiguration, yielding e.g. higher availability.

Both qualities mentioned above are the high-level requirements that must be met to achieve seamless computing. However, real-time performance requirements force some components to be located close to the attached sensors and actuators, i.e., not all components can be deployed in the data center domain or even in the public cloud domain. This causes our hands tied in the way to attain a complete seamless environment. While deployment and in the process of running the multi-domain applications we must always deal with the fact that some segments of the application are critical with respect to real-time processing and cannot be a real part of the desired homogenous environment. This results in a less generic and less flexible, semi-seamless system. Two main constraints that can potentially limit the real-time processing of the workloads are the following:

- Container runtimes do not support real-time code.

- Lack of awareness of Kubernetes scheduler of real-time constraints and capabilities

This paper will be concentrating mainly on the first one, namely, real-time container clusters. However, the scenarios tackling orchestration of real-time containers will also be analyzed.

1.4. Approach and Related Tasks

While Docker enjoys widespread support in server environments, more light-weight container environments exist, especially for more constrained edge nodes.

One of the main goals of this paper is to come up with container solutions to meet the real-time requirements and thus become a better fit for industrial applications. This requires a meticulous selection and installation of appropriate container environment (relevant real-time OS, container runtime). During the work, several OS with real-time support will be chosen for further tests of the runtime containers on them. The idea here is to have several playgrounds with different systems installed and to test their behavior against various possible industrial scenarios and applications. This could help to give better a better understanding of which solution is a better fit to a particular setup.

Among the research goals, there is also to find out existing mechanisms in different container runtimes, such as Docker, to support real-time capabilities. The work would be mainly focused on this container solution unless inadequateness of choice proved underway.

Another topic that will be elaborated in this work is concerning the definition of the real-time scenarios. Before starting to adjust the container runtimes, it is important to identify what are those real-time scenarios. For instance, what is the performance of a real-time task on the host operating system and what is the noise for the same task in the container, as well as the performance of a real-time task on one container against the noise in the other containers? The goal here is to define real-time scenarios, decide on meaningful real-time metrics and the ways they can be measured/monitored.

Another research question of this paper is the degree of fulfillment of fixed real-time requirements by current container runtimes. This chapter is essential for finding out the constraints of present solutions against defined real-time requirements and to discover the causes of their limitations. Here, configuration options in container runtimes and underlying operating systems will be evaluated and practically tested to support real-time tasks. The communication of the container with its hypervisor will be thoroughly investigated. We will try to catch the signals of the Operating Systems which cause interrupts in the works of containers, as well as the tasks of the kernel which are prioritized over the workflow of the containers. This could give an overall picture of the motives of failures in the real-time requirements.

One of the other research goals of this thesis is to analyze how well can Kubernetes orchestrate real-time containers without loss of quality. The idea is to come up with scenarios where real-time requirements can be violated if not treated carefully. As a result, we want

to obtain a scheduler which is aware of the real-time needs of a container in the process of moving it to the nodes. It is essential that after the Kubernetes scheduler recognized a real-time sensitive container, it knows precisely how to handle it. We will work out the scenarios when a container might need special treatment and try to make the scheduler to behave accordingly.

1.5. Outline

In Chapter 2 we dig into the literature to find what is already done in this field and what would fit our research questions the best. The questions we are finding the answers for are related to performance metrics of the real-time systems, benchmarking tools and real-time operating systems. Chapter 3 is entirely devoted to experiments. We describe the experiments made for tackling the real-time performance of the containers. We also discuss the results and try to explain them. In Chapter 4 we check various real-time orchestration scenarios and try to implement them using tools provided by Kubernetes. Chapter 5 talks about the results and future works on this area.

Part II.

Literature Review

2. Literature Review

The main goals of the literature review were the following:

- Figure out what are the performance metrics for real-time systems
- Which benchmarks exist to measure those metrics in different scenarios
- Which real-time solutions exist in the market

2.1. Performance metrics for Real-time systems

The results of this part of the research are crucial for the progress of the whole thesis. It is very important to know for sure which parameters of the system influence on its ability to meet deadline of the real-time software running on it, in other words, fulfill real-time requirements. One of the missions of the thesis is to apply different container technologies to the real-time systems in order to estimate to which extent these technologies can tolerate the harsh time constraints that the real-time systems demand. In order to be able to claim the preservation or deprivation of real-time competence of the systems after alteration applied on them, we must have reliable sources of evidence. Any wrong decision here can potentially affect the accurateness of the results of the whole research, due to the fact that the outcome of this part of literature review will be carried on through the rest of the thesis. For example, if a chosen performance metric is getting affected by only the use of containers and does not indicate the changes in real-time performance in given scenario, this can be very misleading and lead to the wrong conclusions. The opposite scenario (when the behavior of the metric allows us to think that the real-time requirements are met) could be even more harmful. The most challenging part here is the fact that the conclusion should be system-wide and generic, without narrowing down to a particular application.

2.2. Benchmarks

We need to have numbers to be able to compare different systems and scenarios on the subject of RT compatibility. If we have useful, reliable metrics we need to be able to measure them, to see their values in the pure real-time environment and with container technologies applied. One of the ideas is to have the same measurements for three different environments: a standard Linux system without real-time enhancements, pure real-time

Linux and real-time containerized Linux. The assumption is that the results for real-time containerized Linux will land somewhere in between the results for the other two environments. Our interest here is to observe how much closer the results are to the pure real-time Linux.

2.3. Real-time Operating System

The other aspect of the literature review is to choose a suitable RTOS for our experiments. The priority here is non-commercial RTOS which is also widely used by the community. During the experiments, there can be a necessity to dig into the low-level characteristics of the RTOS (which should be accessible) and also to get help from the works done openly to the public as well as from the people working with it. Thus, a broader community would be a higher preference here. Apart from that, it is important to keep in mind the technologies we plan to use on the chosen operating system, making sure from the early stages on that those will be supported.

2.4. Literature review takeouts

Real-time operating systems are claimed to suffer from large memory footprint, jitter, and computational overhead according to [18]. The higher tick frequencies, the more computational overhead occurs, since it is mainly caused by tick interrupt management. Other reasons for CPU getting distracted from running the processes are task scheduling, resource allocation, etc. In [18], they try to reduce computational overhead of task scheduling, interrupt management by dragging out the implementation of task, resource and interrupt management at the hardware level. This is beyond of the scope of this thesis; however, the metrics that they are analyzing and relying on as indicators of improved RT capabilities can be useful to for this research too. As a clear indication of improved RTOS, they illustrate reduced overhead by getting rid of *tick interrupt handling*, *scheduling latency*. In the final tests, their overhead contains only context switching time, and time spent for API class. It is also advised to keep tick frequencies low because they cause bigger overheads.

Some of the software-only methods are considered in [16] as good candidates for measuring software performance of real-time systems.

- Performance profiling is dedicated to showing the developers where precisely in the application the system spends most of its time. It allows function-by-function analysis
- A-B timing is considered as one of the most basic methods for real-time systems. Its idea is to measure time spent between two points in the code. This might be handy to determine which part of the application blocks the system from satisfying real-time requirements

- Response to external events measures *interrupt latency* and is named as chiefly valuable for real-time
- RTOS task performance measurements consist of two types. First is task deadline measurements which concentrate on the distinct tasks meeting their real-time requirements. The time is measured between the event being triggered and the task finishing execution. Task profiling performance measurements are much like the performance profiling described above. [16] The difference is that task profiling measurements record where the system is spending its time on a task-by-task basis

The measurements in [16] may fit the research goals of this thesis in case there is a specific real-time application we want to conduct the tests on.

Like in [18] the idea behind [30] is to exploit some hardware accelerators to improve the efficiency of RTOS. One of the targeted metrics is *latency* which is considered as wasted time in RTOS in [30]. The smaller it gets, the more responsive becomes the system. A specific software latency, namely the kernel lists handling always adds latency to the system. Also, the dynamic data structures used in RTOS often cause *unpredictable response times*, since then the time spent for choosing the next runnable task from the runqueue heavily depends on the position of that task in the data structure used. This behavior is called *jitter*. Another latency that is worth paying attention to is *mutex latency*. Improving system characteristics like *scheduling*, *time* and *task management* improves the performance of RTOS [30]. Thus, these metrics can be taken as indicators of better real-time systems. The degree of their degradation while adding containers to the system can be considered as a worsening of the real-time system.

Worst Case Execution Time (WCET) analysis is preferred over other metrics like response time, throughput, utilization of the resources, etc., in [34] to measure the performance of real-time systems. Having a WCET for each task can help in ensuring that the whole system works correctly [34]. Summing up the WCET for all the task which have to to be completed through different paths will give an estimation about WCET for the whole system [34]. There are several characteristics of real-time systems which are not so easy to handle using WCET analysis [34]. Some of them are the following [22].

- WCET analysis assumes that there is a non-interruptible program execution
- Through go-to statements and having multiple loops, control may flow at different places. This makes it hard to analyze the loop bounds for the static analyzer
- Real-time is about dynamic function calls and dynamic loop bounds which are hard to predict before run-time
- WCET analysis needs to have detailed information about the interrupt calls, which is not easy to provide before the execution
- Overestimation in WCET analysis can also occur due to frequent cache changes in real-time systems because of priorities and task preemption

Despite all the issues as mentioned above, WCET analysis is considered as a proven choice in analyzing the real-time performance [32]. Different categories of WCET tools rely on different techniques.

1. Static tools are based on mathematical modeling of the system. They analyze possible control flows, make many assumptions and provide asymptotic results [25] These tools do not run on any specified processor and are based on the high-level abstractions of the system [34]

2. Measurement based tools compute the execution times by actually running the codes on the real target system or simulated environment. The issue here is the fact that it is tough to ensure that the tool ran through the worst-case input. Thus, the WCET here are mostly underestimated [35] Results usually are not that reliable and tight

Static WCET analysis is considered a better choice by the research groups for analyzing real-time systems [22]

Real-time systems have to be able to react quickly to external triggers. The systems recognize the events they must react to as interrupts. The processor executes certain actions that were programmed to be executed as a reaction to the particular event. The following instructions are expected during the interrupt processing: [23]

- Stopping the currently executing thread and saving its data to the registers
- Interrupt Service routine takes up the control
- Performing processing in the ISR
- Saving incoming data associated with interrupt and processing appropriately the output values
- Figuring out the next thread to executing, taking into account the changes done by the interrupt
- Clearing the interrupt hardware for the next interrupt
- Transferring control to the next thread

Real-time performance of the system depends heavily on the implementation of these operations. [23] The efficient and wise organization of these steps can result in a significant boost of the performance. In addition to system interrupts, the other system services like scheduling, message passing, resource allocation are important in the context of RTOS performance [23]. A free-source benchmark suite called The thread-metric benchmark suite can be used for measuring RTOS performance and is designed to be easily adapted to any RTOS [23]. This suite currently supports the following services:

1. Cooperative Scheduling test runs five threads of the same priority in a round-robin fashion incrementing the counter of each thread. The idea is to verify that the differences in counter values do not exceed one during the test.

2. Preemptive Scheduling. The test consists of 5 threads again but with different priorities. All the threads except the highest priority thread are left in the suspended state. Starting from the lowest priority thread, each thread counts resumes the next thread in the priority list and suspends (except the lowest priority thread). The higher priority thread preempts the previous lower priority one. Eventually, the thread with the highest priority gets its turn and also suspends, which lets the lowest priority thread (the only one not suspended) start the chain again.

3. Interrupt Processing. A single thread causes an interrupt and results in a call to the interrupt handler. The interrupt handler increments a counter and posts to a semaphore. After the interrupt handler completes, processing returns to the test thread that initiated the interrupt. The thread then retrieves the semaphore set by the interrupt handler, increments a counter and then generates another interrupt. [23]

4. Interrupt Preemption. Processing is different from the previous test in one detail. Interrupt handler resumes a higher priority task; thus, the initial thread is getting preempted in return from interrupt.

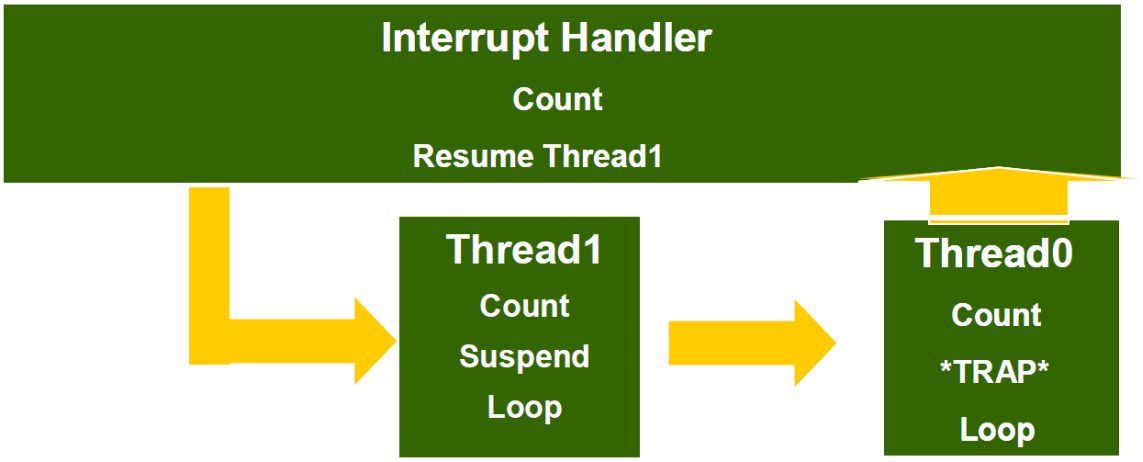


Figure 2.1.: Interrupt preemption testing [23]

5. Message Passing does a basic message passing from the thread to the message queue and vice versa.

6. Synchronization Processing. This test consists of a thread getting a semaphore and then immediately releasing it. After the get/put cycle completes, the thread will increment its run counter. [23]

7. RTOS Memory Allocation.This test consists of a thread allocating a 128-byte block and releasing the same block. After the block is released, the thread will increment its run counter [23].

Six metrics are considered to be the most important real-time characteristics in the Rhealstone Benchmark. [12] Those are average task switch time, average pre-emption time, average interrupt latency, semaphore shuffle time, deadlock break time, and inter-task message latency.

1. *Average task switching time* is the average time for switching between two independent processes

2. *Average preemption time* is average time spent for switching to a higher priority process

3. *Average interrupt latency* is average time between the occurrence of an interrupt at the CPU and the execution of the corresponding interrupt service routines first instruction [11]

4. *Semaphore shuffle time* is average time between a request to lock a semaphore, which is locked by another process, and the time when its request is granted [11]

5. *Deadlock break time* is the time that is spent to resolve a deadlock when the lower priority task holds a resource which is also required by the higher priority task preempting that lower priority task.

6. *Inter-task message latency* is an average delay in message passing between two processes.

The Rhealstone benchmark consists of six C-programs which measure the metrics as mentioned earlier. There are two significant drawbacks of this benchmark. First of all, it measures the average records of those metrics, whereas in real-time systems we are more interested in the worst-case results. Also, this benchmark does not provide separate data for the measured metrics but instead combines them with some weighted value and gives the overall result. This test can be used as an excellent showcase of the overall system performance but does not provide insights into the metrics in separate. Nevertheless, it gives an idea of which metrics can be significant.

It is important to mention that application-oriented performance benchmarks generally offer more realistic results. [11] real applications are not logically too much different from the benchmarks in using the necessary operations. It is always a better idea to try to adjust

the application to the system an make a test on that specific application trying to improve its performance. That would give more stable and reliable results.

It is indicated in [5] that the principal real-time variant of Linux is the PREEMPT_RT patch and the cyclictest test is typically used to measure its scheduling latency. Cyclictest treats the kernel as a black box and reports the scheduling latency right away. Ideally, the higher priority task should be able to grasp the CPU instantly after it becomes runnable. In real life that is not the case [5]. The transferring of the control to an interrupt handler by the processor can be delayed if interrupts are temporarily masked. The job of a scheduler and interrupt handler can also be delayed by cache misses, memory bandwidth, lock contention, lengthy context switches, etc. The scheduler can also be delayed by disabled preemption by the kernel. There can be a bunch of other predictable and unpredictable time-consuming operations popping up during task scheduling. All of these contribute to the overall delay, called *scheduling latency* [5] Scheduling latency has become a standard metric for real-time systems performance evaluation because of its significance [5]. The PREEMPT_RT patchthe de-facto standard real-time variant of Linux aims explicitly to improve scheduling latency by reducing the number and the length of critical sections in the kernel that mask interrupts or disable preemptions [9] [29]. According to [5], Low scheduling latency as reported by cyclictest can be considered the gold standard of real-time performance in the Linux real-time community", whereas one of the missions of PREEMPT_RT patch is redesigning the crucial parts of the kernel to increase the scheduling latency dramatically. To give the rough idea on how cyclictest achieves scheduling latency, it would be worth to mention that the execution of the test is divided into three steps. According to the given parameters, the program starts a number of threads with high priority (which is adjustable). These threads enter to the cyclic execution phase and go to sleep for the desired time. After the threads resume again, the time difference of planned and the actual restart is reported as scheduling latency. In [38] identification and analysis of performance metrics for real-time systems the following ones were selected:

- Context switching time
- Preemption time
- Interrupt latency
- Semaphore shuffling time

2.5. Literature review results

After doing some thorough research on which metrics to choose and how to measure them the choice fell towards the overall scheduling latency and cyclictest respectively. Scheduling latency is accumulated by most of the other performance metrics like interrupt latency, context switch, preemption time, task scheduling, etc. It gives an overall performance information of the system. Cyclictest is an accepted tool for measuring the scheduling

latency and a lot of research groups use it for evaluating real-time capabilities of the systems. As of RTOS the state of art is considered to be Preempt_RT patch which is compatible with most of the linux kernels and is developed to reduce the scheduling latency. One of the other advantages of Preempt_RT is the fact that it uses a single kernel approach and preserves all the features of the distro, which is important for us, since we are going to apply container and orchestration technologies onto the system. Apart from that there are already tons of scheduling latency experiments done on the different Preempt_RT patched machines during the years, which is going to provide a safer and easier start.

2.5.1. Cyclictest

`Cyclictest` is one of the most frequently used tools for evaluating the relative performance of real-time systems [8]. Since most our latency results will be produced by this benchmarking tool, it would be useful to have a look which ideas stand behind it and how it measures the latency.
`Cyclictest` program has a very simple logic. The main function runs with low priority and collects the parameters set by the user. Its main function is to call the `timerthread()` function for each thread and to print statistics during the execution of the test. `Timerthread` has a higher priority and is responsible for setting up each thread (according to passed parameters like) and running a test loop. The logic behind the test loop is the following [14]:

1. It gets the current time.
2. According to the `interval` parameter set by the user the thread defines when it will want to wake up. This value indicates when the thread would wake up if the latency was 0.
3. Then the thread steps into the endless loop. The thread goes to sleep until the time defined in the previous step.
4. When the thread wakes up it takes a snapshot of current time again and finds the difference between the desired time of waking up and the current time. This difference gives us the latency.
5. The threads add `interval` value to the current time again and continues with the next iteration.

There are many reasons that can make the thread wake up later than it wanted. The workflow is roughly the following [33]:

Event 1. External interrupt triggers (clock expires). The following delays occur after that:

- possible delay until `cyclictest` has the highest priority

- scheduler overhead
- possible delay until IRQ is enabled
- IRQ handling
- additional external interrupts
- context switch
- block of sleeping lock
- many other factors...

Event 2. Control is finally transfered control to cyclictest. Time difference is captured.

There can be many other delay factors in addition to the ones mentioned here. Moreover, their order of happening might vary and same factors can occur several times during one control transfer.

Part III.

Body: What Was Done for the Thesis

3. Experiments and Evaluation

3.1. Investigating Linux scheduler

3.1.1. Completely Fair Scheduler

This chapter is dedicated to the Linux kernel scheduler with the intention to illustrate the logic behind the organization of the processes in the kernel. We will first analyze the behavior of the standard Linux scheduler (Completely Fair Scheduler). The idea is to understand the algorithm of ordering the tasks used in this scheduling and the motivation behind it. Then we will see which logic is added to the scheduler implementation to achieve the real-time behavior (`SCHED_FIFO`, `SCHED_RR`). Besides, the means of real-time group scheduling will be discussed to illustrate how the CPU resources can be handed over to the real-time tasks. This is a fundamental feature for preserving real-time scheduling capabilities in containers world.

Linux has a multi-tasking kernel. Due to the fact that more than one process is allowed to run on a single timeframe, the process scheduler has to be able to share the CPU according to the scheduling policy of the processes. The central function that the whole kernel uses to invoke the scheduler to select the next task is `schedule()`. The mission of this function is to decide on the new task to execute and to fulfill the context switch to it. It might as well happen that the current task is the most suitable one to run next. In this case, the scheduler essentially does nothing. The `schedule()` function can be called in several cases:

1. The task that is running at the moment goes to sleep
2. The sleeping task wakes up
3. Regular timer interrupt

Completely Fair Scheduler approach is the process scheduler of the Linux kernel to deal with the regular non-real-time processes implemented by Ingo Molnar and merged in Linux version 2.6.23. The idea behind the CFS is ideally to share the CPU completely equally between all the tasks. In other words, all the active processes are planned to be given the equal amount of the processing time. For instance, if three processes are running on the core, at any given time all of the processes would have been running on the CPU for the equal amount of time. This is where the term fair comes from. With the introduction of task priorities, CFS considers them as well in regulating the proportion of CPU share. Each task has a so-called virtual runtime. Every time a task has run its proportion, its vruntime is updated with the elapsed time since it was scheduled. In CFS the value of vruntime is

crucial in the process of rescheduling. The process with the lowest vruntime is given the CPU share immediately. Splitting the CPU between the tasks, the CFS always try to behave like ideal multitasking hardware as close as possible. That is if there are n processes, each of them should obtain 1/n of processors time. The vruntime (nano-seconds unit) value of the tasks with higher priorities grows slower with the proportional to its priority superiority.

There are two separate priority ranges implemented in the Linux kernel.

1. The standard priority range used in all Unix system is nice value. Its values changes from -20 to +19. Higher priority corresponds to the lower nice value. Consequently, the processes with lower value receive the higher proportion of the core. Default nice value is *0*. Figure 3.5 illustrates how nice levels affect the scheduling.

2. There are also processes that are always prioritized over the normal processes. These are the ones with real-time priority varying from *0* to *99*. Higher real-time priority value corresponds to the greater priority of the process. These priorities enable us to implement the real-time tasks using logic for the scheduling different from CRS. This matter is important for analyzing the real-time behavior of the system and will be elaborated in the further sections.

There are also processes that are always prioritized over the normal processes. These are the processes with real-time priority varying from 0 to 99. Higher real-time priority value corresponds to the higher priority of the process. These priorities enable us to implement the real-time tasks using logic for the scheduling different from CRS. This matter is essential for analyzing the real-time behavior of the system and will be elaborated in the further sections.

Robert Love gives a very intuitive demonstration of the work-flow of processes scheduled using the concept of CFS on the example of two processes [24]: the I/O-bound text editor and processor-bound video encoder. Imagine a system running only these two tasks. The text editor does not consume processor much since it spends the vast majority of its time waiting for the user input. Video encode, in its turn, uses the processor very effectively calculating during the whole time it is given a share. The text editor has to respond to the keypress immediately, whereas user does not care when precisely the video portion he watches was encoded. CFS approach does an excellent job in this types of scenarios, where I/O- and processor-bound processes have to run simultaneously in an efficient way. Rather than giving individual priorities to the tasks that consume processor more, CFS tries to give the same proportion of the CPU to both processes. Since in our example there are only two processes the kernel tries to give each of them 50% of the processors time. The text editor waits for the users key press and is blocked for the most of the time. Thus, the video encoder preempts the CPU. Whenever the user presses the key and wakes the text editor up, CFS observes that instead of the planned proportion of 50% it spent considerably less time on the CPU and rescheduled the text editors process right away. The text editor processes the user input and falls asleep again waiting for

the next user input. That is when the CFS gives the CPU back to the demanding video encoder. This way CFS keeps the processor busy for most of the time running I/O- and processor-bound processes very efficiently.

3.1.2. Runqueue - Red-Black-Tree

As a queue for the tasks, CFS uses a Red-Black-Tree data structure. One of the benefits of this data structure is the fact that there is no path in the from the root to the leaf which is more than twice as long as any other path. A red-black tree is a self-balancing tree with all of its operations in `O(log n)`. With this approach, the addition and deletion of the tasks to the runqueues (nodes to the tree) are very quick and efficient. The tasks are represented by the nodes in the tree and are sorted regarding the task's vruntime value. The leftmost node craves the processor the most since it points to the lowest the vruntime value. The scheduler can access the needed task in `O(1)`. After the task gives the CPU away, its vruntime is updated, and the Red-Black tree balances itself which results in an appropriate task with the lowest time spent on the core landing on the leftmost node. These operations never exceed `O(log n)`. As a result, all of the tasks are getting the chance to become the leftmost child and to get some time on the CPU. This process is described in Figures 3.1 and 3.3

3.1.3. The Linux Scheduling Algorithm

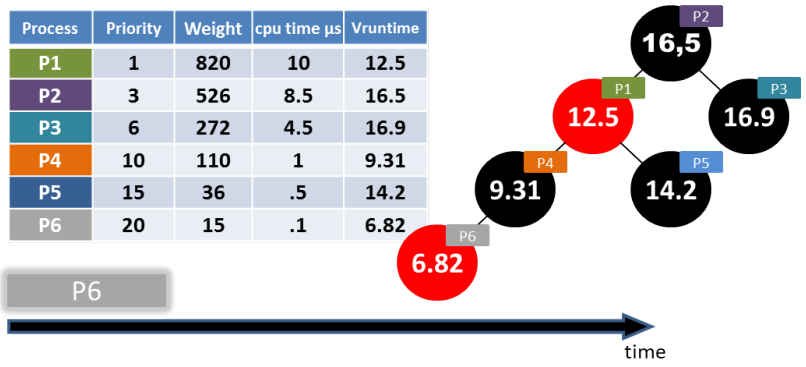
The beauty of the Linux scheduler is the fact that it allows different scheduling strategies for various types of processes. There is a notion of scheduler classes which is used in the implementation of this feature. The Completely Fair Scheduler is one of these scheduler classes, but it is not the only one. According to the `POSIX` standards it is referred to as `SCHED_OTHER`. The behavior of CFR type processes is coded in `kernel/sched_fair.c` [27].

Lets have a look at some important aspects of the implementation of fair scheduling. Whenever a process becomes runnable or blocks, there is a function called that calculates the execution time of the process. This function is called `update_curr()`, it is an important function of CFS and is defined in `sched_fair.c`. It might be invoked by the system timer as well. The main goal of the `update_curr()` function is to update the vruntime of the current process to know which task to run next when the scheduler wants to pick the next process. This function is periodically called to take the time spent since the last update of vruntime while the process was running and add the weighted value (calculated using the nice value and the running time) to its vruntime. The weighted value is calculated as following:

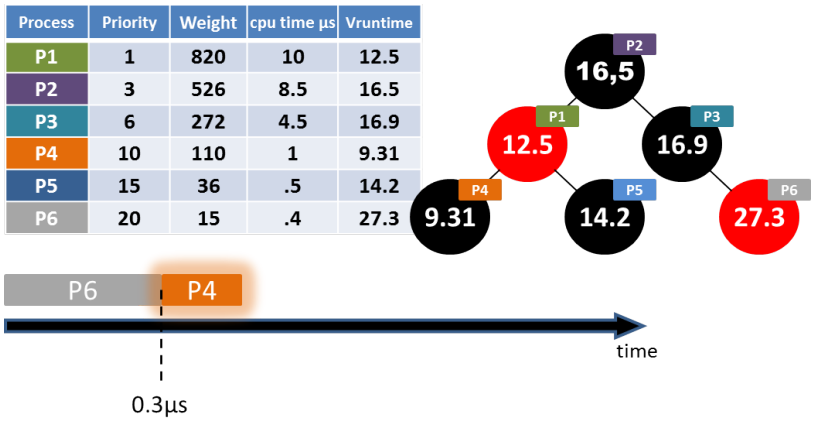
```
delta_exec_weighted = delta_exec * NICE_0_LOAD / se.load
```

- `delta_exec` equals to the time since last update (stored as exec_start)
- `se.load` is calculated according to the nice value of the process.

Figure 3.1.: CFS scheduler workflow. Task is getting scheduled if it is the leftmost node

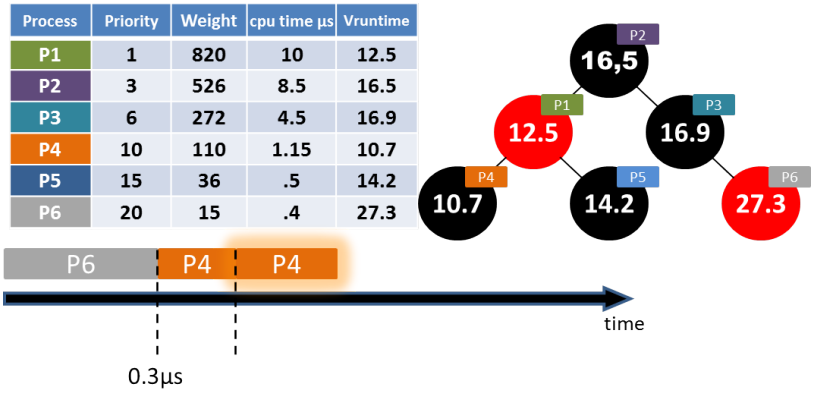


(a) a.Read-Black-Tree runqueue CFS

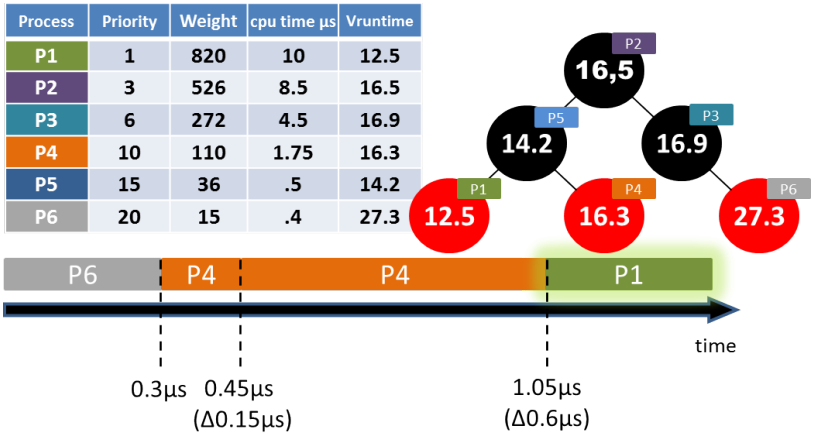


(b) b.Read-Black-Tree runqueue CFS

Figure 3.3.: CFS scheduler workflow. Task is getting scheduled if it is the leftmost node



(a) c.Read-Black-Tree runqueue CFS



(b) d.Read-Black-Tree runqueue CFS

The mapping of the nice values to the weight are defined in the prio_to_weight constant array [36]

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Figure 3.5.: mapping of nice values to weighting factors

Nice levels are multiplicative, with a gentle 10% change for every nice level changed. I.e., when a CPU-bound task goes from nice 0 to nice 1, it will get 10% less CPU time than another CPU-bound task that remained on nice 0. The weight is roughly equivalent to $1024/1.25^{nice}$. The higher the weight of process is, the smaller addition it gets to its vruntime. This allows the process to attain a more left position on run queue of the CLS and get more time on the processor.

The starting time of the process is also getting updated to the current time in order to get the valid delta the next time the function is called again. Calling the `update_curr()` periodically guaranties that the vruntime of the task has a fair value to use on the next rescheduling.

Whenever the scheduler chooses to run the process of this type, the process is scheduled following the rules of CFR. However, these scheduler classes (including CFR) also have their priorities. Any scheduler class which has a task ready to run is a candidate to be chosen by the scheduler. The base scheduler code sitting in `kernel/sched.c` [36] iterates over those classes with runnable processes and pick a class with the higher priority. The process on run queue of that class is then executed according to the algorithm of the scheduling task it belongs to. The following section describes how it is implemented.

To invoke the process scheduler to decide the process to run next and to start that process, the kernel uses the function called `schedule()`, and it is the main scheduler function which is defined in the `kernel/sched.c` [36]. As mentioned above, the mission of this function is to find the scheduler class with the highest priority that has a runnable task ready in its run queue. This function calls `pick_next_task` which iterates through scheduler classes and calls the function (of the same name) of the chosen class directly to return its next runnable process. In fact, in cases when there is only one scheduler class, namely CFS, (which is usually the case), there is no iteration happening.

As a result, only the scheduler class itself is entirely responsible for the logic behind the organization of the processes in its pool. The only thing the kernel scheduler does is picking the appropriate scheduler class. This approach is one of the building blocks making the proper scheduling of the real-time processes possible. Logically, extracting the real-time sensitive processes to a separate scheduler class and giving it higher priorities should do the trick. That is precisely how it is implemented.

There two real-time scheduling policies in Linux:

1. SCHED_FIFO
2. SCHED_RR

The logic that both of these scheduling policies use to arrange the processes is completely different from the one that is used in CFS and discussed above. That is the reason they are implemented in a specific scheduler class, namely `kernel/shed_rt.c`

SCHED_FIFO does not rely on any timing while scheduling the processes. It implements a straightforward logic where the first arrived process with high priority occupies the processor:

- If a task of SCHED_FIFO is in the run queue and has a higher priority, there is no chance for any other task to run until SCHED_FIFO finishes
- Theoretically SCHED_FIFO can run indefinitely if not prevented. It can potentially take up the processor and run until it gives the processor away voluntarily or blocks
- To be able to preempt the task of SCHED_FIFO there should be a runnable task of type SCHED_FIFO or SCHED_RR in the run queue, which has an explicitly higher priority than the current one
- Among two SCHED_FIFO tasks with the same priority the one that arrived later will only get a chance to run when the first one yields the processor
- Obviously, will always be scheduled over the SCHED_OTHER task

In contrast to SCHED_FIFO, SCHED_RR is scheduled for some predefined amount of timeslices. SCHRED_RR tasks are given the default timeslice which hardware specific.
`#define DEF_TIMESLICE (100 * HZ / 1000)` [37]

- HZ is the frequency with which the system's timer hardware is programmed to interrupt the kernel.

In fact it this is mainly the only difference between these two policies. SCHRED_RR is also real-time capable, aggressive towards the lower priority tasks. The processes of this policy have just more dignity to the processes with equal priority.

- SCHED_RR processes run for the predetermined amount of time allowing the rescheduling after that. If that processes has not been finished yet, logically only processes with no lower priority can be scheduled next. As a result, if SCHED_RR process is of the highest priority, it will always get the processor back

- However, unlike SCHED_FIFO, if there are several SCHED_RR processes they will run in round-robin fashion

Since the priorities of real-time tasks and normal tasks are in different ranges, a real-time task will always be preferred over the regular task. Even though these policies are well defined and implemented, the kernel is only considered to fulfill soft real-time requirements. It does not guarantee the consistent absence of any spikes in latencies.

The pure real-time scheduler needs only to know the task with the highest priority to decide on what to execute next. Priorities are static in contrast to the dynamic vruntime values in CFS, which change after every update. That is why storing the tasks in the run queue so that updating and picking the one with the highest priority efficiently is not a big challenge as it was in the fair scheduling. Thus, for the RT scheduling class, the priority-queue data structure is used to store the runnable tasks. Picking the next task can be done in constant time complexity

There is function called `task_tick_rt()` implemented in the kernel/sched_rt.c which is responsible for updating the current timeslice of the running rt processes.This function is called by `scheduler_tick()` defined in the `kernel/sched.c`. `scheduler_tick()` gets executed frequently by the timer code. One of its missions is to call the `task_tick()` function of the running scheduler class to let it know that one cycle is over and enable it to perform the actions according to its policies. For instance, in the case of fair scheduling, this results in updating the current process, which will change the vruntime of the current process and reschedule. As described above, this might well emerge the other task (with lower vruntime value) to take over the processor. In case of real-time scheduler class, the `update_curr_rt()` is also called which updates the statistics of the current task (just like in CFS). The difference is, these changes do not affect the scheduling decisions in this case, since real-time scheduling does not try to achieve equal CPU share. If the policy of the task is `SCHED_FIFO`, the function returns, due to the fact that `SCHED_FIFO` processes do not voluntarily give the processor away also after a CPU cycle. Figure 3.6 visualizes task scheduling with `SCHED_FIFO`.

The only thing left to check is whether the `SCHED_RR` process has run for enough cycles to ask for rescheduling. To achieve this, the current timeslice of the task is decremented. If it does not equal to zero yet the `SCHED_RR` process can keep on running.

Otherwise, the process used all the cycles it was given, and it should be decided which task should run next. Before that, the current task is granted the default timeslice again.

If the task is not the only one in the queue, then it is requeued to the end of it. Otherwise, it can keep on running. Figure 3.7 visualizes task scheduling with `SCHED_RR`.

Process	Priority	Duration	Arrival at
P1	90	8 µs	3rd µs
P2	80	2 µs	2nd µs
P3	70	3 µs	0

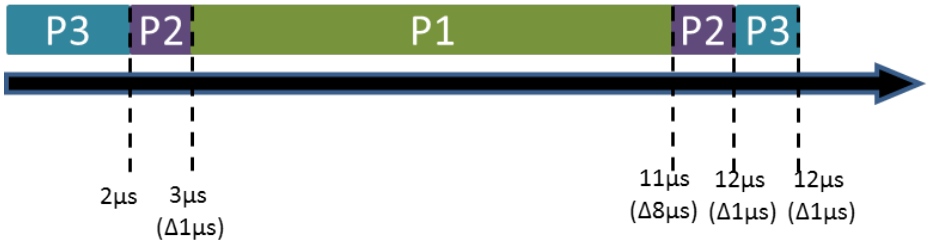


Figure 3.6.: task scheduling with SCHED_FIFO

3.1.4. Real-time group scheduling

The previous chapter provided a detailed view of how the kernel schedules distinct processes according to different scheduling policies. This is very important to understand the general stream of different types of processes and to be able to visualize the workflow of a program running on the kernel. However, the code of scheduler is enormous and contains a bunch of other features which enable us to deal with different real-life challenges occurring while running our processes on the kernel. Lets have a look at some of them:

The logic described above gives enormous privileges to the developer of the real-time application. One important characteristic that is common for both real-time policies is the following: the real-time process will run forever unless the process preempts it with higher priority or it goes to sleep itself. In the case of `SCHED_RR`, it also can be preempted by the process with equal priority. In POSIX standards there is no specific policy defined which would require some allowance for lower priority task to get a fraction of CPU time [15]. Imagine a real-time process that occupies 100% of the CPU and does not give it away. According to the algorithms described above, this behavior will the make normal non-real-time processes starve for the resources. This may well be intentional, but this is not a good idea. First of all, a mistake of the developer can be very costly here and can completely block the system. That would require solutions like artificially injecting some higher priority process to overcome the blocking one. However, even if a developer does everything right wholly monopolizing the CPU with real-time tasks is still something to be avoided, since it blocks the Operating System from performing its housekeeping activities. Eventually, this will lead to a crash of the system. To add some security for the kernel against the blocking FIFO tasks, there is a safeguard mechanism implemented,

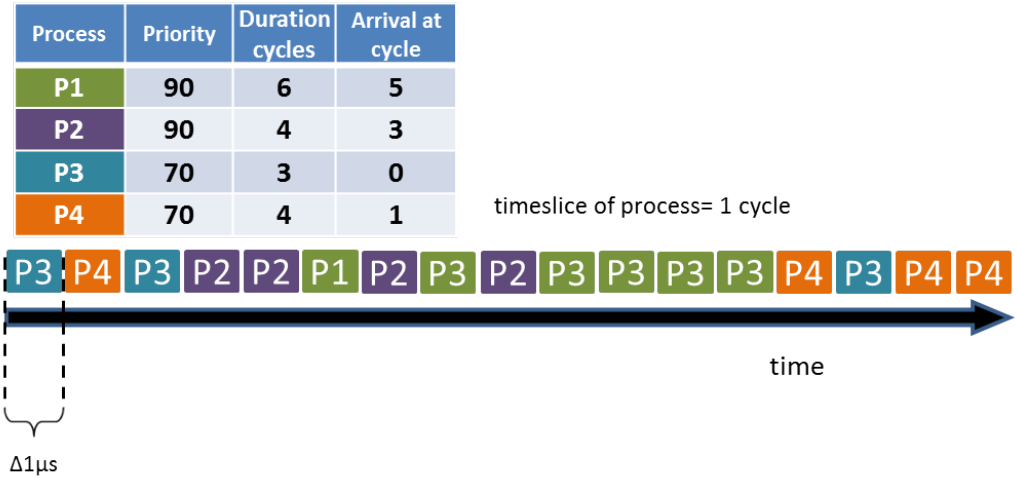


Figure 3.7.: task scheduling with SCHED_RR

which allows adding a specific CPU bandwidth to RT processes. After crossing that limit, RT tasks are forced to get hands off the CPU. This mechanism is called Real-time scheduler throttling and is controlled by two parameters in the `/proc` file system.

- `/proc/sys/kernel/sched_rt_period_us` Defines the period in s (microseconds) to be considered as 100% of CPU bandwidth. The default value is 1,000,000 s (1 second). Changes to the value of the period must be very well thought out as a period too long or too small are equally dangerous. [15]
- `proc/sys/kernel/sched_rt_runtime_us` The total bandwidth available to all real-time tasks. The default values are 950,000 s (0.95 s) or, in other words, 95% of the CPU bandwidth. Setting the value to -1 means that real-time tasks may use up to 100% of CPU times. This is only adequate when the real-time tasks are well engineered and have no obvious caveats such as infinite polling loops. [15]

These values are adjustable, but by default, the proportion is 95% against 5% in favor of real-time tasks. This means that a real-time task can use up to 95% of CPU time but not more. The meaningfulness of the choice of this percentage, however, raised some doubts among the kernel developers and was not appreciated by all of them [6]. Also important to mention that if one real-time process occupies the 95% of the processor, the other real-time processes will not run. This group sharing mechanism described in the next section deals with this issue.

Enabling `CONFIG_RT_GROUP_SCHED` lets you explicitly allocate real CPU bandwidth to task groups. [1] This enables us to control the CPU time dedicated to each control group. This is done by using the cgroup virtual file system and changing the value in `cgroup/cpu.rt_runtime_us"`. Enabling `CONFIG_FAIR_GROUP_SCHED` allows the same

features for SCHED_OTHER tasks. In group scheduling, the kernel first tries to share the resources among the groups and then the received portion is shared among the processes in that group.

Understanding the effects of these kernel features is very crucial in analyzing the behavior of cgroups-based containers runtimes because those technologies rely entirely on these kernel components.

3.1.5. Fair Group Scheduling

Group scheduling is another way to bring fairness to scheduling, particularly in the face of tasks that spawn many other tasks. Consider a server that spawns many tasks to parallelize incoming connections. Instead of all tasks being treated fairly uniformly, CFS introduces groups to account for this behavior. The server process that spawns tasks share their vruntimes across the group (in a hierarchy), while the single task maintains its independent vruntime. In this way, the single task receives roughly the same scheduling time as the group. Let's say that next to the server tasks another user has only one task running on the machine. Without group scheduling, the second user would be maltreated in opposition to the server. With group scheduling, CFS would first try to be fair to the group and then fair to the processes in the group. Therefore, both users would get 50% share of the CPU. Groups can be configured via a /proc interface.

3.1.6. PREEMPT_RT

PREEMPT_RT is a config option that you enable to improve real-time capabilities of the system. The primary goals of this patch are to make a 100% preemptible kernel. That would mean that system can react to any event at any time. 100% preemptible kernel, in theory, is able and will stop any program or process if it has to be interrupted. However, achieving ideal preemption was not possible yet. The first step towards this goal used in PREEMPT_RT is removing the interrupt or other forms of preemption disabling as much as possible. The other step is attaining quick response times by reducing the latencies.

While installing the PREEMPT_RT, the Fully Preemptible Kernel option should be chosen among other preemption models. One of the options there is "Preemptible kernel" which was developed mainly for SMP machines that have critical sections where two or more processes can access (and corrupt the data). The spin-locks protect these critical sections. In the Preemptible Kernel, the idea is to use the spin-locks to represent the places we can not preempt. Thus spin-locks became functional even in uniprocessor systems for disabling preemption. The trick is the preemption is disabled in a single region but enabled anywhere else.

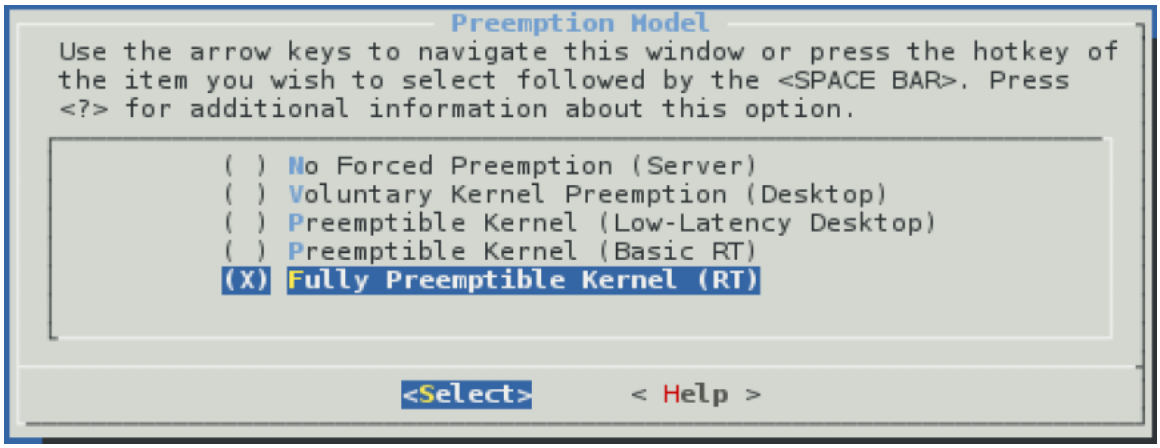


Figure 3.8.: PREEMPT_RT installation procedure [13]

Fully Preemptible Kernel (the RT patch)

This option (which is chosen during the installation) enables the `PREEMPT_RT_FULL` flag. It allows preemption almost everywhere. The spin_locks are converted to mutexes and interrupts - to regular threads that can be scheduled. Sleeping spin_locks, i.e., instead of spinning the thread whenever it faces the critical section, going to sleep or waiting till the lock is released speeds up things tremendously.
The throughput of the system went up tremendously after implementing the feature called PREEMPT_LAZY. It allows the `SCHED_OTHER` tasks to finish up their execution if they hold a lock, even if a real-time task is ready to preempt it.
Priority Inheritance is another feature that helps to lower down the latency by preventing unbounded priority inversion. Priority inversion cannot be prevented entirely, but it is essential to get rid of unbounded priority inversion. Priority inversion happens when a higher priority task is runnable but can not preempt the lower priority task running right now due to whatever reason. The problem occurs when this lower priority process runs forever or does not have any known deadlines. When this happens, the system becomes non-deterministic, which is the first enemy of a real-time system.

Imagine the case of three processes with A having higher priority and C the lowest as shown in Figure 3.9. C is running and then gets preempted by A. Since C holds a lock, A gives the CPU back to it and waits till C is finished. Imagine now process B becomes runnable and preempts C. The problem here is the fact that we have a runnable higher priority task which is unable to run and it is not clear when it will be. Thus the system is not deterministic anymore.

The solution in Figure 3.10 is done by letting the process C inherit the priority of process A., As a result, it was not preempted by B in the meanwhile, and process A (being the higher priority than B) was able to get finished.

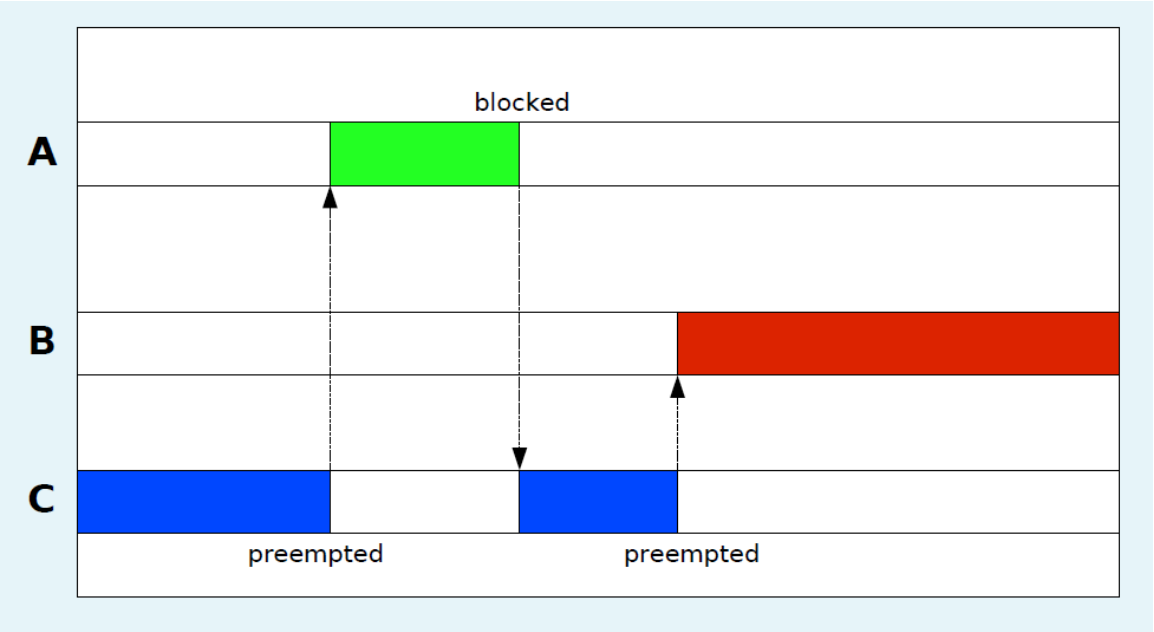


Figure 3.9.: Unbounded Priority Inversion [13]

3.2. Latency tests on Docker containers

This chapter is devoted to the latency experiments on Docker containers running on a machine with real-time extensions. All of the experiments described in this paper are done on HP EliteBook 8750p equipped with Intel i7 4 core processor. The Operation System is Linux based Ubuntu 18.04.1 LTS with kernel version 4.18.7. OS is patched with PREEMPT_RT version 4.18.7_rt5. The Docker version 18.06.1-ce was used.

As a benchmarking tool for a real-time system, we used `cyclictest`. A kernel without good load gives typically low latency results. Thus it makes sense to use one of the stress test programs to keep the kernel busy. During all of the experiments, the stress was applied to the system using a program called `hackbench`. This program creates an enormous number of threads or processes which send data to each other using sockets or pipes [2] Thanks to this stress test throughout all the of the experiments the CPU load on all processors was 100%.

Experiments were done using the following commands:

```
hackbench -l 10000000000000
```

- parameter `-l` indicates how long the program should run

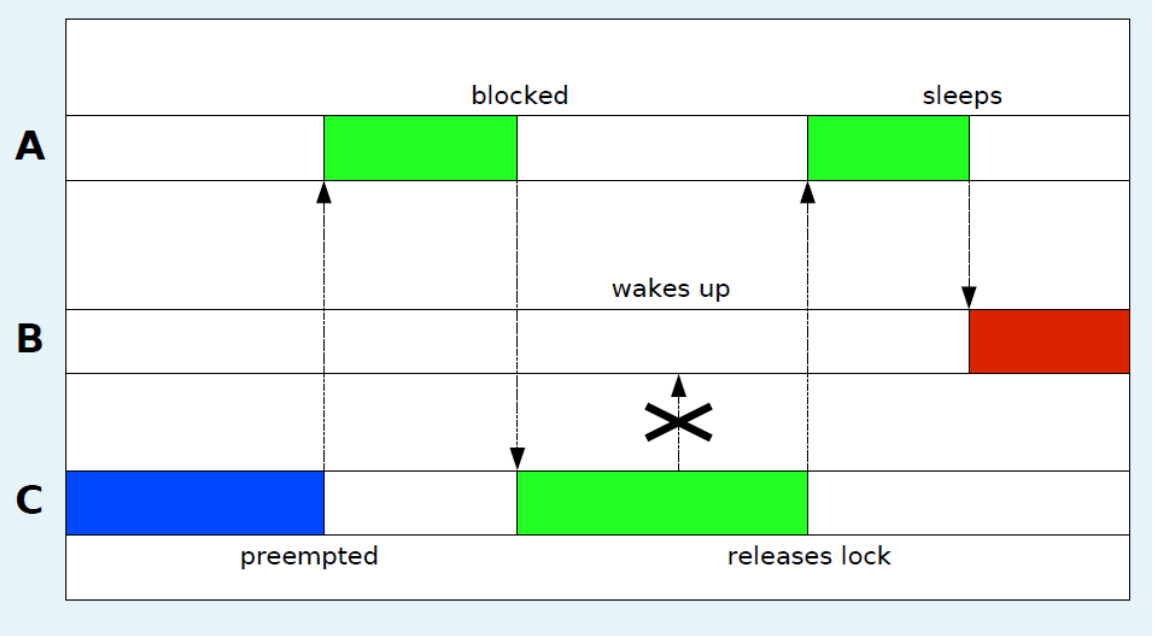


Figure 3.10.: Priority Inheritance [13]

`cyclictest -m -Sp90 -i200 -h400`

- `-m` is used to lock current and future memory allocations to prevent
- `-Sp90` defines the priority a thread strarts executing with
- `-i200` defines intervals between threads
- `-h400` defines how the results are printed out at the end

3.2.1. Experiment on pure machine

It is important to have results that can be used as the best case for comparison during the evaluation process of experiments. For that reason, our first experiment is done without using containers. Cyclictest and hackbench were executed in a patched system for more than six hours. Figure 3.11 is a plot of these latency results. The results were the following:

```
# Min Latencies:  00002 00002 00002 00001
# Avg Latencies:  00003 00003 00003 00003
# Max Latencies:  00035 00040 00037 00036
# Histogram Overflows:  00000 00000 00000 00000
```

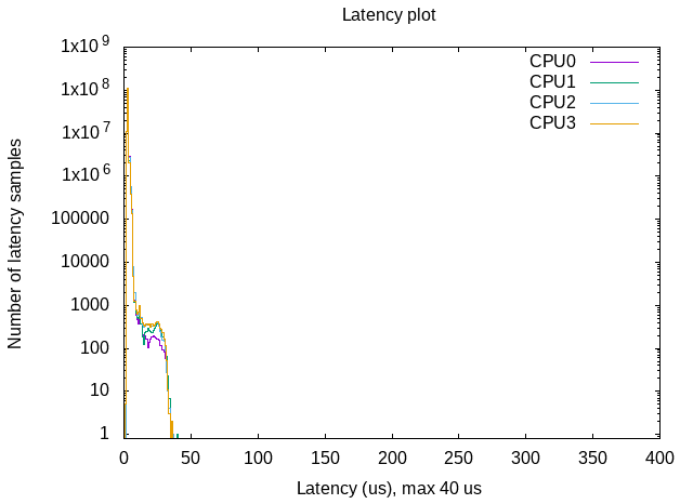


Figure 3.11.: latency results on pure machine

```
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

3.2.2. Experiment with one container running on the machine

Figure 3.12 is a plot of the second experiment which was done by running a container built with the programs mentioned above. This gave us the following results:

```
# Min Latencies:  00005 00003 00003 00003
# Avg Latencies:  00008 00005 00006 00006
# Max Latencies:  00036 00073 00036 00084
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

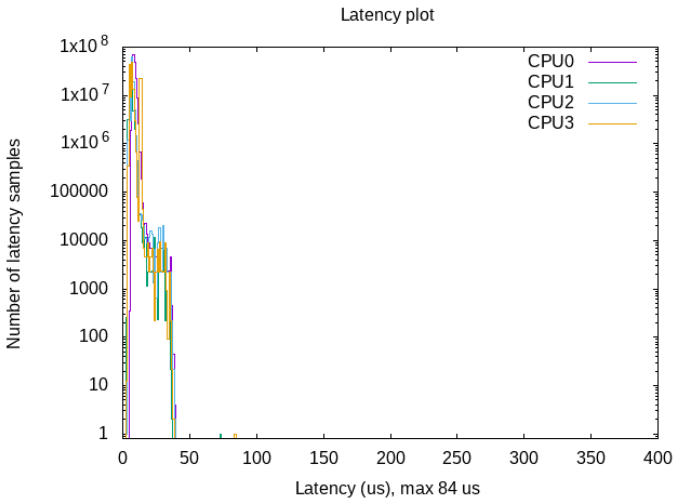


Figure 3.12.: experiment with one container

3.2.3. Running three containers with different priorities

This time three containers were run simultaneously. These containers were built with cyclictest starting with different priorities. The commands are following for each container respectively:

- `cyclictest -m -Sp90 -i200 -h400`
- `cyclictest -m -Sp80 -i200 -h400`
- `cyclictest -m -Sp70 -i200 -h400`

Figure 3.13 illustrates the graphs for each container. The results were following:

- for container running cyclictest with priority 70

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00004 00005
# Max Latencies:  00032 00035 00041 00043
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for container running cyclictest with priority 80

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00004 00003 00003 00004
# Max Latencies:  00040 00034 00032 00043
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for container running cyclictest with priority 90

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00003 00005
# Max Latencies:  00032 00033 00043 00042
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

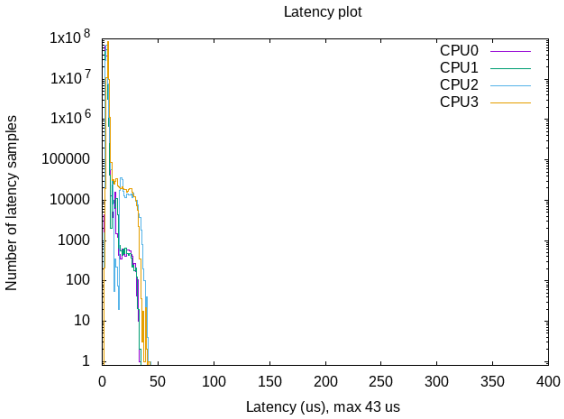
3.2.4. Running three simultaneous containers with different `cpu shares`

Same containers as in the previous experiment are rerunning this time. The only difference is the fact that one of those gets an enormous amount of time on the processor in comparison to the other two. This was done by starting one of the containers with the flag `--cpus=0.5`. See the results below and graphs in Figure 3.15.

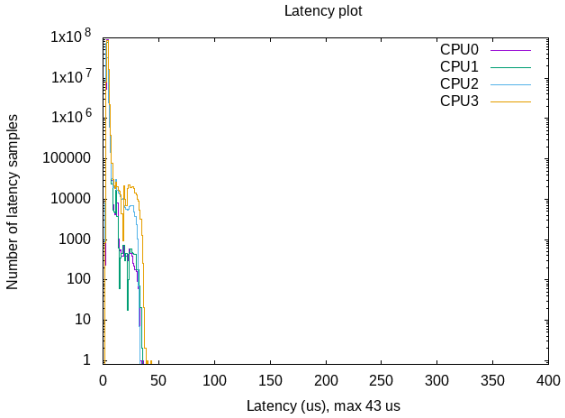
- for container running cyclictest with priority 70

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00004 00004 00004 00004
# Max Latencies:  00040 00042 00039 00039
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
```

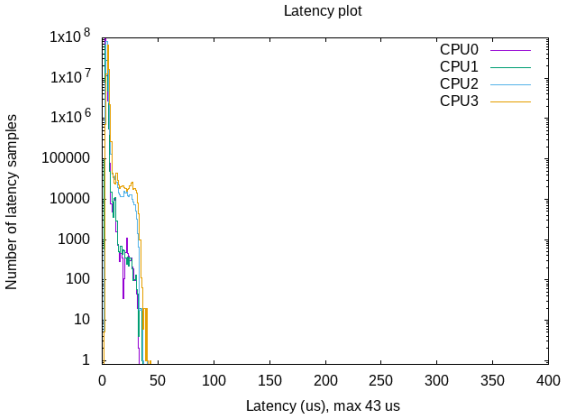
Figure 3.13.: Latency plots for three simultaneous containers



(a) Latency results for priority 90



(b) Latency results for priority 80



(c) Latency results for priority 70

```
# Thread 3:
```

- for container running cyclictest with priority 80

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00005 00003 00003 00003
# Max Latencies:  00039 00034 00037 00043
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for container running cyclictest with priority 90 and big `cpu share`

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00005 00003 00004 00003
# Max Latencies:  00037 00036 00035 00038
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

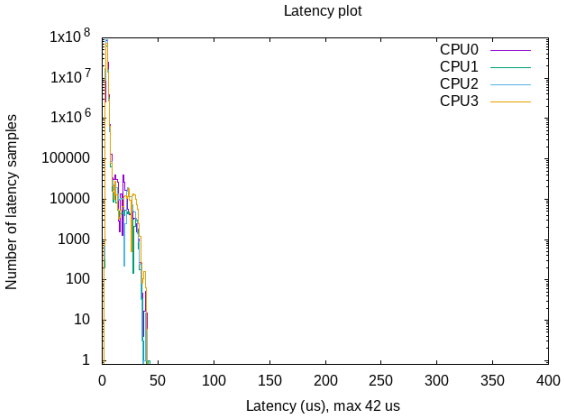
3.2.5. Running six simultaneous containers with cpu limits

In this experiment, the number of containers doubled. Two containers built with cyclictest running the threads with priority 70. Two - with 80. Two - with 90. The other difference this time was that they were started using flag `--cpus=0.5`, which limited the time each container could have on the processor. See the results below. Figure 3.17 illustrates the graphs for each container.

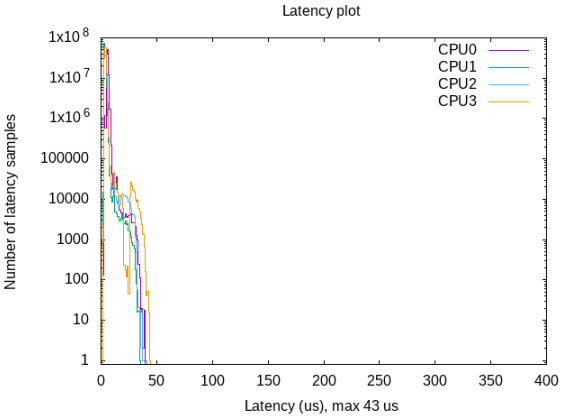
- for the first container running cyclictest with priority 70

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00009 00003 00003
# Max Latencies:  00035 00046 00043 00035
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
```

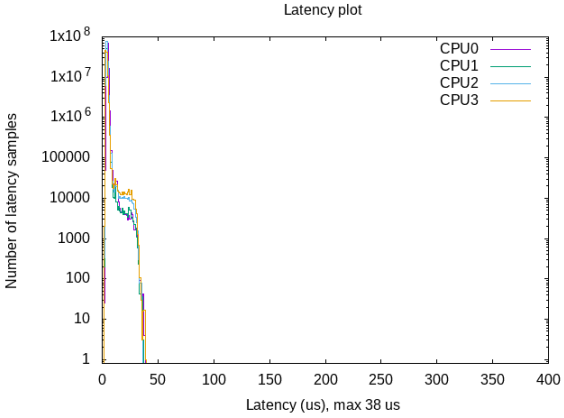
Figure 3.15.: Latency plots for three simultaneous containers with differen `cpu shares`



(a) Latency results for priority 70



(b) Latency results for priority 80



(c) Latency results for priority 90 and highest `cpu share`

```
# Thread 1:
# Thread 2:
# Thread 3:
```

- for the second container running cyclictest with priority 70

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00003 00003
# Max Latencies:  00035 00041 00036 00040
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for the first container running cyclictest with priority 80

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00003 00003
# Max Latencies:  00034 00041 00038 00039
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for the second container running cyclictest with priority 80

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00004 00004
# Max Latencies:  00039 00037 00047 00039
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for the first container running cyclictest with priority 90

```
# Min Latencies:  00002 00002 00002 00002
```

```
# Avg Latencies:  00003 00003 00003 00003
# Max Latencies:  00034 00041 00039 00041
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

- for the second container running cyclictest with priority 90

```
# Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00003 00003 00003 00003
# Max Latencies:  00037 00034 00040 00036
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

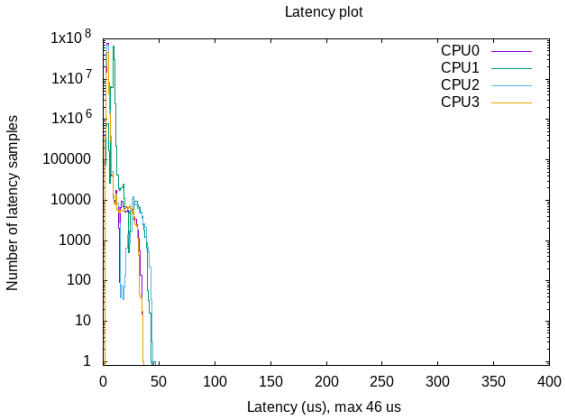
3.3. Docker under the hood

To be able to read and explain the latency results we got during experiments with cyclictest and hackbench conducted on Docker containers running on Real-time compatible operating system, we should first dig into the docker anatomy and be capable to explain in detail how containers work, how they are organized and which effects on the system are foreseeable. Only after having a clear understanding of how docker containers work, we can give sensible and more precise clarifications to the measurement results.

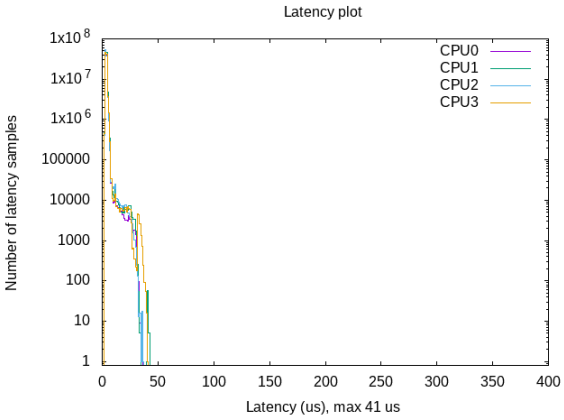
3.3.1. What is a container?

There is a misleading approach of considering the containers as some lightweight virtual machines. Although, for understanding the use cases of containers this way of thinking might give a good foundation. On the high level, containers might feel and behave like virtual machines. We can have our processes and services inside; we can install and use packages, we can get root privileges, own process space, and network interface and so on. Besides, there are the means of accessing the containers just like in virtual machines. One can ssh into it, as well as run a shell in the container [31].

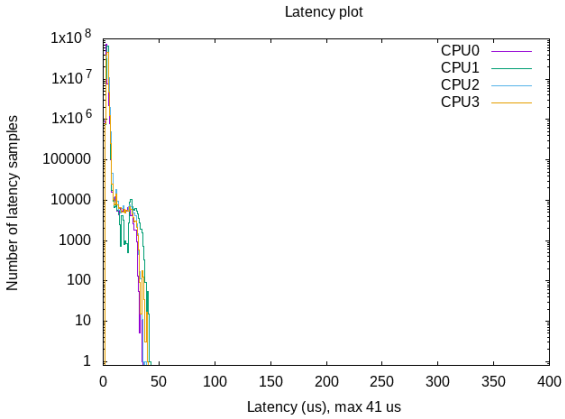
Figure 3.17.: Six simultaneous containers with cpu limits



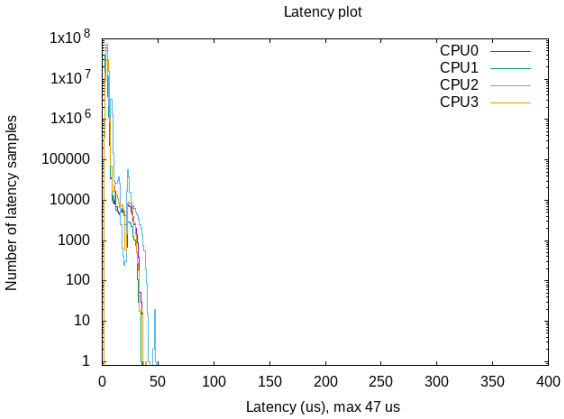
(a) the first container with priority 70



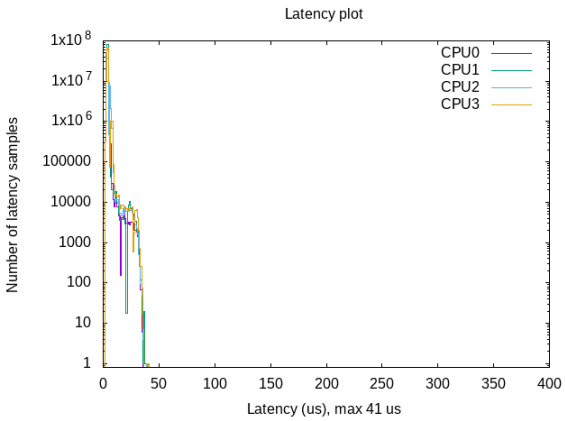
(b) the second container with priority 70



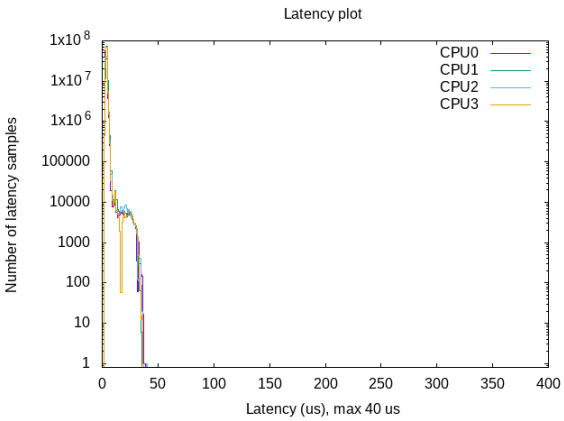
(c) the first container with priority 80



(d) the second container with priority 80



(e) the first container with priority 90



(f) the second container with priority 90

On the other hand, if we take into account the low-level approach, there are significant differences between the virtual machines in the way how containers work. If we scale out the of the container, the processes running inside of it are just a set of normal processes running collectively with other processes on the kernel of the host OS. Unlike VMs, if we take a normal Operating System running several containers and print out the processes running on it, we can see the processes running on all containers. These processes think that they run in their environment whereas outside of their container they run as regular processes alongside the processes of the host or even other containers. Containers use the kernel of the their host and cannot boot a different Operating System or have their modules. These facts make them more transparent in comparison with virtual machines.

3.3.2. Implementation details

The main building blocks of containers are control groups and namespaces. There usually is no evidence of container itself in the kernel. The information about the containers that you might be able to retrieve from the kernel is precisely those cgroups and namespaces. Lets dig into what they are and how they participate in the development of the containers. All containers running on a host ultimately share the same kernel and resources. In fact, Docker containers are not a first-class concept in Linux, but simply a group of processes that belong to a combination of Linux namespaces and control groups (cgroups) [26].

Namespaces Unlike the cgroups, which limit the quantity of resource we can use, namespaces limits what we can see. The idea behind it is to make the containers feel that they host their system and make the systems running in parallel inaccessible to them. Here as well each process is exactly in one namespace of each kind.

Below are some examples of namespaces that are used in container runtimes.

Pid namespace. Pid namespace allows the process to see only the processes in its own PID namespace. Each PID namespace has a process with a process id one inside that PID namespace. However, that processes will be mapped to a different PID in the PID namespace of the higher level.

Network namespace lets each container have its network resources (localhost, eht0, routing tables, iptables, etc.). It is also possible to move one network interface from one network namespace and use it in the other. Another use case could be creating a virtual bridge between the host and containers.

Mount namespaces provide isolation of the list of mount points seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies. It is also possible for processors to have private mounts like temporary files [21].

Control groups

One of the technologies that Docker Engine on Linux relies on control groups (cgroups). A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, you can limit the memory available to a specific container [7]. Besides, other system resources, such as CPU, disk, and network bandwidth can be restricted by these cgroups, providing mechanisms for resource isolation. [26] Just like with a group of processes, when we have a bunch of containers running, we want to prevent any of them from occupying all the system resources. Cgroups is a Linux kernel feature which allows processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on) [19].

Cgroups are used in the implementation of limitations of resources that processes can use. They allow us to measure and limit memory usage, CPU or I/O resources, etc. dedicated to a group of processes. With cgroups each subsystem (CPU, memory, etc.) has its hierarchy. These hierarchies are organized in trees with PIDs as leaf nodes. Every process (the leaf nodes) running in the system is present in every tree and is a child of a specific node which represents the cgroup the process belongs to on that specific subsystem. These specific nodes are a group of processes which share the same resource.

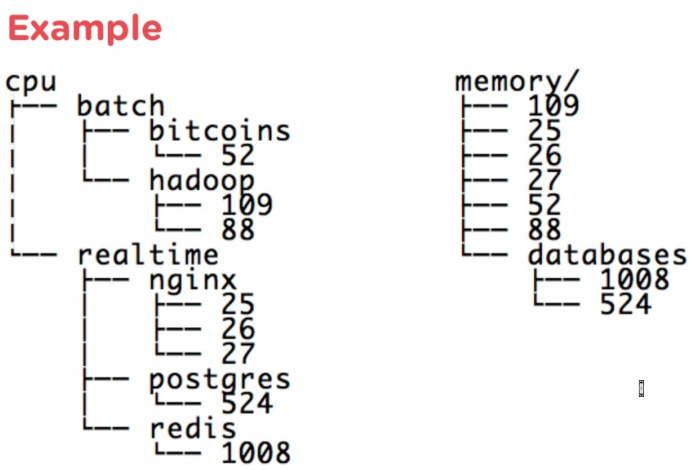


Figure 3.19.: hierarchy trees of resource type subsystems [31]

In the Figure 3.19 [31], we can observe hierarchy trees for two resource-type subsystems and the groups of processes under their relevant cgroups. The process groups under the root node belong to the cgroups with no specified limits for that particular subsystem.

During the booting, there is only one node, and the first process is a child of that node. The whole system can be considered as a container with no specified resource limits, even when there are no containers created. If there is a process which is not assigned to any cgroup, it automatically gets included to the root cgroup. Thus expecting a faster system just by eliminating the usage of containers does not make sense, since, roughly speaking, even the first processes of the system are already running in the container. One can argue, that if any process in the system has namespaces, resource limits and security settings, then it runs in a separate container. Since cgroups are one of the Linux kernel features, every user with sufficient privileges can create cgroups using shell commands, change those cgroups and migrate processes between them.

Types of cgroups:

Memory cgroups: The memory controller isolates the memory behavior of a group of tasks from the rest of the system [20]. For instance, it allows creating a cgroup with a limited amount of memory. The other use-case could be to separate the applications which require much memory from the other applications. Lets have a look at some possibilities of memory cgroups in more detail.

One of the features of memory cgroups is accounting. The smallest unit of memory, in this case, is a memory page. This feature allows us to know how much memory is used by the specific group of running processes and be aware of every memory page involved.

There are two types of memory pages:

- File pages
- Anonymous pages

File pages have their replication in the particular location on the disk on which they are loaded from. The advantage of file pages is the fact that in critical situations there is no fear to lose important data by being forced to eliminate those pages. One can be sure that they are still located on the disk and can be restored at any point. In contrary, anonymous pages are not located on the disk (coming from sources like heaps, stacks, etc.). To be reclaimed afterward, these types of pages should be swapped out beforehand. [31] The pages which are actively used by containers are put in the set of active pages. The ones that after some time are not getting accessed for a long time are marked as inactive and become the first candidates for eviction. This approach is useful when a group of processes starts running out of memory. The system knows which pages are safer to get rid of. Once the page is touched it returns to the active set. Each page corresponds to a particular memory control group. In other words, it is visible to only one group, and only that group pays the costs of having it.

Another kernel feature of cgroups that Docker (and other container runtimes) heavily relies on, is memory limiting. The system allows applying a memory ceiling for the pro-

cesses in one memory group. It is possible to apply those limits to different types of memory, such as kernel memory, physical memory, swap memory. There are two categories of memory limits:

- Soft limits
- Hard limits

Soft limits are not strict. Going above them without any severe consequences is possible. Soft limits are useful in the scenario when the machine is not entirely running out of memory but is already under some stress. In this case, the kernel iterates through the group of processes, and the more the group exceeded its soft limit, the more likely it is to get some memory pages evicted. If a group of processes exceeds its hard limit, then a process from this group gets killed until the rules preserved again. One additional problem of going beyond the hard limit is the fact that is might be the other process of the group that did not necessarily cause the issue that got killed. Thus, in the world of containers, if one container exceeds the hard memory limit, any of the processes running inside of it can be killed by the kernel. That is also one of the reasons, why it is recommended to run one microservice per container. This will establish granularity and make sure that the service will not get down because of another service running contiguously [31].

Memory control groups are vital for us because we want to make sure that we can set manage memory wisely to allow real-time containers with specific memory requirements to be scheduled appropriately to same or different nodes

CPU cgroups: The CPU controller is responsible for grouping tasks together that will be viewed by the scheduler as a single unit. The CFS scheduler will first divide CPU time equally between all entities at the same level and then proceed by doing the same in the next level.

CPU cgroups allow us to check the CPU usage with respect to the whole cgroup, rather than only one process. This relates to container scenarios well and is an excellent enhancement due to the fact that the CPU cgroup allows tracking of many processes simultaneously, which usually is not so easy to do. As the CPU is so central to a computer, it is not surprising that several cgroup subsystems relate to it. The cpuset subsystem that limits physical CPUs a process can run on, and the cpuacct subsystem, which accounts for how much time is spent on the CPU by processes in a cgroup. The third and last CPU-related subsystem is simply called CPU. It is used to control how the scheduler shares CPU time among different processes and different cgroups [4]. Since the way the kernel schedules the processes on the CPU is extremely important for the real-time performance, the behavior of this particular cgroup is one of the essential research questions of this thesis. Thus, we concentrated more on how scheduler is implemented, to which extend the real-time requirements can be maintained and which support does this cgroup subsystem provide to manage the CPU resources for real-time processes. For non-realtime tasks, Linux kernel uses the completely fair scheduler (CFS). The CFS algorithm tries to implement an ideal

multi-tasking processor. The source code can be found [27]. The main idea is to try to share equally the CPU among all the tasks in the system. The concept of Virtual Runtime was introduced to be able to track the time the process occupies the CPU and to make it fair among the other processes. This concept is also used to implement the ceiling enforcement by CPU cgroups in CFS. `cpu.cfs_period_us` specifies a period in microseconds (s, represented here as "us") for how regularly a cgroup's access to CPU resources should be reallocated. `Cpu.cfs_quota_us` specifies the total amount of time in microseconds (s, represented here as "us") for which all tasks in a cgroup can run during one period (as defined by `cpu.cfs_period_us`).

In addition to CFS, the Linux scheduler also supports real-time (RT) scheduling. RT tasks have higher priorities than CFS task and cannot be evicted by the tasks with lower priority. The RT scheduler works similarly to the ceiling enforcement control of the CFS but limits CPU access to real-time tasks only. The amount of time for which real-time task can access the CPU is decided by allocating a run time and a period for each cgroup. The corresponding RT Tunable Parameters are `cpu.rt_period_us` and `cpu.rt_runtime_us`.

As a result, Linux kernel supports CPU access management not only for real-time scheduling processes in separate but also for disjoint groups of real-time processes. We concentrate mostly on this feature of the cgroups to be able to explain better the results of the measurements which predominantly stressed the scheduling capabilities of the real-time kernel.

Apart from ones mentioned above, there are a lot of other control groups (Blkio, Net_cls, net_prio, devices cgroup, freezer cgroup, etc.) that make it possible to run Docker containers on the Linux machine and manage its resources and features.

Obviously, docker also operates with other technologies like overlay data structure, copy-on-write resource management technique, etc. to improve the performance and enhance the range of features it provides. However, the mission of this chapter is to indicate that while running, a container exploits only pure kernel services which otherwise run on the system as well. In a nutshell, Docker is a command-line tool for programmatically defining the contents of a Linux container in code, which can then be versioned, reproduced, shared, and modified easily just as if it were the source code to a program [3]. As mentioned in previous sections, measurements were conducted parallel to heavy load on the system using the hackbench stress test running non-rt processes. Assuming the facts mentioned above, it would not make sense to expect some (or even any) latency inflation.

3.4. Latency tests with Kubernetes

One basic experiment was also done on the Kubernetes cluster. This experiment is similar to the one with one container running `cyclictest` and `hackbench`. This time this container was scheduled by the kubernetes master. The plot is illustrated in figure 3.20

```
  # Min Latencies:  00002 00002 00002 00002
# Avg Latencies:  00004 00004 00003 00004
# Max Latencies:  00039 00022 00039 0045
```

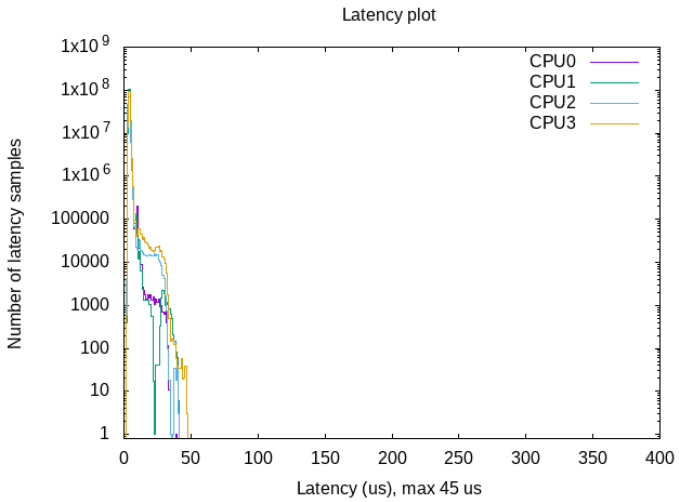


Figure 3.20.: experiment with one container on Kubernetes cluster

```
# Histogram Overflows:  00000 00000 00000 00000
# Histogram Overflow at cycle number:
# Thread 0:
# Thread 1:
# Thread 2:
# Thread 3:
```

3.5. Evaluation of the results

In this chapter we will try to understand the measurement results gotten during experiments conducted with Docker containers and Kubernetes cluster described in the section above.

We described in detail the behavior of the kernel and the organization of processor sharing between the tasks in cgroups. We understood that Docker does not do anything different from what the kernel on its own does. The final state of the tasks in containers before they start running can be replicated by using only native kernel commands. In the scenarios we observe Docker, with tools it provides, makes it easier to group those tasks the way they are grouped, calling native commands under the hood. In the process of running the containers, Docker does absolutely nothing, and the processes are running as they would run if they were coordinated manually.
Let's dive deeper into each experiment scenario. We will try to evaluate the results analyzing the CPU cgroup of the kernel during each experiment.

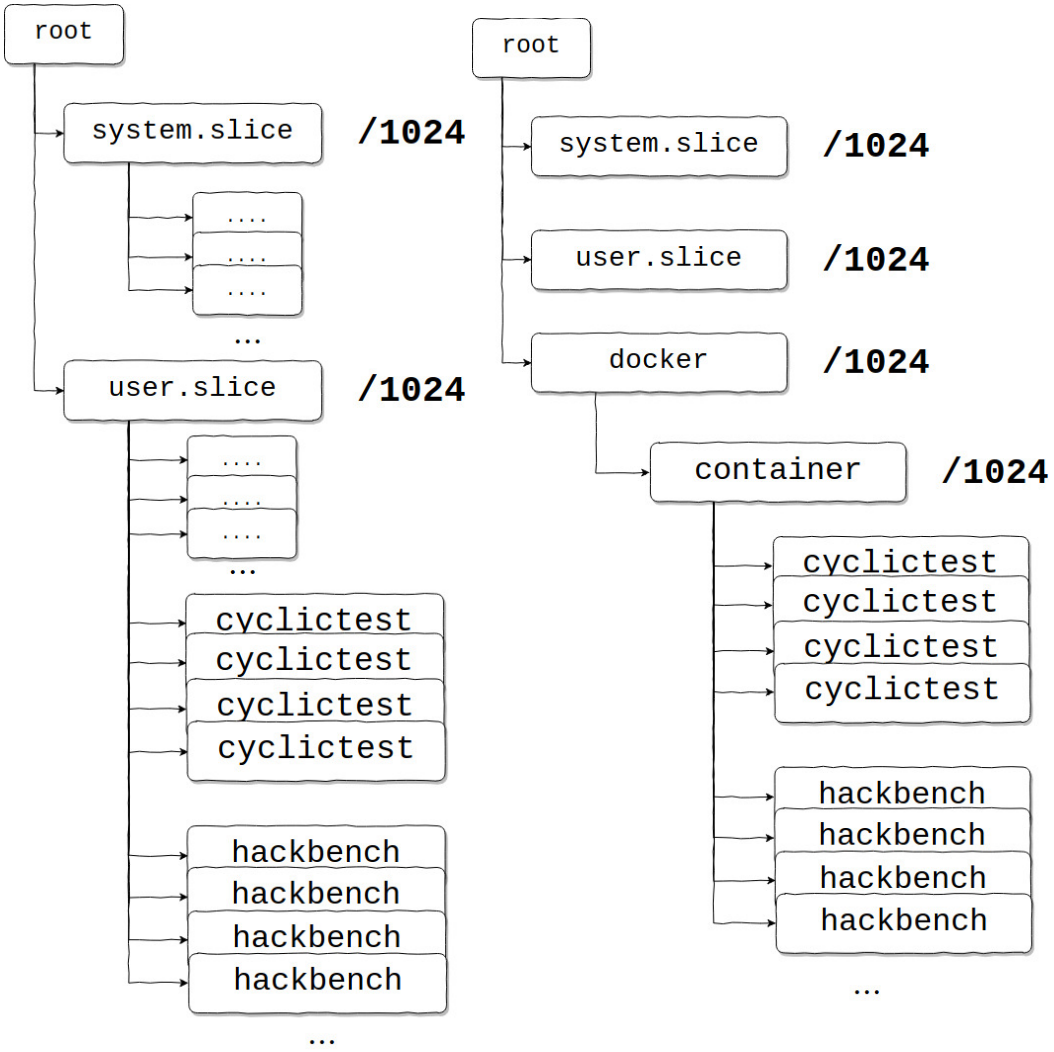
3.5.1. Evaluating the results of the experiment without container

First, we look at the case when the `cyclictest` and `hackbench` were running on a pure machine without any containers. It is visualized on Figure 3.20 (a). CPU is mainly shared between two groups of tasks: `system.slice` and `user.slice`. `1024` is a value of the `cpu.shares`. The fact that those are equal means that each group of tasks is guaranteed to get half of the CPU if needed. Inside those groups, the tasks are managed by the conventional scheduler algorithm. `Cpu.shares` are not the hard limits. If a task group does not use its share, it can be used by another group. Furthermore, there is no upper limit set for a group. If other groups are not starving any group can use the amount of CPU it wants (up to the whole CPU). In our first scenario, we see that real-time tasks are sharing the CPU with all other tasks inside the `user.slice` group. Since those are real-time tasks inside that group, they will, obviously, dominate and take as much CPU as they need.

3.5.2. Evaluating the results of the experiment with one container

Figure 3.20 (b) describes the structure of CPU cgroups of the kernel while running one container with `cyclictest` and `hackbench` inside. We can observe here that in addition to `system.slice` and `user.slice`, there is one more cgroup added. The docker cgroup is also given a share of `1024`, which means the pool of tasks running inside of it is guaranteed to get a third of the CPU if needed. The difference to the case with no containers is the fact that now containers do not share the CPU resources with other tasks. There is one slight difference in the behavior of these two cases. In the first case, if we load the user.slice group like we did (with `hackbench`), this will not disturb the real-time tasks anyhow. The real-time tasks run how much they need, and the rest of the CPU is overtaken by `hackbench`. If we increase the number non-real-time tasks with constant real-time tasks, the potion on non-real-time tasks on CPU does not increase since RT threads get scheduled instead. However, in the case of a container running with RT tasks, there is a way to increase the portion of non-real-time tasks on the CPU. If we increase the load of the tasks in `user.slice group` (by running `hackbench` there), it will affect on their load in CPU, due to the fact that the group is guaranteed to get a third of CPU if it wants to. Still, our `docker` group is still able the get a third of the CPU. Moreover, the scenario described above is not something that we would want to allow in a seamless computing approach. Our goal would be to run all the user tasks inside the containers.
There is no other difference in the real-time threads run in these two experiments, especially no any difference that could cause some visible divergence of the latencies.

Figure 3.21.: Cgroups on the kernel while running one instance of cyclictest



(a) without containers

(b) in a container

3.5.3. Evaluating the results of the experiment with three simultaneously running containers

The CPU cgroup hierarchy is depicted in Figure 3.22 (a). The fact that `docker` cgroup has `cpu.shares` equal to `1024` means that all of the containers combined are guaranteed to get at least the third of the CPU. Since we are intentionally not running any heavy tasks in the `user.slice` the tasks in the `docker` group will overtake the whole CPU. Inside this group, three more groups are corresponding to each container. Since containers started running without any flags, their `cpu.shares` are equal to `1024`, which means each of them is guaranteed to have the third of what `docker` group can have. Thus each of those groups will be able to get at least 1/9 of the whole CPU. Since in this scenario docker grasps the whole CPU, each of the containers loads 33% of it. What is interesting and important to observe here is the fact that the tasks started with priority 90, will not be able to interrupt the tasks with priority 70. This is important because we do not want a real-time application pushed by one user to the system be preferred over a real-time application of the other user just because it was developed setting higher priorities to the tasks. If this is the desired behavior, this should be done by the administrator of the system, by assigning more CPU share to one of the containers and should not be done on the developer side. Next experiment describes this scenario.

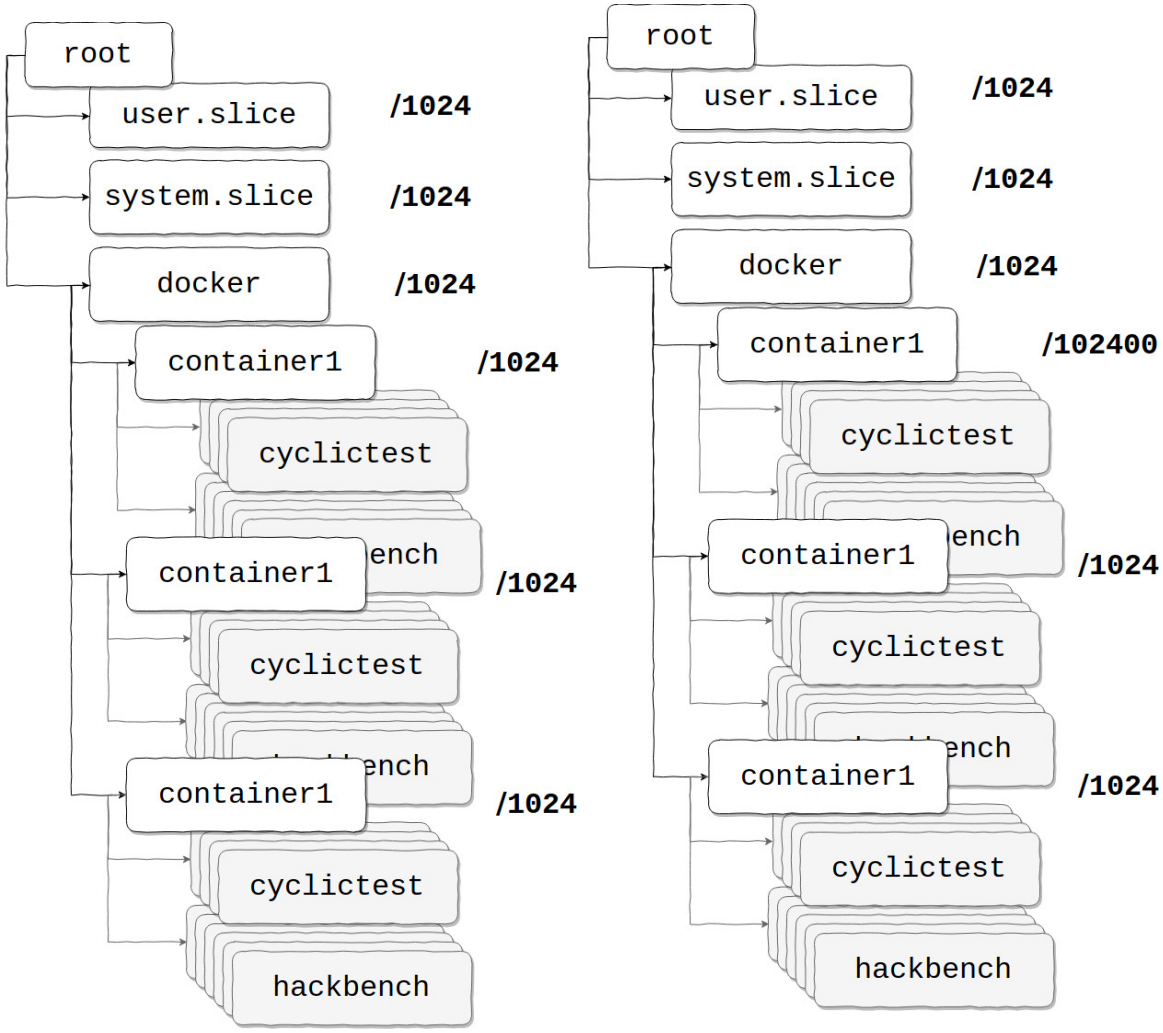
3.5.4. Evaluating the results of the experiment with three simultaneously running containers with different `cpu.shares`

. Figure 3.22 (b) shows how cgroups are in this case. Three containers run again with threads of different priorities, but this time one of them consumes around 100 times more CPU than the other two. `Docker` group takes almost the whole, and the vast majority of it is given to the first container. Note that this was done solely by manually changing the CPU shares while starting the container and is not contributed by different values of priorities at all.
Interesting to notice that containers left with less that one percent of CPU were still capable of preserving the real-time capabilities and produced good latency results. The percentage of the load caused by `hackbench` reduced dramatically which allowed the `cyclictest` threads to accomplished the regular amount of tasks.

3.5.5. Evaluating the results of the experiment with six simultaneously running containers with limited access to CPU

. In this experiment, six containers were run simultaneously having limited CPU access. Every container was started with the flag `--cpus = 0.5`. This means that combined time the tasks of one container spend on CPU every 100 microseconds should not exceed 50 microseconds. Having six containers means that combined they will use three CPUs. Due to the fact that the experiments were done on a four core machine, this will ensure

Figure 3.23.: Three containers running simultaneously



(a) same portion of cpu resources for each container

(b) one container overtakes almost all cpu resources

at 25 percent of CPU resources left untouched by the tasks `docker` group. Default `cpu shares` (1024) were given to the containers in this experiment, which means that the CPU cgroups hierarchy looks the same as in Figure 3.22 (a), but with six container groups. This experiment also resulted in good real-time behavior. The reduced time on the CPU was compensated by reducing the time given for the threads of `hackbench`.

4. Orchestration of Real-time containers

4.1. Kubernetes scheduling for RT applications

Real-time applications are susceptible to the environment they are running in. After showing experimentally that Docker and Kubernetes are not harming the real-time capabilities of the system, it is now important to investigate to which extent can Kubernetes functionality provide a safe residence for real-time critical pods on the real-time enabled node. In this chapter, we will talk about different scenarios that can happen during running and scheduling process of the real-time applications using Kuberentes.

4.1.1. Architecture of Orchestration layer

First of all, let's describe the Architecture of the system which was designed to estimate the capabilities of the Kubernetes orchestration engine in fulfilling the real-time requirements. Our Kubernetes cluster consists of 3 nodes: 1 master node and two minion nodes. One of the minion nodes has a `PREEMPT_RT` patch installed on it; thus, it is a real-time compatible node. This node will be used to shelter the real-time applications. The other minion node does not have any real-time capabilities and can be used only for non-real-time applications. The master node is not going to be used for any containers created by a user. Its primary purpose is to control the scheduling and flow of applications and provide the best possible service for the real-time critical applications.

The master node and the non-real-time node are installed on two separate EC2 instances provided by Amazon Web Services. Due to the fact, that for the real-time node we need we need a specific patched kernel, it is installed on one of the computers in the office which has `PREEMPT_RT` installed.

To make the whole Kubernetes cluster functional, these nodes have to be in the same network. To achieve that, in the cloud master node we use OpenVPN Access Server 2.5. After installation we get an access to the administrator user interface in the master node through

In that admin panel we go to the `User Permissions` in `User Management` and configure the VPN clients. Those clients are two minion nodes: one representing the RT node, the other representing non-RT node which is just like the master node also an EC2 instance. As a result, as depicted in the Figure, using the VPN tunneling those three nodes (2 in AWS, 1 in the office) are connected to the same network. This allows us to connect the minion nodes to the master. The architecture of the Orchestration level is visualized in Figure 3.12.

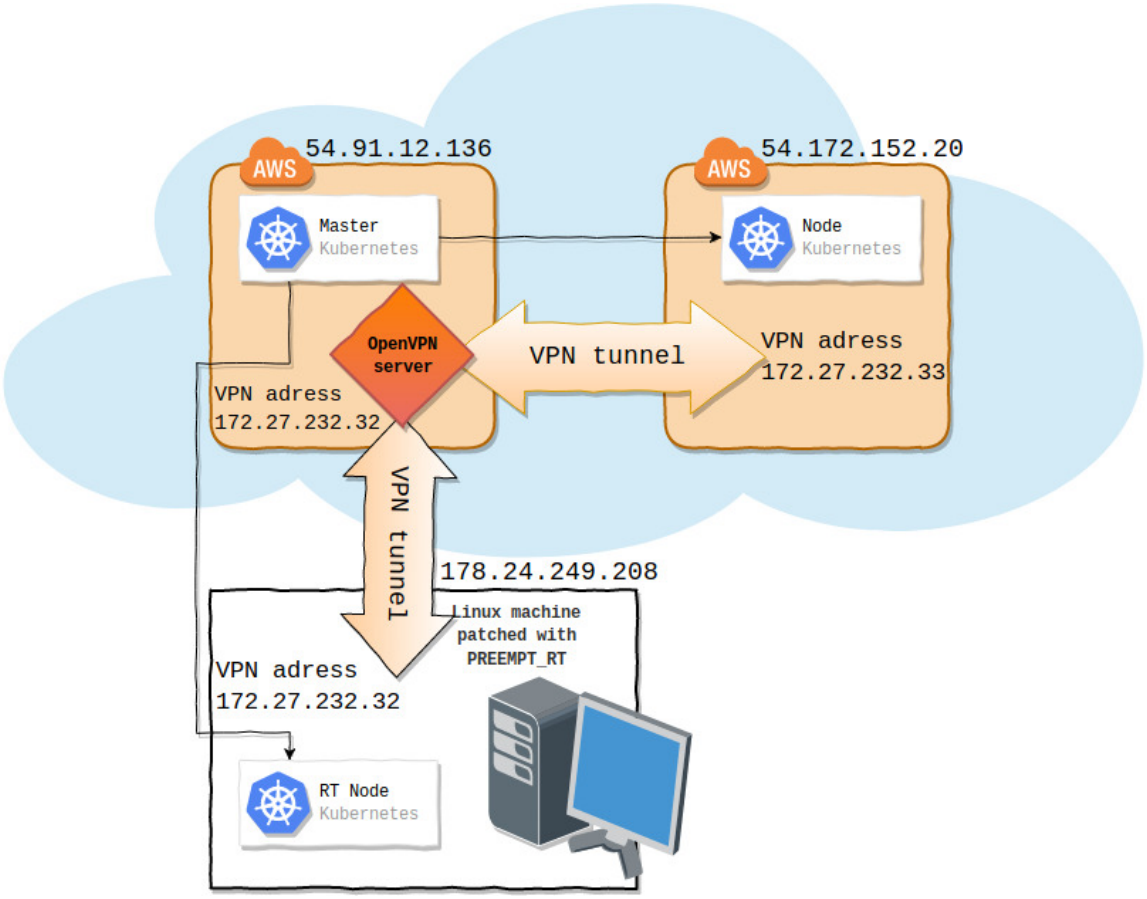


Figure 4.1.: Architecture of Orchestration level

4.1.2. Real-time application scheduling scenarios

In this section, we will discuss the possible scenarios that might occur during scheduling and running of real-time applications on the Kuberentes clusters and present the solutions based on features provided by Kubernetes to respect the real-time requirements as much as possible. One thing and is preserved in any scenario is the following rule: **RT containers can only run on real-time a capable node.** There is not a single case where we can allow the real-time application to run on a machine with no real-time extensions. If it is not possible to schedule the real-time application to such a node, then we prefer to let it wait in the *PENDING* mode until its scheduling to an RT node is possible. This is done using `node labels` on the node and `nodeSelector` on the RT. When we configure a `nodeSelector` on the pod, we force the master to schedule the pod to the node which has that key-value pair of labels.

First, we have to label the node with something like *rt: "true"*. We could do that at runtime from master node using a `kubectl` command. However, in the seamless computing scenario, we presume that thousand of edge devices can register to the master node at any time. We do want the admin to babysit every node and to decide on his own whether the registered node has RT capabilities or not. We want the node to bring this information with it while joining the cluster. We are using a tool called `kubeadm` to create a single master cluster and to register the nodes. To join the master node `kubeadm join` command is used which in its turn uses the systemd service `kubelet`, which is a primary "node agent" and runs on each node. Inside the `/systemd/system/kubelet.service.d` there is config file called `10-kubeadm.conf` which is used by `kubeadm` while calling the `kubelet` service. Inside that file there is the following line:

```
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS
$KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS
```

This line indicates which parameter the `kubelet` service should start with. `$KUBELET_EXTRA_ARGS` is sourced from the file located in `/etc/default/kubelet`. To make the node service start with `nodeSelector`, the following line is added to the file:

```
KUBELET_EXTRA_ARGS==--node-labels=rt=true
```

With this setup, from the master node after the node joined the cluster we can observe in its `Labels` the key-value pair the was assigned by the node itself. The only thing left now is to start up the RT pods with the right `nodeSelectors` and the scheduler will do the rest for us. In the configuration deployment file of the pod, we add the following lines:

```
spec:
  nodeSelector:
    rt:"true"
```

Now that whenever we create a pod from that configuration it only gets scheduled to a node with the label `rt: "true"`, in other words to an RT compatible node. Figure 3.13 illustrates that.

From this point the following terms will be used in corresponding cases:

- **real-time node**: a real-time compatible node with the label `rt:"true"`
- **real-time pod**: a pod which runs critical RT task and has a `nodeSelector rt:"true"`
- **non-real-time pod**: any other pods

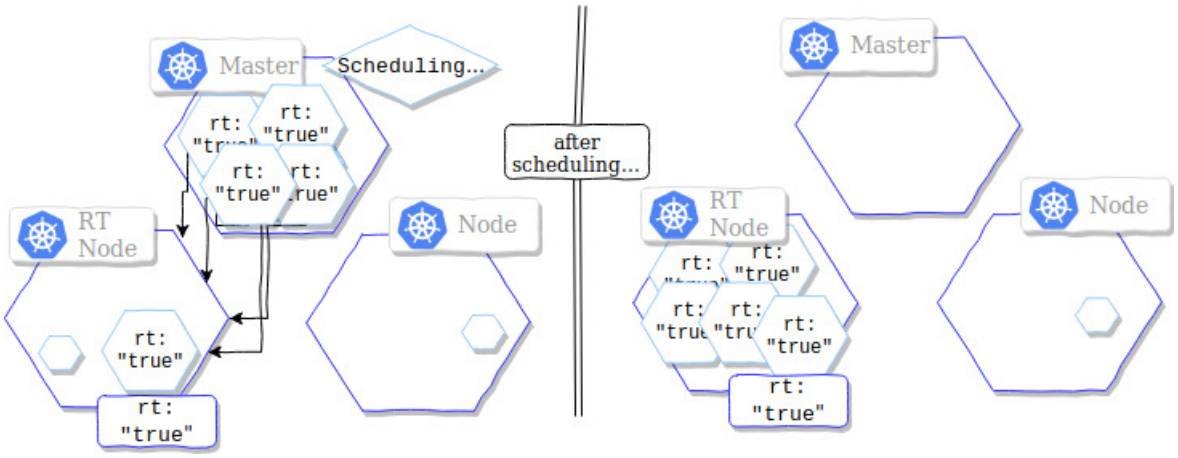


Figure 4.2.: RT pods are scheduled to RT nodes only

Scenario 1: no non-real-time pods on an RT node

Imagine the case when we do not have many RT nodes, and we do know that real-time applications will be coming. In that (or any other suitable) case, it might be reasonable not to let the non-real-time pods be scheduled on the RT nodes. This can save us from additional scheduling when a lot of real-time applications arrive. To make it possible on the nodes we use the concept called `taints`. Just like in case of `labels` we add `taints` to an RT node, and a pod that wants to run their should `tolerate` those taints. This time the nodes get picky and choose itself which pods can run on it. The `/etc/default/kubelet` is now updated to

```
  KUBELET_EXTRA_ARGS==--node-labels=rt=true
--register-with-taints="rt=true:NoSchedule"
```

We create a new deployment configuration for this scenario. In the `spec` field in addition to the previous configuration, we add the following lines:

```
spec:
  tolerations:
  - key:"rt"
    operator:"Equal"
    value:"true"
    effect:"NoSchedule"
```

This configuration will have to be used for the real-time pods otherwise they will not be

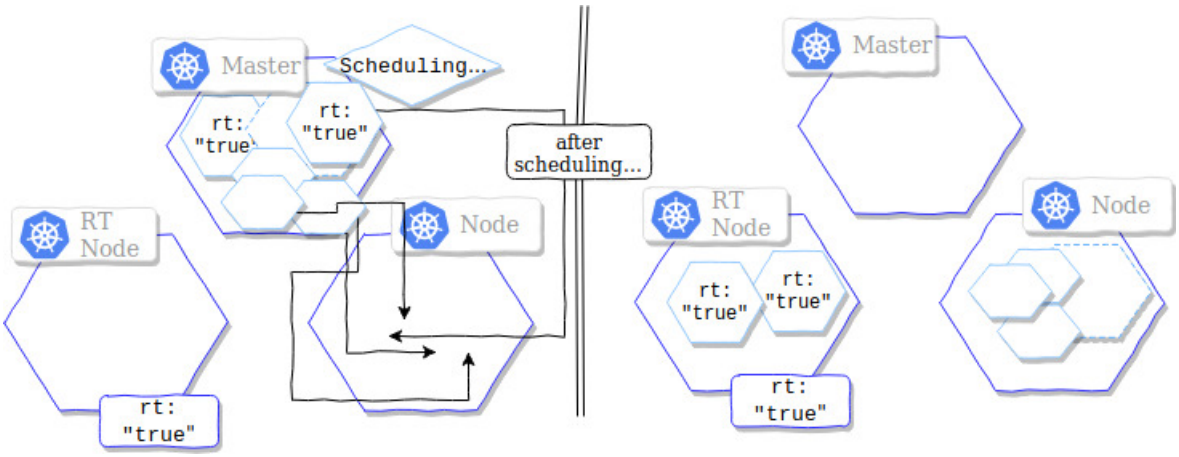


Figure 4.3.: Non-RT pods are not allowed to RT node

able to be scheduled to the RT node. Since the still have the `nodeSelector` field they will be in the *PENDING* state. However, if configured correctly they will be scheduled to RT nodes free from any non-rt pods. A workflow sample is illustrated on the Figure 3.14

Scenario 2: non-real-time pods are allowed until a real-time pod arrives

The more frequent case would be when we allow both RT and non-RT applications run in the RT node. In this scenario, we look at the case when non-real-time pods are allowed to run on RT node but have to free it up when RT pod is getting scheduled. Here, we use two other functionalities of Kuberenetes: `podAntiAffinity` and `podPriority`.

- `podAntiAffinity` helps to separate the pods that we do not want to run together. The problem is it helps the scheduler to make decisions during the scheduling process. There is no way the `podAntiAffinity` can cause any rescheduling on its alone after the pod is already scheduled. Thus, in case the non-RT pod is already scheduled to the RT node and the RT pod arrives, which has a `podAntiAffinity` to the running pod, this pod will not be scheduled in the RT node even in case it has a `nodeSelector` and cannot be scheduled to any other node. Thus, `podAntiAffinity` alone can not be used in this scenario.
- `podPriority` indicates the importance of one pod over the others. Pod with higher priority can cause the scheduler to start the rescheduling process, and the scheduler will try to achieve a setup where the least amount higher priority pods are in *PENDING* state.

First, we create a priority configuration class for RT pods. We set the priority to *1000000*. The priority class looks like the following:

```
  apiVersion:  scheduling.k8s.io/v1alpha1
kind:  PriorityClass
metadata:
    name:  rt-priority
value:  1000000
globalDefault:  false
description:  "This priority class should be used for Real-time thirsty
pods only."
```

Then we create a new pod deployment configuration for this scenario. To be able to make use of `podAntiAffinity` we set a label for each real-time pod.

```
  template:
   metadata:
      labels:
        app:  cyclic-rt
        rt:  "true"
```

In the `spec` field we specify the `podAntiAffinity` bahavior.

```
affinity:
   podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key:  rt
          operator:  NotIn
          values:
          - "true"
        topologyKey:  "kubernetes.io/hostname"
```

Here we say to the scheduler that pods created with this deployment cannot be in the same node with the pods where the `label rt` does not equal to `true`. Adding the priority mentioned above class to this configuration will make scheduler to break ties in favor of this pods.

```
  priorityClassName:  rt-priority
```

Non-real-time pods can get assigned to an RT node. However, whenever an RT pod arrives, non-real-time pods get terminated and scheduled to other nodes and then the RT pod gets scheduled to the RT node. Priorities of all real-time pods have to be the same.

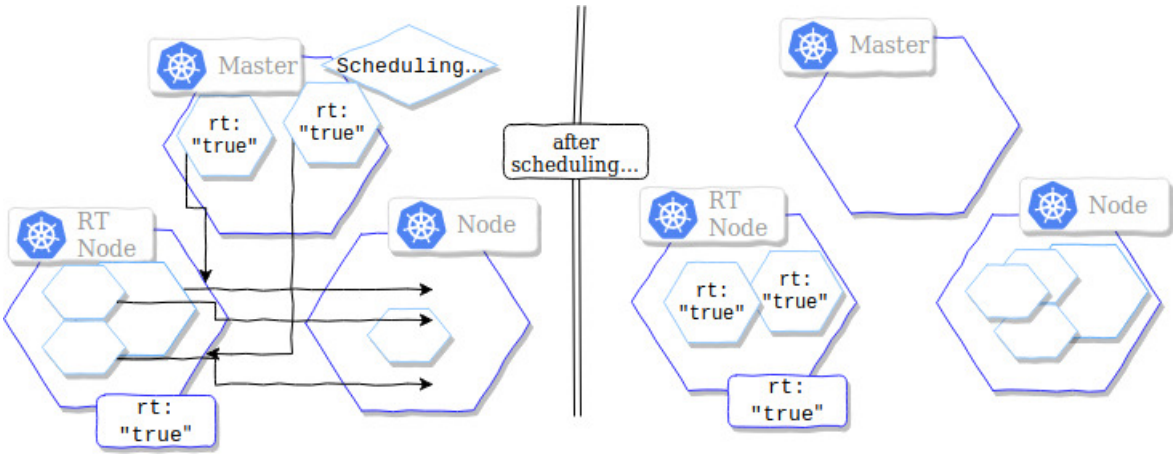


Figure 4.4.: Non-RT pods are moved out of RT node

If a real-time task is scheduled, there should be no way it can be taken out from its node, even if a better arrangement is possible. That would be deadly for critical applications. Thus, making considering all possible scenarios apriori is very important and can be very helpful. Moving real-time critical pods from node to node in the middle of the processing is not the desired approach. This scenario is illustrated in the Figure 3.15.

Scenario 3: non-real-time pods are allowed if there are enough resources on the node

If there is a possibility for a non-real-time pod to run on a real-time node, we usually would not want it to be in the *PENDING* state. If at the moment there is no RT task and a real-time node has an unused resource, it would be undesirable to waste that processing power. Thus, the scenario described in this subsection is the most applicable one if there are not any special requirements.
Until there are enough resources, all the tasks (including the normal once) can run on an RT node. At the moment, when an RT task cannot be scheduled and waits in *PENDING* state, the scheduler will evict a non-real-time pod from the RT node and try to schedule it somewhere else, to enable the RT task to run there. It is also important to mention, that eviction will only happen in case the pending RT application will be able to run in the freed space. If after the eviction of non-RT pods the resources are still not enough to run the desired task, then the eviction will not happen.

This concept can also be implemented using pod priorities. First of all, we specify default recourse requests and limits for the pods. We want each pod to have its specified resource limits. If the user did not specify those limits on his own, the default limit range would be applied on the pod created by him. To make this happen, we create a configura-

tion of type `LimitRange`.

```
apiVersion:  v1:
kind:  LimitRange:
metadata:
    name:  mem-limit-range
spec:
    limits:
    - default:
        cpu:  500m
        memory:  512Mi
      defaultRequest:
        cpu:  250m
        memory:  256Mi
      type:  Container
```

On joining the cluster, a node shares with the master node the information regarding its CPU and memory capacities. Given that all our pods have resource limits and requests applied, the master node will be able to decide beforehand whether to schedule the pod or not. If the amount of resources left on all nodes is less than required by the pod, it will stay pending.
In given scenario, we want to evict non-real-time pods from the cluster if there is a pending real-time pod evicting operation will enable it to run. If we have everything set, the logic of pod priorities from the previous scenario is still applicable here. Starting real-time pods with selected and static priority will help the scheduler to try to free some space up for the real-time pod on a real-time node if it is feasible. Figure 3.16 can give a better idea of the concept.

The user who wants to run a real-time task on a cluster would not normally have too much choice, due to fact that there is a higher chance of not having to much real-time nodes on a cluster, considering the fact that it is not that easy to get one using cloud services. If the requests he sets for his pod are too high his pod might be rejected by the scheduler and stay in the pending mode. To let the user be more flexible, it would make more sense to provide him information about the current resource situation on the real-time nodes. `Kubectl` does not have a command to directly call this information from the node, that is why the implementation of a solution for it required to apply some tricks using the existing commands. The `bash` code is the following:

```
kubectl get nodes --no-headers -l rt=true
| awk 'print $1' | xargs -I  sh -c
"echo NODES; echo ; kubectl describe node
| grep Allocated -A 5
```

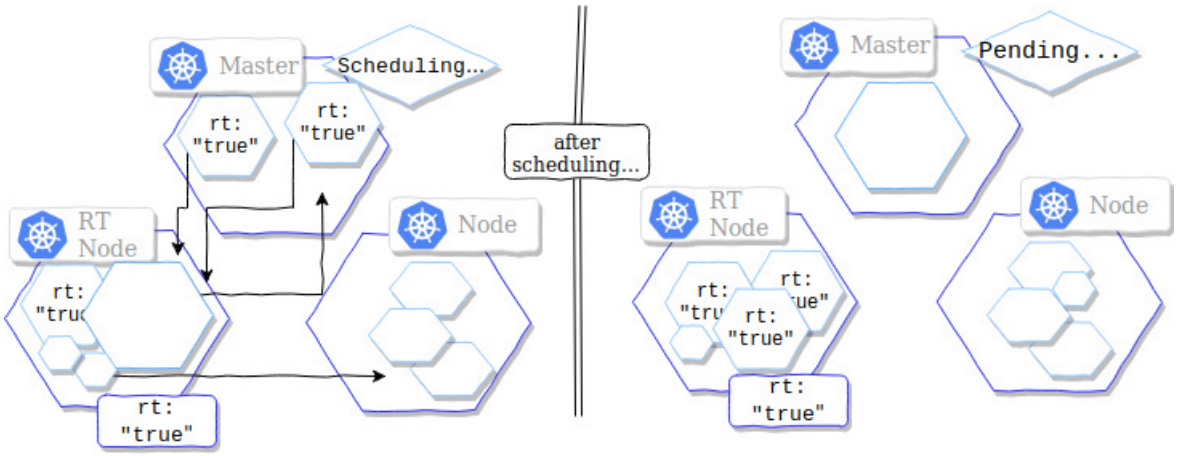


Figure 4.5.: Scheduling according resource availability

```
| grep -ve Event -ve Allocated -ve percent -ve -- ;
echo; echo PODS of ;
kubectl get pods --field-selector spec.nodeName= --no-headers
| awk 'print $1' | xargs -I  sh -c 'echo ;
kubectl describe pod  | grep rt=true;
kubectl describe pod  | grep Limits -A 2;
kubectl describe pod  | grep Request -A 2;'echo "
```

First we look for the nodes labelled with `rt: "true"`, next we print out the resource information for these nodes. Apart from that, we also print out every pod running on that node with its `cpu` and `memory` requests and limits. Also for each pod the information about its real-time needs is also provided.

```
NODE
fariz-celsius-w410
     Resource Requests Limits
     cpu 350m (8%) 600m (15%)
     memory 114Mi (3%) 178Mi (4%)

PODS of fariz-celsius-w410
cyclic-deployment-rt-resources-2-7485d949bd-jdhn6

     Node-Selectors:  rt=true
      Limits:
```

```
        cpu:  500m
        memory:  128Mi
    Requests:
         cpu:  250m
         memory:  64Mi

NODE
fariz-hp-elitebook-8570p
    Resource Requests Limits
    cpu 100m (2%) 100m (2%)
    memory 50Mi (0%) 50Mi (0%)

PODS of fariz-hp-elitebook-8570p
rttest-test-697989b7df-c728x

NODE
ip-172-31-21-193
    Resource Requests Limits
    cpu 850m (85%) 100m (10%)
    memory 190Mi (10%) 390Mi (20%)

    PODS of ip-172-31-21-193
    No resources found.
```

Having all of this information and being aware of scheduling approach used, the user can with certainty determine beforehand whether the pod he plans to run on the cluster will be scheduled or not, and which resource settings he needs to make his pod through. If there is not enough space in one node, the user can use two different nodes by splitting his task into two pieces. Furthermore, if a real-time node has no resources left, however, it is busy with non-real-time pods, and the accumulated resource usage of those pods exceed the needs of user's real-time pod, the user can be sure that those pods will be evicted in favor of his real-time pod. Thus, analyzing the pods of RT nodes in the output as mentioned earlier can save much time and help to find a better approach for the user needs of a user.

Part IV.

Results and Conclusion

5. Summary and Future Work

This chapter briefly outlines the approaches used in finding answers to the questions stated at the beginning of the work, as well as the outcome of the effort and future plans.

The major conclusion that this paper enables us to make is the fact that Docker containers can be peacefully used for running real-time applications in them. The in-depth investigation of Linux kernel scheduler and the ideas used in the implementation of Docker containers enabled us to state that Docker containers can imitate almost any scheduling path occurring in the pure kernel to run a pool of tasks. If we know for sure how the processes are running in the system without containers, we can claim that we are able to achieve the same scheme using containers. This is possible thanks to the implementation approach of this container runtime, which entirely relies on pure kernel features like cgroups and namespaces.
Future plans include using different benchmarking tools and more complicated real-time applications to see how deep can we go stressing out containers for their real-time capabilities.

Another aspect of the research was to suggest solutions for making Kubernetes scheduler aware of real-time tasks. We came up with scenarios where real-time tasks needed to be handled differently and tried to make the scheduler behave appropriately according to the need of the real-time pods. For implementing these scenarios it was not enough to make the scheduler aware of real-time containers. The capability to recognize a real-time node is of the same importance. We came up with a solution where a node can acknowledge the master about his RT capabilities while joining to the cluster. We actively used the features of Kubernetes like `taints`,`tolerations`,`nodeAffinity`,`podAntiAffinity`, `podPriority`, to help the scheduler to pick the best node for real-time pods as well as for non-real-time pods. We also implemented some quick fixes for the users of the cluster to make it possible for them to see the state of the system before they want to push their application. A wise choice made by the user can also help the scheduler in fulfilling the needs the real-time tasks

Much work is planned to on this area afterward. First of all, it is important to consider security issues. Presence of real-time applications can potentially make the security issues more challenging and oversights more costly. It is also planned to make scheduler capable of placing a real-time pod to the most appropriate real-time node among other real-time nodes. For example, there can be cases where a particular real-time software is tested to

be able to run on only specific systems. Using the features like `taints` and `toleratios` it is possible to make real-time node prefer specific real-time pods to other real-time pods. To be able to automize this process, some standardization of the features that specific software might on the node to preserve its real-time capabilities is required. It should know beforehand, which data the node should bring in while joining the cluster and what can be used by the user while pushing his application to the cluster. In addition to implemented scenarios, it is also planned to think of more advanced real-time scenarios and to check the competence of Kuberentes to comprehend those.

Bibliography

[1] Real-time group scheduling. https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt.

[2] Real-time kernel. .https://wiki.archlinux.org/index.php/Realtime-kernel.

[3] What are linux containers? https://opensource.com/resources/what-are-linux-containers.

[4] Neil Brown. Control groups, part 4: On accounting. https://lwn.net/Articles/606004/, 2014.

[5] Bjorn B. Cerqueira, Felipe ; Brandenburg. A comparison of scheduling latency in linux, preempt rt, and litmus.

[6] Jonathan Corbet. Sched fifo and realtime throttling. https://lwn.net/Articles/296419/, 2008.

[7] Docker. Docker overview. https://docs.docker.com/engine/docker-overview/.

[8] The Linux Foundation. Cyclictest. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start.

[9] L. Fu and R. Schwebel. Real-time linux wiki. rt preempt howto. https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO.

[10] Google. Kubernetes — production-grade container orchestration. https://kubernetes.io/.

[11] Roman Gumzej. Real-time systems' quality of service, 2010.

[12] M. Halang W.A.; Gumzej, R.; Colnari. Measuring the performance of real time systems. Real-Time Systems (2000), 2000.

[13] Darren V Hart. Inside the rt patch. https://elinux.org/images/b/ba/Elc2013Rostedt.pdf.

[14] Read Hat. Finding realtime linux kernel latencies. http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt.

[15] Red Hat. Real time throttling. https://access.redhat.com/documentation/en-us/redhatenterpriselinuxforrealtime/7/html/tuningguide/realtimethrottling.

[16] Nat; Hillary. Measuring performance for real-time systems. Freescale Semiconductor, 2005.

[17] Solomon Hykes. Docker - build, ship, and run any app, anywhere. https://www.docker.com/.

[18] Karsten Holm; Schultz Ulrik Pagh; Srensen Anders Stengaard Lange, Anders Blaabjerg; Andersen. Hartos - a hardware implemented rtos for hard real-time applications. Conference: Programmable Devices and Embedded Systems, 2012.

[19] Linux. cgroups - linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[20] Linux. Memory resource controller. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[21] Linux. mount-namespaces - overview of linux mount namespaces. http://man7.org/linux/man-pages/man7/mount-namespaces.7.html.

[22] M Lisper, B.; Santos. Model identification for wcet analysis. 15th IEEE Real-Time and Embedded Technology and Applications symposium, 2009.

[23] Express Logic. Measuring real - time performance of an rtos.

[24] Robert Love. Linux kernel development. https://www.amazon.com/Linux-Kernel-Development-Robert-Love/dp/B009NG8VTO, 2010.

[25] Gu Z.; Guan N.; Deng Q.; G. Yu Lv, M.;. Performance comparison of techniques on static path analysis of wcet. EEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008.

[26] Kevin Lynch. Understanding linux container scheduling. https://engineering.squarespace.com/blog/2017/understanding-linux-container-scheduling, 2018.

[27] Ingo Molnar. Completely fair scheduling (cfs) class. https://github.com/spotify/linux/blob/master/kernel/schedfair.c, 2007.

[28] Spyridon V. Mueller, Harald ; Gogouvitis. Seamless computing for industrial systems spanning. The 2017 International Conference on High Performance Computing Simulation (HPCS 2017), 2017.

[29] McKenney. P. A realtime preemption overview. http://lwn.net/Articles/146861/.

[30] Daniel; Matos Pedro; Machado Rui; Pinto Sandro; Gomes Tiago Manuel Ribeiro; Silva Vtor; Qaralleh Esam; Cardoso Nuno; Cardoso Paulo Pereira, Jorge; Oliveira. Hardware-assisted real-time operating system deployed on fpga. CAlg - Artigos em revistas internacionais/Papers in international journals, 2014.

[31] Jrme Petazzoni. Cgroups, namespaces, and beyond: what are containers made from? DockerCon Europe 2015, 2015.

[32] S. M. Petters. How much worst case is needed in wcet estimation. 2nd International Workshop on Worst Case Execution Time Analysis, 2002.

[33] Frank Rowand. Using and understanding the real-time cyclictest benchmark. https://events.static.linuxfound.org/sites/events/files/slides/cyclictest.pdf, 2013.

[34] Haytham; Gebali Fayez Sharma, Mridula; Elmiligi. Performance evaluation of real-time systems. International Journal of Computing and Digital Systems, 2015.

[35] L. Tan. The worst-case execution time tool challenge 2006. International Journal on Software Tools for Technology Transfer, 2009.

[36] Linus Torvalds. Kernel scheduler and related syscalls. https://github.com/spotify/linux/blob/master/kernel/sched.c.

[37] Linus Torvalds. Scheduler internal types and methods. https://github.com/torvalds/linux/blob/master/kernel/sched/sched.h.

[38] Hussain; Khan Zafrullah ; Mir Sana Ahmed Zahid, Khan; Khalid. Identification and analysis of performance metrics for real time operating system. 2009 International Conference on Emerging Technologies, 2009.