Real-Time Application Development on Linux with PREEMPT_RT

Christopher Besch

Abstract

While Linux is optimized for throughput and efficiency, real-time applications require bounded worst case latencies. However, with the merging of `PREEMPT_RT`, mainline Linux converts to a real-time operating system (RTOS). Through presenting the application developer's perspective and the changes `PREEMPT_RT` introduces to the kernel, we review Linux' real-time capabilities. Additionally, we investigate dynamic worst case execution time (WCET) analysis using a StarFive VisionFive 2 [32] as a case study. Finally, we compare Linux to other RTOSes and argue that the lack of static WCET analysis is Linux' main drawback for real-time applications.

1 Introduction

General purpose operating systems (GPOS) prioritize throughput over deterministic latencies [33]. For many computer systems this focus results in cheap, efficient and performant execution. Other applications, e.g., audio processing, robotics and aviation, need to react to stimuli within certain deadlines [6]. An application is considered real-time when its correctness requires not just correct output but providing this output within a deadline. As most computing use cases do not impose such restrictions, real-time application developers find themselves in a niche. They have to carefully choose what technologies to rely on as most libraries and computer systems do not guarantee bounded latencies. Linux, too, did not guarantee unbounded latencies until the `PREEMPT_RT` patches were merged in September 2024 [28]. Now mainline Linux (v6.12 and onward) is a contender for the operating system of real-time applications [31].

This report gives an overview of what a real-time application developer should be aware of on Linux. Firstly, in section 2 we experimentally show how an application should use the POSIX API to reduce latencies and briefly touch on memory considerations. Secondly, we investigate how `PREEMPT_RT` improves latencies in kernel-space (section 3) and briefly consider hardware implications in section 4. Thirdly, we discuss state-of-the-art tools for analysing Linux' worst case execution time (WCET) in section 5. For this purpose, we test the StarFive VisionFive 2 RISC-V single board computer (SBC) with cyclictest. Lastly, in section 6 we compare Linux with FreeRTOS [1] and RTEMS [2] and discuss what types of real-time applications Linux shows potential for.

2 User-Space

One of the most crucial parts of a real-time system is the application running in user-space. An application can be separated into multiple individual tasks. Such a task becomes runnable at some time point, might require resources during its execution and finishes at a different point in time. If any task is a real-time task, the time between becoming runnable and finishing must be within its deadline. Linux allows both non-real-time and real-time tasks to run on the same kernel [28]. Furthermore, it allows real-time tasks of different priorities to share the same CPU. Shorter deadlines of real-time tasks typically correspond with higher priorities. Here the tasks run in user-space, not on the bare-metal hardware but in the execution environment provided by the Linux kernel. This environment, most notably scheduling and virtual memory, can introduce high latencies (see section 5). Because of this, a real-time application needs to carefully interact with the operating system to fulfil its deadlines. Using a simplified real-time application the following section demonstrates how to do so with Linux.

2.1 POSIX Scheduling Policies

We create an example application with two tasks, a high- and a low-priority one. This is a typical scenario for real-time applications. Take for example the Mars Pathfinder's low-priority, meteorological data gathering task and high-priority bus management task [17]. Here the high-priority task needs to reach completion within a shorter deadline. Whenever the high-priority task is runnable, the system should provide all available resources to this task and no other. In this section we consider the scheduler and thus time as the resource of question. For this we use two versions of the same application, with their source code combined in the appendix Listing 5. Use the preprocessor flag `REAL_TIME_EX` to switch between them.

Firstly, to demonstrate the default scheduling behaviour we implement the tasks with a POSIX `pthread` with default attributes each. Because the scheduling effects are most notable on a uniprocessor (UP) machine, we pin all threads to the same CPU. In Listing 1 the low-priority task completes before the high-priority task. This indicates that while both the high- and low-priority task are runnable, they time-share the CPU. As both threads in Listing 5 (with `REAL_TIME_EX` not set) perform the same kind of work (a simple loop) and both are scheduled with the same settings, this is the expected behaviour of the default `SCHED_OTHER` scheduling policy. Though, this is not our intention; we want the high-priority task to receive the entire CPU while it is runnable.

Listing 1: Output of Listing 5 (with `REAL_TIME_EX` not set), visualized in Figure 1a.

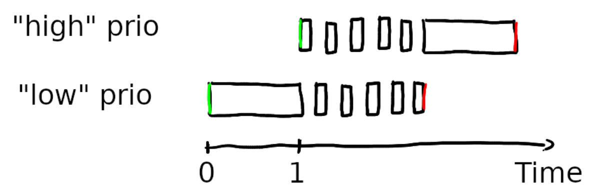
```
low_prio: started
hig_prio: started
low_prio: finished
hig_prio: finished
```

POSIX.4 standardizes the two real-time scheduling priorities `SCHED_FIFO` and `SCHED_RR` [15]. Linux' glibc supports both. These differentiate themselves in the way they handle real-time tasks of the same priority. In Listing 5 (with `REAL_TIME_EX` set) we use the `SCHED_FIFO` real-time scheduling policy. One can reproduce our results with `SCHED_RR`, too. The real-time scheduling policies require the thread to have a static priority higher than 0. For this, we set the `sched_priority` attribute of the `sched_param` struct to 10 and 30 respectively. To run Listing 5 (with `REAL_TIME_EX` set) the executing Linux user needs a high enough `rtprio` configured in her `limits.conf` or be the root user [28]. Now these threads are considered real-time by Linux and scheduled accordingly. Listing 2 shows that the high-priority task finishes earlier.

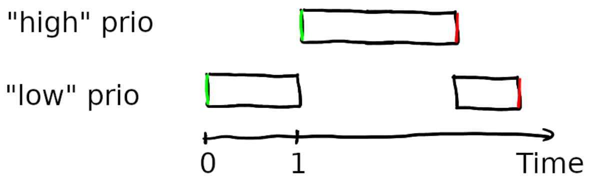
Listing 2: Output of Listing 5 (with `REAL_TIME_EX` set), visualized in Figure 1b

```
low_prio: started
hig_prio: started
hig_prio: finished
low_prio: finished
```

The non-real-time and real-time scheduling policies use different values to prioritize threads: `nice` values and static priorities respectively. The `SCHED_OTHER` scheduling policy grants threads with lower `nice` values more time on the CPU [28]. Counterintuitively, `SCHED_FIFO` and `SCHED_RR` prioritize higher static priorities [28]. Any thread under `SCHED_OTHER` needs a static priority of 0 while the real-time scheduling policies ignore the `nice` value [28]. Furthermore, a real-time task can keep all non-real-time tasks from being scheduled, regardless of their `nice` value. This is a problem on UP with a user interface. If a real-time thread blocks the CPU for a long duration, the interface is unresponsive. For this, consider real-time throttling, which grants real-time threads only e.g., 95% of CPU time and leaves the remainder to non-real-time tasks [28]. Though, critics argue that real-time tasks should only ever execute for very short durations anyway [33]. Then disabled real-time throttling can help to indicate something wrong with the real-time tasks' implementation.



(a) `SCHED_OTHER`



(b) `SCHED_FIFO`

Figure 1: Different POSIX scheduler behaviours. Green lines indicate tasks becoming runnable and red lines tasks completing. Notice that the low- and high-priority task share the CPU in Figure 1a.

2.2 POSIX Priority Inheritance Mutex

Like in subsection 2.1, we provide the two versions described below in Listing 6. Use the preprocessor flag `PRIO_NONE_EX` to switch between them.

Listing 6 (with `PRIO_NONE_EX` set) expands our previous example with a mutex. This accounts for a situation in which the high and low-priority task share a resource. Accesses to this resource must be synchronized, e.g., with a mutex. In Listing 6 (with `PRIO_NONE_EX` set) we utilize a `pthread_mutex` with default arguments for our low and high-priority real-time threads. When the high-priority task becomes runnable, it waits for the lock held by the already running low-priority task. The low-priority task needs to safely leave its critical section before the high-priority task can access the protected shared resource.

The priority inversion problem occurs because we added a third, medium-priority task. Find the output in Listing 3. This medium-priority task becomes runnable while the high-priority task waits for the low-priority task. Because it does not try to lock the mutex and has a higher priority than the currently running task it preempts the low-priority task. Hence, the medium-priority task also prevents the high-priority task from making progress for a long time. This is the priority inversion problem visualized in Figure 2a.

Listing 3: Output of Listing 6 (with PRIO_NONE_EX set)

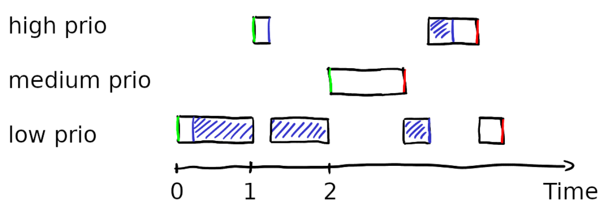
```
low_prio: started, trying to acq mutex
low_prio: acquired mutex
hig_prio: started, trying to acq mutex
med_prio: started
med_prio: finished
low_prio: releasing mutex
hig_prio: acquired mutex
hig_prio: releasing mutex
hig_prio: finished
low_prio: finished
```

On the Mars Pathfinder mission this exact issue occurred [17]. The NASA engineers solved it by swapping the used mutex for a priority inheritance mutex [17]. In Listing 6 (with PRIO_NONE_EX not set) we do the same and use PTHREAD_PRIO_INHERIT as opposed to the default PTHREAD_PRIO_NONE [24]. Listing 4 and Figure 2b show the new scheduling behaviour.

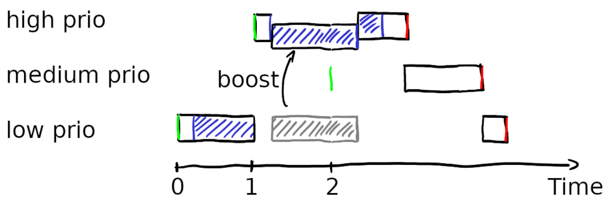
While the low-priority task holds the mutex it prevents the high-priority task from running. Thus, from a real-time perspective, the low-priority task is momentarily more important. The priority inheritance mutex reflects this and boosts the low-priority task's effective priority to match the high-priority. Consequently, the medium-priority task does not receive the CPU and the high-priority task completes its task, solving the problem. Note that the low-priority task's priority is reverted to its original priority once it unlocks the mutex.

Listing 4: Output of Listing 6 (with PRIO_NONE_EX not set)

```
low_prio: started, trying to acq mutex
low_prio: acquired mutex
hig_prio: started, trying to acq mutex
low_prio: releasing mutex
hig_prio: acquired mutex
hig_prio: releasing mutex
hig_prio: finished
med_prio: started
med_prio: finished
low_prio: finished
```



(a) POSIX pthread_mutex Priority PTHREAD_PRIO_NONE



(b) POSIX pthread_mutex with PTHREAD_PRIO_INHERIT

Figure 2: Different POSIX scheduler behaviours. Green lines indicate tasks becoming runnable and red lines tasks completing. Notice how the high-priority task waits for the medium-priority task in Figure 2a. Figure 2b shows how the low-priority task only runs under a boosted priority as long as the high-priority task waits for it.

2.3 Memory Handling

This section gives a brief overview on the memory handling considerations a real-time application developer should make. A real-time application is split into real-time and non-real-time sections. For example, while the startup period of an audio processing application usually does not need real-time deadlines, the subsequent audio processing does. Firstly, Wu [33] argues that real-time applications must never allocate memory in real-time sections. Instead, the non-real-time section right before should do so. Secondly, delays caused by major page faults are potentially unbounded and thus untenable for real-time applications [33]. Therefore, the application should issue a call to libc's `mlockall` with the flags MCL_CURRENT and MCL_FUTURE before the real-time section. This ensures all memory is backed with physical memory and no major page faults occur in the following real-time section [22].

3 Kernel-Space

When a high-priority real-time task becomes runnable, this task needs to receive the CPU as quickly as possible. For example, a timer interrupt or external hardware interrupt could cause this task to become runnable. In section 2 we investigated the implications of user-space code running at this time. In this section we consider kernel-space code instead. Crucially and in contrast to user-space code, the kernel can disable preemption. Only when the kernel is preemptible, can the CPU switch to user-space and the high-priority task be scheduled. The kernel's preemption model decides when the kernel is preemptible [16]. This compile-time option is one of the main ways the administrator trades throughput for real-time performance.

1. With `CONFIG_PREEMPT_NONE` the kernel is never preemptible. Only when the CPU switches back to a user-space context a high-priority task can be scheduled. Batch processes, i.e., servers and high-performance computing can benefit from this setting.

2. A small improvement for real-time applications are the many calls to `might_sleep` in the kernel. Under `CONFIG_PREEMPT_VOLUNTARY` whenever the kernel calls `might_sleep` it allows a high-priority thread to receive the CPU. This setting is typically used in desktop operating systems as it allows some interactivity without excessively degrading throughput.

3. A kernel configured with `CONFIG_PREEMPT` is preemptible everywhere unless the kernel explicitly disable preemption with `preempt_disable()` [16].

4. `CONFIG_PREEMPT_RT` removes these sections with disabled preemption to a minimum. Only in `raw_spinlocks` and IRQ contexts, preemption remains disabled [16]. With these changes a `PREEMPT_RT` configured kernel guarantees bounded latencies (apart from bugs) and becomes suitable for real-time applications [25].

5. `CONFIG_PREEMPT_DYNAMIC` is an additional option allowing one of the first three preemption models to be selected at boot time. The fourth, `PREEMPT_RT`, makes too large changes to the kernel to be enabled at runtime.

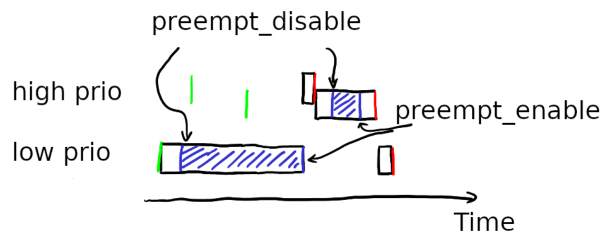
3.1 Spinlocks in `PREEMPT_RT`

In the following section all threads are real-time kernel threads. A spinlock is a type of lock useful for synchronizing access to shared resources. When the thread wants to lock the spinlock it repeatedly checks if the lock is free in a loop. The kernel implements the `raw_spinlock_t` such that preemption is disabled before and during the critical section [19]. Because of that we expect the execution behaviour Figure 3a visualizes. Here, a low-priority thread holds a `raw_spinlock_t`, disabling preemption and thus preventing a high-priority thread from receiving the CPU. This induces unwanted delay.

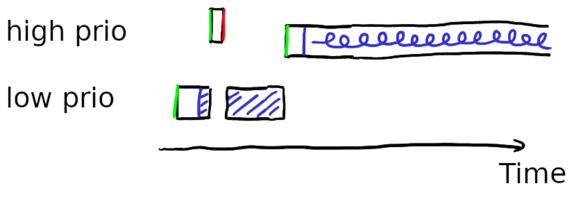
To understand why the kernel disables preemption in this situation consider an imaginary implementation of `raw_spinlock_t` which does not disable preemption. We expect this to lead to a deadlock as Figure 3b shows. The high-priority thread that does not require the lock immediately receives the CPU as desired and completes quickly. But a high-priority thread, which does try locking the `raw_spinlock_t`, causes a deadlock. It uses the CPU to repeatedly checks if the lock remains held by the low-priority thread. That low-priority thread, in turn, cannot run because the high-priority thread holds the CPU. This is not a problem with the actual `raw_spinlock_t` implementation because it disables preemption while it is held and thus no other thread may run.

`PREEMPT_RT` solves this problem without causing real-time delays by using an `rt_mutex`. An `rt_mutex` is a priority inheritance mutex and the underlying implementation of the `pthread_mutex` shown in subsection 2.2 [26]. Figure 3c shows the expected scheduling behaviour with `PREEMPT_RT`. A high-priority task receives the CPU as early as possible. But the moment it waits for the lock, it yields the CPU to the low-priority task. Thus, the system makes progress and no deadlock occurs while keeping the delays in the high-priority tasks as short as possible. The implementation of `rt_mutex` is one example where even `PREEMPT_RT` disables preemption [18]: A `raw_spinlock_t` protects the `rt_mutex` struct's data from unsynchronized access.

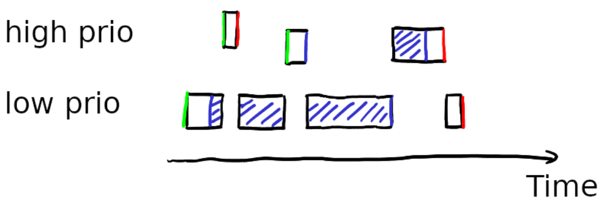
The kernel almost never actually uses the `raw_spinlock_t` directly. Instead, it uses a `spinlock_t`, which in all preemption models but `PREEMPT_RT` maps directly to a `raw_spinlock_t` [20]. When we configure `PREEMPT_RT`, `spinlock_t` maps to `rt_mutex`. Counterintuitively, in this configuration a `spinlock_t` is not a spinlock but a sleeping lock. As is well known, sleeping locks incur a significant overhead for short critical sections. Furthermore, the increase in context switches generally results in a loss of throughput. We see this in section 5. Hence, `PREEMPT_RT` remains a bad choice for non-real-time applications where throughput matters most.



(a) `raw_spinlock_t`



(b) Hypothetical `raw_spinlock_t` with preemption causing a deadlock



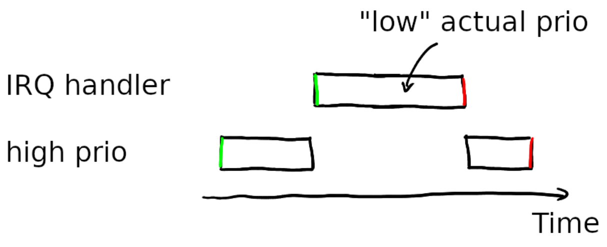
(c) `rt_mutex`

Figure 3: Different spinlocks. Green lines indicate tasks becoming runnable and red lines tasks completing.

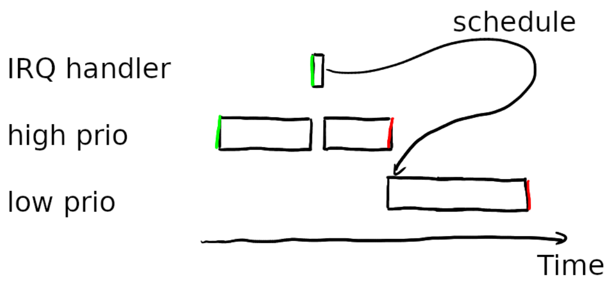
3.2 Interrupt Request Handlers in PREEMPT_RT

In the following section, we briefly discuss the implications of Interrupt Request Handlers (IRQ) on real-time performance. An external interrupt indicates some task needing the system's attention. Some network card receiving a new packet and handing it over to the operating system would be an example. When interrupts are not disabled, the system immediately performs this task in an IRQ handler. To do this the CPU switches from the current context (possibly a user-space or the kernel-space context) to the non-maskable interrupt (NMI), hardirq (top half) or softirq (bottom half) context [31]. While interrupts might remain enabled and thus a hardirq can interrupt a softirq handler, preemption is always disabled. Therefore, user-space application, including high-priority real-time tasks, cannot be scheduled in these contexts, see Figure 4a.

PREEMPT_RT solves this problem by using threaded IRQs [31]. Instead of immediately running the handler, threaded IRQs create a thread doing the actual work, see Figure 4b. Such threads have a static priority of 50 by default and thus can be preempted. In that case they run after all higher-priority tasks yield back the CPU [16]. As a result, application developers should be wary of using static priorities higher than 50 for long-running tasks. This could block the system from handling critical events in a timely manner. While threaded IRQ handlers provide the above advantages, there are situations which do not use threaded interrupts. These include the nanosleep timer used by cyclictest (see section 5) [3] and certain drivers [31]. Another example are System Management Interrupts (SMI), which run firmware defined code and cannot use threaded IRQ handlers [25].



(a) Normal IRQ handler



(b) Threaded IRQ handler

Figure 4: Different types of IRQ handlers. Green lines indicate tasks becoming runnable and red lines tasks completing. Notice how the work initiated by the interrupt runs at a low-priority.

4 Hardware Considerations

While this report focuses on user-space and kernel-space software, another major source of latencies is the hardware and its firmware. Many hardware technologies increase average throughput at the cost of difficult to predict latencies. Classic examples include speculative execution and caching [25]. Because of that Rostedt [25] argues real-time application benchmarks should run in the worst expected cache conditions.

Another major source of latencies outside the control of the kernel-space and user-space software developers are SMIs. An SMI runs firmware defined code blocking the CPU [25]. Recently, Maatallah [21] found a bug in such code to be the cause for millions of Asus laptops stuttering in simple tasks such as video playback. This shows why a real-time application developer must take her hardware into consideration. Other factors include Simultaneous Multithreading (SMT) and Dynamic frequency scaling. Both can induce difficult to determine latency, which is why Wu [33] argues real-time application developers should disable both. Further analysis is outside our scope and left for future work.

5 WCET Analysis

As we showed above, real-time performance depends on many components, not just the user-space application. cyclictest is a tool to empirically determine the system delay, that is all delay not directly caused by the user-space application. For this purpose, it is a synthetic user-space application repeatedly raising timer interrupts. It then compares the desired time of timer expiry with when the timer actually expired. At this time, user-space code (the expiry path) runs [3]. This timer expiry path runs directly in the hard interrupt context. Because an application's interrupt handler could be a threaded IRQ instead, this indirection can cause additional delay not measured by cyclictest [3]. Furthermore, the application will likely need some time to execute the required task itself. Consequently, the cyclictest results are the WCET an optimal application could achieve on the given system. Quite clearly, the application developer should take this system delay into consideration when choosing hardware and operating system (including configuration).

We test the improvements `PREEMPT_RT` provides using a StarFive VisionFive 2 RISC-V [32] single board computer (SBC) and the Linux mainline kernel version 6.12.5. Refer to Besch [8] for a detailed explanation of the setup. "Analysis of Running Real-Time Linux on VisionFive 2" [7] already performed these tests on kernel version 5.15.0 with `PREEMPT_RT` patches applied. We attempt to reproduce parts of this benchmark. The paper fails to specify the exact kernel configuration. Thus, we use the default provided in [4] only altering the preemption model. Nevertheless, we use the same testing procedure:

1. Boot the system and leave it idle for at least five minutes.
2. Request the maximal CPU frequency using the performance scaling governor.
3. Perform synthetic loads: `stress-ng --matrix 4` as a CPU load and `stress-ng --vm 4` for memory load (mmap, munmap calls).
4. Run cyclictest with the options `--mlockall --smp --priority=99 --interval=200 --quiet --duration=10m --histofall=200`.

Out of the cyclictest measurements the WCET is the only relevant metric for real-time applications. Therefore, we only present the max latencies measured in Table 1. Most importantly, we noticed a 19644% latency reduction from the two extreme preemption models with CPU load: `PREEMPT_NONE` to `PREEMPT_RT` (see the left column in Table 1). Interestingly, our observed latencies are significantly higher than "Analysis of Running Real- Time Linux on VisionFive 2" [7] measured. We suspect they used a different configuration, further improving the system delay. Additionally, we noticed a noticeably longer boot-up with `PREEMPT_RT`.

	max latency in our measurement (µs)	**max latency in "Analysis of Running Real- Time Linux on VisionFive 2" [7] measurement** (µs)
CPU load with `PREEMPT_NONE`	884	
CPU load with `PREEMPT`		88
CPU load with `PREEMPT_RT`	45	39
Memory load with `PREEMPT_RT`	127	95

Table 1: cyclictest measurements

Another observation is the stark latency difference for the different types of synthetic load we replicate. CPU heavy loads incur lower latencies than memory operations. This is because these cyclictest measurements do not take the actual production application into account. The different latencies for different loads show how non-real-time tasks of an application can influence the system latencies. Emde [12] proposes enabling latency measuring monitors in the production system itself. If the system does not exceed its deadlines over a long end-to-end test one might consider it real-time compliant. [6] however argues that dynamic latency analysis of any kind cannot measure the actual WCET. The worst case conditions are usually not obvious and thus difficult to test for. Therefore, one cannot consider the dynamic latency analysis results as a safe WCET to rely on. This is especially important for any preemption model other than `PREEMPT_RT`. Linux does not guarantee bounded latencies for these. Consequently, the actual WCET of `PREEMPT_NONE` could theoretically be unlimited. Heuristics can only make these dynamic WCET measurement safe by overestimating the actual WCET by several orders of magnitude [6].

Static code analysis attempts a different approach, specifically path analysis. Path analysis does not execute the software but analysis its source code statically. It finds the execution path with the worst execution time and simulates the cache and CPU pipeline to calculate the actual WCET [29]. However, for Linux (with `PREEMPT_RT`) static WCET analysis is infeasible [12]. This stems from Linux' broad feature set and original intended use case as a GPOS. Thus, Linux relies entirely on dynamic analysis. The Open Source Automation Development Lab (OSADL) QA Farm on Real-time of Mainline Linux performs continuous dynamic real-time analysis, including cyclictest [23]. They test many hardware platforms and Linux kernel versions. This provides application developers some assurance that the operating system fulfils their real-time requirements. Still, there is no proof that Linux with `PREEMPT_RT` does not contain a bug causing unbounded latencies. Depending on the specific application this may be problematic. In section 6 we argue that this is the main drawback of Linux compared to other RTOSes.

6 Related Work

The following section investigates two contenders to Linux for real-time applications: FreeRTOS and RTEMS. We investigate these as they are heavily used in critical real-time applications like those found in spacecrafts. A notable example application is Outpost [11], the flight software library by the DLR Institute for Space Systems. It is compatible with both FreeRTOS and RTEMS.

In contrast to Linux, [30] shows that static WCET analysis is possible for applications running on FreeRTOS. This means that a developer on FreeRTOS can proof that her application will fulfil the deadlines even in the worst of conditions. Though, FreeRTOS comes with certain limitations. Firstly, FreeRTOS is a much smaller operating system than Linux or RTEMS. It does not allow dynamic app loading and requires compiling the application with FreeRTOS in combination. Furthermore, FreeRTOS only provides basic features such as process synchronization, timers and memory allocation [10]. Any application using TCP, for example, must do so with a separate library [1]. Additionally, while FreeRTOS supports a Memory Protection Unit (MPU), there is no virtual memory [5]. This enables it to easily run on CPUs without an MMU. In addition, real-time tasks do not need the benefits of virtual memory considering the paging overhead. Non-real-time applications, however, might do require virtual memory. This makes it more difficult to have both real-time and non-real-time applications running on the same system. Moreover, FreeRTOS itself does not support multiprocessor CPUs; though there are forks that do (see ESP-IDF) [13]. In general FreeRTOS is a much smaller operating system than Linux. Especially the lack of full POSIX support hampers porting aplications to FreeRTOS [14].

RTEMS is another operating system specifically developed for real-time applications. Contrary to FreeRTOS it does support the POSIX API and many of the features a GPOS provides. This includes a file system and a networking stack among others [2]. Static path analysis is possible with RTEMS and allows the accurate calculation of the WCET [9]. Again, RTEMS comes with multiple downsides: Most importantly it lacks the ability to dynamically link shared libraries and only provides a single global address space [27].

As stated in section 5 Linux does not allow static WCET analysis. In return, the application developer finds herself surrounded with most of the open source ecosystem. The Debian 12 official repositories alone contain over 60000 packages. Only very little of this software is designed to fulfil real-time requirements. However, it facilitates certain applications with both real-time and non-real-time tasks running on the same system. The non-real-time tasks can make use of the entire Linux ecosystem while the real-time task takes advantage of the kernel's guaranteed bounded latencies. An alternative system design could separate real-time and non-real-time tasks on two CPUs with different operating systems. We suspect that such a design would be more expensive and less energy efficient, having to power two instead of one CPU. Furthermore, we expect data exchange and synchronization to be more difficult than on a single CPU.

Additionally, we argue that the size of the Linux ecosystem results in a much easier development experience. There are more projects realized with Linux and thus more documentation, tutorials and experiences than with FreeRTOS or RTEMS. Experience with FreeRTOS and RTEMS is much more limited to embedded and real-time applications. This is the advantage of using an operating system with so much use as a GPOS.

7 Conclusion

In this report we showed how real-time applications can use the POSIX.4 real-time API and `PREEMPT_RT` on Linux. We explained how `PREEMPT_RT` converts Linux into a real-time operating system (RTOS) with guaranteed bounded latencies. Furthermore, we argue that an application developer needs to pay careful considerations to the hardware she chooses. She can use cyclictest as dynamic WCET analysis to empirically measure the system delay and verify her choice. Unfortunately, she cannot proof that her application will not encounter any worse latencies than cyclictest measured. The lack of static WCET analysis is the main drawback of Linux with `PREEMPT_RT`. FreeRTOS and RTEMS, which we compared to Linux, however, do permit static WCET analysis. On the contrary, Linux enables using a large portion of the open-source ecosystem and offers easier development than FreeRTOS or RTEMS. We conclude that Linux with `PREEMPT_RT` is an apt choice for certain real-time applications; especially those with both non-real-time and real-time tasks and where dynamic WCET analysis suffices.

References

[1] Accessed: 2025-08-07. URL: `https://www.freertos.org`.

[2] Accessed: 2025-08-07. URL: `https://www.rtems.org`.

[3] Accessed: 2025-08-08. URL: `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`.

[4] Commit: 3a681f7. URL: `https://github.com/starfive-tech/linux`.

[5] Accessed: 2025-08-07. URL: `https://forums.freertos.org/t/free-rtos-memory-management/6267/3`.

[6] AbsInt GmbH. *Worst-Case Execution Time Analyzer*. Accessed: 2025-08-06. AbsInt GmbH. URL: `https://www.absint.com/ait/slides/3.htm`.

[7] "Analysis of Running Real- Time Linux on VisionFive 2". In: (2024). URL: `https://doc-en.rvspace.org/VisionFive2/PDF/VisionFive2_RT_Linux.pdf`.

[8] Christopher Besch. *Real-Time Linux on RISC-V*. Accessed: 2025-07-07. 2025. URL: `https://chris-besch.com/articles/riscv_rt`.

[9] Antoine Colin and Isabelle Puaut. "Worst-case Execution Time Analysis of the RTEMS Real-Time Operating System". In: ().

[10] *Creating a New FreeRTOS Project*. Accessed: 2025-08-07. URL: `http://www.openrtos.org/Creating-a-new-FreeRTOS-project.html`.

[11] DLR Institute for Space Systems. Accessed: 2025-08-07. URL: `https://github.com/DLR-RY/outpost-core`.

[12] Carsten Emde. *Long-term monitoring of apparent latency in PREEMPT RT Linux real-time systems*. Open Source Automation Development Lab (OSADL) eG. URL: `https://www.osadl.org/fileadmin/dam/articles/Long-term-latency-monitoring.pdf`.

[13] Espressif. *ESP-IDF — FreeRTOS Overview*. Accessed: 2025-08-07. Espressif. URL: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/api-reference/system/freertos.html`.

[14] FreeRTOS. *FreeRTOS+POSIX*. Accessed: 2025-08-07. FreeRTOS. URL: `https://www.freertos.org/Documentation/03-Libraries/05-FreeRTOS-labs/03-FreeRTOS-plus-POSIX/00-FreeRTOS-Plus-POSIX`.

[15] Michael González Harbour. "Real-time posix: an overview". In: *VVConex 93 International Conference, Moscu*. 1993.

[16] Jim Huang and Chung-Fan Yang. "Effectively Measure and Reduce Kernel Latencies for Real-time Constraints". In: Embedded Linux Conference. 2017.

[17] Mike Jones. *What really happened on Mars Rover Pathfinder*. Accessed: 2025-07-07.

[18] *Linux v6.15.3 include/linux/rtmutex.h l.23.*

[19] *Linux v6.15.3 include/linux/spinlock_api_smp.h l.130.*

[20] *Linux v6.15.3 include/linux/spinlock_types.h l.14*.

[21] Mohamed Maatallah. *The ASUS Gaming Laptop ACPI Firmware Bug: A Deep Technical Investigation.* Accessed: 2025-09-29. URL: https://github.com/Zephkek/Asus-ROG-Aml-Deep-Dive.

[22] *mlock(2) — Linux manual page.*

[23] OSADL. Accessed: 2025-08-07. URL: https://www.osadl.org/OSADL-QA-Farm-Real-time.linux-real-time.0.html.

[24] *pthread_mutexattr_setprotocol(3) — Linux man page.*

[25] Steven Rostedt. *Real-time programming with Linux.* Accessed: 2025-07-07. Kernel Recipes. 2016. URL: https://youtu.be/w3yT8zJeOUw.

[26] *RT-mutex subsystem with PI support.* Accessed: 2025-07-07. Kernel.org. URL: https://www.kernel.org/doc/html/v6.13-rc5/locking/rt-mutex.html.

[27] RTEMS. *RTEMS — Dynamic Loader.* Accessed: 2025-08-07. RTEMS. URL: https://docs.rtems.org/docs/main/user/exe/loader.html.

[28] *sched(7) — overview of CPU scheduling — Linux manual page.*

[29] Anthony Serino and Liang Cheng. "A Survey of Real-Time Operating Systems". In: (2014). URL: https://engineering.lehigh.edu/sites/engineering.lehigh.edu/files/_DEPARTMENTS/cse/research/tech-reports/2019/LU-CSE-19-003.pdf.

[30] Josef Stmadel and Peter Rajnoha. "Reflecting RTOS Model During WCET Timing Analysis: MSP430/Freertos Case Study". In: (2012).

[31] *Understanding Linux real-time with PREEMPT_RT training. Constraints.* Bootlin. 2025.

[32] *VisionFive 2 Datasheet.* Accessed: 2025-09-25. URL: https://doc-en.rvspace.org/VisionFive2/PDF/VisionFive2_Datasheet.pdf.

[33] Shuhao Wu. *Real-time programming with Linux.* Accessed: 2025-07-07. URL: https://shuhaowu.com/blogseries.html#rt-linux-programming.

8 Appendix

8.1 Real-Time Scheduling Policy

Listing 5: POSIX real-time scheduling example

```c
#define _GNU_SOURCE
#include <printf.h>
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void pin_to_cpu(int cpu)
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(
        ↪ cpu_set_t), &cpuset);
}

void* low_prio_thread(void* arg)
{
    pin_to_cpu(0);
    printf("low_prio: started\n");

    for(volatile long i = 0; i < 1e10; ++i)
        ;

    printf("low_prio: finished\n");
    return NULL;
}

void* high_prio_thread(void* arg)
{
    pin_to_cpu(0);
    // ensure low prio thread has started
    sleep(1);
    printf("hig_prio: started\n");

    for(volatile long i = 0; i < 1e10; ++i)
        ;

    printf("hig_prio: finished\n");
    return NULL;
}

int main()
{
    // initialize threads
    pthread_t          low, high;
    pthread_attr_t     attr;
    struct sched_param param;
    pthread_attr_init(&attr);
#ifdef REAL_TIME_EX
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO)
        ↪ ;
#else
    // this is the default
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER
        ↪ );
#endif
    pthread_attr_setinheritsched(&attr,
        ↪ PTHREAD_EXPLICIT_SCHED);

#ifdef REAL_TIME_EX
    param.sched_priority = 10;
#endif
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&low, &attr, low_prio_thread,
        ↪ NULL);

#ifdef REAL_TIME_EX
    param.sched_priority = 30;
#endif
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&high, &attr, high_prio_thread,
        ↪  NULL);

    // cleanup
    pthread_join(low, NULL);
    pthread_join(high, NULL);
    pthread_attr_destroy(&attr);
    return 0;
}
```

8.2 Priority Inheritance

Listing 6: Priority Inheritance with POSIX example

```
#define _GNU_SOURCE
#include <printf.h>
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex;

void pin_to_cpu(int cpu)
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(
        ↪ cpu_set_t), &cpuset);
}

void* low_prio_thread(void* arg)
{
    pin_to_cpu(0);
    printf("low_prio: started, trying to acq mutex
        ↪ \n");
    pthread_mutex_lock(&mutex);
    printf("low_prio: acquired mutex\n");

    // simulate long operation
    // both sleep and busy wait work, though
        ↪ differently
    // sleep(4);
    for(volatile long i = 0; i < 1e10; ++i)
        ;

    printf("low_prio: releasing mutex\n");
    pthread_mutex_unlock(&mutex);
    printf("low_prio: finished\n");
    return NULL;
}

void* high_prio_thread(void* arg)
{
    pin_to_cpu(0);
    // ensure low prio thread has lock
    sleep(1);
    printf("hig_prio: started, trying to acq mutex
        ↪ \n");
    pthread_mutex_lock(&mutex);
    printf("hig_prio: acquired mutex\n");

    printf("hig_prio: releasing mutex\n");
    pthread_mutex_unlock(&mutex);
    printf("hig_prio: finished\n");
    return NULL;
}

void* medium_prio_thread(void* arg)
{
    pin_to_cpu(0);
    // ensure high prio thread waits for lock
    sleep(2);
    printf("med_prio: started\n");

    // simulate CPU-intensive task
    // don't sleep here because we want to block
        ↪ the CPU
    for(volatile long i = 0; i < 1e10; ++i)
        ;

    printf("med_prio: finished\n");
    return NULL;
}

int main()
{
    // initialize mutex
    pthread_mutexattr_t mutex_attr;
    pthread_mutexattr_init(&mutex_attr);
#ifdef PRIO_NONE_EX
    pthread_mutexattr_setprotocol(&mutex_attr,
        ↪ PTHREAD_PRIO_NONE);
#else
    pthread_mutexattr_setprotocol(&mutex_attr,
        ↪ PTHREAD_PRIO_INHERIT);
#endif
    pthread_mutex_init(&mutex, &mutex_attr);

    // initialize threads
    pthread_t          low, medium, high;
    pthread_attr_t     attr;
    struct sched_param param;
    pthread_attr_init(&attr);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO)
        ↪ ;
    pthread_attr_setinheritsched(&attr,
        ↪ PTHREAD_EXPLICIT_SCHED);

    param.sched_priority = 10;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&low, &attr, low_prio_thread,
        ↪ NULL);

    param.sched_priority = 30;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&high, &attr, high_prio_thread,
        ↪  NULL);

    param.sched_priority = 20;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&medium, &attr,
        ↪ medium_prio_thread, NULL);

    // cleanup
    pthread_join(low, NULL);
    pthread_join(medium, NULL);
    pthread_join(high, NULL);
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&mutex);
    pthread_mutexattr_destroy(&mutex_attr);
    return 0;
}
```