School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Guaranteed response time for mixed-criticality systems on seL4

by

Curtis Millar

Thesis submitted as a requirement for the degree of
Bachelor of Science (Honours) in Computer Science

Submitted:	May 2021	Student ID:	z3424700
Supervisor:	Prof. Gernot Heiser	Topic ID:	COMP4953

Abstract

This report provides an overview of strategies used to guarantee the timing behaviour of hard and mixed-criticality realtime systems. It then proposes a set of changes the the MCS seL4 extensions that ensure the kernel-level scheduler can guarantee the assumptions made by common timing analysis processes for fixed-priority scheduling.

Abbreviations

DM Deadline-monotonic scheduling

EDF Earliest deadline first

IPCP Immediate priority ceiling protocol

IPC Inter-process communication

MCS Mixed criticality system

OS Operating system

RM Rate-monotonic scheduling

RSC Resource scheduling context

RTOS Real-time operating system

SC Scheduling context

SS Sporadic server

SWaP Size, weight, and power

TCB Trusted computing base

VCPU Virtual CPU

Contents

0.1 Introduction

Computerised systems are widely used in situations where the security of lives and assets is paramount and where the systems have strict timing requirements for how they must interact with each other and with the physical world. The level of assurance required for these mixed-criticality real-time systems demands precise guarantees to demonstrate that the timing requirements for any highly critical components are always met. The cost of formally and mathematically verifying these guarantees is immense and grows exponentially with system complexity and is in tension with the desire to consolidate the software of such systems into fewer hardware components, particularly in applications which must minimise size, weight, and power consumption (SWaP) of the physical system.

In this thesis, we demonstrate an approach that provides the required guarantees utilising seL4, a verified protected-mode microkernel, to enforce the scheduling behaviour required for high-criticality components without needing to perform any verification or provide any level of assurance for low-criticality components. This approach drastically reduces the cost of providing absolute assurance of real-time systems used in medical, aerospace, automotive, and military applications.

This project will outline a number of changes made to the seL4 microkernel, ensuring that it enforces scheduling behaviour precisely and correctly, and then it will demonstrate how those guarantees are used to construct real-time systems with highly-critical components that will always satisfy their real-world timing requirements.

The remainder of this report will be structured as follows.

- Chapter 0.2 will cover the background of real-time systems, mixed criticality systems, and protected-mode kernels.
- Chapter 0.3 will look at work related to consolidating real-time systems into single physical systems and the construction of mixed-criticality real-time systems.
- Chapter 0.4 will introduce the concept of response time analysis and its use in determining the schedulability of high and mixed-criticality real-time systems and the requirements imposed on the scheduler implementation.
- Chapter 0.5 will introduce seL4 as a protected-mode microkernel with extensions to guarantee scheduling behavior for mixed-criticality real-time systems.
- Chapter 0.6 will outline the approach taken to adapt the extensions in seL4 to the theory of response time analysis and demonstrate how seL4 can be used to guarantee schedulability of mixed-criticality systems.
- Chapter 0.7 will outline the implementation of the changes made to seL4 to guarantee scheduling properties of system components and the guidance of applying time demand analysis to real-time systems on seL4.
- Chapter 0.8 will review the efficacy of the implementation and strategy to show that the system does provide the guarantees necessary for mixed-criticality systems in all contexts.

- Chapter 0.9 will outline direction for future work to build on what has been discussed and implemented.
- Chapter 0.10 will conclude the work of this thesis.

0.2 Background

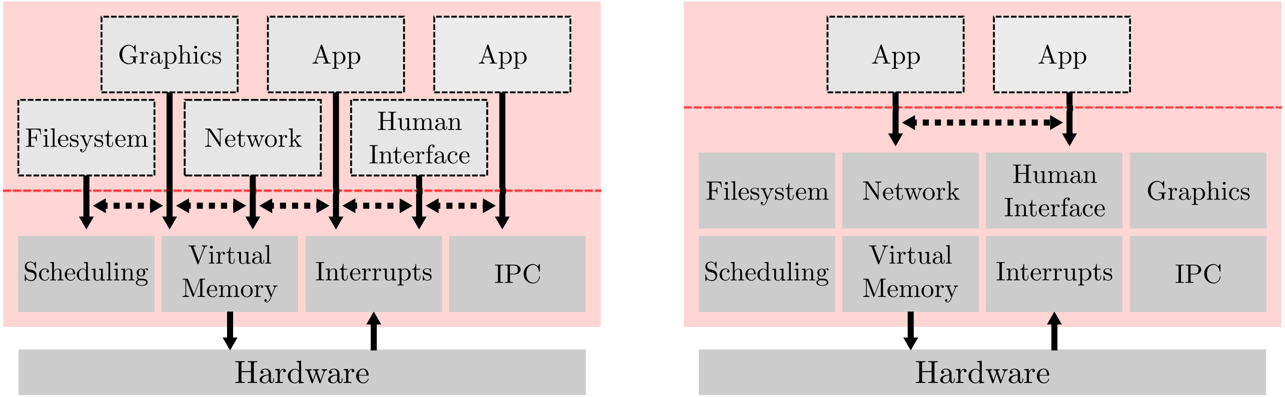
0.2.1 Protected-mode microkernels

A *protected-mode* operating system kernel is the component of an operating system (OS) that operates with greater access to hardware mechanisms than all other software in the system. Hardware that provides a greater *privilege level* at which a kernel can operate will also enforce protections for software operating at levels of lesser privilege. These protections prevent execution of privileged instructions and access to all but the memory to which access has been explicitly granted when unprivileged software is executing. When unprivileged software attempts to violate these protections the hardware *traps* the operation and invokes the kernel to respond to the fault.

A kernel operating in protected mode can construct isolated *threads* of execution where execution of one thread has restricted access to read from and write to a subset of the machine's physical and device memory, respond to external events, and to invoke the kernel to perform work on its behalf. When operating on behalf of a *user-level thread*, the kernel can pass information between *protection domains* that would otherwise isolate the threads from each other. This is used as the basis for inter-process communication.

[Liedtke, 1995] characterises a *microkernel* by stating "a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."

The L4 family of microkernels implement a minimal feature-set that includes a threading construct, virtual addressing abstractions, mechanisms for communication between threads, mechanisms for communication with hardware, and capabilities that describe what components of the system and underlying architecture any particular thread may be able to access. All other operating system features are implemented at user-level and are managed by software executing in user-level threads.



A microkernel system

A monolithic kernel system

Figure 1: A comparison of the services that operate with full privilege in a microkernel and a monolothic kernel

A major advantage of microkernel-based systems is that a component is not *implicitly* required to trust operating system components upon which it does not depend. This bares a stark contrast to *monolithic* kernels, where much of the operating system runs within the kernel at its higher privilege level. In either system, any component is required to trust anything that runs in at the kernel's privilege level as it has unrestricted access to the entire system and the underlying hardware.

The concept of trust also extends beyond the code that runs at the privilege level of the kernel; the *trusted computing base* of an application is the set of all software and hardware that is required to work correctly in order for the application itself function correctly. If an application is highly critical then all the components in its TCB are at least as critical as it is and require at least the same level of guarantee. If the process of determining the guarantees of a highly critical application is expensive, then it is desirable for that applications TCB to be as small as possible.

[Klein et al., 2009] presented seL4, a L4 microkernel with a complete functional correctness proof. [Sewell et al., 2011] outline the formal verification of enforcement of access control in seL4. The formal verification proves that the kernel is guaranteed to correctly isolate user-level components which makes seL4 a trustworthy basis for safety-critical or security-critical systems.

0.2.2 Formal verification

Formal verification of software is a process whereby the implementation of that software is said exhibit the same behaviour as the specification of that system. Where the implementation may exhibit a potentially greater set of behaviours than the specification, the implementation is said to *refine* the specification. The refinement process generally requires a relation to be made between the formal semantics of some representation of the implementation, such as the source code or the binary machine code, and the possible behaviours of the specification.

Additionally, or alternatively, formal verification of software may be the process of proving that certain the specification (or the implementation) satisfies certain, formally defined properties.

0.2.3 Real-time systems

Real-time computing systems are those with specific requirements regarding the ordering, duration, and completion time of individual operations. Such requirements are commonly necessary in control systems, where the state of hardware needs to be maintained in response to environmental changes and user input, multimedia applications, where audiovisual information needs to be transmitted and synchronised locally or over large networks, and digital signals processing, where large amount of data must be processed with a high input rate. The physical processing hardware used in such applications can also vary widely, from low-power embedded microcontrollers to large multiprocessor systems including processors dedicated to particular processing operations.

[Liu, 2000] characterises real-time applications as a collection of *tasks* and a set of *resources*. Each task is a sequence of *jobs* that must be allocated some of the system's resources in order to complete. A resource may be *finite*, where only a limited number of tasks can access the resource concurrently, or *infinite*. Resources are also generally assumed to be *reusable*, in that when one task is done with it another task can be granted access.

Jobs within a task are often modelled with the following common properties:

- *release time* or *arrival time*; the instant after which a job may begin execution,
- *deadline*; the instant at which the job must complete,
- *execution time*; the amount of time a particular job takes to execute, and
- *blocking time*; the amount of time for which a particular job is preempted by execution other than that of higher-priority tasks.

For a particular task, we may not know some (or any) of the properties of the individual jobs until execution time. Instead we determine the bounds of all jobs within a task in order to reason about that task.

The *period* or *minimum inter-arrival time* is the minimum time between the release of consecutive jobs. A task where the duration between the release times of consecutive tasks is known a priori and is constant is known as a *periodic task*. A task where only the *minimum* time between consecutive releases is known is referred to as *aperiodic* or *sporadic*.

The *worst-case execution time* (WCET) is the longest possible execution time of any job within the task. This is generally determined by analysis of the software with knowledge of the hardware that will be used. A prediction of a WCET is useful only if it is sound, i.e. if it is not less than the actual WCET.

The *worst-case blocking time* is the longest possible time any task may be preempted by execution other than that of higher priority tasks. This is determined by any non-preemptable operations, such as privileged operations that modify privileged state, and priority inversions, where tasks are able to preempt other tasks with a higher priority.

Within a particular task, all jobs share a common set of properties, namely:

- the *relative deadline*; the maximum duration after the release of any job in a task within which the job must complete (this is often equal to the period of the task), and
- the *laxity*; a function used to determine the degree to which a job is still useful if it misses its deadline.

Tasks are typically implemented within a real-time operating system (RTOS) using operating-system threads with each job within the task being a *release* of that thread by the operating system's scheduler. This implementation does not permit more than a single job in any task from executing at a time. In order to allow a single task to operate across multiple protection domains, i.e. in contexts with different access control restrictions, some RTOSs separate scheduling configuration from threads and release scheduling objects rather than threads. The scheduling object can then be passed between the threads in distinct protection domains that perform work on behalf of a job within a task.

0.2.4 Real-time scheduling algorithms

A schedule must be chosen for a real-time system to determine when each job in a task may be exclusively allocated finite resources. This is particularly useful when assigning the use of a physical CPU for the execution of a job. Different scheduling algorithms offer different advantages and prioritise different functional requirements.

An admission test must be applied for a particular set of jobs when paired with a particular scheduling algorithm. The test ensures the algorithm will produce a schedule where the temporal requirements of every task will be met, i.e., if every job in every task will be able to execute to completion before its deadline. In cases where the real-time parameters of every job is not known, every job is assumed to execute with the worst case parameters of its task.

One can quantitatively compare scheduling algorithms using a number of metrics. The *utilisation* of an algorithm refers to the proportion of total system time which can be utilised by running tasks. More pessimistic algorithms will only schedule tasks for a relatively small portion of the available time. A schedule can also be compared by the *latency* or *response-time* of the jobs that it schedules, i.e., the time between a job's release and its completion. *Jitter* measures of variance in latency for tasks of a given schedule.

Priority-driven scheduling algorithms

Priority-driven schedules are determined as the system is running and are able to adapt to the actual execution time of jobs and the later release times of aperiodic jobs. In doing so they can reduce the time between the release and completion of a job if higher priority tasks don't consume their full WCET. These algorithms can also account for tasks being added and removed over the lifetime of the system by applying the admission test at runtime.

Every job in a priority-driven schedule is assigned some numeric priority. Whenever a job is released, the released job with the highest priority is chosen to continue execution. When a job is released that has a higher priority than the currently executing job, the higher priority job *preempts* the already executing job, with the higher priority job executing to completion before the lower priority job is resumed.

Fixed-priority scheduling algorithms assign the same priority to every job within a task, allowing priorities to be assigned offline. *Rate-monotonic scheduling* (RMS) assigns all jobs of a task the same priority with the priority for a task being greater than the priority of all tasks with a longer period. In a RMS schedule, jobs in a task with a shorter period may preempt the jobs of a task with a longer period. *Deadline-monotonic scheduling* (DMS) assigns priorities such that the priority for a task will be greater than all tasks with a longer relative deadline. In a DMS schedule, jobs in a task with a shorter relative deadline may preempt the jobs of a task with a longer relative deadline. In systems where the relative deadlines of all tasks are proportional to their periods (i.e., the deadline is implied by the period), the two algorithms are equivalent.

Dynamic-priority scheduling algorithms may change priorities of jobs within the set of currently released jobs. One of the most common dynamic-priority schedules is the *earliest deadline first* (EDF) schedule. This assigns priorities to the released jobs in the order of their absolute deadlines, such that the task with the earliest absolute deadline at any given time has the highest priority.

Sporadic servers

Sprunt et al. [1989] present *sporadic servers*, a model of a task that that may not have hard realtime requirements or any notion of jobs, but that is bounded by the worst-case behavior of a fixed-priority aperiodic task. A sporadic server can be considered as the equivalent aperiodic task for all timing analysis purposes.

A sporadic server is assigned a *period*, *budget*, and a priority, which are respectively the same as the minimum inter-arrival time, worst-case execution time, and priority of of the equivalent aperiodic task. The sporadic server is scheduled as though it is an infinite set of aperiodic tasks, each of which has a worst case inter-arrival time equal to the period of the sporadic server and a priority equal to the sporadic server. The worst-case execution time of each task in the infinite set is infinitesimally small, but the sum of the worst case execution times is equal to the budget of the sporadic server.

In a fixed priority system of aperiodic tasks, the worst-case interference from a set of tasks with an equal priority and minimum inter-arrival time and with a known combined worst-case execution time is equal to the interference produced by a single aperiodic with the same priority and minimum inter-arrival time and with combined worst-case execution time.

When a sporadic server executes, it can only consume as much time as the total WCET of *released* tasks from the equivalent infinite set of tasks, i.e., only those tasks for which it has been more than the period since they last executed. Unlike an aperiodic task, when a sporadic task suspends no execution time is forfeited. A subset of the tasks with a combined WCET equal to the time the sporadic task spent executing are considered has having their jobs completed and are scheduled for future job execution. If this is less than the total WCET of all tasks that were released before execution began, the rest of the tasks remain released and allow the task to execute immediately when it is resumed or becomes unblocked. The key to scheduling sporadic tasks as an infinite set of tasks is that we only determine which task in the infinite set were released once the server suspends. We only consider enough tasks to have been released so as to cover the execution cost.

This model is useful for scheduling soft realtime, non-realtime, and low criticality tasks in a system of hard realtime and high criticality tasks, i.e. a mixed-criticality system, without restriction on the scheduling properties of any tasks.

0.2.5 Shared resources

Tasks in a real-time system may require access to a common set of resources, with some of those resources only being usable by a limited number of tasks at a time. In order to ensure that only one task has access to an instance of any resource at a given time, a resource access protocol is applied to the schedule of a real-time system. The protocol provides imposes a set of requirements on how resources are considered in the schedule and provides a set of guarantees regarding the timing of each resource access.

The *immediate priority ceiling protocol* is a resource access protocol for fixed-priority scheduling algorithms. It assigns a priority to each resource, one that is higher than the priority of all tasks that may access that resource. When a task executes, it operates at the highest priority of all those assigned to the resources to which it has obtained exclusive access. If it does not have exclusive access to any resources, it operates at its own priority. This ensures that the task will always complete any access before any task at a lower priority than the resource would begin execution of a job. As such any task that may wish to access a resource is guaranteed immediate access to all resources it may require once execution for a job commences. If the job of any higher-priority task were to preempt it before it accesses a certain resource, that preemption, including all resource accesses, would complete before the task could continue execution. Additionally, all resource access from lower-priority tasks must complete before the job can begin execution as such accesses occur at a higher priority than the task.

0.2.6 Mixed criticality systems

[**?**] describe criticality as "a designation of the level of assurance against failure needed for a system component". A *mixed criticality system* (MCS) is comprised of components of differing criticalities. Examples of such designations include:

- *safety critical*; wherein a component must be *guaranteed* against failure,
- *mission critical*; wherein the operations of a component must be prioritised over less critical components, and
- *non-critical*; where such components may be allowed to fail temporarily or completely.

The trusted computing base of any highly critical component is also at least as critical as the component itself. This means that a high-criticality component cannot depend on the correctness of a low criticality component and must be properly isolated from all lower-criticality and untrusted components.

Industry standards used to certify MCSs often provide a specific set of such criticality designations, although they may refer to them by a different name. Many real-world real-time systems are naturally mixed criticality by their specification, as they must provide differing levels of guarantee to different tasks.

Reservation-based schedulers

Reservation-based schedulers offer a mechanism to enforce the execution bounds assumed by any timing analysis for a real-time system, rather than either formally verifying the implementations of tasks to show that they do not exceed those bounds or simply assuming that they do not.

Each task in a reservation based system is represented by a *reservation*. The reservation is controlled by some scheduling component and encapsulates time that is made available for execution associated with the task. When the available time associated with a reservation is depleted, the scheduler is invoked to suspend execution associated with the task and schedule more time to be made available in the future.

Only the portion of the system implementing and scheduling reservations, in addition to any part of the system that is entirely non-preemptable, can produce violations of the scheduling assumptions; every other part of the system is forcibly bounded to operate within the configured schedule. As such, only the implementation and scheduling of reservations and non-preemptable code sections need to be verified to show that tasks are *temporally isolated*, i.e., that a change in behaviour of one task can prevent another from satisfying its timing requirements.

0.3 Related work

0.3.1 KeyKOS

The KeyKOS microkernel [Hardy, 1985] implements hierarchical scheduling using *meters* to control delegation of execution time on the CPU and *meter keys* which act as capabilities that confer access to delegated execution time with which a *domain* can execute. All time used via a meter key is tracked in every meter from the root to the meter which produced the key. When the time associated with a meter key is exhausted, the *meter keeper* is invoked to manage the delegation of further time.

This system allows for a *meter keeper* to make scheduling decisions at user level by judicious choice of when to grant access to processing time.

Implementing a general-purpose real-time system using this microkernel would require more complex user-level services to be created and would impose a large overhead for the user-level operations required to respond to scheduling events.

0.3.2 NOVA Microhypervisor

The NOVA microhypervisor [Steinberg et al., 2010] provides scheduling contexts (SC) that encapsulate a priority level and a time *quantam*. The time quantum describes the amount of time for which a thread may execute before it is preempted by a thread of equal priority.

When a client thread performs a blocking call that is handled by a lower priority server thread on the same core, the client donates the SC to the server and the server executes using the SC of the client. If a second client with a higher priority than the first calls to the server while the server is still processing the request of the first client, the second client *helps* by allowing the server to run with its higher-priority SC. A server will always run with the highest-priority SC of all of its blocked clients.

While this system may be useful in limiting *priority inversion*, when work on behalf of a low-priority task prevents the progress of a high-priority task, it does not provide the bandwidth or scheduling guarantees of a system with hard real-time components nor does it provide any level of temporal isolation between components.

0.3.3 Composite

The Composite microkernel [Parmer and West, 2008; **?**] makes a wide variety for schedulers possible at user-level. *TCaps* [Gadepalli et al., 2017] are temporal capabilities that control explicit access to execution time on a particular CPU. They provide a single finite amount of time at a particular global priority. Components within Composite encapsulate a set of access controls, such as accesses to system resources and physical memory, and implementation of a set of routines that can execute within it. A user-level scheduler can construct access to multiple instances of time to implement recurring releases of a thread with a TCap for each release. As each TCap has an associated global priority, distinct jobs within a task can be assigned different priorities if they are released using distinct TCaps.

Composite uses a migrating thread model for IPC. When a thread executing in one component calls into another component, the thread context is migrated to the called component for the duration of the call. This allows a TCap to remain associated with a thread when it calls into another component such that time spent execution in that component still executing time on the original thread's TCap.

Priority inversion can be minimised in Composite with TCaps using user-level scheduling operations. When a resource within a component is locked and a client with a greater priority requests access, it can communicate with the scheduler component to allow the thread with exclusive access to continue execution using the TCap with the highest priority of all clients waiting to access the component.

Utilising this system requires complex user-level components to implement the scheduling decisions and handle scheduling events which add to overhead and implementation complexity. Each scheduling operation also imposes a considerable cost as it includes a switch to a dedicated thread execution within a scheduling component.

0.3.4 Quest-V Hypervisor

[Danish et al., 2011] utilise the sporadic server model described by [Stanovic et al., 2010] to implement real-time scheduling for virtual-CPU contexts in the Quest-V hypervisor. All tasks are executed using a *Main VCPU* context which is scheduled using the algorithms from [Stanovic et al., 2010]. When a task needs to perform an I/O operation with an external device, it communicates with a driver on an *I/O VCPU* which is responsible for directly communicating with hardware. Each I/O VCPU is scheduled with minimal logic to preserve bandwidth. When an I/O VCPU handles a request from a Main VCPU, it inherits the priority of the Main CPU until the request is complete.

[Danish et al., 2011] note that the overheads from the complexity of the sporadic server implementation have a noticeable impact on throughput and that the I/O VCPUs benefit from the simplified bandwidth preservation logic.

This demonstrates how a scheduling system built on bandwidth constrained scheduling contexts can be made effective but it does not address scenarios where tasks of differing criticalities must share hardware and software resources.

0.3.5 Flattening hierarchical mixed criticality scheduling

[Völp et al., 2013] describe a way in which a system of temporally isolated real-time tasks, encapsulated with *scheduling contexts*, can be used as the basis for a hierarchical system of independent real-time components. They also describe how different scheduling algorithms may be mapped onto such a system of scheduling contexts and what modifications may be required to adapt the scheduling contexts to allow for different algorithms.

They introduce a system where a task is represented by a *scheduling context* (SC), a kernel object which can be released by the operating system scheduler, which is attached to the thread responsible for executing in response to a job being released in a task. Each scheduling context is given a priority, with the highest released SC being executed at any given time. Each scheduling context may be used for execution up to its assigned budget in each window of time equal to its period. Each SC is also assigned a fixed criticality level.

They extend these fixed-priority SCs with additional behaviour that is required for particular mixed criticality scheduling algorithms. Scheduling contexts are extended with a relative deadline which is used to determine when a job has not completed by its deadline and preempt it in such a case. They also allow a task to execute with a series of SCs with each SC describing a single job. This is done for every job in the systems hyperperiod, the least common multiple of the periods of all tasks in the system.

Each of the changes increases the applicability of the underlying model to allow a greater set of single and mixed criticality scheduling algorithms to be implemented on top of the primitives of the model.

While this system does show the flexibility of the underlying primitives, the guarantees to real-time tasks depend entirely on the correctness of every scheduling component within the system. This work does not include the necessary mechanisms to adequately enforce temporal isolation such that untrusted and low-criticality tasks cannot interfere with the correct operation of high-criticality tasks.

0.3.6 MC-IPC

[Brandenburg, 2014] describes a system of encapsulating shared resources in *resource servers* and describes a protocol, *MC-IPC*, for communication between tasks of varying criticality that preserves 'temporal and logical isolation'. This allows for the effective use of resources shared between tasks of differing criticality. The protocol implements a priority inheritance that is fair across multiple cores.

The system reduces the assurance burden and the level of trust required of low-criticality tasks that share resources with high-criticality tasks. Resources are shared between components of differing criticality and assurance by encapsulating them in a shared resource server that inherits the priority and execution time of its highest priority client.

The protocol requires all tasks provide sufficient time for all lower priority tasks on all cores that have been granted access to the server to complete their request. This ensures that even when a lower-priority task is able to access the server ahead of a high-priority task, the lower-priority task cannot prevent the use of the server from the high-priority task by exhausting its available budget while the server is responding to it. This also ensures that in a schedulable system, all real-time tasks will have all of their requests serviced by the shared resource server, such that even high-criticality tasks with low priority will be able to complete.

The protocol makes two fundamental assumptions about the scheduler to which it is applied. The first is that reservations of the highest priority level that have available budget may be selected for execution and may thus have their time consumed, even when the thread executing on the reservation is inactive or blocked on IPC. The second is that the priority of a reservation does not change until its budget has been exhausted or replenished.

This work effectively demonstrates how resources can be shared between mixed-criticality tasks using priority inheritance without preventing the correct execution of high-criticality tasks. As such, it would be useful component of a more complete real-time operating system. However, it makes assumptions that restrict the operating system in which it operates: a server is tightly coupled to the IPC medium used to request service such that it is always aware of all blocked clients and the time the server spends executing can be charged to any one of its blocked clients.

0.3.7 seL4 mixed criticality scheduling

[Lyons, 2018] presents a modification to the scheduler used by seL4 microkernel, enabling the construction of mixed criticality real-time systems. It introduces explicit scheduling context objects that represent access to processor time which can be managed at user-level. These changes allow for a number of real-time scheduling decisions to be made with user-level components.

A thread must have access to a SC with available budget in order to execute on a CPU. Each SC is bound to a particular CPU core and enforces a maximum bandwidth by replenishing a particular amount of time throughout a given window of time. This ensures that, at any instant, the amount of CPU time that may have been consumed by threads associated with an SC does not exceed a configured portion of all time equal of its *budget* divided by its *period.* Only the threads of the highest configured priority that are not blocked and with a *released* scheduling context are eligible for execution. Time from a scheduling context's available budget is only consumed when it is associated with the currently executing thread.

In addition to the changes to scheduling, the semantics of blocking IPC is changed to guarantee that whenever there is more than a single TCB waiting on a kernel object to send or receive IPC, the thread with the highest priority will always perform its operation first.

When a thread exhausts it available execution budget or loses access to a scheduling context a user-level monitor is able to respond and reconfigure the task such that it can be recovered. This allows for management of soft real-time tasks and low-criticality tasks and enables the monitor to recover resource servers shared between mixed criticality clients.

The bounds on execution are too strict to be applied to mixed-criticiality and hard realtime systems without losing any notion of temporal isolation. Additionally, there are issues regarding the accounting of kernel execution time. These issues are explained and resolved in the remainder of this thesis.

0.4 Response Time Analysis

The timing analysis of a real-time system determines whether or not that system will, under all circumstances, satisfy its expected timing behaviour. Any timing analysis will make assumptions regarding the behaviour of each job of each task in the system. To ensure that we guarantee the results of the timing analysis, we must determine these assumptions and guarantee they hold, either by verification of the tasks themselves or by enforcing them in a scheduling monitor. As we will see, the scheduler must enforce specific guarantees regarding how execution time is made available to a task at a given priority, how that execution time may be consumed by a resource at a higher priority, and how unused execution time is lost.

Vestal [1994] presents a process that has come to be known as *response time analysis* (RTA) that produces a critical scaling factor, the maximum factor by which the execution time of all tasks can be scaled without loss of schedulability. While Vestal [1994] details further ways in which this analysis can be used to adjust a system and account for variability, we only seek to demonstrate the bounds that must be enforced for the analysis to apply.

To determine the schedulability of a task set we must first produce some mathematical model of that task set. We describe a set of tasks as $\{\tau_1, \tau_2, \ldots, \tau_n\}$ where each task τ_i has an assumed worst-case execution time C_i, a known minimum inter-arrival time T_i, and some fixed priority P_i. We also assume a worst-case blocking time, B_i, which is the worst-case amount of interference time from sources other than tasks with an equal or higher priority, which can include lower-priority tasks accessing resources shared with tasks at an equal or higher priority. For each task, we also consider the set of task indices at an equal or higher priority, $\mathbf{H}_i = \{x \mid P_x \geq P_i\}$.

To produce a potentially large overestimate of the worst-case response time R_i, for a given some task τ_i, we assume that all tasks at an equal or higher priority execute for their worst case execution times as many times as they may be released in the minimum inter-arrival time of τ_i. We also assume that the worst-case blocking time B_i is also observed for the task with the worst-case response time. This task will then be guaranteed a response within the total of these execution times if that total is no greater than the inter-arrival time T_i.

$$R_i = B_i + \sum_{j \in \mathbf{H}_i} C_j \left\lceil \frac{T_i}{T_j} \right\rceil$$

If the actual worst-case response time is even smaller than this estimate, we can determine a smaller overestimate. For each task at an equal or higher priority to τ_i, we consider each multiple of its minimum inter-arival time that is no greater than the inter-arrival time of τ_i, each of these being a member of $\mathbf{S}_i$.

For each such duration, we consider the total of the worst-case blocking time B_i and the total execution of all equal or higher priority jobs that can be released in that duration. .

$$\mathbf{S}_i = \left\{ kT_u \;\middle|\; u \in \mathbf{H}_i \,, k \in 1 \mathinner{..} \left\lfloor \frac{T_i}{T_u} \right\rfloor \right\}$$

For each duration in $\mathbf{S}_i$, we consider the total of the blocking time B_i and, for each task at an equal or higher priority than τ_I, that task's worst-case execution time multiplied by the number of releases of that task that may occur in the duration from $\mathbf{S}_i$.

$$R_i = \min_{s \in \mathbf{S}_i} \left(B_i + \sum_{j \in \mathbf{H}_i} C_j \left\lceil \frac{s}{T_j} \right\rceil \right)$$

The worst-case response time is the smallest of these total execution times and task is schedulable, if that time is no greater than the duration for which it is considered. i.e., τ_i is schedulable within the given task set if the following is true:

$$\exists s \land s \in \mathbf{S}_i \land \left(B_i + \sum_{j \in \mathbf{H}_i} C_j \left\lceil \frac{s}{T_j} \right\rceil \right) \leq s$$

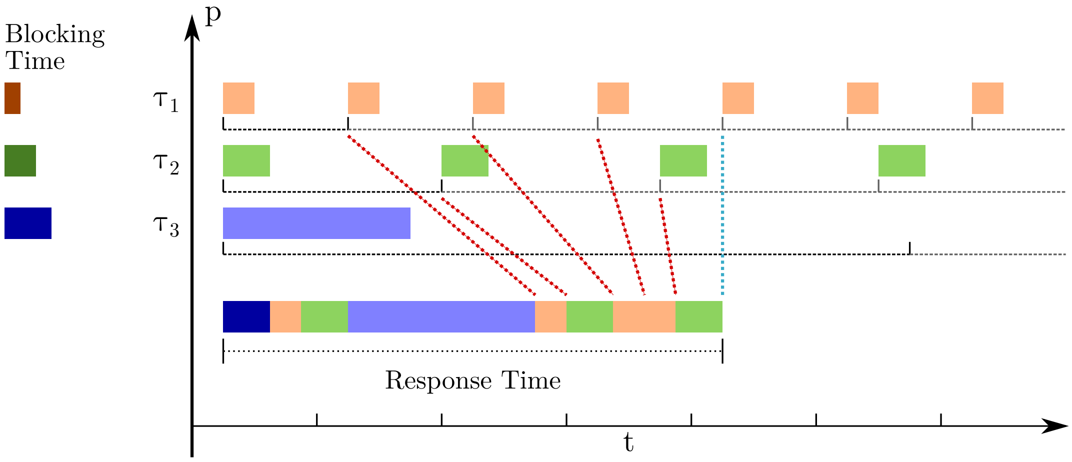


Figure 2: Response time analysis for a single task within a simple set of tasks
This shows un upper bound on the response time of τ_3 when scheduled in a system with τ_1 and τ_2 at a lower priority than both. The darkened durations to the left indicate the worst-case blocking time for each job of that task. The bar along the bottom shows the total of the blocking time for a job in τ_3 and the worst case execution time as jobs are released from all tasks after the release of a job in τ_3. The coloured dotted lines compare the multiples of the minimum inter-arrivals of each task with the total worst-case execution for the jobs released within that time. The blue dotted line indicates the smallest multiple of a task's inter-arrival time that is no less than the total of the worst-case execution times of jobs released, and thus, a bound on the response time of τ_3.

For each such period we can also determine a slack time, X_s, which is the amount of execution time available after completion of all tasks at a priority equal to or greater than τ_i in the period s. The slack time of τ_i is then the greatest such X_s.

$$s \in \mathbf{S}_i \implies X_s = s - \left(B_i + \sum_{j \in \mathbf{H}_i} C_j \left\lceil \frac{s}{T_j} \right\rceil \right)$$

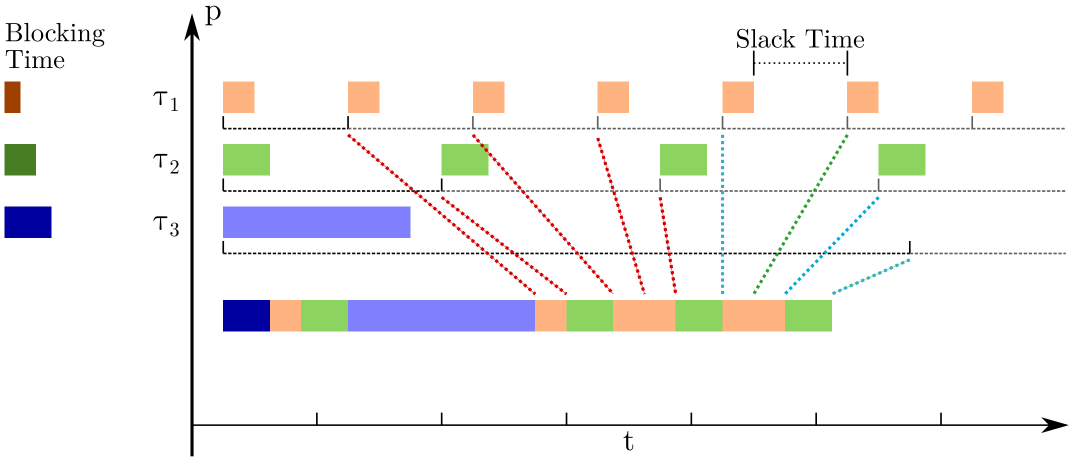


Figure 3: Slack time for a single task within a simple set of tasks
This shows the *slack time* for τ_3 when scheduled as in Figure 2. The sum of the blocking time and all jobs released is extended up to the minimum inter-arrival time of τ_3, with lines relating the multiples of inter-arrival times of each task to the sum of the total of the blocking time for τ_3 and the WCET of all tasks released. The green dotted line shows the multiple of an inter-arrival time with the greatest value when the total of the blocking time and WCET of released jobs is subtracted, and this, the slack time for τ_3.

From this, we can also produce a critical scaling factor, Δ^*, which is the greatest factor by which total system execution can scale before becoming unschedulable. A critical scaling factor of less than 1 implies an unschedulable system as well as the factor by which execution time must be scaled to achieve schedulability.

$$\Delta^* = \min_{i \in 1..n} \left(\max_{s \in \mathbf{S}_i} \frac{s}{s - X_s} \right)$$

We can also observe that if we substitute a task τ_j with an arbitrary set of tasks $\mathbf{U}_j$ that have the same priority and minimum inter-arrival time as τ_j and that have a combined worst-case execution time equal to that of τ_j, then the interference on lower-priority tasks is unchanged.

$$\forall t. \sum_{v \in \mathbf{U}_j} C_v = C_j \implies (\forall v \in \mathbf{U}_j \implies T_v = T_j) \implies C_j \left\lceil \frac{t}{T_j} \right\rceil = \sum_{v \in \mathbf{U}_j} C_v \left\lceil \frac{t}{T_v} \right\rceil$$

As sporadic servers [Sprunt et al., 1989] model a sporadic tasks as an infinite number of tasks with an equal minimum inter-arrival time and priority and a known total worst-case execution time, they can be considered in this scheduling analysis as the equivalent aperiodic task.

This model only captures execution time consumed in two manners: when a task at a fixed priority executes, and when a task at some priority level is preempted due to execution not attributed to a higher priority task. This model also assumes that budget for job is lost when that job completes. For an enforcing scheduler to both satisfy these requirements, it must ensure:

- that the actual execution time of every task is divided into jobs,

- that each job of a task not be allowed to execute for more than the assumed worst-case execution time,
- that any consecutive jobs of a task are never permitted to execute with less than the minimum inter-arrival time of their task separating their releases, and
- that all execution time that preempts a task is either attributed to a task with a fixed higher priority or being bounded by the assumed blocking time.

If we apply these requirements for a scheduler of tasks with a *deadline-monotonic* (DM) or *rate-monotonic* (RM) fixed priority assignment with resources accessed using the *immediate priority ceiling protocol* (IPCP), we find that any execution time that that preempts a task that is not attributed to higher-priority task must be bounded by the blocking time of of that task. We also find that all resource accesses from lower-priority tasks to resources higher priorities must be additionally considered in this bound; no task should be able to access a resource for longer than the blocking time of each task at a lower priority than that resource. Additionally, we must guarantee that a task does not change its priority relative to other tasks by any means other than a resource access within these bounds. In a system where tasks may suspend or block, all available execution time must be either forfeited or the task must consume time as though it were executing for each duration for which it was blocked or suspended.

If we can guarantee these bounds in the seL4 kernel, then the ability for any task to meet any deadline depends only on its own correctness, the correctness of any shared resource implementation, that the WCET enforced by the kernel is no less than its actual WCET of the task, and that the kernel enforced priority inversion bound at any priority level is no less than the WCET of any shared resource access that occurs at that priority. This ensures that the correctness of any task does not depend on the correctness of any other task when no dependency relationship exists.

Whilst the constraints regarding scheduling algorithm and resource sharing protocol here do simplify the means by which we perform the check, the general approach can be applied to a wider range of scheduling configurations without significant change to the properties that must be enforced by the kernel or the trust relationship between tasks. This is what will ensure that we can provide the robust scheduling guarantees necessary for mixed-criticality systems.

For seL4 to guarantee the assumptions of fixed-priority scheduling analysis such as RTA (section 0.4), it must allow for the modelling of tasks with fixed priorities, enforce a maximum bound on the time for which any task may execute at a raised priority level, and enforce a known bound per job on preemption from sources other than resource accesses and higher priority tasks. If such a system is to be configured by user-level components, any component with the ability to configure the scheduling configuration of a task, including its priority, worst-case execution time, and inter-arrival time, or the scheduling configuration of a resource, including its priority and the bound on the execution time of each access, must be verified to conform to the configuration assumed by any timing analysis.

0.5 seL4

Klein et al. [2010] presents seL4, a formally verified microkernel that has been proven to provide guaranteed integrity, availability, and isolation between software components in correctly configured systems [Murray et al., 2013]. seL4 uses capabilities to model and control access to all resources of a system, both those provided by the kernel implementation itself and those managed by user-level components. The seL4 kernel directly manages access to physical memory, virtual to physical address translation configuration, capabilities themselves, and generic communication channels.

Certain capabilities may represent access to specific resources, such as a particular region of physical memory or the physical address space, with some constrained set of rights to act on that resource. For a thread to act on any such resource, it must have direct access to a capability to said resource with the rights sufficient for the desired action.

Some capabilities represent communication channels that are mediated by the kernel and that may alter the scheduling state of the communicating threads such as to block and unblock threads when a communication occurs. Synchronous communication channels can be used to send capabilities as well as data between threads.

0.5.1 Mixed-criticality scheduling

Lyons [2018] introduces a set of extensions to the seL4 microkernel referred to as the *mixed-criticality scheduling* (MCS) extensions. Prior to these extensions, the scheduler provided within seL4 allowed for round-robin scheduling over a set of threads with fixed priorities. A queue of threads was managed for each fixed priority level, with the thread from the head of the highest-priority non-empty queue being selected for execution. After a number of kernel ticks, 5 by default with each tick being 2ms apart, the currently executing thread would be placed on the end of the queue of its priority and the new head would then be chosen for execution. This implementation also allowed a simple form of *timeslice donation* to occur, whereby a communication to a blocked thread at new highest priority would allow that thread to consume the remaining time before the next kernel tick.

Whilst this scheduling policy is sufficient for many non-realtime systems, it is not sufficient for realtime systems, as it does not satisfy any of the guarantees mentioned in Chapter 0.4. A user-level scheduler can be implemented on top of this scheduler, although the amount of overhead introduced by performing scheduling operations at user-level is undesirable.

The MCS extensions introduce new resource capability, a *scheduling context* (SC) that models execution time made available on a specific core of a system. It also introduces a *scheduling control* capability that is required to allocate time on a particular core to a scheduling context. A scheduling context is configured with a number of refills, a budget, and a period. A scheduling context can be donated from one task to another via synchronous IPC, emulating the behavior of thread migration. This can also be used to model a resource server by constructing a thread that waits on an IPC object without a configured SC and that is donated the SC of any client such that it executes using the client's SC until it responds.

The MCS extensions do not alter the manner in which a thread's priority is configured. A capability to a pair of *thread control blocks* (TCB) is all that is required to alter priority of one of those threads. Of the pair, one provides a *maximum configured priority* (MCP), which enforces an upper bound on the new priority and MCP assigned to the other thread.

The scheduler enforces a sliding window constraint on the execution associated with each scheduling context. In any period of time equal to the scheduling context's configured period, no more than the scheduling context's budget may be consumed by a thread executing using the time of the scheduling context. This places a fixed *bandwidth* of time attributed to any SC, irrespective of the priority at which the time of the SC is consumed and any blocking behaviour that the thread associated with the SC may exhibit. The number of refills assigned to a task bounds the degree to which the execution time associated with a SC can be fragmented, with bandwidth being lost whenever a preemption or blocking operation would fragment the execution into a number of distinct durations per period that is greater than the number of refills.

Whilst this model provides a number of useful mechanisms for dealing with time as an explicit resource, the guarantees provided by the scheduler under this implementation do not relate in any useful way to the scheduling assumptions of our timing analysis. Whilst the bounds enforced are no greater than the bounds on execution assumed by the real-time analysis, they do not provide any guarantee that sufficient execution time would be provided for a given task to execute to up to its worst-case execution time within the provided budget as execution budget is reduced whenever a task is not executing, even if only due to preemption. This model also fails to enforce sufficient bounds on execution for accesses to resource servers, as the scheduling context donation effectively allows a task to permanently change its priority to that of the resource server. Any task with access to a TCB capability is able to modify the priority of that TCB, although this could be configured in a manner such that components other than any trusted schedulers can only reduce the priority of those tasks. Any task with access to multiple TCBs and multiple SCs is also able to swap the priorities at which any of those SCs expend budget.

In addition to the issues present in the model provided by the MCS extensions, there are a number of issues present in the implementation itself, particularly with regards to preemption from the kernel, both for external interrupts and for the purposes of context switching. Each kernel entry will include a read of the scheduler timer, with all time prior to the timestamp being attributed to whichever SC was active at the point the kernel entry began and all time after the timestamp attributed to the SC that was active at the point where the kernel entry completed. When a kernel entry occurs due to an external interrupt, this time is attributed to the executing SC even if it is unrelated to the execution of the executing task. When a kernel entry occurs as part of a preemption to begin a higher priority task, part of that kernel execution will be charged to the lower-priority task, despite being for the purposes of the higher priority task.

Finally, the scheduler will preempt the currently running task each time a job of a task is released asynchronously, even if that task is at a lower priority than that of the task executing at the time. While this does incur a cost to the executing task, the number of lower priority releases is per job is bounded by the number of lower priority tasks. As the cost is bounded, it can be considered as part of the worst case execution, although it would be more accurately tracked as component of the blocking time for that task.

In order to ensure that the assumptions of scheduling analysis are guaranteed by the kernel scheduler, substantial changes are required of the scheduling model provided by the kernel. Preemptive kernel execution must be clearly attributed and bound by some explicit task that is responsible for the preemption, or bound and attributed to the blocking time assumed for each task. Priority increases, such as resource accesses, should be modelled such that the both appear as normal task execution to tasks with a lower priority than any client and must be bounded by the blocking time of any tasks at a higher priority than any client.

0.6 Approach

To produce a real-time scheduler that correctly guarantees the assumptions of our timing analysis, we will model our system of tasks and resources using seL4 primitives, modifying the implementation as needed. The modifications should avoid introducing unnecessary overheads on other uses of the kernel such that we could not introduce them into the distributed seL4 implementation. As we are only seeking to model a deadline-monotonic (DM) schedule with the immediate priority-ceiling protocol (IPCP) for shared resource access, the system can be modelled directly on the existing primitives introduced with the mixed-criticality scheduling (MCS) extensions.

0.6.1 Modelling tasks with scheduling contexts

As our timing analysis only models aperiodic tasks with jobs that cannot self-suspend in any way, we must constrain execution of all tasks in the system to be either running jobs or completed jobs of aperiodic tasks. For each task we must ensure that the jobs never execute for more than the assumed worst-case execution time and that no two consecutive job releases of a task are less than the task's minimum inter-arrival time apart.

seL4 provides mechanisms to directly suspend and resume threads and allows threads to suspend for the purposes of inter-process communication (IPC). We must ensure that the effects of these operations don't break the assumptions of the timing analysis, either by ensuring that a blocked thread with an SC is scheduled and charged time as though it was running for the duration it was blocked or suspended, or by scheduling SCs as though they have no released jobs when they become blocked.

As the intent of the blocking mechanisms in the kernel is intended to allow lower-priority tasks to execute when higher-priority tasks are waiting for some external event or a signal from another task, choosing to model the SCs associated with suspended threads as having no released jobs makes the most sense. If a thread must wait in a way that does not complete a job, a user-level spinlock would be the most appropriate mechanism to remain compatible with timing analysis.

With this approach, we have two conditions for a job of an aperiodic task to be released: the minimum inter-arrival time since the last job of that task was released was passed and the task is associated with a thread that is not blocked. We also have three conditions that can lead to an aperiodic job being completed: the task executes for its worst-case execution time, the task forfeits any remaining execution time, or the thread associated with the task suspends or blocks.

We only ever need to model a single job of any aperiodic task. When that task completes, we configure a release of the next job in the future no earlier than the task's minimum inter-arrival time after the release of the completed job. If we reach the point in time of the subsequent release and the associated thread isn't blocked, the job is released. If the associated thread is unblocked or resumed and it is after the time of the subsequent release, the job is released at the instant it becomes unblocked.

As each SC models a set of aperiodic tasks with a combined worst-case execution, we track the total of the worst-case execution of the tasks that are more than the minimum inter-arrival time since their last release, the *total available budget* of the SC. We also track the individual worst-case times of the unreleased tasks in the set and the point in time at which they would next release using the replenishments of the SC. For periodic and aperiodic tasks modelled with a dedicated SC, this requires only a single replenishment in addition to the record of total available budget.

Not all tasks that we wish to include in a system may necessarily behave as a single aperiodic task. For tasks which may have more *sporadic* activations, we can instead use the sporadic server (SS) model [Sprunt et al., 1989] which models sporadic tasks as an infinite set of aperiodic tasks with equal priority and minimum inter-arrival time and a bounded total worst-case execution. To model this in seL4, a scheduling context (SC) will be used to model an infinite set of aperiodic tasks, each with the same priority and minimum inter-arrival time and with a bounded total worst-case execution time.

A sporadic task can also be modelled with a dedicated scheduling context with the total worst-case execution time of all released tasked tracked in the total available budget and each replenishment recording an instant where an infinite set of aperiodic tasks would be released and the total worst-case execution of those tasks. As the number of replenishments that can be tracked for an SC is bounded, when a set of aperiodic tasks completes and the maximum number of replenishments are already in use, the last replenishment is delayed to the time when the new replenishment would need to be created. As can be seen by this model, the actual number of replenishments does not bound the *number* of aperiodic tasks modelled by the single sporadic tasks, only the number of grouped releases of infinite subsets of those aperiodic tasks.

0.6.2 Removing the sliding window constraint

seL4 enforces a sliding-window constraint on the execution of all tasks, preventing a task from executing when its continued execution would the violate the constraint. The constraint requires that for any window of time equal to the SC's configured period, no more than the SC's configured budget in execution may be charged to that SC. A well behaved real-time task may be expected to expend as much as twice its configured budget in any window of size equal to its period in a fully schedulable system. This can occur if its worst-case response time is equal to its inter-arrival time when one job observes the worst-case response time and is followed immediately by a job that is not preempted.

When this constraint is applied to tasks where the budget is configured to the task's WCET and with period configured to match the minimum inter-arrival time of the task, any interference to a task's execution from higher-priority tasks introduces delays to the task's future release times, leading to inevitable deadline misses. This generally leads to all but the highest priority tasks missing deadlines almost shortly after the system has started.

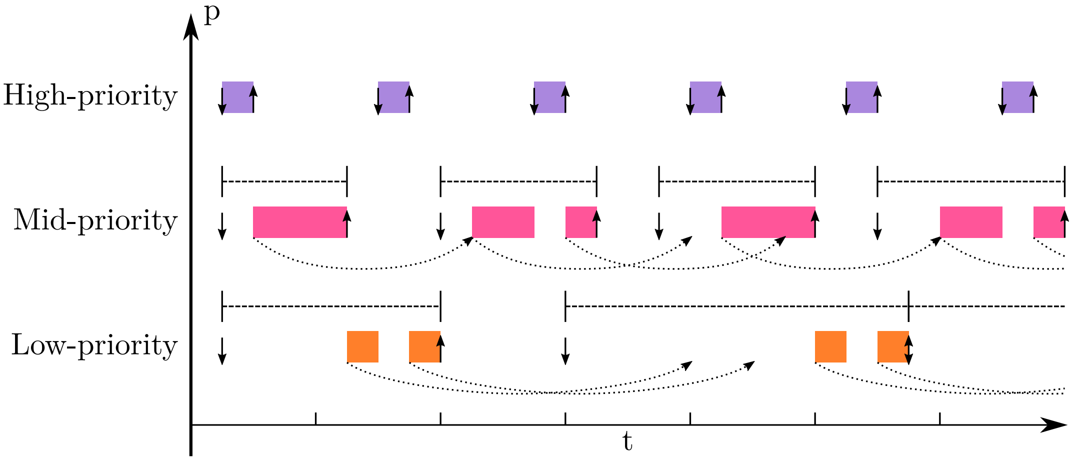


Figure 4: Three periodic tasks with the sliding window constraint applied

The task set contains:

- a high priority task with a WCET of 1 unit and a minimum inter-arrival time of 5 units,
- a medium priority task with a WCET of 3 units and a minumum inter arrial time of 7 units, and
- a low-priority task with a WCET of 2 units an a minimum inter-arrival time of 11 units.

With the sliding window constraint applied to each of these tasks, when a unit of time is consumed a release of that unit of time is placed in the future by the minimum inter-arrival time. Whevever a job of task is preempted, not only is the response time of the preempted job increased, but the response time of all subsequent jobs of that task is also increased. This produces an unbounded worst-case response time for all but the task with the highest priority. Response time analysis gives a worst-case response time of 5 units for the medium priority task and 7 units of time for the second case. The second job of the medium-priority task can exceed its assumed worst-case response time by its fourth job and the low-priority task can exceed its worst-case response time by its second job.

The current implementation tracks a set of replenishments for each task. A task stops consuming budget when it stops executing for any reason, such as due to preemption from another task, explicitly suspending, or by blocking to wait for an external event or IPC from another task. At this point, the execution time consumed while the task was running is scheduled as a release one period after the task last began execution. Any available budget left unused can be consumed when the task next runs. When a task exhausts all available budget, it is forcefully stopped and either faults or cannot execute again until it reaches the point of a future replenishment. If a task is stopped, preempted, or calls into the kernel when the maximum number of future replenishments are configured, the task loses all available time until the next replenishment becomes available. A task may also voluntarily forfeit any available budget.

To resolve this, we change the constraint on SCs to be that of the sporadic server algorithm [Sprunt et al., 1989], which models each task as an infinite set of aperiodic tasks with the same priority and period and with a bounded total budget. As we cannot model each task of an infinite set individually, we instead only track the total WCET of subsets of tasks that are released at the same time and bound the number of these groups that we track. Where we would need to record the release of a new subset but have reached the bound on the number of subsets we can track, we take the latest tracked subset and delay it to be released at the same time as the new subset to be scheduled. As such, this produces a reduced bandwidth for any sporadic tasks when it attempts to execute across more subsets than there are refills in an SC.

0.6.3 Correct attribution of execution time

As we intend to enforce execution time by charging scheduling contexts (SCs) for time spent executing, we must take care to ensure that each SC is only charged for the execution time that our timing analysis has assumed is associated with the task that the SC represents. For each task, we attribute all user-level execution while that task is running to that task. We also attribute the cost of all explicit kernel entries, i.e. system calls, made while that task is running to that task. We can also attribute kernel entries due to user-level exceptions, such as arithmetic exceptions or invalid memory accesses, to the SC as they should be triggered by the execution of user-level code.

This leaves two cases where the task that must be charged for execution is less clear: when the kernel is entered due to a task with no released jobs receiving a job release, and when an external interrupt triggers a kernel entry.

Ideally, the kernel would only enter for the release of a job of a task of a strictly higher priority. The current implementation of timed releases in the seL4 kernel uses a single queue sorted by release time, requiring a kernel entry whenever the head element of the queue would observe a release, even if it is at a lower priority. This is required as a higher-priority task may appear later in the queue. An alternative could be to find the first element in this queue with a higher priority and configure the timeout for that point in time, or to maintain a separate release queue for each priority level. Both cases could increase some scheduling operation in a potentially unbounded manner.

The contribution of releases of lower-priority tasks to the WCET of any given task can be considered as bounded. The number of these releases is bounded by the total number of lower-priority tasks and the cost of each release is reasonably bounded by some WCET to enter the kernel and remove an element from the release queue. Each task in the system must include the cost of all such lower and equal-priority releases in that task's WCET.

If a release of a job from a higher-priority task occurs, this will also result in a switch to that higher-priority task. We can attribute the entire cost of the kernel entry for that release to the released task and account for it once in each job of that task. This is preferable to attributing that cost to the task that was running at the time the release occurred as this can occur as many times as higher priority jobs can be released. Where higher priority tasks are only periodic or aperiodic, this is strictly bounded by the minimum inter-arrival time. In the presence higher-priority sporadic tasks, however, a task may see an effectively unbounded number of higher-priority releases. If this cost were to be considered part of the preempted task's execution, the assumed cost would be unbounded.

Attributing kernel time to external interrupts is more difficult, particularly given the implementation of interrupts within seL4. seL4 divides interrupts into 4 general types:

- the timer interrupt, for triggering events in the in-kernel scheduler;
- signals, which produce a signal to an IPC object that can be received by a user-level thread;
- inter-processor interrupts (IPIs) for asynchronous signalling in the kernel between cores; and
- reserved interrupts, for internal kernel mechanisms.

As the timer interrupt is only used for the timer itself, indicating either that the current task has exhausted its budget or that a job for another task has been released, kernel entries for this interrupt have already been correctly attributed.

If a signal interrupt occurs, it would be ideal if the cost of that interrupt were charged to the task that receives the interrupt. If we only allow an interrupt to be received when a task is waiting, and associate that waiting task directly with the interrupt, then that time could be charged to the task. The kernel implementation currently separates the association between an interrupt and the IPC object that it signals from the association between the IPC object and any thread blocked waiting for receive. This makes it difficult to robustly ensure that an IRQ is only unmasked when a thread is blocked waiting for a signal on an IPC object associated with the IRQ. Another complication with this approach is that it can cause interrupt events to be unexpectedly missed.

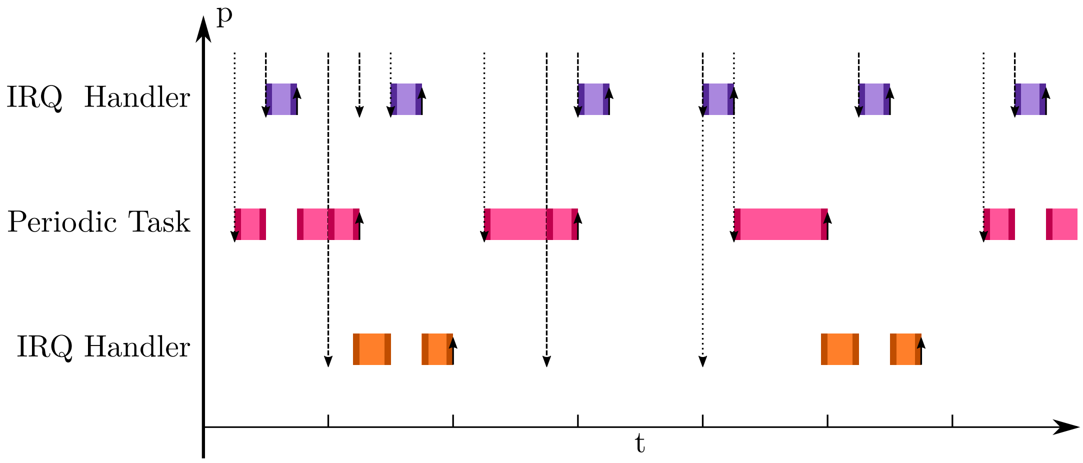


Figure 5: Execution trace of a task set including IRQ handlers and a periodic task
This task set contains a high-priority aperiodic task handling IRQs, a medium-priority periodic task, and a low-priority aperiodic task handling IRQs of a different device. Dotted arrows indicate a timer IRQ marking the minimum inter-arrival time after the previous release of a task. Dashed arrows indicate an external device IRQ handled by a specific task. Dark regions of an execution trace indicate preemptive execution in the kernel, either to switch between tasks or to handle an IRQ.

The approach we use here instead treats all signal IRQs as explicit tasks. Absent any ability to assign priorities to IRQs, we instead treat all IRQ tasks as having an equal priority above all non-IRQ tasks. Each IRQ is assigned a SC such that it is only unmasked when it has available budget and is charged the cost of any kernel entry where the IRQ is delivered. This also provides a convenient mechanism for bounding the rate at which certain interrupts are delivered. This approach could be further extended by allowing IRQs to be assigned specific priorities such that they are also masked while higher priority tasks are executing. This would also allow IRQs to have independent bounds on minimum inter-arrival time in rate monotonic and deadline monotonic schedules. This approach is not considered in depth here as it goes beyond what is strictly required to ensure that the execution time associated with kernel interrupt handling is attributed correctly.

We do not consider multicore systems in any depth in this thesis, so we assume IPIs do not occur. In a multicore system, they would be used to trigger kernel operations across cores for synchronising state and communicating between threads. If a real-time system isolated to a single core of an seL4 system does not receive changes to its scheduling state, changes it its virtual addressing, or communications from other cores via the kernel, then these IPIs are avoided. This would not, however, exclude communication to other cores via shared memory or signalling external cores from the real-time core.

Reserved IRQs are architecture-dependant and are handled by the kernel itself. IRQs that indicate a user-level exception within the architecture's virtualisation mechanisms can be treated the same as any other user-level exception, with the associated kernel entry time attributed to the running task. IRQs associated with external devices managed in the kernel, such as the IRQs associated with managing device MMU faults, are more complex to correctly attribute time to and left out of the scope of this thesis.

The current implementation of kernel time only splits time once for each kernel entry, soon after the most recent point where an external interrupt could be received. All time before this is attributed to the task that was running when the kernel entry began and all time after is charged to the task that is running when the kernel then exits. This approach makes it difficult to account for the cost of kernel entries in any of the timing analysis, as it doesn't strictly associate the kernel costs with any particular task.

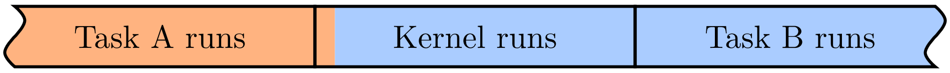


Figure 6: Time charged to tasks for all kernel entries
When a kernel entry occurs, part of each entry is charged to the task that was active when the kernel entry began and part is charged to the task that was active when the kernel entry completed.

To implement the more accurate assignment, we only need to split time at each kernel entry and exit. The time between entry and exit is charged to either an interrupt task, the task that was running when the kernel entered, or the task that was running when the kernel exited. We can also use an overestimate of this kernel entry duration as long as it is smaller than some known worst-case bound.

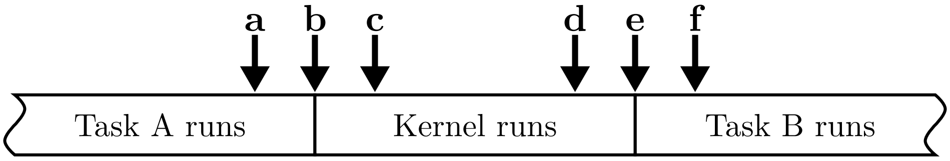


Figure 7: The timing points related to the kernel entry and exit time
The kernel attempts to overestimate its entry time, **b**, by reading the timestamp soon after entering the kernel at **c** and subtracting a worst-case entry time to produce an estimate, **a**, that is no later than the actual entry. The kernel also attempts to overestimate its exit time, **e**, by reading the timestamp soon before exiting the kernel at **d** and adding a worst-case exit time to produce an estimate, **f**, that is no earlier than the actual exit.

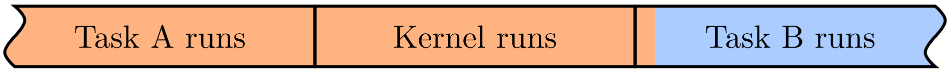


Figure 8: Time charged to tasks for a system call or user-level exception
When a task calls into the kernel or produces a user-level exception, the resulting kernel entry is charged to the task that was current at the point where the kernel entry began.

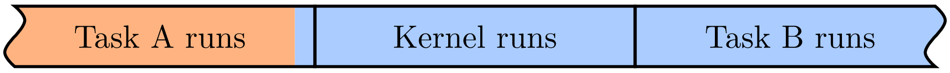


Figure 9: Time charged to tasks for a preemptive task switch
When the timer causes a higher priority task to begin execution, the kernel time is charged to the task that was current at the point where the kernel entry completed.

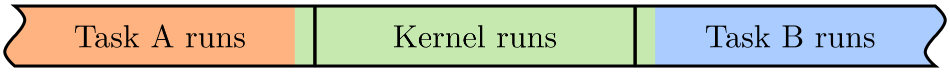


Figure 10: Time charged to tasks for an IRQ with no switch
When the kernel enters due to an IRQ handled at user-level, the kernel time is charged to the SC associated with the IRQ.

The changes to the implementation to track the kernel implementation itself should be minimal, only requiring a read of the current time either at the point the kernel is entered or the earliest point after a interrupt can be received and then immediately before a return to user-level, with an over-assumption kernel entry and exit time added. The changes to bind scheduling contexts directly to interrupts is also likely to be straightforward, only requiring SCs to track when they are associated with an IRQ and for the kernel to maintain a table of SC capabilities associated with the IRQs.

0.6.4 Bounding priority inversion

When a task raises its priority, it must still be charged for all time consumed at a higher priority. With the immediate priority ceiling protocol (IPCP), a task operates at the highest priority of all acquired resources, raising its priority on access and lowering its priority on release. In seL4, we model mutually exclusive resources as threads. When a task accesses a resource, execution switches to the thread of the resource with the resource thread consuming time from the SC of the task. This provides the ability for operations on a resource to operate in a private virtual address space, such that only the IPC interface provided by the resource can be used to access that resource.

The current model of seL4 assigns priorities to threads. When one thread performs a synchronous IPC to another and that thread does not have a bound SC, the SC of the calling thread is *donated* to the receiving thread. The thread that receives the SC executes at its own priority under the bandwidth constraints of that SC. When that thread later replies to the calling thread, the donated SC is returned.

SCs can also be bound to asynchronous IPC objects (seL4 *notification* objects). When a thread with no bound SC receives a notification from an asynchronous IPC object with an SC, that SC is donated to the thread. The SC is later returned when that SC performs a blocking receive operation. This allows a single thread to act both as a resource, receiving synchronous requests from lower-priority tasks, as well as an explicit task at the same priority level. This may be useful if the task implements a driver service that may also communicate with an external device asynchronously and ensures that all operations on that resource are atomic.

seL4 also allows the holder of a capability to a thread to change that thread's priority and it's maximum controlled priority (MCP). In this case, a capability to another thread provides an upper bound on the newly configured priority and MCP as neither can be configured greater than the MCP of the second thread.

This implementation effectively emulates *thread migration*, where a single entity in the scheduler is able to execute across multiple protection domains. This allows for a protection domain to encapsulate access to certain system resources in a particular virtual address space and the implementation of certain routines or services for accessing those services. This has the advantage that the correct protocols required to correctly access the state and resources of a protection domain are always used as only the implementation of the routines within the protection domain can operate on those resources. A task to execute routines across multiple protection domains without requiring synchronised communication to separately scheduled entity in the scheduler that executes within that protection domain.

The fundamental issue presented by this implementation is that implementing resource access, i.e. the ability for a task to raise its priority to access a resource, is conflated with this emulation of thread migration. This is achieved by configuring priorities as part of thread object rather than as part of the scheduling context, allowing a task to raise its priority *only* when it is bound by the same scheduling context.

To ensure that tighter bounds are enforced when a task accesses a resource, we must not allow the scheduling context of that to be transferred in in its entirety to that resource, as this would not enforce the necessary bounds on execution time for access to that resource. Instead, we will separate these two concerns by ensuring that priority is configured directly on the scheduling context and by bounding the scheduling of accesses to resources by scheduling contexts representing those resources. We will still allow this model of thread migration, by allowing a task to pass its scheduling context to a task with no scheduling context, as the entirety of the scheduling bounds are now configured on the scheduling context.

Rather than using TCBs to provide the authority to set the priority of a scheduling context, the scheduling control capability is used as it would also be used to configure the other scheduling parameters of any SC. Each scheduling control capability is assigned a badge of the highest priority it can assign and can be used to mint a copy with an equal or lower priority.

As the bounds for resource accesses must be stronger, we introduce a new kind of scheduling context, a *resource scheduling context* (RSC). Unlike SCs used to represent tasks, a RSC only receives budget via a donation mechanism and never as the result of a job release. A RSC is also configured with a maximum budget which can be set to the minimum blocking time of all tasks at an equal or lower priority.

When time is donated to an RSC, the SC of the task performing the access is also linked to the RSC. When nested accesses occur, a linked list is formed between RSCs. When an SC donates time to an RSC, the donated budget is set for the RSC. When a RSC donates time to another

RSC, the budget is deducted from the source RSC and returned when a matching reply occurs. We do not allow the donation of budget to an RSC with a priority lower than that of the SC that originally donated the budget. Doing so would lead that task to be preempted by lower priority tasks, which in turn can transitively interfere with the tasks at a lower priority that is configured for the SC.

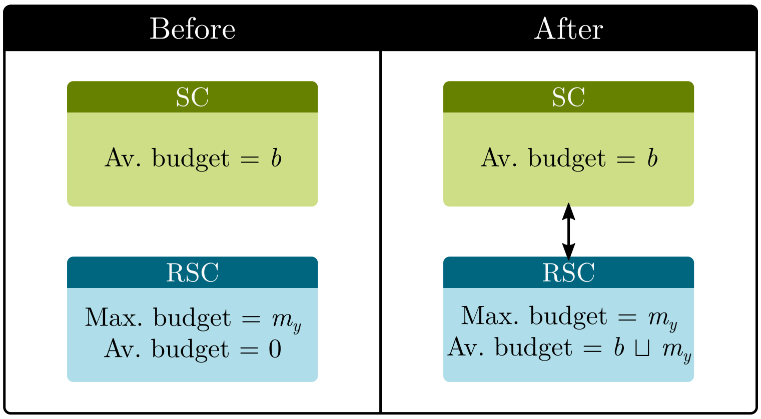


Figure 11: A task with a SC sending to a blocked task with a RSC

When a task with a SC performs a blocking send to a task with a RSC, the RSC is linked to the SC and is given the minimum of the available budget in the SC and the RSC's configured maximum budget as its available budget.

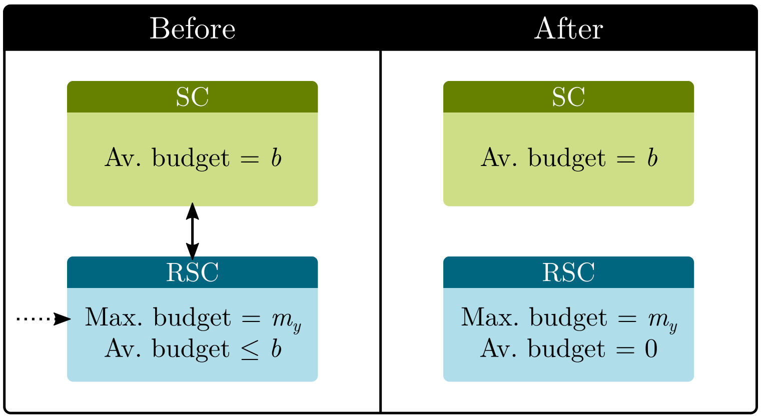


Figure 12: A task with a RSC replying to the task bound to the SC from which budget was originally donated

When a task with a RSC performs any send to a task with a SC to which it is linked, the SC and RSC are unlinked, the RSC is unlinked from any other RSC, and the budget of the RSC is set to 0.

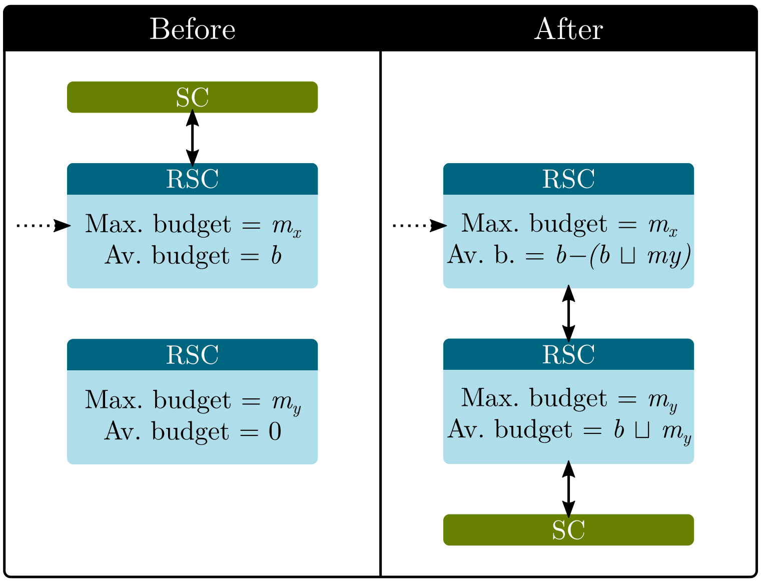


Figure 13: A task with a RSC sending to a blocked task with a RSC
When a task with an RSC performs a blocking send to another task with an unlinked RSC, the two are linked, the SC is unlinked and linked to the receiving RSC; pushing an RSC onto a linked list of RSCs. The receiving RSC is given the minimum of the sender's budget and its own configured maximum as its available budget with any remaining budget being left in the sending RSC.

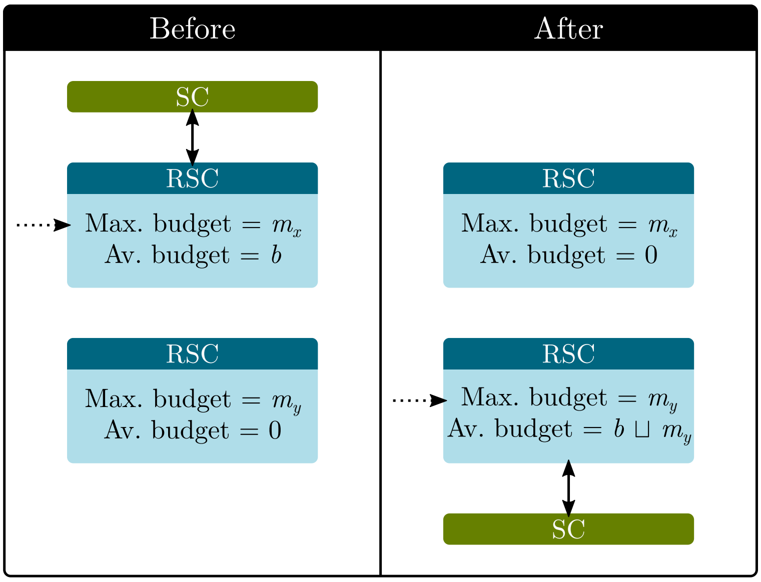


Figure 14: A task with a RSC performing a non-blocking send to a blocked task with a RSC
When a task with an RSC performs a non-blocking send to another task with an unlinked RSC, the SC is unlinked and linked to the receiving RSC and any link to the sending RSC is moved to the receiving RSC; swapping the head of the linked list of RSCs. The receiving RSC is given the minimum of the sender's budget and its own configured maximum as its available budget and the sending RSC has its budget set to 0.

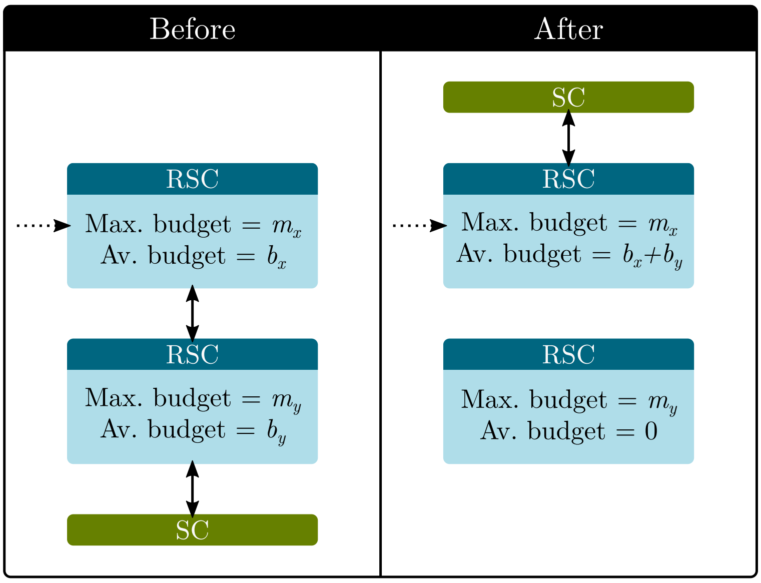


Figure 15: A task with a RSC sending to the task bound to the previous RSC in the list of RSCs

When a task with an RSC performs a send to the task with the previous RSC in the list, the SC is unlinked and linked to the previous RSC and the RSCs are unlinked from each other. This effectively pops the the end off of the linked list of RSCs. The sender's available budget is added back to that of the receiver and the sender's budget is then set to 0.

If a task with an RSC blocks in a manner that does not donate, the SC loses all available budget and the SC is unlinked. This ensure that the task with the RSC can only later continue execution when donated budget from a running task. When a task with an RSC is unblocked while in a queue of RSCs, it is unlinked from that queue and any budget it had retained is returned to the previous RSC.

An advantage to this approach is that timing guarantees of the system now only depend on the set of scheduling contexts that exist, rather than depending on the ways in which they can be assigned to and passed between threads. Further, the ability to alter the guarantees at any particular priority only depends on the components with a scheduling control capability with an equal or greater maximum configurable priority. This makes it substantially easier to ensure the timing properties of any system built with seL4 whilst still allowing soft realtime and non-realtime tasks to be configured independently at lower priority levels with reduced timing guarantees.

Given the above changes of introducing RSCs, bounding budget donated to RSCs, tracing the SC associated with donated budget, and utilising the scheduling control capability to assign priority directly to SCs and RSCs, we can, with minimal effort, ensure that a task never lowers its priority and never executes with a raised priority for longer a bound specified for a given priority level.

0.7 Implementation

To implement our model we must make several modifications to the seL4 kernel, particularly to its mechanisms for time keeping, charging time, inter-process communication (IPC), and managing IRQs.

0.7.1 Sporadic servers

The current implementation of a *scheduling context* (SC) consists of a bounded list of refills and a set of configuration parameters including the SCs period, assigned CPU core, and references to other objects. When an SC is configured, the period and core are set and a single refill with the SCs full budget and a release time of the instant of configuration is created as the only element in the list. The available budget of an SC is the total budget in all refills with a release time before the current time. The kernel uses a separate counter tracking time consumed for the current SC. When the SC currently executing changes, the consumed time is *charged* to the current SC and the next SC is *released*. The charged time is deducted from the first refill in the list and added one period after that refills release time. When an SC is released, all refills with a release time before the time of the SC's release are delayed and merged into a single refill with a release time of the instant the actual release occurred.

If a SC does not have a refill with a release time before the current time, it is placed in the release queue, which is an queue of threads ordered by the release time of the first refill in their SC. When the current time is no less than the release time of the first refill of an SC in the release queue, that SC is removed from this queue.

The implementation effectively treats each change of an SC as release of a task, including changes due to the execution of a higher priority task in favour of a lower priority task. To change this implementation such that it models sporadic server [Sprunt et al., 1989], we only need to change the times at which an SC is released. Rather than releasing on every change of the current SC, we only release an SC at the points where we want to model the release of a job or group of jobs for the task of that SC. As described in subsection 0.6.1, this is any instance where the thread bound to the SC of a task is manually resumed, unblocked from an IPC object, or where an unblocked task's first refill becomes available.

0.7.2 Correct and precise time attribution

Our timing analysis assumes that all execution for a task is bounded by that tasks assumed worst-case execution time (WCET) and its minimum inter-arrival time. As we enforce these bounds by charging budget to a *scheduling context* (SC) that represents the task, we must ensure that only the execution related to that task is charged to the SC of that task. All user-level execution time is considered part of that task's execution. Not all kernel execution time that occurs while a given SC is current is part of that task's execution. As such, we require mechanisms to determine the duration of any kernel entry and to charge that time to the SC of the task that the timing analysis assumed bounded that execution time.

The current seL4 implementation only reads the scheduling timestamp once per entry. Whichever SC is current on entry is charged all time prior to the timestamp and whichever SC is current on exit is charged all time after the timestamp. To enable us to specifically charge the time of the kernel entry itself to a particular SC, we require knowing both the time of entry into the kernel and the time of exit. Both times need to be known at the point where the execution time before the kernel entry and the execution time of the kernel entry itself will be charged to an SC.

Determining the time of entry is easiest done soon after entry, preferably a consistent time after the last user-level instruction is executed. At this point we can read the timestamp and subtract some over-estimate of the worst-case time to get from user-level to reading the timestamp. Determining the time of exit should similarly occur a consistent time before the next user-level instruction is executed. The kernel implementation generally charges time to scheduling contexts as the last operation before returning to user-level. This is already an optimal point in the kernel entry to read the timestamp and add some overestimate for the worst case time between reading the timestamp and user-level execution resuming.

The difference between the estimated entry and exit will be an overestimate of the actual time spent in the kernel. If there is a bound on this execution time, then this can be accounted for in the timing analysis. The bound on such execution times depends on what capabilities are available to a task as it executes with some capabilities allowing a task to execute for longer periods in the kernel. The bound on these durations also impacts the blocking time of higher-priority tasks.

When the kernel is entered due to a system call or user-level exception, the kernel entry duration is charged to the SC that is current on entry. Were the timestamp read on entry only used for charging time to the current SC, we could avoid reading it as the both the time before the kernel entry and the time between the entry and exit is charged to the same SC. The timestamp read upon entry is used for more than just charging time, it is also used determine whether the SC has sufficient budget to complete a kernel operation and to determine if any SCs in the release queue can be removed. As such, this timestamp read cannot actually be avoided.

When the kernel entry is due to a preemption, such as when a SC waiting for the release of a refill can once again run, the kernel entry is charged to the released SC. If this is the same as the SC that is current when the kernel exits, i.e., it has preempted the task that was running when the kernel entered, then all time after kernel entry is charged to the released SC. In this case, we can avoid reading the timestamp before exiting the kernel.

Another case where kernel entries occur is in response to *interrupt requests* (IRQs) from external devices. These kernel entries cannot be guaranteed to be related to the currently executing task, and in general are not. These kernel entries must be charged to SCs that are associated with the IRQs themselves. In order to associate IRQs with SCs, we allow SCs to be bound to IRQs using an IRQ handler capability. The SC then bounds when interrupts can be delivered and is charged for the kernel entries associated with that IRQ. When the SC associated with an IRQ has no available budget, that IRQ is masked until the next refill and the SC is added to the release queue. This allows a bound on IRQ delivery to be configured via the SC bound to that IRQ. To facilitate the presence of IRQ-bound SCs in the release queue, we change it to be a queue of scheduling contexts rather than a queue of thread control blocks (TCBs).

0.7.3 Bounding priority inversion

To allow tasks to access resources at an increased priority in accordance with the *immediate priority ceiling protocol* (IPCP), we must provide a way for higher priority execution to be attributed to a task in a manner that is bounded by the blocking time of all tasks between the baseline priority of the task and the priority of the resource accessed.

The current seL4 implementation allows the *scheduling context* (SC) of a task to be used indefinitely at the priority of a resource that it accesses, effectively placing that task at that priority.

Rather than allow the resource to execute using the SC of the task, we assign the resource a *resource scheduling context* (RSC). A resource SC only executes on budget donated from a SC, either directly or via another RSC. A RSC is also configured with an upper bound on the budget that can be donated. When a RSC has donated budget, it is associated with the SC from which the budget was originally donated and added to a list of RSC that have performed nested donations. Any execution time that would be charged to a RSC is deducted from the RSC's available budget and charged to the associated SC. If the thread associated with the SC resumes running and is associated with a RSC, that RSC is disassociated and looses all available budget.

If a thread associated with a RSC blocks without donating to another RSC or yields, it looses all available budget and is disassociated from the SC that donated the budget. If the thread blocks for a synchronous IPC with another RSC, budget up to the bound of the receiver is donated and the associated SC is transferred. Any budget not donated is kept in the RSC blocked waiting for a reply. When the receiving RSC later responds, any unused budget is returned and the associated SC is transferred back so long the receiving RSC has not been used to receive more donated budget. If a RSC or SC to an RSC at a lower priority than the SC from which budget is originally donated, the SC is disassociated and all budget is lost.

This ensures that once a resource thread suspends, execution for that RSC can only continue if it is invoked by a task at the highest runnable priority, as is required by IPCP, instead of being able to resume execution when a lower-priority task would receive a new job release.

To ensure that a task can never be preempted by lower-priority tasks, i.e., that it lowers its priority, we never allow a SC to donate budget to a RSC executing at a lower priority than the SC as the lower-priority resource could then be preempted by a task that was not considered for the response time of the SC that donated the time.

To ensure that a task only ever executes at the priority assumed by the scheduling logic, i.e. its base priority or the priorities of the resources that it accesses, we configure the priority of tasks and resources directly on their SCs and RSCs. This has the benefit of ensuring that the timing analysis holds based only on the knowledge of which configured SCs and RSCs exist in the system and does not depend on the SCs or RSCs being bound to particular threads with the correct priority. To enable delegated configuration of SCs and RSCs at priorities below that of all tasks and resources with bounded response times, we allow the scheduling control capability to be minted with a badge denoting the maximum priority it can configure. The minted scheduling control capability can only have a maximum configured priority no greater than that of the source capability.

0.8 Evaluation

The tests here are performed on a Hardkernel ODroid C2 with an Amlogic S905 SoC containing 4 ARM ® Cortex ® A53 cores running at 1.5GHz and 2GiB of physical RAM. All threads were pinned to the first CPU core.

0.8.1 Correctly charging preemption time

As identified in subsection 0.6.3, one way in which the execution time of the kernel can be mis-attributed is that a running SC may be charged the cost of the kernel entry to switch to a higher prior task when a job of that task is released.

This test uses a low priority periodic task and a set of high priority periodic tasks that are all out of phase, each with a bound SC. The lower priority SC has a minimum inter-arrival time of 12500µs and a bounded WCET of 8332µs. Each high priority task has a minimum inter-arrival time of 400µs and a bounded budget of 24µs. All tasks will perform a tight loop that increments a counter at user-level while it runs.

As a greater number of high-priority tasks are added to the task set, the number of iterations counted by the low-priority task is reduced. This indicates that there is execution other than that of the low priority task being attributed to that task's SC, specifically the execution of the kernel switching to the higher priority tasks.

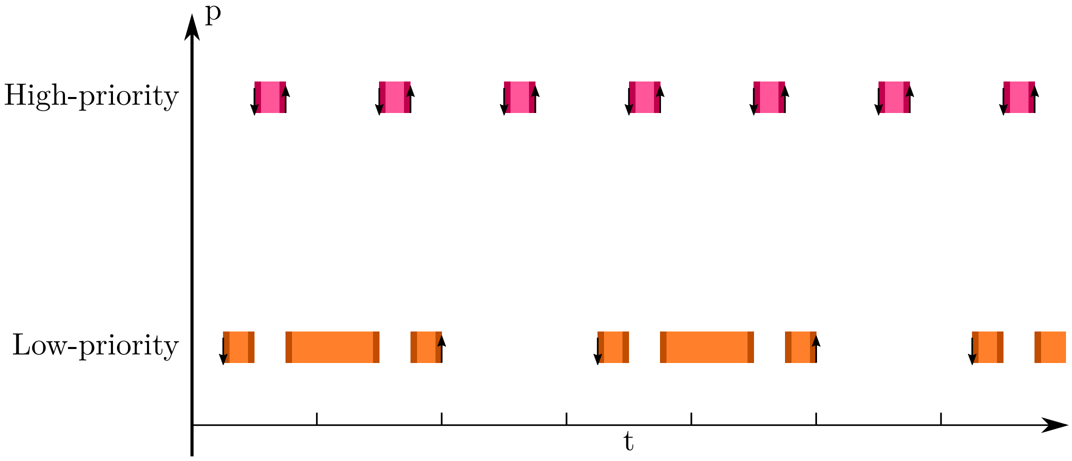


Figure 16: A pair of periodic tasks and the kernel execution time charged to those tasks. Dark regions of the execution trace indicate areas where a task is charged from preemptive kernel execution. The low-priority periodic task expends some of its budget each time it is preempted, reducing the amount of budget available for user-level execution.

With the updated implementation, which charges such kernel execution to the task that is released in such cases, the number of iterations counted be the low-priority tasks remains effectively unchanged as the cost of releasing higher-priority tasks is no longer charged to the SC of the low-priority task.

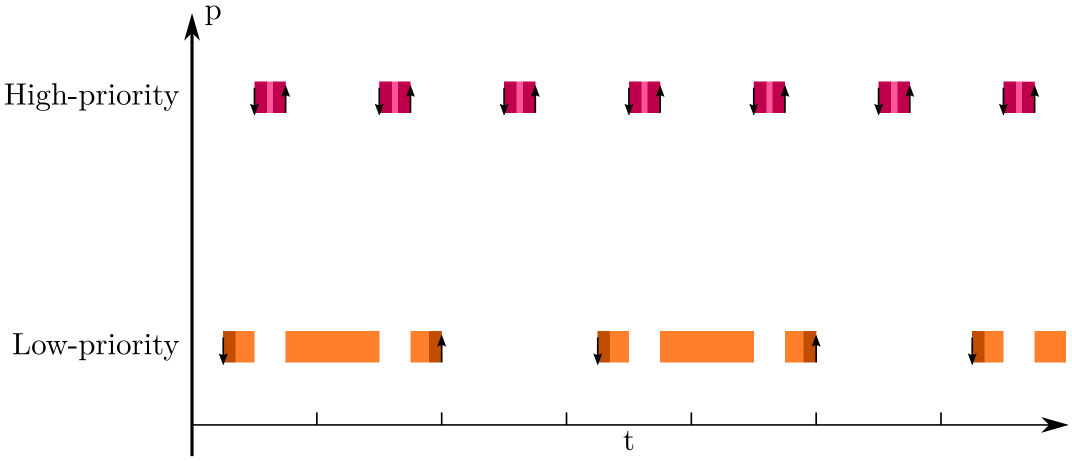


Figure 17: A pair of periodic tasks with kernel execution time charged to the preempting task Diagram is as in Figure 16, except the cost of preemption is always charged to the high-priority task. The low-priority task no longer expends budget when preempted ensuring user-level execution always has same budget available.

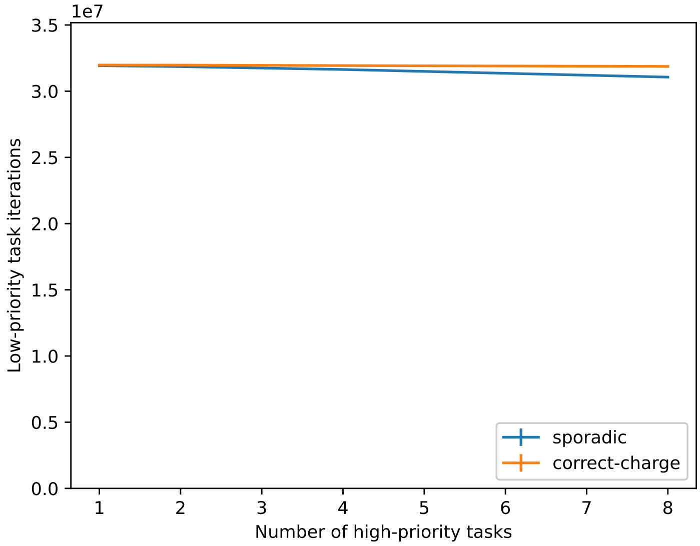


Figure 18: The number of iterations counted by the low priority task.
This is relative to the number of out of phase higher priority tasks that preempt the lower priority task. The *sporadic* dataset indicates the count when only the change implementing sporadic server is applied. The *correct-charge* dataset indicates the count when the preemption time is attributed to the SCs of the higher priority tasks.

0.8.2 Bounded interrupt delivery

As identified in subsection 0.6.3, a second way in which the execution time of the kernel can be mis-attributed is that a running SC may be charged the cost of the kernel entry caused be the delivery of an *interrupt request* (IRQ) from a device.

This test uses a low priority periodic task and a high priority sporadic tasks controlling a pair of timer devices, each with a bound SC. The lower priority SC has a minimum inter-arrival time of 12500µs and a bounded WCET of 8332µs. The high priority task has a minimum inter-arrival time of 500µs and a bounded budget of 240µs, but acts as a sporadic task. The timers are configured such that the second arrives 120 degrees out of phase with the first, i.e., 1/3rd of a period after the first.

The low task will perform a tight loop that increments a counter at user-level while it runs. The timer task will configure both timers and then block, waiting for the IRQ from the second of the two timers. The IRQ for the first timer will be delivered when either the low-priority task or no task is executing.

As the frequency of the timers increases, the number of iterations counted by the low-priority task is reduced. This indicates that there is execution other than that of the low priority task being attributed to that task's SC, specifically the execution of the kernel switching receiving the IRQs and the kernel switching the released to the higher priority task.

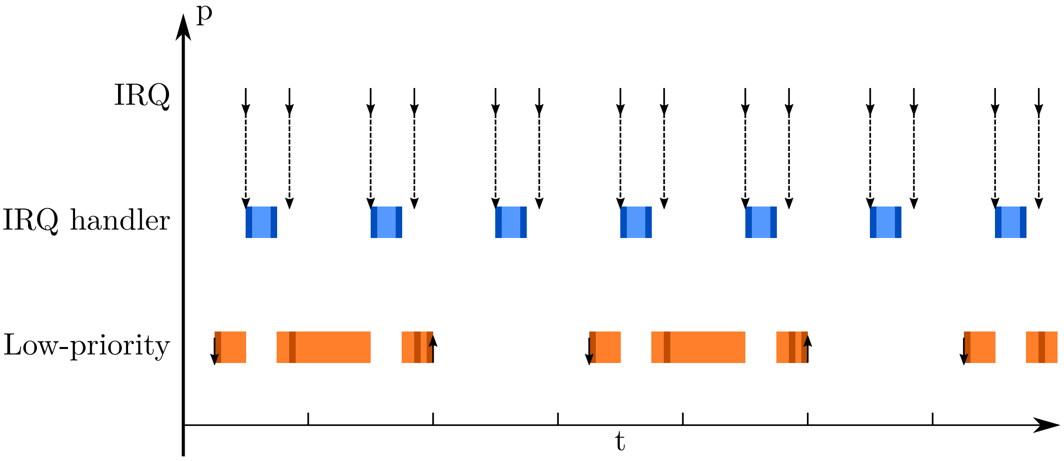


Figure 19: IRQs releasing a handler and preempting a low-priority task
When an IRQ arrives that can wake a high priority task, that task is charged the cost of switching to that task. When an IRQ arrives and does not release a task, the current task is charged the cost of the kernel handling the IRQ.

With the updated implementation charging the released task for the kernel entry used to switch tasks, the number of iterations counted by the lower-priority task decreases by less with the increased timer frequency. There is still a substantial loss in time spent in the loop of the low-priority task as the SC of that task is still charged for the cost of the kernel responding to IRQs that do not release the higher priority task. It is only once we introduce the *resource scheduling contexts* (RSCs) that we see the low-priority task execute the same number of iterations regardless of the activity of the timer and high-priority task.

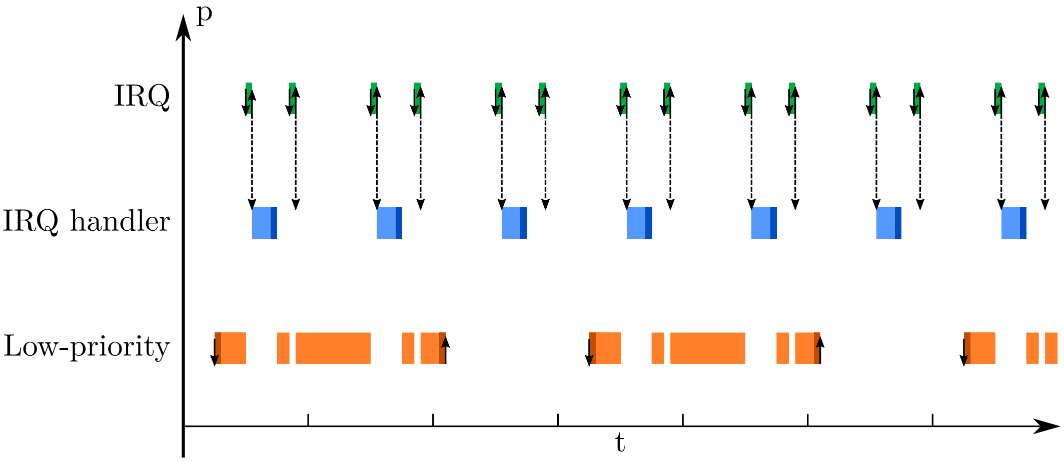


Figure 20: IRQs with assigned scheduling contexts
As in Figure 19, but IRQs can only be delivered when their associated SC has available budget. When IRQs do arrive, the SC for the IRQ is charged rather than any task that could be preempted.

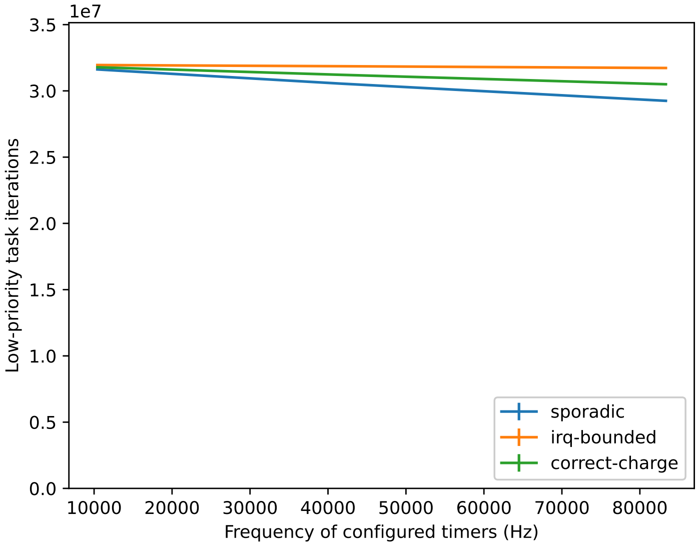


Figure 21: The number of iterations counted by the low priority task.
This is relative to the frequency of the configured timers. The *sporadic* dataset indicates the count when only the change implementing sporadic server is applied. The *correct-charge* dataset indicates the count when the preemption time is attributed to the SCs of the higher priority tasks. The *irq-bounded* dataset indicates the count when IRQs are assigned SCs that are charged for the kernel entry that responds to the IRQ.

0.8.3 Bounded priority inversion

As identified in subsection 0.6.4, we must ensure that a lower priority task cannot access a resource for longer than the blocking time of tasks with a higher priority than the task accessing the resource and a lower priority that the resource itself. If we do not enforce this bound, the response time assumed by the timing analysis will be incorrect.

This test uses a low priority periodic task, a medium priority periodic task, and a high priority resource. The lower priority SC has a minimum inter-arrival time of 12500µs. The medium priority task has a minimum inter-arrival time of 400µs, a bounded budget of 24µs, and a blocking time of 50µs.

The low task will call the resource in a tight loop. The resource will then spin in a tight loop to exhaust all budget. When the budget of the resource is exhausted, a fault handler for the resource thread will reply to the low task and reset the resource. The medium task will also spin in a tight loop. When the medium task exhausts its budget, a fault handler for that thread will log the time at which it ran out of budget as this is essentially the completion time of that job. The medium task is then reset and continues at the release of the next job. At the end of the test we subtract the release time of each job of the medium task from that jobs completion time as recorded by the fault handler.

As the bounded WCET of the low-priority task increases the worst-case observed response time of the medium task also increases. This is due to the entire budget of the low-priority task being raised to the priority of the resource. If the resource thread had no fault handler, it could continue executing on the time for future jobs of the low-priority task, effectively allowing that task to execute at a raised priority indefinitely.

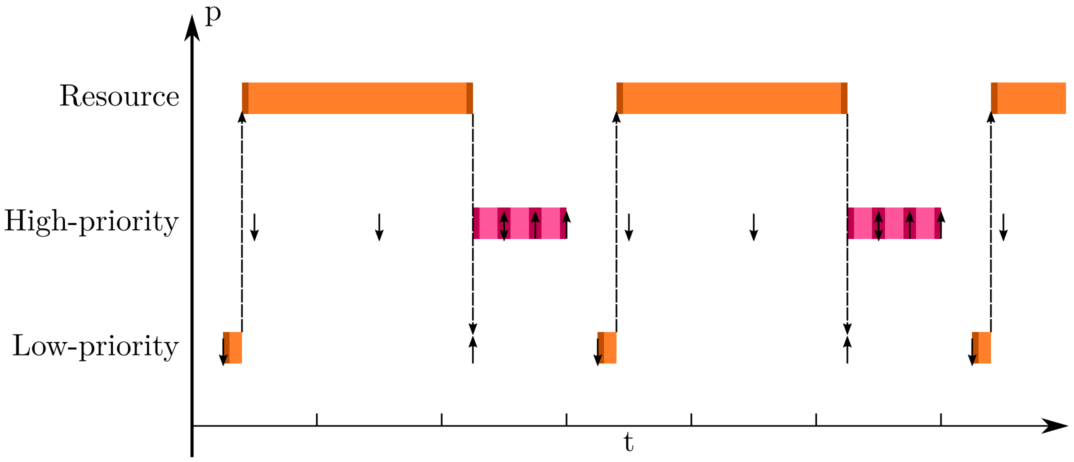


Figure 22: A pair of periodic tasks with a low priority task accessing a high priority resource
When a low-priority task accesses a high-priority resource, its entire set of scheduling bounds are then applied at that higher priority. This allows the low priority task to preempt the high-priority task for its enitre WCET.

With the updated implementation, we assign the resource thread a *resource scheduling context* (RSC) with a budget bound no greater than the blocking time of the medium task. Whenever the low-priority thread calls the resource, no more than the bounded amount of budget is donated. As the budget of the low priority task increases, the worst-case observed response time of the medium priority task remains the same.

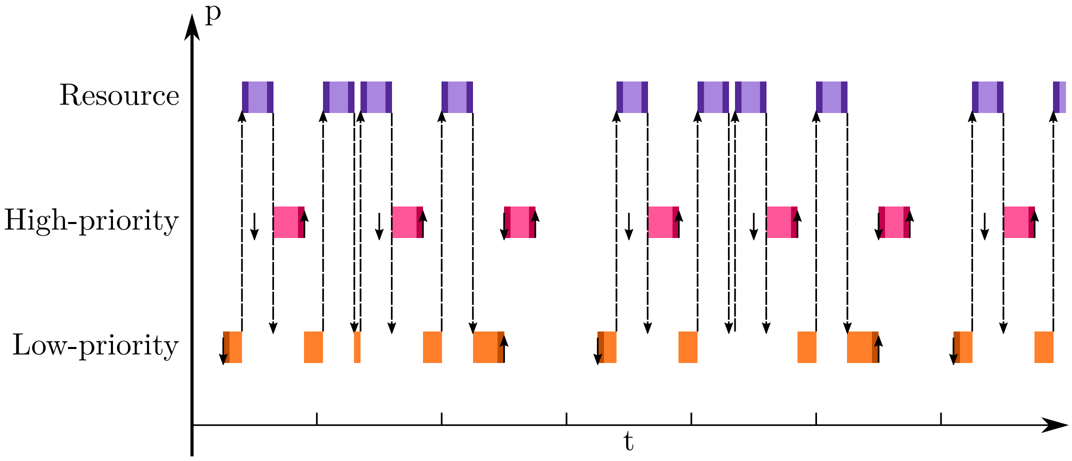


Figure 23: A pair of periodic tasks with a low priority task accessing a high priority resource bounded by a RSC
As in Figure 22, but the resource is bounded by a RSC and is only donated available budget from the low-priority task up to the RSC's configured maximum budget.

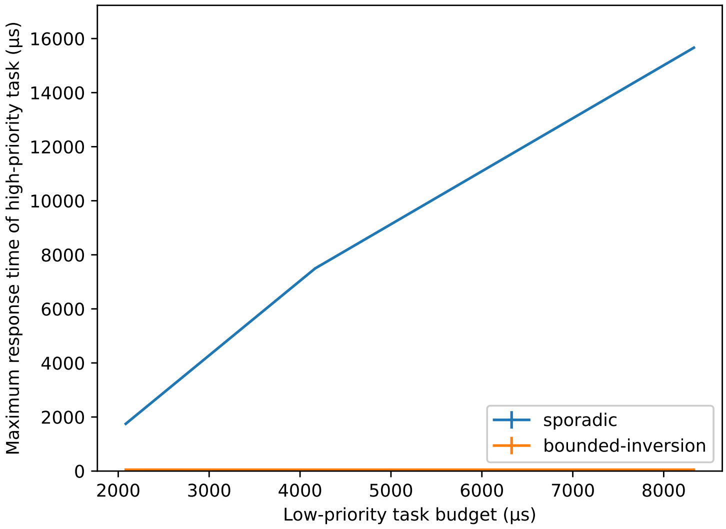


Figure 24: The worst case observed response time of the medium priority task.
This is relative to the bounded WCET of the low priority task. The *sporadic* dataset indicates the response time when only the change implementing sporadic server is applied. The *bounded-inversion* dataset indicates the response time when the resource is configured with a RSC with a bounded budget no more than the blocking time of the medium task.

0.8.4 Summary

These test clearly show that the effect of the existing implementation, even with the modifications to implement the sporadic server model, is quite pronounced and poses a great challenge to predicting scheduling behavior. Inaccuracies in how the implementation charges execution time to SCs and accounts for kernel execution time can greatly reduce the effective execution time of tasks within a system and the lack of execution bound on resource accesses effectively eliminates the assumption of the priority associated with any task. In summary, it would be effectively impossible to produce a system with any guarantee of a response time without thorough verification of every component in the system.

The results of applying the changes proposed by this thesis demonstrate their clear efficacy. By accurately attributing kernel execution time we can remove the need to consider loss of usable budget by a task due to preemption. By allowing a resource to be configured with a bound on execution time of each access, we can ensure that the access to a resource never exceed the blocking time of tasks at lower priorities, regardless of the correctness of scheduling configuration of clients to those resources.

These mechanisms allow us to build a system utilising rate-monotonic scheduling and the immediate priority ceiling protocol and guarantee the response time of the tasks within that system, down to some minimum priority. These guarantees are provided only on the basis of the scheduling contexts configured by tasks with access to a scheduling control capability and do not depend on the implementation or configuration of the rest of the system.

0.9 Future work

seL4 is still far from being a suitable kernel for mixed-criticality or hard-realtime systems. This project has outlined some changes that could be made to resolve this, however there are still many issues that must be resolved before this kernel would be an appropriate choice in this domain, particularly when targeting multicore systems.

As seL4 is a formally verified, many of the mechanisms may not be constructed in such a way as to be easily verified. Some consideration would need to be given to implementations that are compatible with the existing verification work of the kernel.

The systems discussed in this paper only considered those scheduled with rate-monotonic and deadline-monotonic algorithms and the immediate priority ceiling protocol. More work is needed to determine if the policies enforced by this system are sufficiently general that they may apply to other scheduling algorithms and resource access protocols.

Little consideration is given in this thesis to real-time systems configured dynamically at run-time. Although the assumptions of this paper do not include the systems in question to be configured statically, more consideration should be given to the interactions between mechanisms used to reconfigure real-time systems dynamically and the guarantees provided by the scheduler.

The detailed effects of constructing system on multicore processing hardware have not been considered in depth here. More work is needed to determine how inter-processor interrupts (IPIs) used by the kernel can interact with the scheduler's ability to maintain the execution guarantees.

This thesis did not consider mutually exclusive resource access across processing cores. Extending the IPC mechanisms from this thesis to apply to such resource accesses would require further research work and consideration of the assumptions such resource accesses introduce that must be guaranteed by a multicore scheduler.

This thesis did not explore the ability to use the mechanisms discussed to enable assigning priority to *interrupt requests* (IRQs). This would appear to follow naturally from the fact that IRQs would now require bound scheduling contexts and that all scheduling contexts have a configured priority. Implementing IRQ priorities in seL4 requires further work.

While this thesis has demonstrated that seL4 can, with some modification, be used as the basis of robust real-time systems with string timing guarantees, there is still great opportunity to adapt these mechanisms for even more general systems and more work is needed to understand the interactions between the operating systems primitives provided by the kernel and the guarantees that the kernel's scheduler offers.

0.10 Conclusion

In this thesis we presented a set of modifications to the seL4 scheduling extensions for mixed criticality systems. The modifications allow the scheduler to guarantee the assumptions made regarding execution by common scheduling analysis techniques. We have demonstrated that the modifications produce a scheduler that allows for simple assumptions to be made regarding scheduling and timing behaviour such that hard realtime and mixed criticality systems can be ensure guaranteed response times of critical tasks with a minimal subset of tasks requiring verification.

Although the changes proposed produce a set of useful scheduling mechanism, further work is needed to determine a cohesive set of operating system primitives that be offered by a formally verified microkernel, particularly one that can benefit from modern multiprocessor architectures.

Bibliography

Björn B. Brandenburg. A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems. pages 196–206, Rome, IT, December 2014.

Matthew Danish, Ye Li, and Richard West. Virtual-CPU scheduling in the quest operating system. pages 169–179, Chicago, IL, 2011.

Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. Temporal capabilities: Access control for time. pages 56–67, Paris, France, December 2017.

Norman Hardy. KeyKOS architecture. 19(4):8–25, October 1985. URL http://www.cis.upenn.edu/~KeyKOS/OSRpaper.ps.gz.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, et al. seL4: Formal verification of an OS kernel. pages 207–220, Big Sky, MT, US, October 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. doi: 10.1145/1743546.1743574.

Jochen Liedtke. On μ-kernel construction. pages 237–250, Copper Mountain, CO, USA, December 1995.

Jane W. S. Liu. *Real-Time Systems*. Pearson, 2000.

Anna Lyons. *Mixed-Criticality Scheduling and Resource Sharing for High-Assurance Operating Systems*. PhD thesis, 2018. Available from.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. pages 415–429, San Francisco, CA, May 2013. doi: 10.1109/SP.2013.35.

Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the Composite component-based system. Barcelona, ES, November 2008.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. pages 325–340, Nijmegen, The Netherlands, August 2011. doi: 10.1007/978-3-642-22863-6_24.

Brinkley Sprunt, Lui Sha, and John Lehoczky. Scheduling sporadic and aperiodic tasks in a hard real-time system. Technical report CMU/SEU-89-TR-011, Carnegie Mellon University, Software Engineering Institute, April 1989. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=10919.

Mark J. Stanovic, Theodore P. Baker, An-I Wang, and Michael González Harbour. Defects of the POSIX sporadic server and how to correct them. pages 35–45, Stockholm, 2010.

Udo Steinberg, Alexander Böttcher, and Bernhard Kauer. Timeslice donation in component-based systems. pages 16–22, Brussels, BE, July 2010.

Steve Vestal. Fixed-priority sensitivity analysis for linear compute time models. *IEEE transactions on software engineering*, 20(4):308–317, 1994. ISSN 0098-5589.

Marcus Völp, Adam Lackorzynski, and Hermann Härtig. On the expressiveness of fixed-priority scheduling contexts for mixed-criticality scheduling. pages 13–18, Vancouver, Canada, 2013.