
UNSW
AUSTRALIA

**School of Computer Science and Engineering**

**Faculty of Engineering**

**The University of New South Wales**

# Strengthening scheduling guarantees of seL4 MCS with IPC budget limits

by

Mitchell Johnston

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: November 2022 Student ID: z5161146

Supervisor: Prof Gernot Heiser Topic ID: 423

# Abstract

This thesis evaluates some areas for improvement regarding the scheduling behaviour of the seL4 MCS kernel, improving its applicability to mixed-criticality read-time systems. It proposes a set of changes that, when configured correctly, improve system scheduling behaviour regarding budget expiry and budget overrun.

# Acknowledgements

I would like to thank Gernot Heiser and Kevin Elphinstone for their input as supervisors for this project. I would also like to thank the other members of the trustworthy systems group for their general assistance and suggestions. Finally, I would also like to thank my family for their support during this time.

# Contents

# Chapter 1

# Introduction

The *criticality* of a component refers to the severity of failure of a that component. Broadly, low-criticality components can fail with little consequence on the objectives of the system. On the contrary, the failure of high-criticality components have a major impact on the overall system goal. This generally necessitates that high-criticality software be written to a higher standard, such that it is more reliable. Unfortunately, this also leads to a correspondingly higher cost to develop higher-criticality software. However, software of varying criticality must be isolated from one-another to prevent potentially malfunctioning low-criticality software from interfering with high-criticality software.

In the past, software of varying criticality were seperated physically with dedicated hardware, which provided extremely strong isolation. However, this is no longer practical. Firstly, the development of more complex systems requires communication between components, preventing them from being fully isolated. Secondly, in some applications, such as aeronautical or space-flight, weight is crucial and consolidating software functions onto fewer processors is greatly beneficial. Even in applications where weight is not as essential, cost-savings can still be gained.

However, consolidating software of varying criticality onto a single system requires strong isolation. Spatial isolation has been extensively studied and some operating systems, such as seL4, exist which have been formally proved to support it Klein et al. [2009], Murray et al. [2013]. However, temporal isolation, whereby software components cannot cause other components to complete late, is also required for mixed-criticality systems (MCS). These are a form of real-time system, where the correctness of a result depends both on its output and when it is computed. Therefore, these systems must also support temporal isolation.

The seL4 MCS kernel Lyons [2018] was developed with the aim of meeting this requirement. It was built upon the strong formally verified guarantees of the base seL4 kernel, extending them to include a formal treatment of time. In this thesis we aim to further development of the seL4 MCS kernel, improving the guarantees that it offers to mixed-criticality systems built upon it.

This involved investigating and addressing some of the issues previously brought up with the kernel both in previous work Millar [2021] and other identified issues. The issues that we focus

on in this thesis revolves around seL4's support for threads to donate their time allocation to another thread. Prior to our work, there were few restrictions on this time donation, which lead to weaker scheduling guarantees. Specifically, threads could donate insufficient time for the receiver to complete, causing the receiver to experience a timeout exception. Additionally, the receiver of a donated time allocation was not required to return it, which would prevent the sender from continuing execution and causing priority inversion in the system.

As a result, threads involved in time donation required a high level of mutual trust. This interdependence reduced support for temporal isolation, increasing the number of components that must be considered high-criticality. As a result, there is an increase in the cost to develop these systems. We aim to resolve these issues, moving towards a dependable system that enforces scheduling behaviour in a predictable and reasoned manner, while retaining the existing spatial isolation guarantees.

Specifically, we present two changes that improve the scheduling properties of inter-thread communication. The first change guarantees a thread a minimum quantity of budget, preventing timeout exceptions from occurring. The second change proposes a mechanism to limit priority inversion from inter-thread communication, while also reducing the level of mutual trust required by threads.

We have successfully implemented both proposed changes. When properly configured, the first change completely eliminates timeout exceptions, while imposing at worst a 13% overhead on the IPC path. With the second change, we successfully restricted the amount of donated time that can be consumed before it is guaranteed to be returned to the sender. However, this change introduced larger IPC overheads, up to 35% in the worst case.

The remainder of this report is structured as follows:

- In Chapter 2 we will cover the background of real-time systems and scheduling, microkernels and mixed-criticality system.
- In Chapter 3 we will look at other work related to mixed-criticality systems and specifically the seL4 MCS kernel.
- In Chapter 5 we will provide an assessment of some limitations with the seL4 MCS kernel.
- In Chapter 6 we will present our design and the rationale behind the decisions chosen.
- In Chapter 7 we will review the efficacy of our design, showing it provides the desired guarantees, while not excessively compromising performance.
- In Chapter 8 we will conclude this thesis and present directions for future work.

# Chapter 2

# Background

## 2.1 Mixed Criticality Systems

*Criticality* refers to the severity of failure of a system component. If a high-criticality component fails, the ability of the system to accomplish its objective will be significantly degraded. On the other hand, if a low-criticality component fails, the impact on the success of the system objective will be minimal.

For some highly-critical systems, all the processes and their properties may be known in advance. This means it is possible to provision the system such that all the processes always have sufficient resources to succeed. However, the processing requirements of software vary, so a system provisioned for the worst-case resource requirements may have spare capacity under typical circumstances. It may be desirable to run other, lower-criticality tasks using this spare capacity. However, during atypical circumstances, there will then be insufficient processing capacity to successfully run all processes. Further, a system should support isolation properties such that the high-criticality tasks can be guaranteed to succeed without requiring any assumptions of the lower-criticality tasks.

This is the core of the *mixed-criticality system (MCS)* problem: Allowing for a system to support processes of varying criticality, while ensuring that high-criticality tasks can still succeed when there are insufficient resources available for all tasks.

In the following sections we will formalise many of the concepts alluded to here.

## 2.2 Real-time Systems

The distinguishing feature of real-time systems is the *non-fungibility* of time: that distinct intervals of time are not equal, even if they are of equal length. Consider the example of an autonomous vehicle in danger of crashing. If some software responsible for obstacle avoidance

is executed beforehand, the crash can be averted. However, if the obstacle avoidance program is run after the crash, then regardless of its output, the crash would still have occurred.

Informally, the non-fungibility of time means that it is important both how much execution time software receives, along with when it receives that execution time. Note that execution time has specifically been referred to here, but access to other system resources can also be time-dependent and will be covered in more detail in Section 2.7.

In real-time theory, the term *task* refers to an abstraction of a series of logically related executions. Tasks are modelled as an infinite series of *jobs*, which each represent a piece of computation with a *deadline*, which is when the job must be completed by. Jobs have the following properties:

- *Release time:* The instant when a job is ready to run.
- *Execution time:* The amount of processing time that a job requires to complete successfully.
- *Worst Case Execution Time (WCET):* An upper bound on the amount of processing time a job requires to complete successfully.
- *Response time:* The duration from a jobs' release until it is completed.
- *Blocking time:* The time that a job is preempted from execution for reasons other than the execution of higher priority tasks.
- *Deadline:* The time by which a job must be completed.

There are three broad categories of deadlines in real-time systems: hard, soft and best-effort:

- Tasks with *hard* deadlines are considered incorrect if the deadlines is missed.
- Tasks with *soft* deadlines can still be considered partially correct if the deadline is missed. However, the usefulness of the result typically diminishes the more the deadline is missed by.
- *Best-effort* tasks do not have temporal requirements and commonly run as background tasks in a real-time system. However, for these tasks to succeed they still require processing time, so starvation must be avoided.

Tasks can also be categorised by the bounds between their release times into *periodic*, *aperiodic* and *sporadic* tasks:

- Periodic tasks have a fixed interval between the release of consecutive jobs, known as the *period*. These tasks commonly have hard deadlines. These can be *implicit*, where the deadline is equal to the next job's release time. Alternatively, the deadline can be *constrained*; where the deadline is defined relative to the release time, but must be prior to the next job's release. Finally, the deadline can also be *arbitrary*, where the deadline is defined relative to the release time, but need not be prior to the next job's release time.

- Aperiodic tasks are released randomly, with no useful upper or lower bound on the time between subsequent releases.
- Sporadic tasks are also released randomly, with no upper bound on the time between subsequent releases. However, there is a lower bound, known as the *minimum inter-arrival time.*

Before continuing, we wish to distinguish between criticality and time-sensitivity. We recall that criticality is, broadly speaking, the importance of a task. Time-sensitivity is how urgently a task needs to be completely, the length of time between its release and deadline. While both of these are properties of tasks, they are not directly related. High-criticality tasks are not necessarily urgent, and urgent tasks are not necessarily highly-critical, though they can be.

As an example, we consider a system running both a control loop and a network driver. The control loop operates an important, but slow moving, piece of machinery. This piece of machinery must avoid physical collisions to avoid costly damage. Therefore, this control loop is not highly urgent, as the machine moves slowly and there is plenty of time to avoid a collision. Nevertheless, the control loop is highly-critical, as if it fails to prevent a collision, costly damage will occur. In contrast, a network driver is urgent, as the timings involved in high-speed networking are very small. However, the impact of missing a single, or even multiple network packets is minor, and therefore the network driver is considered low-criticality,

## 2.3 Formal Verification

Formal verification describes a process where a system undergoes a mathematical proof to establish that it behaves as specified. It involves proving a correspondence between a high-level abstract specification of behaviour and the low-level implementation. Formal verification provides an extremely strong guarantee that the implementation will function according to the specification.

## 2.4 Operating Systems

An *operating system* is a software system that serves two broad goals. First, it acts as an abstraction for hardware, presenting a common interface to application. Its other role is to manage finite resources, allocating them to user processes. On hardware which supports multiple privilege levels, the *kernel* is the portion of the operating system that runs at the highest privilege level.

The kernel can use its privilege to separate threads of execution into *protection domains*, where threads have limited access to system resources, for example only having access to a subset of physical memory. This serves to isolate threads, preventing a thread from unauthorised access or from tampering with another thread's data. However, there are many cases where threads need to share information. To support this without compromising isolation, the kernel transfers

messages between protection domains on the behalf of user threads. This functionality is known as *inter-process communication (IPC)*.

Many operating systems in common use today, such as Windows, MacOS and Linux are *monolithic* kernels. This means that the majority of operating system functionality is supported by code running within the kernel. In contrast, *microkernels* aim to remove as much functionality from the kernel as possible.

They were characterised by Liedtke [1995] as *"A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."* This is known as the *minimality* principle.

The services that have been removed from the kernel must are instead implemented at user-level. Other threads can invoke these services over IPC, fulfilling the role of a remote procedure call. To facilitate this with high performance, microkernels must support very efficient IPC.

## 2.5 Trust

*Trust* refers to the level of reliance in the correct operation of a system component. Trusted components must operate correctly for system success, while untrusted components can fail without compromising the overall system objective. For example, the operating system kernel has full access to the system and can cause failures in any task by overwriting memory, for example. Therefore, the kernel must be highly trusted.

These are examples of the *trusted computing base* of an application, which is the set of all software and hardware that must operate correctly for an application to function correctly. The trusted computing base of an application is therefore as critical and as trusted as the application itself.

The counterpart to trust is *trustworthiness*. A component is trustworthy if it can be relied upon to operate as expected. The trustworthiness of a component can be established by, in increasing strength: testing, certification and formal verification. However, these processes are generally quite costly to perform, with higher levels of trustworthiness being correspondingly more expensive and time-consuming.

The previously mentioned worst-case execution time is an element of trustworthiness. For example, a completely untrustworthy program may contain an error that causes an infinite loop. Such a program would run forever, therefore it must be assigned an infinite WCET.

We also outline the relationship between criticality and trust. High-criticality tasks, due to their importance to mission success, must be trusted. Generally, high-criticality tasks should also be trustworthy. However, this relies on the high-criticality components having undergone a process that makes them trustworthy, which is not guaranteed.

External enforcement of various properties, such as a WCET, potentially by the system kernel, can reduce the trust that system components need to have in one another. This has the benefit of reducing the trust required of various system components, making it easier and cheaper to develop highly-critical systems.

## 2.6 Scheduling

The goal of scheduling is to determine which tasks to run at which time. This is generally accomplished by assigning a *priority* to each task and then executing the highest-priority task that is ready to run. Priority is not directly equivalent to either criticality or time-sensitivity. Rather, it is instead derived from both, the particulars of which is determined by the type of scheduling algorithm used.

To introduce scheduling, we begin by considering a non-MCS example. We assume we have a collection of periodic tasks where their worst-case execution times and period between job releases are known. We also assume that these tasks are trustworthy and that the tasks will honour their WCET and period, which for the *ith* task we assign the variables $C_i$ and $T_i$ respectively. A collection, or set of tasks is considered *feasible* if it's possible to schedule them such that all tasks always receive sufficient execution time to complete before their deadlines. The *utilisation* of a specific task $i$ is equal to $C_i/T_i$ and represents the proportion of execution time that the task requires. This introduces an upper bound on the feasibility of task sets. The sum of utilisation of all tasks in the system, known as *total utilisation* must be less than or equal to one for each processor in the system, represented in Equation 2.1.

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.1}$$

Otherwise, there is simply insufficient execution time to satisfy all tasks in the system.

When the inequality in Equation 2.1 does not hold, we consider the system to be *overcommitted*. However, the WCET covers a task's worst-case requirements and under typical circumstances, tasks may require less execution time than their WCET. Therefore, an overcommitted system may still be able to satisfy all of its task's execution requirements. However, if all, or even some of the tasks require their full WCET, then there will be insufficient execution time to satisfy all tasks, leading to *overload*.

Returning to consider MCS, the assumption that all the tasks were trustworthy is too strict a requirement. To allow for tasks with varying levels of trustworthiness to run on a single system, we require some form of *temporal isolation*, where one task cannot cause a temporal failure (a deadline miss) in another task. However, on an overcommitted system, this is unfeasible as the system cannot guarantee that all tasks will always meet their temporal requirements. Instead, we require *asymmetric protection*, where low criticality tasks cannot cause failures in high-criticality tasks.

Due to the cost of establishing trustworthiness, lower-criticality tasks are generally not as trustworthy as higher-criticality tasks. As a result, high-criticality tasks cannot be required to trust

lower-criticality tasks. This means that *cooperative* scheduling protocols, where threads voluntarily yield to one another, are inherently unsuitable for MCS, due to the required trust between tasks. This requires *pre-emptive* scheduling, where the scheduler can pre-empt executing threads, allowing for enforcement of threads execution time.

### 2.6.1 Periodic scheduling

For sets of periodic tasks, there are two broad categories of scheduling algorithms:

- *Fixed priority* scheduling, whereby all jobs in a task are assigned the same priority.
- *Dynamic priority* scheduling, where jobs are individually assigned priorities.

Fixed-priority scheduling algorithms assign fixed priorities to tasks. There are two main basis for these priorities:

- *Fixed-priority rate monotonic* scheduling allocates higher priorities to tasks with higher *rates*, where the rate is defined as the inverse period $\frac{1}{T}$.
- *Fixed-priority deadline monotonic* scheduling allocates higher priorities to tasks with shorter deadlines relative to release times.

Fixed priority scheduling can suffer from *schedulability bound* issues Liu and Layland [1973], which is an upper limit on the CPU time that is spent executing tasks in the system. It has been proved that for large task sets, the maximum CPU utilisation of fixed priority scheduling approaches 70%, which is not ideal.

An example of a dynamic priority scheduler is *earliest deadline first* (EDF). A key benefit of dynamic scheduling is that their schedulability bound is 100%, when not considering algorithmic overheads. EDF scheduling assigns higher priorities to jobs that have deadlines closer to the current time. EDF scheduling is theoretically optimal where a single resource can be scheduled pre-emptively.

A key difference between FP scheduling and EDF scheduling is how they behave under overload. FP scheduling will continue to run jobs with high priorities, which can lead to complete starvation of jobs with lower priorities. In contrast, EDF scheduling will allow all jobs to make progress, but at a lower rate. This means that FP scheduling will deterministically drop jobs from lower-priority tasks, while EDF will drop jobs from all tasks. Either behaviour can be desirable depending on the situation, though generally deterministic behaviour is preferred.

As FP scheduling predictably prefers high priority tasks under overload, one possible approach is to simply assign high priorities to high-criticality tasks. Unfortunately, this approach does not work well in mixed-criticality systems. We consider the example of a control loop and a network driver that we presented in Section 2.2. This could lead to circumstances where the

system is not overloaded, but prioritising the high-criticality, non-urgent control loop causes the network driver to miss some of its deadlines. While the network driver is of a lower-criticality, we would still prefer to meet all the deadlines in the system where possible. Therefore, simply assigning high priorities to high-criticality tasks is not suitable for mixed-criticality systems.

### 2.6.2 Sporadic scheduling

We now investigate algorithms designed to encapsulate sporadic tasks in a manner that is compatible with scheduling analysis for periodic tasks. We consider a sporadic task with a minimum inter-arrival time and WCET. We assign all the following algorithms a period $T$ and budget $C$ equal to those, respectively.

First, *periodic servers* store sporadic jobs in a queue. The server *activates* at intervals separated by the server's period and then processes the queued jobs. The server can execute for up to its budget, otherwise it is pre-empted and must resume execution at its next activation. Therefore, this server has a schedulability impact identical to a periodic task with an equivalent budget and period. However, it can lead to long response times for sporadic jobs, especially if a job is released just after the server has finished its activation. The job would then need to wait the servers full period until its next activation.

An improvement is the *sporadic server*, introduced by Sprunt et al. [1989]. However, the original sporadic server algorithms presented in the Sprunt paper contained defects. We will instead focus on the models presented in Stanovich et al. [2011].

We begin by considering the *primitive sporadic server*. Instead of activating on a consistent periodic basis, these only activate when there are jobs ready to be processed. However, the server must still obey the *sporadic constraint*, whereby the time between two subsequent executions of the server $t_i$ and $t_{i+1}$ must be greater than or equal to the period, i.e. $t_{i+1} - t_i \geq T$. This is to ensure that the worst-case pre-emption caused by a sporadic server is no greater than would be caused by an equivalent periodic task.

We illustrate this with an example in Figure 2.1. Both servers experience a job release and activation at time $t_0$, which causes them to execute for some time and complete the job. Then, one period (T) later at time $t_0 + T$, the periodic server activates, finds no waiting jobs and forfeits its remaining execution time. At time $t_2$, a job is released. The sporadic server can activate immediately and process it, as the sporadic constraint is satisfied $t_2 - t_0 \geq T$. However, the periodic server cannot activate again until $t_0 + 2T$. This leads to longer response times with the periodic server.

The *ideal sporadic server* is a generalisation of the sporadic server model. It consists of a set of primitive sporadic servers, all with equal period and unit budget. This unit budget is the smallest possible division of time in the system. An ideal sporadic server with a given budget and period is modelled as a set of these unit-servers with equal period and a sum of unit budgets equal to the overall budget. This allows for the sporadic server to divide up its budget, while the worst case interference still occurs when all the tasks are released at once. Therefore, for

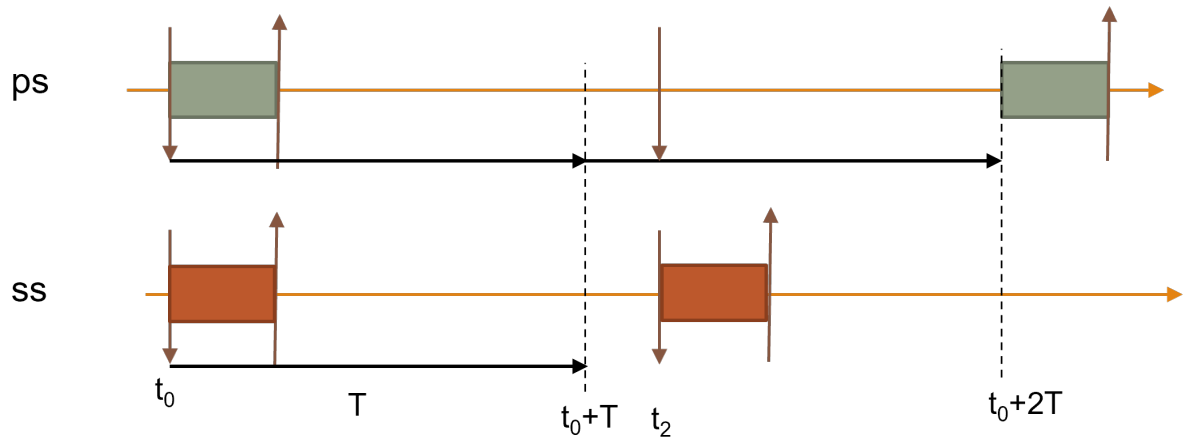


Figure 2.1: Periodic servers (ps) vs sporadic servers (ss). Downward arrow represent job releases, while upward arrows represent job completion. Dashed lines are used to indicate moments in time. Horizontal black arrows represent 1 period T

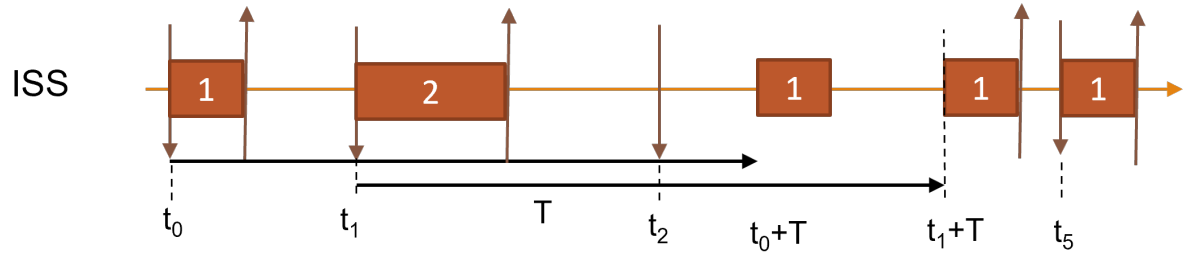


Figure 2.2: Ideal sporadic server. Numbers over blocks of execution represent time taken.

schedulability analysis, the ideal sporadic server can still be treated as a periodic task with equivalent period and budget.

We illustrate this with an example in Figure 2.2. We consider an idealised sporadic server with a budget of 3 units and some period T. For this example, there is no relation between the task's minimum inter-arrival time and the period. At time $t_0$, a job is released that requires 1 unit of time. That completes, leaving the sporadic server with 2 units of budget remaining. Then, at time $t_1$, a job is released requiring 2 units of processing time, which completes, depleting the budget of the sporadic server. At time $t_2$ another job requiring 2 units of execution time is released, but as the server's budget is exhausted, it cannot run. One period after $t_0$, at time $t_0 + T$, the budget is replenished and the server can run. Note however, that as only a single unit of budget was expended at $t_0$, only a single unit is replenished at $t_0 + T$ and this limits the server's execution. It cannot complete the job until time $t_1 + T$, one period after $t_1$ when the two units of budget are replenished. The server consumes one unit to complete the job. The second unit remains as the server's budget, allowing it to execute immediately when a job is released at $t_5$.

However, ideal sporadic servers can lead to many fragmented budget replenishments which

causes significant overhead to track. So, in practice, sporadic servers commonly have a limit on the number of replenishments that are tracked at any one time. If a server runs out of replenishments, any excess budget is generally forfeited.

#### Constant-bandwidth server

Constant-bandwidth servers are a variant of sporadic servers that strictly enforce the *sliding-window constraint*. These also possess a budget and a period. The sliding window constraint means that over any length of time equal to the period, the server can consume no more CPU time than their budget. This is in contrast to sporadic servers, where, as seen above, they can preserve their budget and exceed the sliding-window constraint in some circumstances.

### 2.6.3 Criticality scheduling

All of the aforementioned scheduling algorithms have only taken into account the periods and deadlines of tasks, not their criticalities.

One modelVestal [2007] involves each task having a vector of WCET's. Each of these WCET's correspond to a particular criticality level. As the criticality level and required level of assurance increases, the WCET requirements also increase. Further, each task also has a maximum criticality level, which if exceeded by the system criticality level, then that task will no longer be scheduled. This model can be interpreted as a form of graceful degradation, where increasing the criticality level and required level of assurance sheds tasks such that the task set is feasible with the WCETs at that criticality level.

This model lends itself to schedulers that implement a *criticality mode-switch*. There are multiple forms, but one version involves differentiating tasks into low and high criticality. When the criticality mode-switch is activated, low-criticality tasks are either prevented from running altogether or have their priorities dropped below all high-criticality tasks. This is a coarse mechanism for managing an overloaded system.

However, we note that multiple WCET values do not necessarily neatly translate to real-world systems. Alternate models instead require sufficient tepmoral isolation, such that the scheduling properties of a particular criticality level can be determined without making any assumptions on lower-criticality components.

## 2.7 Shared resources

Tasks in a system may need to share resources, such as shared memory. To maintain consistent state, in some circumstances only a single task can access the shared resource at a time. Otherwise, the interleaved execution of multiple tasks could lead to inconsistency and race conditions.

This access restriction is known as *mutual exclusion*, whereby only a single task can access a resource at a time. Sections of code where resources must be accessed mutually exclusively are referred to as *critical sections*. Generally, once started, critical sections must be run to completion before the resource can be used by another task. This remains true even if the task is pre-empted.

A common approach to ensuring mutual exclusion is locking, whereby tasks coordinate to ensure that only a single task accesses the resource at a time. However, basic locking is an inherently cooperative protocol that requires tasks to trust each other. All tasks in the system have access to the shared resource at all times, the lock only serves to help the tasks coordinate. A faulty or malicious task could access the shared resource without the lock, causing corruption and leading to the failure of other tasks. Given that tasks of varying criticality may share a resource, this would violate the asymmetric protection requirement. In the next section, we introduce an alternate mutual-exclusion protocol that will allow for the required protection.

### 2.7.1 Resource servers

*Resource servers* are a means of encapsulating shared resources to allow tasks to safely share a resource. The resource server is a dedicated process that has sole access to the shared resource, with all other tasks having no access to the resource. If a task needs to access the resource, it executes a remote procedure call via inter-process communication into the resource server. The resource server then operates on the resource on the tasks behalf. This mean that with the resource server model, tasks must trust the server, however they do not need to trust the other tasks that share the resource.

### 2.7.2 Priority inversion

However, mutual-exclusion protocols, whether based on locks or resource servers, can introduce *priority inversion*. This is where a lower-priority task prevents a higher-priority task from running, which inverts the meaning of priority.

We will explore an example using locks for simplicity, however the principle is identical with resource servers. We consider a system containing 3 tasks, $p_1$, $p_2$, $p_3$, with priorities corresponding to their subscripts, with larger numbers corresponding to higher priority. Tasks $p_1$ and $p_3$ share a resource $r$, while $p_2$ does not.

Consider a situation, illustrated in Figure 2.3 where initially $p_1$ is running and acquires a lock on $r$. Task $p_3$ then releases a job at time $t_1$, which pre-empts $p_1$ and then attempts to acquire the lock on $r$. This fails, as $p_1$ holds the lock, so $p_3$ blocks at time $t_2$. This allows $p_1$ to continue running until $t_3$, when $p_2$ releases a job, pre-empting $p_1$ as it has a higher priority. This job completes at $t_4$, allowing $p_1$ to resume execution until $t_5$, when it releases the lock on $r$. This causes $p_3$ to become unblocked, immediately pre-empting $p_1$. $p_3$ then runs to completion at time $t_6$, at which point $p_1$ begins executing again until completion.

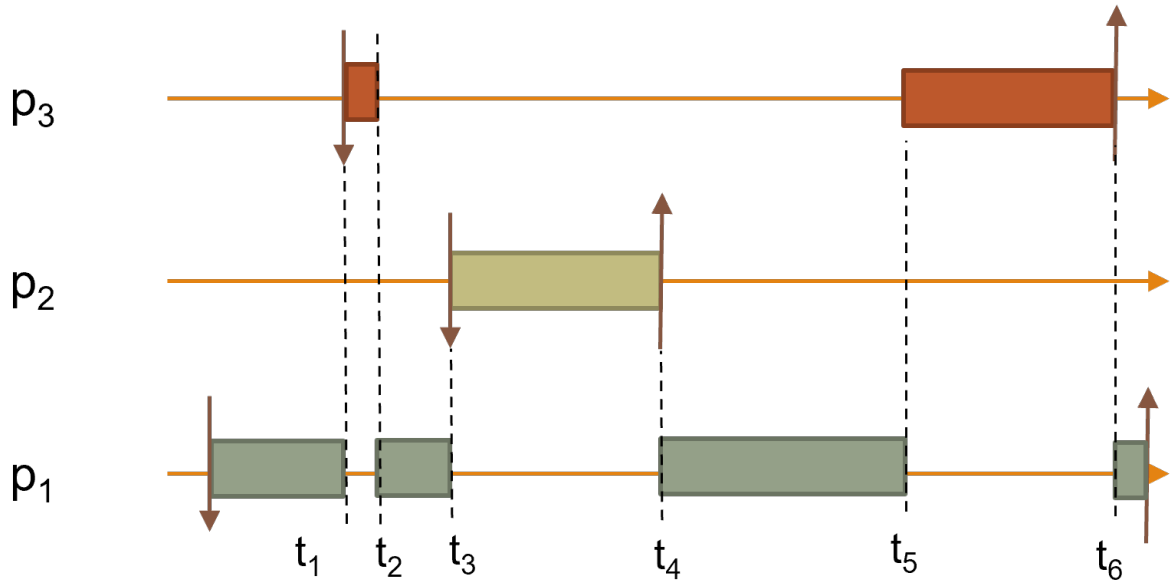


Figure 2.3: An example of priority inversion. Downward arrow represent job releases, while upward arrows represent job completion. Dashed lines are used to indicate moments in time.

There are two types of priority inversion present in this example. Firstly, during the time between $t_2$ and $t_3$, due to holding the lock on resource $r$, $p_1$ is preventing $p_3$ from executing, despite $p_3$ having a higher priority. This from of priority inversion is bounded by the longest critical section involving $r$, as once the lock on $r$ is released, $p_3$ will be able to run. The second form of priority inversion is where $p_2$ delays $p_1$ from executing, from time $t_3$ to $t_4$. This can lead to a potentially unbounded delay in the execution of $p_3$, especially considering that there could be multiple tasks with a priority between $p_1$ and $p_3$ that would collectively prevent $p_1$ from finishing its critical section.

We now formalise this concept of tasks delaying the execution of other tasks. *Blocking time* is the time that a task spends pre-empted for reasons other than the execution of a higher-priority task. It is desirable to reduce the amount of priority inversion that occurs, specifically the second form where the priority inversion may be unbounded. In the next section we investigate several protocols with this aim.

### Real-time locking protocols

In this section, we introduce four protocols aimed at reducing priority inversion:

- *Non-prempetive critical sections protocol (NCP)*: Under this protocol, pre-emption is completely disabled while any task is in a critical section. While simple to implement, this can lead to a large amount of unnecessary blocking of higher-priority tasks.

- *Priority inheritance protocol (PIP)*: With PIP, if a task requires a resource locked by a lower-priority task, it donates its priority to the lower-priority task. This allows the lower-priority task to finish its critical section and release the lock, reducing priority inversion. However, this approach is complex to implement with costly system overheads, as there can be nested priority inversion, additional pre-emptions and care must be taken to avoid deadlock.

- *Immediate priority ceiling protocol*: Under IPCP, all resources are assigned a ceiling priority, which is 1 greater than the highest priority of all the tasks that access it. Whenever a task accesses a resource, it immediately runs at the ceiling priority. However, this requires that all task's priorities be known in advance, though this is required for schedulability analysis regardless. Additionally, it can lead to unnecessary blocking of intermediate priority tasks that do not require the resource. We also note that if the ceiling priority is the highest in the system, this protocol is equivalent to NCP.

  We now illustrate the IPCP protocol with the example in Figure 2.4, similar to the example presented previous in Figure 2.3. A new row has been added above with the label $r$, to represent the ceiling priority of resource $r$. This ceiling priority is 4, because that is one greater than the priority of the $p_3$, the highest priority task that accesses the resource. Once again, $p_1$ releases a job, however, once it acquires the lock on resource $r$ at time $t_0$, its priority is promoted the priority of $r$. Thus, at times $t_1$ and $t_2$, when $p_3$ and $p_2$ release jobs, they do not pre-empt $p_1$. However, at time $t_3$, $p_1$ releases the lock and returns to its original priority. This allows $p_3$ to then pre-empt $p_1$ and begin executing. When $p_3$ then acquires $r$ at time $t_4$, its priority is also promoted to the priority of $r$. It then runs to completion at $T_5$, then allowing $p_2$ to run. Finally, once $p_2$ completes at time $t_6$, $p_1$ can resume execution and complete.

  This protocol thus prevents the unbounded blocking that could occur from the intermediate priority task $p_2$.

- Finally, the *original priority ceiling protocol (OPCP)* is a combination of the approaches of PIP and IPCP. Resources retain a priority ceiling, exactly as in IPCP. However, tasks do not have their priority boosted immediately upon acquiring a resource. Instead, they are only boosted when another tasks attempts to acquire that resource, though they are increased to the resource's ceiling priority rather than inheriting the other tasks priority. This reduces the level of unnecessary pre-emption that occurs. Additionally, it also introduces a *system ceiling*, which is the highest priority ceiling of all currently locked resources in the system. Tasks can only lock resources if their priority is greater than the system ceiling, which prevents deadlock. Unfortunately, this system ceiling requires a global state to be tracked, which increases the implementation complexity of the protocol.

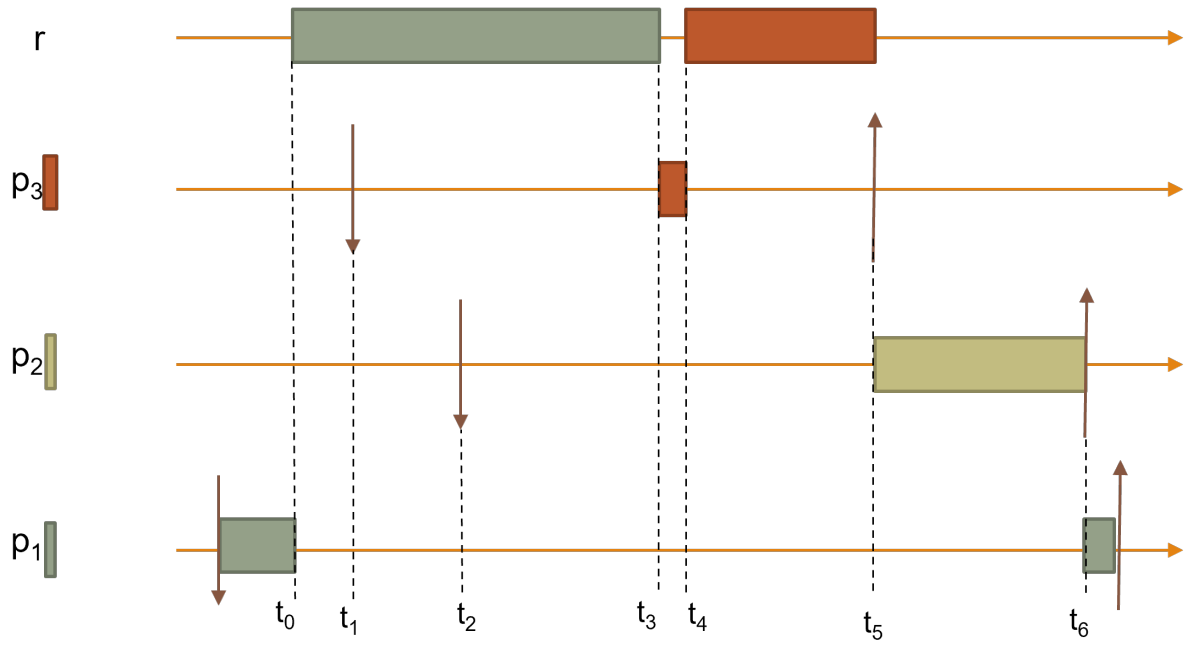


Figure 2.4: An example of IPCP. Downward arrow represent job releases, while upward arrows represent job completion. Dashed lines are used to indicate moments in time.

# Chapter 3

# Related Work

## 3.1 COMPOSITE and Temporal Capabilities

COMPOSITE Parmer [2009] is a component-based microkernel. It does not provide a scheduler or blocking within the kernel. Rather, these scheduling decisions are the responsibility of user-level, which leads to total policy freedom in regards to scheduling.

We focus our attention on temporal capabilities (TCaps) implemented into COMPOSITE, introduced by Gadepalli et al. [2017]. These aim to *"decouple scheduling decisions from the ability to consume time"*. TCaps possess a budget, which represents a slice of time. When user-level schedulers provide a thread to run, they must also provide a TCap from which time is drained from. Budget can be delegated between TCaps, however, budget is not automatically replenished by the system. Rather, there exists an original capability called *chronos*, which possesses infinite budget. All other TCaps are replenished by delegating budget from chronos.

Additionally, TCaps are assigned a *quality*, which represents the importance of the time held by that TCap. This quality is not equivalent to priority and is not used to make general scheduling decisions. Rather, if an interrupt activates a TCap, it will only preempt execution of the current thread if the activated TCap has a higher quality than the current thread's TCap.

However, a TCap cannot be revoked unless it has no budget. This design is intentional, such that a subsystem that has been delegated time has a guarantee of that full available budget. As the budgets do not self-replenish, this will be a finite amount of time.

## 3.2 NOVA microhypervisor

The NOVA microhypervisor Steinberg and Kauer [2010] Steinberg et al. [2010] enforces temporal separation of components running upon it. It provides scheduling contexts, which combine a time *quantum* with a priority. These are associated with execution contexts, which abstract

away the difference between threads or virtual CPU's. When invoked, the kernel scheduler chooses the highest-priority scheduling context and then allows its associated execution context to run. The execution context, regardless of whether it is a thread or a virtual CPU, can then run until either the SC's time quantum expires, or it is preempted by a higher-priority SC being released.

If a client thread sends an IPC message to a lower-priority server, the client donates its SC to the server, even if the server already possesses its own SC. The server then runs on the donated SC, also inheriting the higher priority of the client. This prevents a medium priority thread from pre-empting the server and causing priority inversion, which would occur if the server ran on its own low-priority SC.

While the server is still processing the first client's request, it may receive a request from a second, higher-priority client, which we refer to as $H$. $H$ will be initially unable to rendezvous with the server, so $H$ raises the priority of the server to $H$'s priority, helping it. This further prevents priority inversion from occurring.

This design is effective at preventing priority inversion from occurring, though it does enforce policy on user level, which we consider undesirable.

## 3.3 L4Re microkernel

The L4re microkernel, also known as Fiasco, encompasses multiple versions of an L4-based microkernel.

Fiasco Hohmuth and Härtig [2001], developed in 2001, used non-blocking synchronisation, using both wait-free and lock-free techniques, aimed at multicore systems. Wait-free synchronisation eliminates blocking caused by critical sections, by effectively implementing the priority inheritance protocol. We recall that the PIP protocol involves a higher priority thread donating its priority to lower-priority thread to help it finish the critical section.

In lock-free synchronisation, critical sections are designed such that they prepare their results separately and then commit them to shared data without needing a lock. The critical sections use an atomic instruction to first compare for potential conflicts, then write the data. If the compare operation fails, the entire operation is restarted, with exponential back-off used to avoid retry contention.

Notably, Fiasco was written in a higher-level language, specifically C++.

A further iteration of L4re was known as Fiasco Object Capabilities (Fiasco.OC) Lackorzynski et al. [2012]. This version of the kernel introduce capabilities, token that provide specific privileges to threads. We focus predominantly on the scheduler implementation, based around scheduling contexts (SC). SC's in L4re have the following attributes: a global priority, a budget and a budget replenishment rule. For example, this replenishment rule can support a fixed-priority scheme where each SC has a period controlling when its budget is replenished. These

SC's are distinct from threads, instead being considered as a reservation for a given amount of CPU time, that can be associated with a virtual CPU. Notably, a virtual CPU may have multiple SC's attached to it, in which case the guest running the virtual CPU can choose which SC to run on.

Further work focused on mapping mixed-criticality concepts into scheduling contexts Völp et al. [2013]. If a high-priority thread experiences a delayed job release, its execution will also be deferred, causing other lower priority, but potentially higher-critically tasks to be delayed and miss their deadlines. To address this, a deadline was introduced to SC's to ensure that a SC's budget had been consumed by a particular point in time. This would ensure that the thread's execution would not interfere with other tasks in the system. If the deadline was overrun, the task associated with the SC would be preempted by the kernel.

They also introduced mechanisms to effectively support using multiple SC's for quality-assuring scheduling. The basic principle involves jobs where there is a core mandatory component and then multiple additional optional sections, each improving the quality of the result. The authors proposed mapping a scheduling context to each section of work, with the SC's mapped to the optional sections possessing gradually reduced priorities. To support this use, additional configuration options were added defining whether multiple SC's bound to an SC were available simultaneously, or sequentially.

## 3.4 seL4 mixed-criticality systems kernel

The seL4 mixed-criticality systems (MCS) kernel was introduced by Lyons et al. [2018]. These changes to the kernel allowed for principled control of time, manageable at user-level, leveraging seL4's capability model.

While user-level control and policy freedom is a core design principle, it was decided to retain the scheduler within the kernel. For any thread not within the same protection domain, a user-level scheduler would require two context switches for each dispatch. While this overhead can be minimised with efficient context switches, it can be avoided altogether with a kernel-based scheduler. This scheduler is based on fixed-priorities, as it is easier to reason about, and it is simpler to implement dynamic scheduling policies at user-level. Further, while a kernel scheduler does slightly compromise on policy freedom, seL4 MCS contains various mechanisms that allow for users to add their own scheduling policy. In particular, the author demonstrated how an EDF scheduler can be implemented at user level with low overheads.

User-level control of scheduling is supported by the introduction of scheduling contexts (SC), which represent access to processor time. SC's possess a budget $C$ and a period $T$, where the budget is the maximum amount of time that can be consumed over the period. This imposes a utilisation limit $\frac{C}{T}$ that can be consumed by a thread associated with the SC. Budget is automatically replenished via a sporadic server algorithm, though the number of replenishments is limited.

Threads are only permitted to run if they are associated with an SC that has available budget.

SC's additionally support donation, whereby a thread can pass its SC to another thread via inter-process communication (IPC). This supports the existence of *passive servers*, which do not possess their own SC, but use a clients SC when working on its behalf.

The SC's introduced in seL4 are similar, but distinct from those present in Fiasco.OC. Notably, similar to all other kernel objects, SC's in seL4 are controlled by capabilities, however this is not the case in Fiasco.OC. Additionally, priority is an attribute of SC's in Fiasco, while in seL4, priorities are kept distinct in thread control blocks (TCB), which control most other non-scheduling attributes of threads.

If a thread's SC runs out of budget while executing, the thread is pre-empted by the kernel to provide enforcement. This raises a *timeout exception*, which can optionally be handled by a user-level monitor.

In line with the seL4 design principle of policy freedom, the seL4 MCS kernel does not restrict permissible task sets and allows for an overcommitted system.

## 3.5 Slite scheduling

Gadepalli et al. [2020] describe the new Slite scheduling protocol for *"near zero cost scheduling of system-level threads at user-level"*. Slite was initially implemented on the Composite microkernel.

In their system, both the kernel and a user-level scheduler track the same set of threads. However, they allow for direct user-level dispatch with no kernel involvement. This leads to an incoherency between the kernel and user space of which thread is current active. To resolve this incoherency, they describe a protocol involving a buffer shared between user-level and the kernel. Changes to the active thread are recorded to this buffer so that when required, the view of the active thread can be re-aligned.

Slite strongly aligns with two key principles of seL4, namely performance, by supporting user-level dispatch, and policy freedom, by moving scheduling policy entirely into user-space. However, direct user-level dispatch is only possible within the same protection domain. Threads switched to in this manner would then potentially be able to interfere with the scheduler. The scheduler is inherently one of the most trusted component of the system, as it can cause any task to fail by preventing it from accessing processor time. Therefore, it would only be permissible for the most trusted tasks in a system to share a protection domain with the scheduler. This greatly limits the use of the fast user-level dispatch as dispatching to other protection domains still requires a kernel entry and two context switches.

Overall, while Slite is closely aligned with the principles of minimality and policy freedom, its user-level dispatch is only possible for the most trusted tasks. Even then, it is generally considered best practice to isolate components whenever possible. We conclude that while the Slite scheduling model has some merit, a more detailed analysis of the security and isolation consequences is required before it would be suitable for consideration in seL4.

## 3.6 Mginkgo microkernel

Hu et al. [2021] introduce a set of modifications made to the Mginkgo microkernel to support mixed-criticality systems. The Mginkgo microkernel is described in the paper as being *"designed and implemented by our lab with reference to sel4 [sic]"*. Mginkgo is capability based and supports IPC and interrupts. Memory control and allocation is handled by userspace. Finally, threads are scheduled from within the kernel based on properties configured from userspace.

The authors define a criticality level for each thread along with a system wide criticality level. Threads whose criticality level is greater than or equal to the system criticality level are considered critical threads. In the kernel, a dedicated queue contains these critical threads sorted in reverse order of deadline. Upon each scheduler entry, the *critical point* of each task is calculated. This appears to be the latest time that a task can be scheduled and still complete successfully. If the current time is a critical task's critical point, then that task's priority is raised such that it is scheduled. Otherwise, the task with highest priority is chosen for execution. The authors also describe a bitmap algorithm that can determine the highest priority thread in $O(1)$ time. The algorithm "records the existence of ready tasks in each priority level", using a "bidirectional cyclic list to organise the tasks".

Unfortunately, the design presented in this paper is of limited use in seL4. The scheduler system, while potentially effective, adds significant policy into the kernel, mandating the use of a system-wide criticality level. This causes it to be unsuitable for seL4, where policy freedom is a core principle of the kernel. The $O(1)$ bitmap algorithm for determining the highest priority task would potentially be useful, though seL4 already contains an $O(1)$ scheduler. Unfortunately, insufficient details were provided to recreate the algorithm, so we are unable to make use of it.

In general, a system-wide criticality level is not the direction we wish to pursue for seL4. Building on seL4's history of verification and strong guarantees, we instead seek to create a system where the scheduling and isolation guarantees are very strong. This would allow the scheduling properties of high criticality tasks to be reasoned about without requiring any assumptions of lower-criticality tasks.

## 3.7 seL4 Response time analysis

Millar [2021] investigated the behaviour of the seL4 MCS kernel, to ensure that it correctly enforced scheduling behaviour. They constructed a relation between scheduling theory and the implementation of seL4 MCS. Using this, they identified a three deficiencies within the existing implementation and then proposed solutions.

The first issue was a defect within the sporadic server algorithm that controlled budget replenishment for scheduling contexts. The previous algorithm enforced a sliding window constraint, such that an SC could consume no more than its budget over any window of time equal to its period. This constraint was overly restrictive and would lead to lower-priority tasks having long

response times. The author successfully resolved this defect by altering the budget replenishment to obey the sporadic constraint rather than the sliding window constraint. This allows the system to correctly schedule periodic tasks.

The second investigated issue involved bounding priority inversion. Various protocols for handling priority inversion caused by resource sharing were introduced in Section 2.7. However, while those protocols prevented unbounded priority inversion, in general, they still create a large amount of priority inversion from priority ceilings. The author proposes that it is desirable to enforce tighter bounds on this priority inversion, so that the response time of higher-priority tasks can be guaranteed with a lower bound.

To accomplish this, they introduce *resource scheduling contexts* (RSC), which are a specialised form of SC that can only receive budget via donation. However, the authors did not consider potential alternative solutions. We consider this worthwhile of further investigation and cover this in more detail in Section 5.2

The third identified issue concerned the correct attribution of execution time in the kernel. Ideally, the kernel time would be attributed to the task that the kernel is operating on the behalf of. The author investigated all causes of kernel entry and where the time should be attributed to.

Further, in the previous version of the kernel, only a single timestamp was read, with all time charged being split at that point. The authors changed this to use two timestamps, one read just after kernel entry and one just prior to kernel exit. This allowed for a more accurate estimation of the time spent in the kernel. These changes were generally effective, however kernel time attribution that involved *interrupt requests* (IRQ), were only fully resolved with the introduction of RSC's.

# Chapter 4

# seL4

In this chapter, we present the current state of the seL4 microkernel, the operating system that our work will focus on. First, we will present some general background regarding the seL4 microkernel. Afterwards, we cover a limited number of areas in more technical detail, focusing on those relevant to our work. Specifically, we will cover seL4's capability system, system-calls, inter-thread communication and scheduling.

## 4.1 General background

seL4 (non-MCS) is a third generation L4 microkernel that has been formally verified Klein et al. [2009]. In particular, this provides the strongest guarantee of isolation between various components in the system.

seL4 controls privileges via *capabilities* Dennis and Van Horn [1966], which are unforgeable tokens that allow access to a resource. By requiring explicit delegation of rights, they support the principle of least privilege. In addition, they allow for extremely fine-grained access control while also supporting strong delegation properties.

However, the non-MCS seL4 kernel did not extend this capability control of resources to time. This was rectified in the seL4 MCS kernel, introduced by Lyons 2018, explored in more detail in Section 3.4.

seL4 has a number of design principles Heiser and Elphinstone [2016], that any addition or extension to the kernel should follow. We focus on *policy-freedom*. This involves having as few policy restrictions present in the kernel. System designers and users should then be able to define system policies outside the kernel, in user level. This is a natural consequence of minimality, while also making the kernel applicable to the widest possible range of situations.

The practical result of policy-freedom is that when designing the kernel, it is both acceptable and desirable to move policy decisions out of the kernel and into user-space. However, this

must only be done when it would not unduly compromise the other goals of the system, such as performance or security.

For the remainder of this chapter, we will exclusively discuss the seL4 MCS kernel.

## 4.2 Capabilities

seL4 is a capability-based OS, meaning that access to all resources is controlled via capabilities. The first user-level thread started in a system is provided with capabilities that grant access to all resources in the system. That initial thread is then permitted to allocate resources as required.

Capabilities are tracked within capability space (*cspaces*). These consist of capability nodes (*cnodes*), which are capabilities that contain capabilities. These may include capabilities to additional cnodes, allowing for multi-level cspaces to be constructed. An individual entry in a cnode is known as a *slot* in the cspace, which can either be empty or contain a capability to a kernel object.

Capabilities have associated access rights, of which there are four: read, write, grant and grantreply. These have different effects depending on which kernel object the capability provides access to.

The kernel provides a number of system calls for interacting with and manipulating cspaces and capabilities. First, there are some basic operations, move and delete. Move simply moves a capability from one slot to another, while deletion removes a capability from a cspace.

New capabilities can be *derived* from an existing capability. A copy operation derives a new capability with equal access rights as the original. However, it is also possible for the derived capability to have diminished rights, which occurs via a mint operation. Minting also supports a special type of derivation known as *badging*, the details of which we cover below in. A capability can also revoked, which leaves the capability itself unaffected, but deletes any capabilities that were derived from it.

## 4.3 Memory management

We now briefly outline the basics of seL4s memory management. In line with policy-freedom, in seL4, memory is managed by user-level. Firstly, the kernel reserves a small amount of system memory for its own use. The rest of system memory is then provided to the initial startup thread in the form of *untyped* capabilities.

Untyped capabilities represent unused memory. These memory regions can be split into smaller untyped capabilities, or *retyped* into kernel objects for use by threads. All memory used by

user-level threads, whether explicit kernel objects, or memory frames, must be retyped from untyped capabilities.

## 4.4 System calls and invocation

seL4 supports two broad categories of system calls. The first category is true system calls, distinguished in the kernel by system call number. First, there is the Yield system call used for scheduling, which we cover in more detail in section below. The other true system calls all facilitate message passing, fundamentally consisting of sending and receiving. When performed on endpoint and notification objects, this supports inter-thread communication, which we will detail further in the following section.

However, it is also possible to invoke a send operation on a kernel object, which forms the second category of system calls, object invocations. This sends a message with the kernel as the implicit destination. The other parameters of the invocation are encoded in the message parameters. The kernel then decodes these parameters to determine which should be taken. Where a response is required, the user-level thread receives the kernel's reply in the same format as a message from another thread.

## 4.5 Scheduling

seL4 supports a kernel-based fixed-priority scheduler. However, the scheduling properties of threads are split between two separate kernel objects. Threads themselves are represented by *Thread Control Blocks (TCB)* and this contains the thread's priority. However, a separate object, *Scheduling Contexts (SC)* represent access to CPU time. For a thread to run its TCB must have a SC bound to it. This binding is restricted to being one to one, only a single SC can be bound to a TCB at any point.

We now describe SC's in more detail, before further describing the behaviour of the kernel scheduler.

### 4.5.1 Scheduling Contexts

Scheduling contexts are an abstraction for the allocation of CPU time, based on the sporadic server algorithm. The fundamental parameters on an SC are its period $T$ and total budget $C$, where $C \leq T$. The budget is the maximum amount of CPU time that the SC allows to be consumed over the period.

SC's support this behaviour with a queue of refills, analogous to the replenishments of sporadic server theory. Each of these refills have an amount and a timestamp. The amount tracks how much time a thread can execute for, while the timestamp tracks when that time is eligible for

use. There is a limit on the maximum number of refills a SC can possess, configurable by system designers. This allows system designers to trade-off more accurate tracking of replenishments with increased preemption overhead.

The seL4_Yield system call depletes the head refill of the scheduling context. This causes the bound thread to not be runnable, triggering a reschedule. When the SC later has a refill released, it becomes runnable again.

Capabilities to a scheduling context provide the right to bind and unbind that SC to another object and check the amount of time consumed on the SC. Notably however, the SC capability does not provide the right to set the SC's budget or period. We recall that untyped memory can be used to create any kernel object, and this includes scheduling contexts. Control of scheduling properties needs to be restricted to allow strong enforcement of scheduling guarantees. Allowing any thread with access to untyped capabilities to configure scheduling properties would be an unacceptable compromise of this.

Instead, there exists a schedControl capability, which provides access to CPU time management on a single core. Therefore, a distinct schedControl capability exists for each core. A schedControl capability can be used to set the scheduling parameters, such as budget and period of an SC, however this binds the SC to that core, only allowing the use of CPU time on that core.

Scheduling contexts support a number of different scheduling behaviours. With the period set to zero, SC's operate in a round-robin fashion.

SC's also support constant-bandwidth server behaviour. This strictly enforces the sliding window constraint, where over any period T, the thread can consume no more than its budget. This is in contrast to the sporadic server behaviour, which is additionally supported by SC's. The sporadic server mode allows the SC to continue to accumulate budget while preempted by a higher priority thread. The sporadic thread can then use the accumulated budget to briefly exceed the sliding-window constraint. SC's can be toggled between constant-bandwidth and sliding window behaviour when configured by the schedControl capability.

### 4.5.2 Kernel scheduler

The kernel scheduler supports 256 priority levels, (0-255), based on the priority set in the thread's TCB. All threads that are currently runnable, which means they are not blocked and have a bound SC with a released head refill, are tracked in queues in the scheduler. The exception is the currently running thread, as in many cases the currently running thread directly changes state from running to blocked. By not keeping the currently running thread in the scheduler, enqueue and dequeue operations can be avoided, improving performance. This is known as Benno scheduling Heiser and Elphinstone [2016].

**Release queue**

The release queue tracks threads which would otherwise be runnabled, except their bound SC does not have a released head refill. The queue is ordered by the release time of the SC's head refills and a timer interrupt is programmed for the release time of the head of the queue. At that point, the relevant SC will have a released head refill, so its bound thread is removed from the release queue and placed in the scheduler.

## 4.6 Communication

seL4 supports communication between threads both through IPC and notifications. IPC predominantly supports protected procedure calls, while notifications function as a synchronisation mechanism. We explore IPC in more detail below, but we do not cover notifications in further detail, as they do not significantly impact our work.

### 4.6.1 IPC

In seL4, IPC is facilitated by threads sending and receiving messages over endpoints, which function as message ports. The payload of these messages are stored in an IPC buffer, of which one is associated with each thread The message payload can consist of both data and capabilities.

When a sending and receiving thread meet on an endpoint, the kernel sends the message by copying the contents of the IPC buffer from sender to receiver. This is known as an *IPC rendezvous*. For efficiency, smaller messages are sent only using CPU registers, avoiding the copy operation.

If a thread sends or receives onto an endpoint without a corresponding thread ready to rendezvous, the thread can block on the endpoint. It remains associated with the endpoint, waiting for another thread to later invoke the endpoint and trigger a rendezvous.

Multiple threads can be queued on an endpoint, if multiple threads invoke the endpoint in the same manner without corresponding threads to rendezvous with. However, endpoints only consist of a single queue, which contains sending threads or receiving threads as necessary. By design, an endpoint cannot contain threads waiting to both send and receive, as these threads would have rendezvoused with each other, rather than blocking and waiting on the endpoint.

Sending and receiving system calls are available in both blocking and non-blocking variations. The blocking variations operate as described above, permitting the thread to block on the endpoint if it cannot immediately rendezvous. In contrast, if there is not another waiting thread immediately ready to rendezvous, the non-blocking variations do not perform the IPC and continue execution.

In seL4, IPC should primarily be viewed as supporting protected procedure calls, whereby a thread can invoke a procedure in a separate thread or protection domain. The fundamental

operations that support this are seL4_Call, invoked by the caller and then seL4_ReplyRecv, which is invoked by the callee. We briefly outline these operations.

- **seL4_Call**: Performs a send followed by a receive on the same endpoint. Blocks the thread until the recipient replies. In addition, it sets up a separate reply channel that the server can use to reply directly to the client.
- **seL4_ReplyRecv**: Replies to a client, then receives on the endpoint waiting for a new request.

Other IPC invocations exist, including standalone sending and receiving invocations. However, these are primarily used for system initialisation or exception handling.

#### Badging

Endpoint capabilities additionally have a *badge* associated with them. When a capability is invoked to send a message to an endpoint, the kernel transfer the sender's badge to the receiver.

A capability with a badge of zero is said to be *unbadged* and carries no specific meaning. However, when an unbadged endpoint capability is derived, the child capability can have a badge set on it. An example use is allowing a receiver to differentiate between multiple senders.

Finally, a badged capability cannot be unbadged or used to create further child capabilities with a different badge.

#### Scheduling context donation

seL4 also supports scheduling context donation, whereby the sending of a message can donate its SC for use by the receiver. When the receiver replies, the SC is sent back with the return message.

SC donation is not explicitly chosen by users, rather, if the sender uses an invocation that supports it and the receiving thread does not have a SC bound to it, the sender's SC will be donated. Servers that do not have a SC are known as *passive* and receive scheduling contexts via donation. *Active* servers possess their own SC and do not receive scheduling contexts via donation.

Finally, as mentioned, scheduling context donation is only permitted via some system calls. Specifically, those that combine sending and receiving in a single system call. Otherwise, a client that donates its SC with only a sending invocation would then be unable to run as it no longer possesses an SC. It would then be unable to perform the receiving invocation and could not receive the reply that would return the SC. Similarly, a server could reply, returning the SC to the client, but then be unable to run. The server would then be unable to invoke a receive on the endpoint and be unable to process and further requests.

Therefore, SC donation is only permitted using the combined send and receive system calls.

### Reply objects

Reply objects allow the recipient of a message (i.e. a server) to track the sender and later reply to them. When a server receives on an endpoint, it must also provide a reply object. When a client invokes a server over an endpoint using a seL4_Call, the server's reply object becomes active, representing a reply channel back to the client. By invoking the reply object, the sever can reply directly to the client.

A server can also call onwards to another server. This leads to a doubly-linked stack of reply objects to be created, allowing the call chain to be tracked. If a scheduling context has been donated, the stack of reply objects is used to track it.

Each reply object contains two pointers, used for tracking the reply stack in the usual manner for a doubly-linked list. There is an additional pointer referencing TCB's with two possible meanings, depending on the thread's state. The referenced thread can be receiving on an endpoint, with the reply object ready to track the call stack. Alternatively, the referenced thread is the reply target of the reply object. A reply IPC message will be sent to that thread when the reply object is invoked.

### Faults and exception handling

The IPC mechanism is also used to support exception handling for threads. All threads can have two exception-handling endpoints associated with them, a *standard* exception handler and a *timeout* exception handler. When a thread causes an exception, the kernel generates an IPC message with the relevant details and sends it to the relevant endpoint. A thread waiting on that endpoint to receive can then take an appropriate corrective action.

The standard exception handler covers errors where the thread can only be recovered with the aid of another thread. For instance, if a thread accesses an unmapped virtual memory page, the thread will only be able to continue if that page is mapped.

In contrast, timeout faults occur when a thread attempts to run but has no available budget. Unlike standard faults, timeout faults do not need to be handled. If a thread does not have a timeout fault handler, no fault message will be sent and the thread will continue running after its budget is replenished.

# Chapter 5

# Approach

## 5.1 Goals

The primary goal of this thesis is to improve the model and implementation of the seL4 MCS kernel, originally developed by Lyons et al. [2018]. Specifically, we aim to address issues with the model that force inefficiencies and complexity on system built upon it. As we still desire to retain the achievements of the original seL4 MCS kernel, we align our broad goals with the original goals for that system.

Specifically, the system should support:

- Capability-controlled enforcement of time
- Policy freedom
- Efficiency
- Temporal Isolation
- Overcommitment
- Safe resource sharing

We do however, emphasise that in line with the principle of minimality, the seL4 MCS kernel offers only a small set of features. It is not a mixed-criticality system in its own right, rather it aims to be a platform to allow system designers to create secure, temporally isolated systems. Therefore, policy freedom is a key goal, and it is both acceptable and desirable to leave policy decisions to system designers where possible.

### 5.1.1 Scope

We also outline the scope of this thesis, in particular focusing on items that will be out of scope and left for future work:

- Multiple processing cores: In this thesis we will focus on single-core processors. Multiple cores can require more complex resource control and synchronisation, complicating scheduling analysis.
- Variable processor speed: Processors with variable clock speed can mean that CPU clock cycles are not a consistent length of time. This leads to additional challenges with benchmarking and comparing performance.
- Verification: The aim is for the seL4 MCS kernel to be verified, just as the baseline seL4 kernel was Klein et al. [2009]. This will be taken into consideration when evaluating design choices, however, the actual task of verification is beyond the scope of this thesis.

This thesis will instead focus on investigating and a resolving a number of key issues present within the kernel, presented in the next section.

## 5.2 Assessment of Issues

In this section we present some current issues with the seL4 MCS kernel along with a discussion of some potential solutions.

### 5.2.1 Budget Expiry in Passive Servers

We recall that the seL4 MCS kernel supports passive resource servers, commonly used to encapsulate shared resources. They generally do not run on their own, but instead operate on behalf of other tasks. As passive servers, these do not have their own budget, but are donated budget by the task they are operating on behalf of. If a passive server runs out of budget while executing, there are three broad options available to the system:

- Do nothing: This is the simplest option and is the default behaviour. When the scheduling context associated with the passive server is replenished at its next period, the server will be able to run again. While this may be acceptable for some systems, it is not appropriate for a server shared by clients of mixed criticalities. This is because any other tasks that require the shared resource will be blocked while the server is waiting for the budget to replenish. This can lead to a significant increase in the blocking time of tasks in the system. Crucially, this can cause a high-priority task to be dependent on a low-priority task, breaking the isolation between criticality levels that we desire seL4 to support.

- Supply additional budget: Through a thread's timeout exception, another thread can be informed when the passive server exhausts its budget. Provided this thread has appropriate privileges, it can provide additional budget to the server, allowing it to continuing executing. If the task using the passive server is a high-criticality task, this may be the most desirable option. However, lower-criticality tasks that exceed their allocated budget are violating their contract with the scheduler. It is preferable to not reward tasks that violate their contract.
- Revert the server: Some passive servers can be easily restored to a previous clean state. This allows the server to be used by other tasks. However, reverting the server has a non-zero cost and not all resources support revertion. Further, this requires that all operations involving the passive server follow a transaction model, whereby each operation can be reverted and restarted if required. This enforces policy on user level, which violates one of the seL4 design principles.

While there are options that can be taken when a task's budget expires inside a passive server, all of them have limitations. Ideally, we would prefer to prevent budget expiry from occurring inside passive servers altogether.

One option for supporting this is enforcing a minimum budget that a task must possess to be able to enter a passive server. These properties should be configurable at user level, but if the minimum budget was set to the passive server's WCET, then budget expiry inside the server could be prevented. The check for sufficient budget would most likely need to be implemented on the IPC kernel path. Further, a policy decision exists regarding how tasks that attempt to access a resource with insufficient budget should be handled. This policy decision should be made at user-level, but some options the kernel could support include:

- An exception handler, allowing a user-level fault handler to intervene.
- Block the thread associated with the task until its budget is replenished and exceeds the required minimum budget.
- Return to the task with an error indicating insufficient budget available and allow the calling task to decide what action to take.

### 5.2.2 Bounding Priority Inversion

Priority inversion is an unavoidable consequence of sharing resources within a system. The protocols presented in Section 2.7.2 prevent priority inversion from becoming unbounded. However, with the IPCP protocol, when a low-priority task accesses a resource, intermediate tasks between the ceiling priority and the low-priority task will be blocked, even if they have no dependence on the resource. Currently, the bound on this blocking time is length of the single longest critical section associated with the resource. Once any critical section ends, the task's priority should drop down from the ceiling priority and the intermediate task will be able to pre-empt it. However, when determining the worst-case response time of tasks, this contribution to the blocking

time from resource-based priority inversion must be considered. Therefore, it can be desirable to introduce kernel-enforced bounds on priority inversion caused by resource access.

We additionally reframe this issue from an alternate perspective. Currently in seL4, clients of a passive server must completely trust that server. For example, whether maliciously or due to an error, if a server never replies to the client, the server can keep and run on the client's SC indefinitely. Introducing a kernel mechanism that limits the time a server can consume on a donated SC would reduce the trust needed in a system, making it easier to perform schedulability analysis on a system.

We present the existing solutions that have been proposed:

- Nothing: Doing nothing is not strictly a solution. However, all other solution protocols are likely to introduce some form of performance overhead. Therefore, on a sufficiently congested system it is possible that doing nothing is the best solution.
- OPCP: The original priority ceiling protocol can result in reduced preemption by delaying the priority boost. However, the need to track global state significantly complicates the protocol and can lead to greater overhead. An implementation of OPCP in user-space has already been demonstrated Lyons [2018], however it is possible that better kernel support would improve the usefulness of OPCP. However, this could compromise on policy-freedom, as a kernel implementation of OPCP may strongly lead to it being the preferred policy.

  Unfortunately, the global state of OPCP would almost certainly compromise confidentiality between tasks if supported in the kernel. The system ceiling of OPCP restricts all tasks in the system from locking resources. Two otherwise isolated subsystems could then potentially communicate by manipulating the system ceiling and then testing whether they can acquire a lock. This would violate confidentiality and isolation and therefore, we do not consider OPCP to be a viable option.
- Resource scheduling contexts: Resource scheduling contexts (RSC) were introduced by Millar Millar [2021]. They are a variation upon scheduling contexts designed for use with passive resource servers. We recall that the current model of SC donation occurs when a server does not possess its own SC. The caller's SC is then transferred to the server, allowing it to run.

  In contrast, RSC's are designed to be permanently bound to passive servers. RSC's are configured with a maximum budget, however, this is not automatically replenished based on a period. Therefore, the sever is still passive, as it can only receive budget and run via donation. Donation still occurs when the server is called by a client, however RSC donation is distinct from normal SC donation. Rather than the client's SC being transferred in its entirety, the caller's SC and the RSC become linked. The server with the RSC is then permitted to consume up to the minimum of the donating SC's available budget and the RSC's maximum budget. Therefore, the server can consume no more budget than is available in the donating SC, in line with the scheduling properties required by donation. However, the RSC's maximum budget also restricts the maximum continuous time that the server can run for. When the server replies to the client, its usage is charged against the clients SC and the RSC forfeits any remaining budget it holds. Thus, in line with the passive server model, it is unable to run until another client donates some budget.

As a result, the RSC's maximum budget restricts the maximum continuous time that the passive server can run for. This limits the blocking time that can be caused by the passive server.

However, it is possible that when the RSC's budget depletes, the critical section has not completed. It is not fully clear what would happen in such a situation. The resource server would be unable to run as it has no budget and will not be replenished automatically, while the original task would be blocked waiting for a reply from the server. This would need to be resolved by an external process. This specific issue is permissible, as it would be up to system designers to prevent such a situation from occurring. However, it would make the system more complex and would need demonstrable benefits to be worth the trade-off. In general, as outlined above, budget expiry in a passive server is preferably avoided.

This model also associates thread priority with scheduling contexts, rather than TCB's. This priority is therefore only settable using a schedControl capability, which we recall is used to configure all scheduling properties of a SC. The model also restricts donation to only be permitted to RSC's with a higher or equal priority than the original SC, to prevent the task from being pre-empted by lower-priority tasks. We do note that this is a restriction on policy freedom, while also being relatively straightforward for system designers to enforce at user level.

The schedControl capability is also changed under this model. In line with its additional role of setting priorities, it would possess a maximum controlled priority (MCP). This is the maximum priority that a particular schedControl capability can configure on a SC or RSC. Presumably, when schedControl capabilities are derived, the child capabilities MCP can be set to less than or equal to the original capability. This would allow for weaker schedControl capabilities to be created that only have the privilege to affect the system below a certain priority level.

Further, this model would require switching scheduling contexts upon every IPC, which would result in performance degradation due to the overheads involved. In particular, every time the active scheduling context changes, the consumed budget needs to be charged to the previously active SC. The existing model of SC donation avoids this overhead as the active SC remains the same, it is simply passed between the client and the server.

# Chapter 6

# Design and Implementation

## 6.1 Budget Expiry in Passive servers

We now present our design and model for preventing budget expiry from occurring within a passive server.

Our goal is to support better scheduling behaviour, in particular by preventing blocking time caused by budget expiry within passive servers. As outlined in subsection 5.2.1, this issue can be addressed reactively through timeout handlers. However, we believe that a proactive solution, preventing budget expiry from occurring is a superior option. We therefore propose the introduction of a *threshold* to IPC, a minimum budget required to complete an IPC operation. By setting this threshold to the WCET of a passive server, we could completely eliminate budget expiry.

Our core design goals are:

**Policy freedom:** As far as possible, not restrict the possible designs that can be implemented by system designers and users.

**Minimality:** In line with the microkernel principle of minimality, we seek to keep the kernel as simple as possible while implementing the required behaviour.

### 6.1.1 Thresholds

There are a number of design choices associated with the threshold itself:

- Which kernel object should a threshold be associated with?
- What do we consider a thread's available budget?
- How should thresholds be configured?

We investigate these design options in the following sections.

### Thresholds and kernel objects

There are two main candidates for kernel objects that can be associated with the threshold:

- Endpoints
- The server's thread control block

A sensible choice for the threshold value would be the WCET of the passive server. This value would ensure that the server always has sufficient budget to complete its execution and would prevent budget expiry from occurring. However, to maintain kernel minimality, we leave the responsibility of determining the correct threshold value to system-designers. This additionally supports policy-freedom, by allowing system-designers to set the threshold as desired.

This link with the server's WCET most naturally lends the association of the threshold with the server's TCB. Unfortunately, this can lead to a long-running kernel operation which we now illustrate with an example.

We will cover the specific details of how the kernel handles clients with insufficient budget below. For now, we assume that clients with sufficient budget are allowed to proceed with the IPC, while other threads are delayed until they accumulate sufficient budget.

We consider an endpoint with no server waiting on it, but $n$ clients queued to send. When a server then receives, the kernel needs to check each client's available budget against the server's threshold before allowing the IPC rendezvous to occur. Potentially, all $n$ threads would need to be checked, if the first $n-1$ clients had insufficient budget, but the final thread was sufficient.

We can avoid this long-running kernel operation by associating the threshold with the endpoint. This allows the budget check to occur before the client is permitted to enter the endpoint, either rendezvousing with a waiting server, or enqueuing on the endpoint queue. As the kernel only needs to check the budget of a single thread at a time, the long-running operation is avoided.

On a single-core system, there is typically a single passive server associated with an endpoint. Therefore, the link between the threshold and the server's WCET remains clear. Where there are multiple servers on a single endpoint, setting the threshold to the maximum WCET of the servers would work well. Further, multiple servers receiving on an endpoint must be equivalent and therefore would very likely have similar WCETs regardless.

Therefore, we choose to associate threshold values with endpoints as this avoids a long-running kernel operation, with only minimal impacts on system designers.

### Endpoint object vs capability

Now that we have decided on making the threshold a property of endpoints, we must further decide between the endpoint capabilities or the objects themselves. The principal distinction is that there is only a single underlying kernel object, but potentially multiple capabilities, distributed out to multiple threads. Associating the threshold with the object itself would support only a single threshold value which all clients must obey.

However, as each thread must possess an endpoint cap, thresholds associated with the caps would allow for system designers to assign different thresholds to different threads. Different endpoint capabilities could be badged to access different operations provided by the passive server. These operations could have different WCETs and thus there may be a use case for different threshold values for different capabilities.

Another possible use would be setting a lower threshold for a critical client, to ensure that it can enter the endpoint. However, we argue that in this case, the correct design would be to allocate that client more budget, rather than reducing the endpoint threshold for it. This is because setting a lower threshold does not guarantee that that client will be successfully served, rather it creates the possibility of budget expiry.

Both the endpoint object and the endpoint capability would need to be increased to size to store the threshold value. This is straightforward for the endpoint object and its size can be increased with no major repercussions. However, all capabilities share a common size. This is required so that capability nodes can hold any capability within their slots. Therefore, increasing the size of endpoints caps would require increasing the size of every capability. As all sizes in seL4 work with powers of 2, capabilities would need to be doubled in size, which would double the memory footprint of the entire capability system. The majority of this space would be unused and wasted. This would also require moderate changes to the kernel to increase the size of all capabilities. These changes, if implemented, would compromise on simplicity and kernel minimality.

Overall, we choose to associate the threshold with the endpoint object. This is a modest restriction on user policy-freedom, with regards to setting different thresholds for different operations of a passive server. However, implementing thresholds on endpoint capabilities would require moderate changes to the kernel, which we currently consider an unacceptable compromise of minimality in light of the potential benefits. However, if in future the benefits are considered to outweigh the drawbacks, we believe its implementation is fairly straightforward.

### Calculation of available budget

We recall that the available budget in a scheduling context is composed of a number of refills. It is possible that a client has multiple released refills, which must all be summed together to calculate the clients total available budget. This means that the maximum number of refills, which are configured from user-level, affects the kernel's worst-case execution time.

While this is not ideal, we believe this is the only viable option. The alternative would be to only consider the budget in the head refill when comparing against the threshold. This would limit the kernel time spent checking available budget to a constant time operation. However, this design has some unacceptable compromises that we now illustrate with an example. In the following section we will cover in detail the behaviour of threads with insufficient budget. For now, we assume the client threads are blocked until they have sufficient budget.

We consider a client thread that calls in to a thresholded endpoint. It has multiple released refills, the sum of which is sufficient, but the head refill alone is insufficient. As the head refill is insufficient, the thread will initially be rejected and blocked. The normal behaviour for blocked threads is to have their refills merged, which will lead to the head refill having sufficient budget. This would then lead to the thread being permitted to enter the endpoint.

There are a number of issues with this example where we only check the budget of the head refill. First, it imposes an extra constant-bandwidth restriction. At all points in the example, the thread possessed sufficient budget, yet its refills were merged and deferred regardless. This violates the sporadic server algorithm and negatively affects schedulability analysis.

Secondly, the time savings from checking only the head refill are made irrelevant. Once the thread blocks due to insufficient budget, all the refills need to be merged regardless. This is again a kernel operation dependent on the maximum number of refills.

We therefore conclude that summing all the released refills is the best option. While the kernel WCET becomes dependent on the maximum number of refills, there are some mitigating factors. The privilege to set the maximum refills of a SC is restricted to the schedcontrol capability and does not need to be widely distributed. Therefore, a static analysis of the contribution to the kernel WCET from this behaviour will still be possible.

### 6.1.2 Behaviour with insufficient budget

We now consider the behaviour that occurs when a client thread calls into an endpoint with a threshold set. Before continuing, we clarify our terminology:

- Available budget: Sum of the budget in released refills.
- Maximum budget: Maximum budget that can be held in the client's SC.
- Sufficient budget: A client's available budget exceeds the thresholds.
- Insufficient budget: A client's available budget is less than the threshold.

If a client calls into the endpoint with sufficient budget, the IPC operation continues normally. A client that calls with a maximum budget less than the threshold will be rejected immediately with an error returned to the user. However, if the available budget is insufficient, but the maximum budget is greater than the threshold, we have a number of options regarding how the system should behave:

1. Fail the IPC immediately and return to the user.
2. Wait until the thread has sufficient budget, then return to user level.
3. Wait until the thread has sufficient budget, then complete the IPC operation without explicitly returning to the user.

Options 1 and 2 have the benefit of allowing clients to choose how they wish to handle failed calls due to insufficient budget. For instance, clients could try and use their remaining budget to perform another operation before retrying the call. However, clients may not have sufficient budget to complete another operation or may not have other useful work to perform. Further, such a change would disrupt the existing model of call invocations as analogous to function invocation. Unlike function invocations, calls may need to be retried, not due to a parameter error, but simply insufficient budget. Therefore, we consider it an acceptable compromise of policy-freedom to not allow clients to choose alternate behaviours upon a failed call.

We also note that the main behaviour for clients in option 1 would be to wait for sufficient budget and then retry the call invocation. This wait operation would need be implemented as a system call, as a thread's available budget is known only to the kernel. Therefore, regardless of the chosen option, a kernel mechanism to wait for sufficient budget needs to exist. Once this mechanism exists, it is trivial to configure the kernel to invoke it automatically after an invocation with insufficient budget. Further, a kernel mechanism for restarting system calls already exists. Therefore, the additional kernel complexity to implement option 3 over option 1 is small.

As a result, we choose option 3 for its superior user experience, no major restrictions on policy-freedom and minimal additional kernel complexity.

#### Waiting for sufficient budget

As detailed above, a kernel mechanism needs to be provided to allow threads to wait until they possess a certain amount of budget

First, we note that the existing yield system call is unsuitable for this use. We illustrate this with an example in Figure 6.1. Consider a thread with a bound SC, where the maximum budget is 100 units, currently split into two refills of 50 units each. Finally, the thread bound to SC requires 60 units of available budget, to pass a threshold.

If the existing yield system call is used, we recall that it depletes the head refill and schedules it for replenishment one period later. As this will not cause refills to be merged, this does not guarantee that the thread will have sufficient budget in the future. In the example presented, even after yield is called, there are still two refills with 50 units of budget each, which are insufficient to pass the threshold. Therefore, we must introduce a new kernel mechanism.

There are two broad alternatives we could pursue:

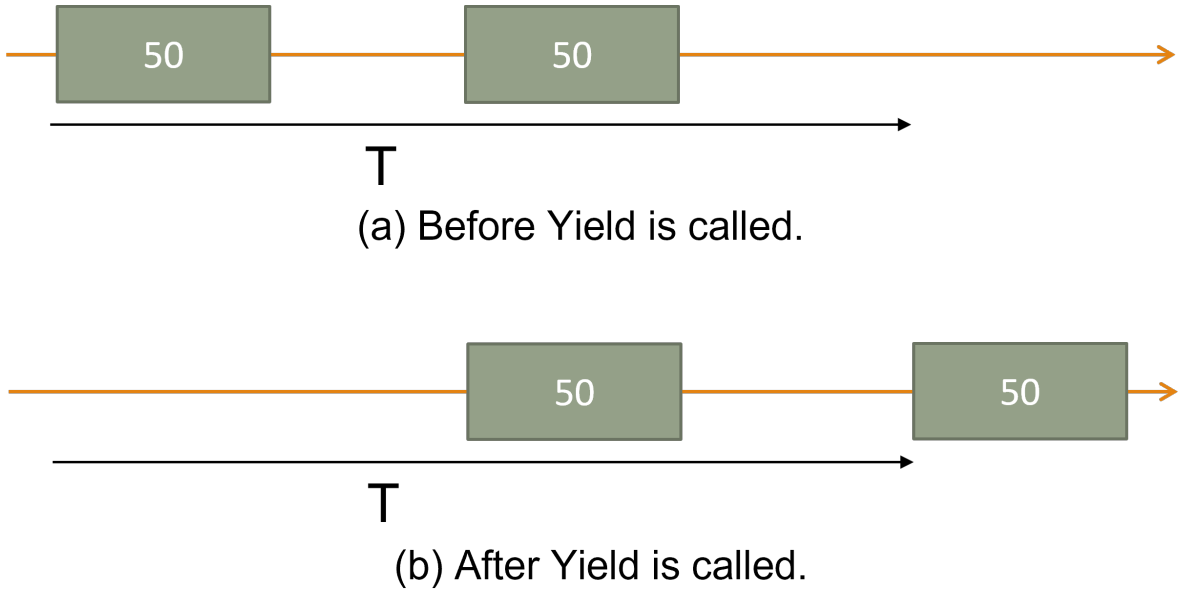


Figure 6.1: An illustration of the yield system call. Shows the state of SC refills before and after Yield is called.

1. The first option is to leave the SC refills untouched. Then, we would create a separate mechanism that would keep the thread blocked until future refills are released such that its total budget becomes sufficient.

2. The second option is to immediately defer and merge the SC refills such that a single refill with sufficient budget is created in the future. When the thread resumes running on that refill it will have sufficient budget.

The first option would require us to create a new blocking mechanism to keep the thread blocked until it has sufficient budget through the release of (potentially multiple) refills. The existing release queue waits for the release of a SC's head refill only. Once a thread's SC has a valid head refill, the thread is moved into the scheduler. However, when we consider thresholds, it is possible for a thread to have a released head refill that is insufficient and still need to wait for the release of additional budget. This would require either significant modification to the existing release queue, or a new additional release queue with different semantics. Both of these option would introduce additional kernel complexity.

In contrast, the second option allows us to implement our desired behaviour using the existing scheduler mechanisms. We would immediately defer and merge refills until the SC's head refill, at some point in the future, contains sufficient budget. Then, the thread would not be runnable, as its SC would not possess a released head refill. The thread could then be inserted into the existing release queue and, once the head refill is released, become eligible for scheduling. By configuring the kernel to then restart the IPC operation, the thread should now have sufficient budget to succeed. By avoiding the need to introduce a new queue, this approach better aligns with kernel minimality.

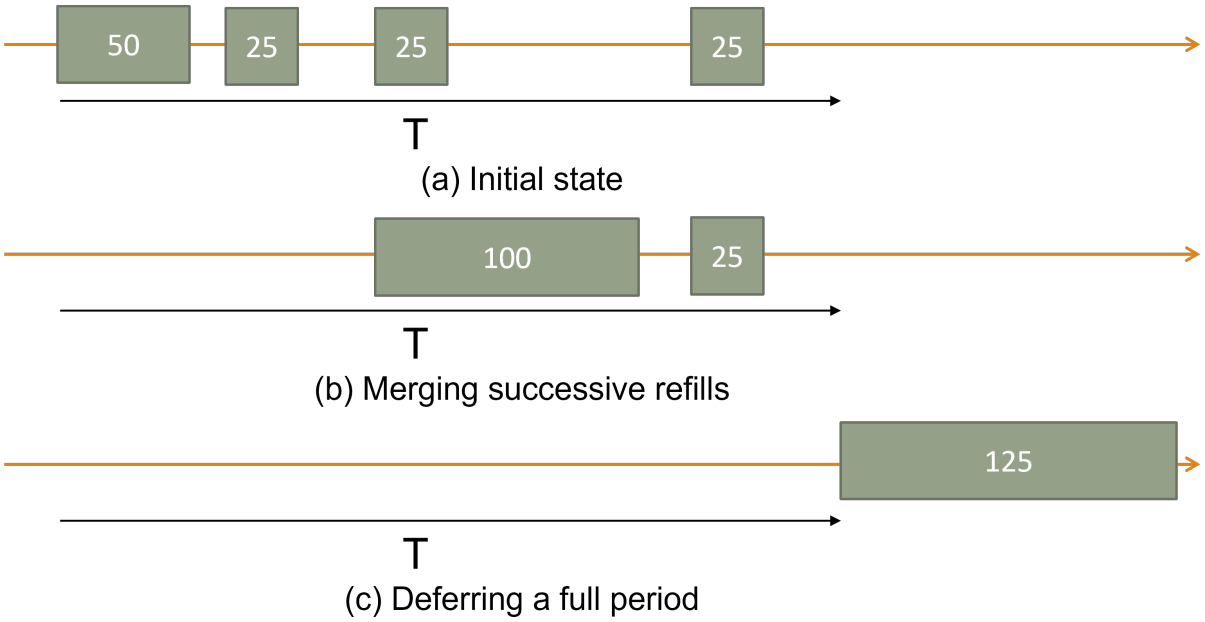


Figure 6.2: An illustration of the two options for deferring budget. Shows the state of SC refills before and after budget is deferred for the two options.

We choose option two due to its much simpler implementation requirements, with minimal drawbacks. Now that we have decided to defer and merge refills immediately, there are again two options of how this can be performed:

1. Merge successive refills until a refill with at the least desired budget is created.

2. Remove all refills and create a single refill, 1 period in the future with the SC's full budget.

We illustrate these two options in Figure 6.2. We show refills for a SC with a total budget of 125, fragmented into multiple refills. The SC needs its budget merged up to 100 units of budget. Part (b) of the figure shows the final state after successive refills are merged, only up to the required amount. Part (c) of the figure shows the final state where the budget has been deferred a full period.

The first option will allow threads to execute earlier, as they will be allowed to run once their released refills would have sufficient budget. In contrast, the second option would cause us to always defer threads for a full period of their SC. This would be equivalent to the worst-case for the first option. However, while the worst-case behaviour is equivalent, our average case performance will be superior for the first option.

The trade-off we need to consider is the kernel time required for these operations. The first option has a kernel run-time dependent on the maximum number of refills in the thread's SC. The second option would instead be a constant time kernel operation.

An increased kernel WCET dependent on the maximum number of refills is not ideal. However, we recall that checking the available budget of the thread is also dependent on the maximum number of refills. Therefore, the additional kernel complexity required for option two is relatively minor, with potentially significant average-case performance benefits.

Additionally, we choose to make this new behaviour of deferring and merging refills explicitly available for users as system call. As we already need to implement this for internel kernel operations, providing it to users as a system call requires minimal additional complexity. Therefore, we introduce a new variant of yield, which we call *seL4_YieldUntilBudget*. This accepts a single parameter, whereby threads can specify a desired quantity of budget that the kernel will merge and defer until a refill is created with that quantity of budget. If the requested budget is greater than the SC's maximum budget, an error is instead returned to the user.

### 6.1.3 Thresholds - revisited

We now revisit some finer details of thresholds, specifically how we permit users to configure them and which invocations can be performed on an endpoint with a threshold set.

Also, we expect that endpoint thresholds will involve some overhead on the IPC path, even when threshold behaviour is not enabled on that endpoint. To avoid imposing a performance penalty on systems that do not make use of thresholds, we place threshold support for the kernel behind a compile flag. This should reduce the performance impact on systems that do not use threshold behaviour. However, we choose to make the new seL4_YieldUntilBudget system call available to all builds, regardless of whether they compile threshold support into the kernel.

#### Configuration of thresholds

The key design choice is how we should permit thresholds to be configured:

- Set during endpoint creation and unchangeable afterwards.
- Configurable by users at any time via a system call.

We believe it would be a rare occurrence for a server's WCET to vary during the execution of a system. However, we concede that it is not impossible. As we expect the threshold value to be based on the WCET of a passive server, there should generally be no need for it to change while the system is running. Nevertheless, if we can support varying the threshold value at any time, without excessive compromising of kernel minimality, that would be preferable.

Supporting arbitrary configuration by users is possible, but we must impose some restrictions. Firstly, only a subset of capabilities to the endpoint should possess the right to set the threshold value. Otherwise, any client could reduce the threshold value before calling, which would effectively invalidate the protection the threshold provides. Unfortunately, none of the existing rights on capabilities (send, receive, grant and grandReply) map to the privilege of setting a threshold

value. To avoid needing to introduce a new right on capabilities, maintaining minimality, we choose to restrict this privilege to the original unbadged endpoint capability.

When considering arbitrary threshold changes by users, we must also consider how the changes in threshold value will be propagated to threads that are associated with the endpoint. For instance, if the threshold value is increased, should threads that are already enqueued on the endpoint have their available budget checked against the new threshold value? Also, consider a thread that previously had its budget deferred, but after the threshold value was decreased its available budget would have been sufficient. Should that thread now be enqueued on the endpoint immediately?

Given that we expect changing a threshold value after initialisation will be an exceedingly rare operation, we choose to prioritise kernel minimality and simplicity. With this consideration in mind, we decide on the following design. We permit the threshold value to be changed by user-level at any time, however only the thread possessing the original unbadged endpoint capability has this right. In line with minimality, we decide that any threads enqueued on the endpoint or that have had their budget deferred will not be immediately affected by the threshold value change. Those threads will only interact with the new threshold value when they next attempt to invoke the endpoint.

To support this, we introduce a new system call for setting the threshold of an IPC endpoint "seL4_Endpoint_SetThreshold". We recall that seL4 system calls are divided into true system calls, recognised by the kernel, and object invocations, which are invoked via a message passing system call on an object, with the kernel as the implicit recipient. Nearly all existing configuration system calls fall into the latter category, however this direct approach cannot be used for endpoints. The kernel has no way of distinguishing whether a sending invocation on an endpoint is intended as a system call or simply a normal message that happens to have the same parameters. We wish to avoid placing restrictions on messages that can be passed via IPC, so we must use this alternate approach.

This model allows users the freedom to change threshold values at any time, allowing us to support cases where this may be desirable, even though they may be rare. As we can support this without requiring significant increased kernel complexity, we settle on this model over only allowing configuration upon endpoint creation.

#### Permitted invocations

Currently, a client could invoke a passive server using a system call that does not permit SC donation. The IPC rendezvous would still occur, but the SC would not be donated. The passive server would therefore be unable to run, causing similar issues to budget expiry.

As our desired behaviour for endpoint thresholds is that passive servers are protected from budget expiry, this is unacceptable. To address this, we also place a restriction on the invocations that can be used to invoke an endpoint with a threshold set. Specifically, we only allow sending invocations that permit SC donation. Other invocations fail immediately and return an error

to the client. However, we place no restriction on what invocations can be used to receive on an endpoint with a threshold set.

### 6.1.4 Summary

We now present a summary of the design choices that we have made.

We have introduced a new *threshold* onto endpoints, associated with the object. By default, the threshold would be set to zero and in this state it would have no effect. This threshold value can be set using a new invocation, however, only the original unbadged endpoint capability possesses this right.

When an endpoint has a threshold set, threads are only allowed to send on the endpoint using invocations that permit SC donation. After a thread invokes a send operation, the available budget in the thread's SC is compared to the threshold value. If the available budget is sufficient, then the IPC continues as normal.

However, if the budget is insufficient, the refills in the SC are deferred and merged into a single refill that exceeds the threshold. The thread is then placed in the scheduler release queue to wait until this new head refill is released. At that point, the IPC invocation is retried by the kernel in a manner transparent to the thread. If instead the maximum budget of the SC is less than the threshold value, an error is returned to the calling thread.

We present a flowchart of this in Figure 6.3.

## 6.2 Passive server budget limits

In this section, we present a design that aims to limit the budget that a passive server can consume on a donated scheduling context. This will both bound priority-inversion and reduce the trust that clients must place in passive servers that they call.

### 6.2.1 Desired semantics

We base our design on the threshold behaviour presented above, proposing an extension to support the desired properties. We extend the meaning of endpoint thresholds, such that they can additionally represent a limit on the budget that can be consumed on the SC before it is returned to the client. We refer to this restriction on budget consumption as a *budget limit*.

If the passive server consumes more than this limit, we consider this to be a *budget overrun* and the kernel will forcibly return the SC to the client. The client will have its IPC aborted and an error message passed to it from the kernel. This may leave the server in a stuck state, so we would invoke the server's timeout handler to restore the server to a sane state. We note

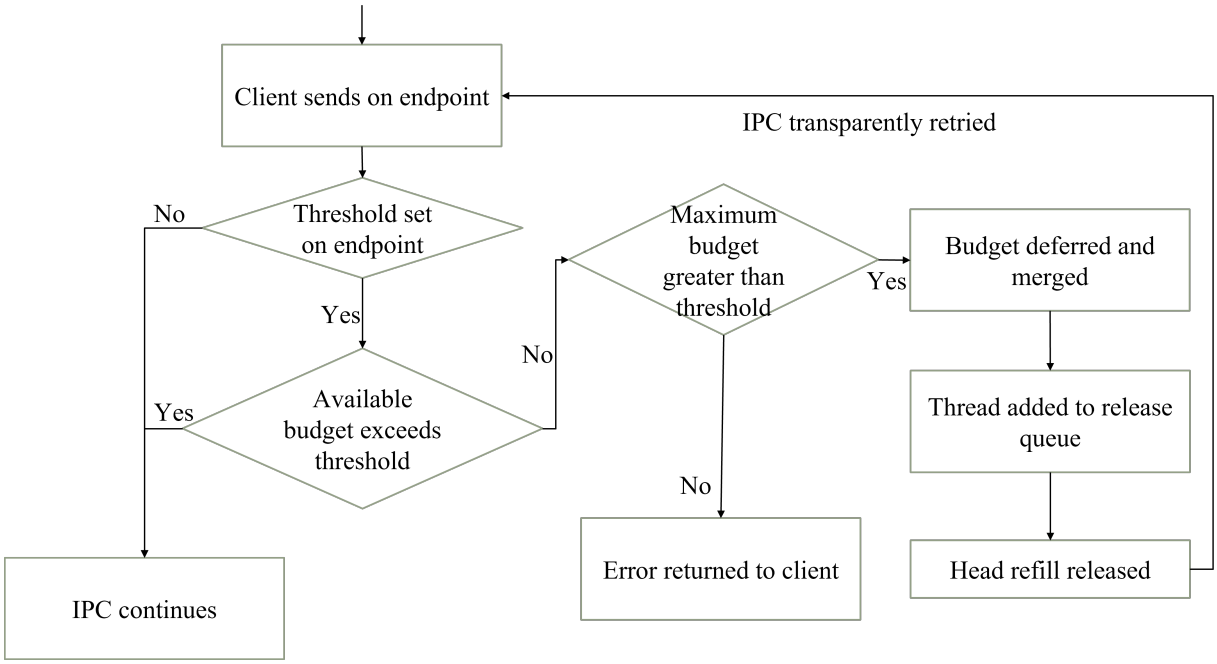


Figure 6.3: A flowchart of the processes that occur when an IPC is invoked on an endpoint with a threshold.

that we consider the guarantee to be between the SC and the client. Even if the SC is further donated to other servers, the guarantee between the SC and the original client should remain in effect. These semantics should ensure that a server can consume no more budget than the budget limit, bounding priority inversion caused by servers.

We note that we consider the threshold value to be the maximum budget that the server is permitted to consume. Therefore, the client will observe slightly more budget consumed on its SC, as a result of the kernel execution time involved in the IPC operation.

Additionally, the budget limit behaviour should be an optional extension of the threshold behaviour above. System designers should be able to choose whether an endpoint enforces solely threshold behaviour, or threshold and budget limit behaviour.

### 6.2.2 Configuring budget limit behaviour

We recall from the previous section that thresholds are individually configurable on endpoints. We now consider how we can permit system designers to enable budget limit behaviour on endpoints with thresholds set.

There are three broad options for how we could permit budget limits to be configured.

1. Configurable system-wide.

2. Configurable on each endpoint object.

3. Configurable on endpoint capabilities.

We consider the first option to be an excessive restriction on policy-freedom. Especially when considering servers of mixed-criticalities, it may be desirable to enforce differing strengths of guarantee on them.

We recall from the previous section that we decided to associate thresholds with endpoint objects rather than capabilities. We primarily chose this design as there was insufficient space to store the threshold value in the capability. However, unlike thresholds, a budget limit toggle requires only a single bit of space. Therefore, these could be accommodated within endpoint capabilities without requiring them to be increased in size.

This supports better policy-freedom, without the major implementation compromise that would occur if thresholds themselves were stored in endpoint capabilities. We therefore decide to associate the budget limit toggle with endpoint capabilities, allowing greater freedom for system designers.

However, we believe that it is not a reasonable design for budget limits to be set more granularly than threshold values. One of the goals of budget limits is to bound priority inversion. This bound will only be effective if it applies regardless of which client is calling into the passive server. If there is a client that can call into the passive server without budget limits in effect, it can cause longer priority inversion than the expected bound. Therefore, the scheduling guarantees offered would be significantly weakened. As a result, we consider the best design to associate the budget limit toggle with the endpoint object

However, if in the future, endpoint thresholds are moved to endpoint capabilities, then it would be natural for the budget limit toggle to also be moved with them. The obligation would then rest with system designers to ensure the budget limit is enabled if a bound on priority inversion is desired.

### 6.2.3 General design

For a single client and server pair, the implementation of our desired semantics is relatively simple. Reply objects are already used to track donated SC's, so we extend them with a new *sub_budget* field. Additionally, we add a field to SC's that tracks the budget consumed on it, along with a boolean that marks the SC as having budget limits enabled. When the client calls into the server over an endpoint with a threshold with budget limits enabled on the capability, the SC's consumed budget tracking is reset to zero, and it is marked as having budget limits enabled. Additionally, the reply object's sub_budget is set to the threshold value. If the SC's consumed budget reaches the sub_budget set in the reply object, the SC is returned to the client. The IPC operation is aborted, and then an error is returned to the client. Once the SC is returned to the client, regardless of the means, it is marked as no longer having budget limits enabled.

We now consider how this design would support nested calls, where in all cases we assume that SC donation is occuring. For brevity, we refer to an endpoint without a threshold set as a *normal endpoint* and an endpoint with a budget limit enabled as a *limit endpoint*. First, two nested call over normal endpoints do not require any specific behaviour, so this case is straightforward. No changes are required to support this case. Next we consider the behaviour where a server is first invoked over a normal endpoint, which then calls over a limit endpoint. As the call over the normal endpoint does not require any form of guarantee, we can treat this case effectively identically to the single client and server pair. The call over the limit endpoint requires a guarantee to enforced, but the original call over the normal endpoint does not impact the guarantee.

When we explore the case of two nested calls over limit endpoints, the situation is slightly more complex. We consider a client $C$ that calls over a limit endpoint into a server $s_1$, that then again calls into a server $s_2$ over a limit endpoint. In this situation, we restrict $s_1$ to only be able to call into $s_2$ if $s_1$'s remaining budget limit exceeds the threshold. We consider this a natural interaction, as the budget limit represents the budget that $s_1$ has available for use before the SC must be returned to the client $C$. This necessarily limits the second threshold value to be less than the first threshold value. There are two budget guarantees in effect, both to $C$ and $s_1$, that operate independently. If $s_2$ exceeds its budget limit, the SC will be forcibly returned to $s_1$, which will be allowed to continue running normally up to its own budget limit. If $s_1$ then exceeds its budget limit, then the SC will be forcibly returned to the client $C$.

With this design, the budget limits and guarantees of both the limit endpoints are enforced.

The final case involves an initial call over a limit endpoint, followed by a call over a normal endpoint. We again consider a client $C$ calling over a limit endpoint into a server $s_1$, then $s_1$ calls over a normal endpoint into $s_2$. In this case, there is a budget limit guarantee made only to $C$. If the budget limit is exceeded, the SC should be returned to $C$, regardless of whether it is bound to $s_1$ or $s_2$ at the time. If a budget overrun occurs in $s_2$, the SC will be returned directly to $C$, skipping over $s_1$. We therefore consider it necessary to invoke the timeout handlers of both $s_1$ and $s_2$, as otherwise the servers could be stuck and unable to receive new IPC messages.

However, this poses implementation issues. Reply objects form a doubly linked list, but only contain a reference to the thread directly prior in the call chain. In this example, the reply object associated with $s_2$ would only contain a reference to $s_1$. Therefore, when we consider how to return the SC to the client $C$, we have 3 broad options.

1. Walk down the reply stack to find the relevant thread.
2. Add direct return pointers to reply objects.
3. Prevent this call structure from occurring.

The first option requires no additional object changes as it involves iterating down the reply stack until the relevant thread is found. However, the kernel execution time for this operation is $O(n)$ based on the size of the reply stack. Additionally, our design requires that the timeout

handler for each thread skipped over to be invoked, until the thread the SC is returned to is found. This design would lead to longer kernel execution times and potentially multiple timeout handlers all running consecutively. This could result in significantly increased response times for the client thread, as a budget overrun would require a long set of operations before the client can run again. These issues reduce the usefulness of the budget limit for protecting clients from servers, as the scheduling guarantees it offers to clients are weaker. Overall, while this design is simple, we consider a kernel execution time heavily dependent on user level parameters undesirable and therefore consider other solutions.

The second option involves add a direct pointer from each reply object which references the thread that the budget limit guarantee is made to. When a budget overrun occurs, this direct pointer would theoretically allow for an $O(1)$ operation to handle the overrun. Notably, this allows us to find the thread the SC needs to be returned to without walking the reply stack. However, we do still need to walk the reply stack in order to invoke the timeout handlers of all the intermediate threads. Therefore, that requirement means that this design is not a major improvement from the first option.

Further, seL4 maintains a general symmetry, where if an object $X$ contains a reference to $Y$, that object $Y$ must contain a reference to $X$. This helps during the deletion of objects, to ensure that all references are cleaned up before the object is deleted. However, with the second option, there could potentially be many reply objects that all reference a client TCB. Therefore, it is not possible to store back references to all the reply objects and this option is not feasible.

The third option involves imposing a restriction on IPC to ensure that the call stack grows in a manner that we can account for. We saw above that the only problematic case is where a thread is first invoked over a limit endpoint and then calls over a normal endpoint. To prevent this, we create a restriction that a donated SC with a budget limit enabled can only be donated further over an endpoint with a budget limit set. When considering whether donation should be permitted, the kernel will additionally compare the threshold value against the remaining budget limit and only permit it if the budget limit is greater. We do not enforce any restriction on calling active servers. In a call chain that does not involve any endpoints with budget limits enabled, this change would have no effect. In that situation, threads would be free to donate SC's over any endpoint. However, once the SC has been donated over an endpoint with a budget limit set, that SC can only be donated further over other endpoints that also have thresholds set.

The basic effect of this is that servers invoked over endpoints with budget limits can only further call other servers over other budget limit endpoints. We note that this restriction does not apply to endpoints with thresholds where the budget limit is disabled.

We first consider how this restriction resolves the issue we encountered earlier. We consider an example of servers $s_1, s_2, s_3..s_n$ all calling the subsequent server, where a budget limit is in effect and SC donation is occurring. If budget overrun occurs in any server $s_x$, then the server that the SC should be returned to is always $s_{x-1}$, exactly one level down. This avoids the need to either walk the reply stack or maintain an additional pointer reference.

Calling into an active server involves a separate SC, on one where the budget limit behaviour is

not enabled. Therefore, we do not need to impose any restriction on calls into active servers.

The primary downside of this approach is the significant restriction of policy freedom, limiting how system designers can construct servers that communicate over IPC. However, we argue that in a time sensitive system, the restriction we impose is not unreasonable. We now explore the designs that we are restricting and consider the alternatives that are available to system designers.

First, the presence of an endpoint threshold and budget limit signifies that the server receiving on that endpoint has time guarantees associated with it. Specifically, the server is making a guarantee that the donated SC from the client will have at most 'threshold' budget consumed on it before it is returned. It would be a poor design pattern for that server ($s_1$) to forward donate the SC to another server ($s_2$) without any time guarantees, which is the case over an endpoint without a budget limit.

Instead, we believe that $s_1$ should only donate to $s_2$ over an endpoint with a budget limit set. This will create another budget limit guarantee, one enforced by the kernel, from $s_2$ to $s_1$. This helps $s_1$ ensure that it will be able to meet the budget limit obligation that it owes to the original client.

In the aforementioned example, a client donated a SC to $s_1$ over an endpoint with a budget limit, and then attempted to further donate that SC to $s_2$ over an endpoint without a budget limit. This design would not be permitted, but we believe there are two readily accessibly solutions:

1. Firstly, if strong timing guarantees are required, the WCET of $s_2$ should be determined and a budget limit threshold set on its endpoint. Then, donation can occur in a manner permitted by our model.
2. Alternatively, if strong timing guarantees are not required, then the budget limit on the first endpoint should be disabled. The threshold can remain enabled as it does not restrict the calls that can be made.

In the first instance, we recognise that to implement our design, the WCETs of all servers involved in the call chain must be known. However, in this case, we argue that the benefit of the budget limit is to strengthen the guarantee of the server's WCET value.

It is possible that a WCET is computed in a manner that is overly optimistic, such that the server's execution time is only bounded by it in most cases. However, under rare and exceptional circumstances, the server may exceed its WCET value. Such a WCET would be relatively untrustworthy, but a possible reason this may occur is to reduce the cost of a developing a system, as verifying the WCET of code can be very expensive. Enforcing this WCET with kernel budget limits increases the trustworthiness of the WCET, as even if the server exceeds it, the WCET will be enforced by the kernel. This strengthens the WCET and makes it more useful for schedulability analysis of the system.

The second alternative allows system designers to still prevent budget expiry from occurring in passive servers, without requiring any restrictions on how they configure their systems.

Therefore, we consider this restriction to not be a major issue, despite it infringing on policy-freedom. While it does restrict some user-level designs, there are readily available alternatives that we consider acceptable.

### 6.2.4 Revocation and deletion

In a capability-based system, such as seL4, kernel objects are owned by user threads. Therefore, kernel objects can be deleted at any time and we must consider the consequences of this on the guarantee that we are establishing.

We consider three main relevant kernel objects, thread controls blocks, scheduling contexts and reply objects. First, thread control blocks and scheduling contexts are core objects to the guarantee. If either the scheduling context or any threads that are part of call chain are deleted, we consider the guarantee invalidated. If the scheduling context is manually bound or unbound, we also consider the guarantee to be invalidated.

However, reply objects are a unique case. Generally, threads do not hold capabilities to their own TCB or SC. However, servers do own capabilities to their reply objects. The consequence is that a server can potentially delete its reply object, disrupting the tracking of the call chain that we require to maintain the budget limit guarantee.

We consider it potentially desirable for the budget-limit guarantee to be preserved even if a reply object is deleted. Otherwise, the guarantee made to the client could be easily invalidated by the server by deleting its reply object. This means that client's still need to maintain relatively high levels of trust in their servers, reducing the benefit of budget limits.

However, we leave this extension for future work, and in the current model presented we still allow the guarantee to be invalidated if reply objects are deleted.

# Chapter 7

# Evaluation

We now evaluate our design using a series of benchmarks. First, we present an example demonstrating endpoint threshold's preventing budget expiry from occuring, even in the prescence of a malicious client.

Secondly, IPC cost is well known to be a major performance factor of microkernels Liedtke [1993]. As our design involves change on the IPC path, we also aim to demonstrate that the overhead of our changes on the IPC path is acceptable.

## 7.1 Hardware

We perform these tests on an NXP i.MX8-MM evaluation kit, containing the ARMv8-A based Cortex-A53 SoC running at 1.2GHz with 2GiB of physical RAM. The system contains 4 processing cores, but all threads are pinned to the first CPU core.

## 7.2 Endpoint thresholds

We first consider the performance impact where only endpoint thresholds are compiled into the kernel, without kernel support for budget limits. In the following section we will consider the additional performance overhead introduced by budget limits.

Unless otherwise specified, all tests are performed using SC's with no additional refills.

### 7.2.1 Preventing Timeout Exceptions

Firstly, we test that endpoint thresholds can prevent timeout exceptions from occurring in a passive server, even where a client is malicious. We set up a passive server that loops tightly,

with the loop parameters being carefully set such that the server requires very close to $10ms$ of budget. We can therefore also consider this value its WCET.

Then, we also set up a malicious client of the passive server. This client has a SC with a budget of $12ms$ and a period of $20ms$. The client calls the server in a loop, but before each call it consumes an incrementally larger amount of budget, with the goal of causing a timeout exception in the server. We configured our client to burn its budget by also executing a tight loop. Over subsequent calls, the client extended its loop to burn increased quantities of budget. There is additionally a timeout handler configured for the server, which exists only to count the number of timeout exceptions caused. We first run a test using the baseline kernel, then our modified kernel with a range of threshold values set. We present the results from this test in table Table 7.1 below. The tests were repeated multiple times, but there was no variance between runs. In the table, we additionally present the budget gap, which is the difference between the threshold value set and the server's WCET.

| Threshold value ($\mu S$) | Budget gap ($\mu S$) | Timeout Exceptions |
|---|---|---|
| 0 (baseline kernel) | 10000 | 260 |
| 0 (modified kernel) | 10000 | 260 |
| 10000 | 0 | 0 |
| 9950 | 50 | 7 |
| 9900 | 100 | 16 |
| 9500 | 500 | 85 |
| 9000 | 1000 | 170 |

Table 7.1: Timeout exceptions with varying threshold values

Without a threshold set, on either the baseline or the modifed kernel, the server experiences many timeout exceptions. When we instead set the threshold value to $10000\mu s$, equal to the WCET of the passive server, timeout exceptions are completely eliminated.

However, if we reduce the threshold, even slightly to $9950\mu s$, a few timeout exceptions occur. This aligns with our expectations, as we have created a budget gap of $50\mu s$, between $9950\mu s$ and $10000\mu s$. If the client donates a budget above $9950\mu s$, but below the server's WCET of $10000\mu s$, donation will be permitted as it exceeds the threshold, but budget expiry will still occur as the donated budget is insufficient. In line with our expectations, as we further reduce the threshold value and the budget gap increases, the number of timeout exceptions that occur corresponding increase, relatively linearly. Finally, we note that the client did not reduce its donated budget down close to zero. Therefore, the linear relationship does not map neatly to the $0\mu s$ threshold cases.

Overall, we observe that when properly configured, thresholds can completely eliminate timeout exceptions. They behave as expected when configured lower than the server's WCET, predictably allowing only a subset of budget values to pass.

### 7.2.2 System call overheads

As part of the changes made to support endpoints, we introduced a new variant of seL4_Yield, seL4_YieldUntilBudget. However, this introduces some additional kernel overhead even on unrelated slowpath invocations. This is likely because there is an additional system call numbers that the kernel must branch through when checking.

As we do not intend to restrict this new variant of yield to only builds with threshold behaviour enabled, this overhead will affect all systems running on the MCS kernel. These overheads were measured using the existing seL4Bench benchmarking suite and we present results for a number of the basic seL4 system calls.

| System call | Baseline | With seL4_YieldUntilBudget | Overhead |
|---|---|---|---|
| seL4_Call (Fastpath) | 265 (0) | 267 (3) | $< 1\%$ |
| seL4_ReplyRecv (Fastpath) | 290 (0) | 289 (0) | $< 1\%$ |
| seL4_Call (Slowpath) | 947 (14) | 948 (14) | $< 1\%$ |
| seL4_ReplyRecv (Slowpath) | 972 (16) | 975 (14) | $< 1\%$ |
| seL4_Send | 783 (9) | 798 (8) | 2% |

Table 7.2: System call overheads from the introduction of seL4_YieldUntilBudget. Results are cycles, presented as: mean (standard deviation)

As the new system call is only introduced on the slowpath, as expected, the fastpath results are essentially unaffected. Various operations on the slowpath are all slightly slower, but none of the overheads introduced exceed 2%.

While we consider these overheads acceptable, they are surprisingly high. The new system call we have introduced is not being executed in these examples, rather its mere presence is causing moderate overheads for unrelated system calls. On the slowpath, the kernel uses a switch statement to choose which operation to take based on the system call number passed from user level. Our current hypothesis is that the additional system call caused a change in how the compiler handled this switch statement, leading to slightly increased slowpath costs. However, we leave a thorough investigation of this behaviour for future work.

### 7.2.3 IPC overhead

We now specifically investigate the overheads we have introduced on the IPC path due to thresholds. We first consider the overheads on a successful IPC rendezvous. Afterwards, we will consider the kernel cost involved when the available budget is insufficient and budget must be deferred and merged.

### Successful IPC overhead

To measure the additional overhead introduced by endpoint thresholds, we set up IPC path cost tests based on the seL4Bench benchmarking suite. The existing IPC bencmarks set up a client and a passive server. The client reads the CPU cycle counter, then calls onto the endpoint. The passive server is waiting to receive on the endpoint and immediately afterwards, it also reads the CPU cycle counter. The difference between the two cycle counts read is computed as the kernel cost of the IPC operation.

We first run this test unmodified on the baseline kernel. Then, we again run the same test with threshold support compiled into the kernel, but the threshold value on the endpoint is disabled by setting it to zero. Next, we modify the test to set a very small threshold on the endpoint, such that the client's SC will always have enough budget and the IPC will succeed. However, the kernel will still need to compare the SC's available budget against the threshold.

We perform these tests for both the fastpath and the slowpath. We force the kernel to use the IPC slowpath by setting the message length to 10, which means the message cannot fit solely in CPU registers. The results of these tests are presented in table Table 7.3.

| | Fastpath | Fastpath overhead | Slowpath | Slowpath overhead |
|---|---|---|---|---|
| Baseline | 265 (0) | N/A | 947 (14) | N/A |
| Thresholds disabled | 279 (4) | 5% | 959 (10) | 1% |
| Thresholds enabled | 304 (2) | 15% | 976 (12) | 3% |

Table 7.3: IPC Call overhead from endpoint thresholds. Results are cycles, presented as: mean (standard deviation)

From these results, we can see that the introduction of endpoint thresholds has a measureable cost, however in general these overheads are relatively modest, all being under 15%. Notably, with thresholds compiled into the kernel, but disabled on the endpoint, the fastpath overhead is 5%. For the fastpath, the overhead with a threshold enabled is around 15%, while on the slowpath, the overhead is 3%. On the fastpath, this overhead can be entirely attributed to the cost of comparing the available budget against the threshold value. On the slowpath, the overhead is proportionally lower as the overall path cost is higher.

### Deferred IPC overhead

With the addition of threshold endpoints, there is a new case that can occur when calling into an endpoint. If the donated SC's maximum budget is greater than the threshold, but the current available budget is insufficient, the refills of the SC are merged are deferred. The broad kernel operations that are required during this process are

1. Checking the available budget against the endpoint threshold
2. Deferring and merging the SC's budget

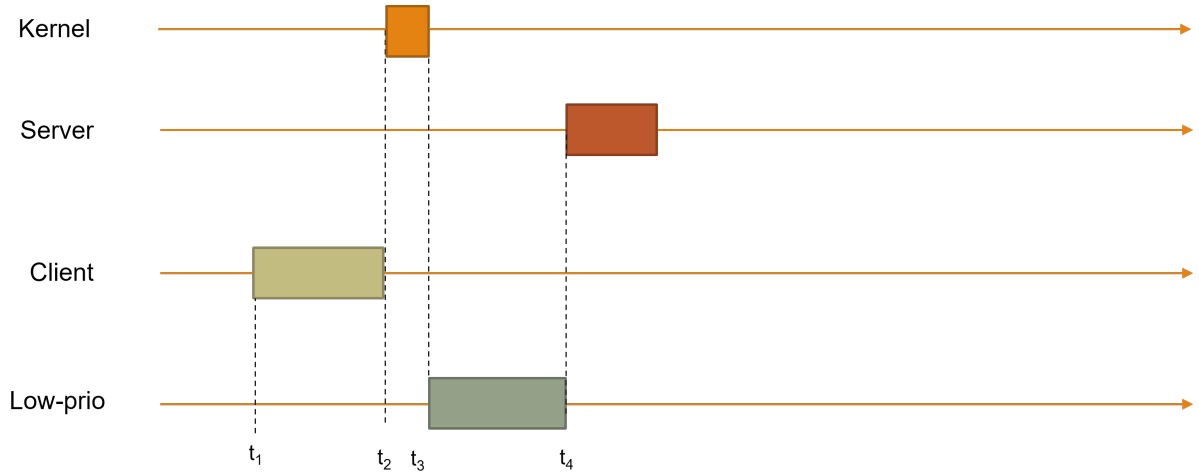


Figure 7.1: The structure of the deferred IPC overhead test. We omit kernel execution times that are not of interest in this test.

3. Inserting the thread into the release queue

4. Choosing another thread to schedule

To benchmark this, we set up a new test involving a client and a low-prio thread, with a lower priority than the client. We illustrate the structure of this test in Figure 7.1. Note that in the figure we only show the kernel execution time we are interested in, omitting the rest. First, we set up the low-prio thread as runnable, but pre-empted by the client. This is the state of the test at $t_1$. Then, between $t_1$ and $t_2$, the client consumes some of its budget so that its available budget is insufficient to pass the threshold. Then, at $t_2$, the client reads the cycle counter and calls onto the endpoint. Between $t_2$ and $t_3$, the kernel will defer and merge the client's SC's budget. Once this operation is complete, at $t_3$, the low-prio thread will no longer be pre-empted and start running. It then immediately reads the cycle counter. The difference can again be used to determine the kernel time required for the operation. At $t_4$ the passive server receives and replies on the endpoint, however it does not play a role in benchmarking, rather its role is to reset the test and allow the client to continue running. Finally, we also configure a high priority thread, omitted from the that yields its budget in a tight loop. Its role is to pre-empt the client, causing its budget to be split into multiple refills. We perform the test with the client's SC configured to support various numbers of extra refills.

There is not an existing direct analogue to use as a baseline, so we instead compare against two existing kernel operations. For this section, we use our modified kernel for all tests. First, we compare against a successful call, for a comparison against the existing successful IPC costs. Then, we also set up a similar benchmark to above, but the client calls onto an endpoint with no receiver, but also no threshold set. This will cause the client to be blocked, which will also require the kernel to invoke the scheduler and resume the low-prio thread. As before, we present results for both the fastpath and the slowpath, in Table 7.4. We force the kernel to use the slowpath by increasing the size of the message payload such that it does not fit in CPU

registers.

| Operation | Extra refills | Fastpath | Slowpath |
|---|---|---|---|
| IPC call | N/A | 279 (4) | 959 (15) |
| IPC block | N/A | 852 (11) | 796 (17) |
| Threshold defer | 0 | 1150 (24) | 1015 (17) |
| Threshold defer | 10 | 1391 (18) | 1300 (26) |
| Threshold defer | 20 | 1665 (25) | 1545 (19) |
| Threshold defer | 30 | 1942 (30) | 1819 (20) |
| Threshold defer | 40 | 2174 (18) | 2079 (23) |
| Threshold defer | 50 | 2446 (24) | 2331 (29) |

Table 7.4: IPC Call overhead from endpoint thresholds. Results are cycles, presented as: mean (standard deviation)

First, while we present the IPC fastpath call costs for comparison, as expected, its cost is far lower than all the others. The seL4 fastpath is a highly optimised codepath that is only applicable under a specfic set of conditions, such as the IPC payload fitting in CPU registers. If any of these conditions are not satisfied, then the kernel switches over to the slowpath. Blocking on an IPC endpoint is a case where the kernel switches over from the fastpath to the slowpath.

We note that the fastpath IPC block cost is higher than the slowpath cost. Our current hypothesis is that the larger message payload causes the kernel to immediately switch over to the slowpath code. However, with the smaller number of message registers, the kernel performs more checks on the fastpath, only switching over to the slowpath when it finds that there is no waiting receiver thread. This would lead to the increased observed fastpath cost.

We believe a similar effect causes the fastpath threshold defer costs to be higher than the slowpath costs. In the fastpath cases, the kernel only switches over to the slowpath after it has performed the budget calculation and determined that the budget is insufficient. This would explain the extra cost observed in the fastpath results..

As is expected, deferring and merging budget involves significant cost compared to either a slowpath call or a thread blocking on an endpoint. With a large number of refills (50), the threshold defer cost is about 2.5 times the cost of a slowpath IPC, which is a substantially larger kernel cost. However, with a more modest quantity of refills, the defer cost is significantly lower. With 10 refills, the kernel cost required to defer and merge the budget is around 1.5 times the cost of a slowpath call.

This is still a significant increase in cost, however, we still consider these increased kernel costs acceptable in the context of avoiding budget expiry.

## 7.3 Budget limits

In this section, we present the additional overheads introduced by the budget limit mechanism. We evaluate the performance of a kernel compiled with support for both thresholds and the

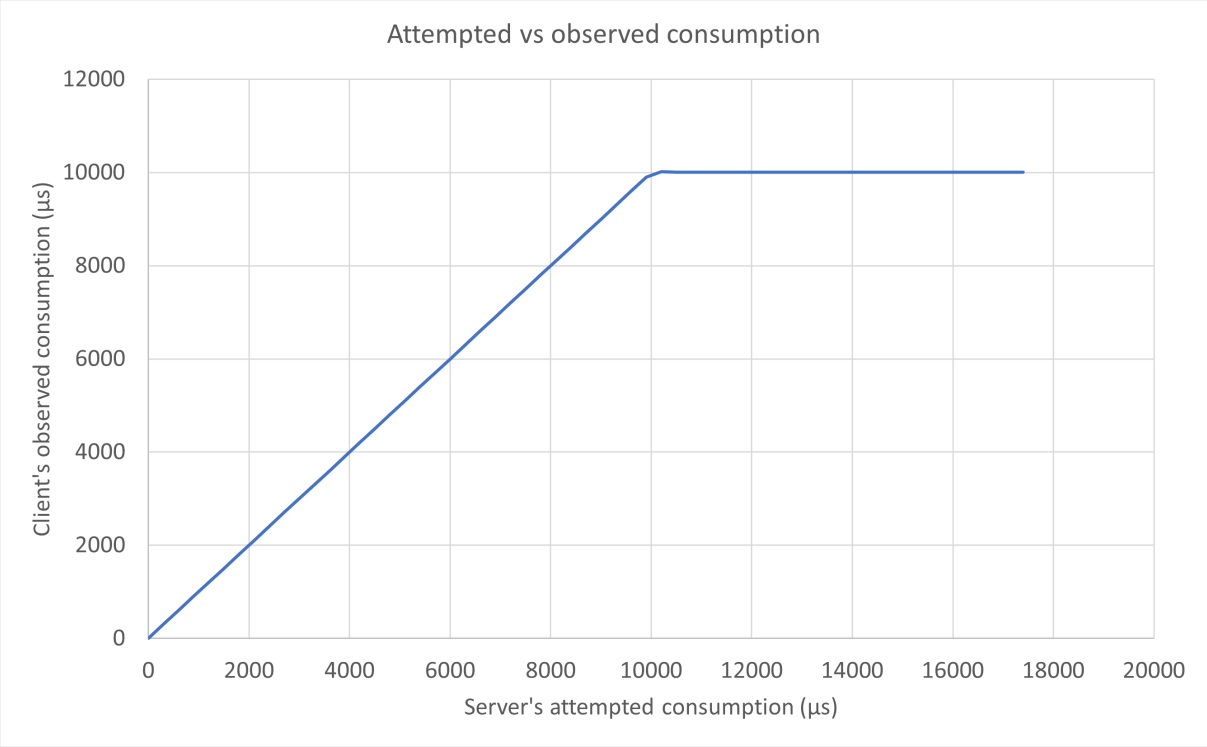


Figure 7.2: Budget consumed on the client's SC against the budget that the server tries to consume.

budget limit extension. Similar to the above, we first demonstrate that our solution is effective and then consider the overheads we have introduced on the IPC path.

### 7.3.1 Preventing budget overrun

We first test that thresholds with budget limits can prevent budget overrun. Our setup involves a client and a server. The client first resets the consumed time tracking on its SC. It then calls into the server and after the server replies, or the IPC is aborted, the client checks how much time has been consumed on its SC. This allows us to determine how much time the server consumes.

Before replying to the client, the server tries to consume a particular amount of budget using a carefully crafted loop. Over subsequent runs the server tries to consume increasing quantities of budget, in increments of $300\mu s$. We perform all these tests over an endpoint with a threshold set to $10000\mu s$ with budget limits enabled.

In Figure 7.2, we observe that when the server attempts to consume less budget than the threshold of $10000\mu s$, the relationship is extremely linear. This reflects that in these cases, as expected, the budget limit is having no effect. However, once the server tries to consume more than the threshold, the budget limit takes effect, capping the consumption at the threshold

value. Regardless of how much extra budget the server attempts to consume, the budget limit successfully limits the consumed budget to the threshold value.

We do note that the consumption is not strictly capped at the threshold value of $10000\mu s$. Rather, when the server tried to consume more budget than the threshold, the client observed $10011\mu s$ of budget consumed on its SC. However, this is in line with our expectations.

Our model considers the threshold value the amount of budget that the server is permitted to consume. However, the version of the kernel that we based our work on does not precisely account for its own execution time. Rather, it attributes its execution time to the thread that runs afterwards. Therefore, when the kernel sets the timer interrupt to enforce the budget limit for the server, it must allow for its own execution time during the call operation. As a result, the kernel must set an interrupt for the threshold value plus the kernel WCET. This ensures that even if the kernel consumed its full WCET, the server is still able to run for the full threshold value. There is then some further kernel execution time that occurs when the SC is returned to the client. On our test system, the kernel WCET was configured as $10\mu s$. Therefore, the observed consumed budget of $10011\mu s$ is in line with our expectations.

In this manner, we can see that the budget limit is effectively limiting the budget consumed by a passive server, reducing the trust that a client must place in passive servers that it calls.

### 7.3.2 IPC overheads

To measure the IPC path costs, we again run a test based on the seL4bench suite. Similar to the above, it involves two threads that perform IPC operations with one another. By reading the cycle counter before and after, the kernel time taken can be determined.

In this evaluation we present results for both seL4_Call and seL4_ReplyRecv, as our design required changes on both code paths. We again present results for endpoints with the threshold disabled (set to zero) and budget limits enabled, where the threshold is a non-zero value. We present our results for seL4_Call in Table 7.5 and seL4_ReplyRecv in Table 7.6.

| | Fastpath | Fastpath overhead | Slowpath | Slowpath overhead |
|---|---|---|---|---|
| Baseline | 265 (0) | N/A | 947 (14) | N/A |
| Thresholds disabled | 282 (0) | 6% | 1085 (12) | 15% |
| Budget limit enabled | 363 (0) | 37% | 1127 (11) | 19% |

Table 7.5: seL4_Call overhead from endpoint thresholds. Results are cycles, presented as: mean (standard deviation)

From these results, we see that the overheads introduced by budget limits are far more significant. Where thresholds are disabled, the additional cost on the fastpaths are relatively small, both less than 6%. However, even with thresholds disabled, the slowpaths experienced a more significant increase in overhead of 13% to 15%. We attribute this additional cost to the additional budget tracking that the kernel must perform. This includes checking whether a thread has exceeded its

| | Fastpath | Fastpath overhead | Slowpath | Slowpath overhead |
|---|---|---|---|---|
| Baseline | 290 (0) | N/A | 972 (16) | N/A |
| Thresholds disabled | 300 (3) | 3% | 1102 (13) | 13% |
| Budget limit enabled | 352 (5) | 21% | 1208 (17) | 24% |

Table 7.6: seL4_ReplyRecv overhead from endpoint thresholds. Results are cycles, presented as: mean (standard deviation)

budget limit and setting the timer interrupt to enforce when a budget limit would be exceeded. We do however, believe that there is room for further optimisations in this area.

When budget limits are enabled, the overheads on the fastpath are significantly greater, up to 37%. We attribute this cost to the additional bookkeeping required for enforcing budget limits, the additional fields to be written to the SC and reply objects. In addition, the timer interrupt may also need to be set. Finally, there is also an increase in the cost for the slowpaths where budget limits are enabled, which can be similarly attributed to the additional bookkeeping required.

Overall, while these overheads are significant, we believe that the stronger scheduling guarantees they provide are worth the trade-off for mixed-criticality real-time systems. While performance is always important, for these systems, reliability and guaranteed behaviour are essential. Further, we believe it is highly likely that additional optimisations are possible, reducing the overheads introduced. However, we leave such optimisations for future work.

## 7.4 Summary

Overall, these results demonstrate that the proposed changes effectively manage budgets, completely preventing both budget expiry and budget overrun when correctly configured. The overheads from threshold behaviour alone are modest and acceptable, allowing for the system to enforce stronger scheduling guarantees without compromising on performance. The overheads from the introduction of budget limits are more significant, but we believe these are acceptable in exchange for the stronger scheduling guarantees provided.

# Chapter 8

# Conclusion

In this thesis, we presented improvements to the seL4 model for mixed-criticality systems. Our changes improved the scheduling behaviour of seL4 in regard to avoiding budget expiry and preventing budget overrun. We have demonstrated that these changes are effective, improving the scheduling behaviour and schedulability analysis of seL4 systems.

However, our changes introduced moderate overheads into the kernel, and we left a portion of our design, specifically the revocation behaviour of budget limits unimplemented. These areas would benefit from future development, improving the performance and further strengthening the scheduling properties that seL4 offers.

## 8.1 Future Work

There are still a number of issues and model improvements that need to be addressed before seL4 can be considered a truly effective kernel for mixed-criticality systems. A development of the aforementioned revocation behaviour for our budget limit design would strengthen the guarantees provided and further reduce the trust that threads in the system must place in one another. Further, the designs presented in this thesis would also benefit from further performance optimisations, reducing the overheads they impose.

This thesis has not given any consideration to multicore systems, instead we focused our attention on single-core systems only. A future rigorous analysis of the scheduling behaviour of seL4 on multicore systems would be important to effectively apply seL4 to a wider range of hardware platforms.

This thesis also did not explore the impact of hardware interrupts on scheduling behaviour. Further work is required to enforce scheduling properties on interrupts and interrupt handlers.

Finally, after further improvements to the scheduling behaviour and guarantees, the scheduling properties of seL4 could be verified. This would build upon seL4's heritage of being a ver-

ified microkernel, and would provide the strongest proof of seL4's scheduling properties and guarantees.

# Bibliography

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, mar 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL `https://doi.org/10.1145/365230.365252`.

Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. Temporal capabilities: Access control for time. In *Proceedings of the 38th IEEE Real-Time Systems Symposium, pages 56–67*, December 2017.

Phani Kishore Gadepalli, Runyu Pan, and Gabriel Parmer. Slite: OS support for near zero-cost, configurable scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 160-173*, 2020.

Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1), apr 2016. ISSN 0734-2071. doi: 10.1145/2893177. URL `https://doi.org/10.1145/2893177`.

Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. pages 217–230, 01 2001.

Zubin Hu, Jianchao Luo, Xiyu Fang, Kun Xiao, Bitao Hu, and Lirong Chen. Real-time schedule algorithm with temporal and spatial isolation feature for mixed criticality system. In *7th International Symposium on System and Software Reliability (ISSSR), pages 99-108*, 2021.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. Technical report, October 2009.

Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. pages 93–102, 10 2012. doi: 10.1145/2380356.2380376.

Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 175–188, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897916328. doi: 10.1145/168619.168633. URL `https://doi.org/10.1145/168619.168633`.

Jochen Liedtke. On $\mu$-kernel construction. *SOSP*, December 1995. URL `www.cs.uah.edu/~weisskop/papers/LiedkeSOSP.pdf`.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, jan 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL `https://doi.org/10.1145/321738.321743`.

Anna Lyons. *Mixed-Criticality Scheduling and Resource Sharing for High-Assurance Operating Systems*. PhD thesis, University of New South Wales, September 2018.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190539. URL `https://doi.org/10.1145/3190508.3190539`.

Curtis Millar. Guaranteed response time for mixed-criticality systems on seL4. May 2021.

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. In *IEEE Symposium on Security and Privacy, pp. 415–429)*, May 2013.

Gabriel Parmer. *Composite: A component-based operating system for predictable and dependable computing*. PhD thesis, Boston Univertisy, August 2009.

Brinkely Sprunt, Lul Sha, and John Lehoczky. Scheduling sproad and aperiodic events in a hard real-time system. Technical report, April 1989.

Mark Stanovich, Theodore P. Baker, An-I Wang, and Michael Gonzalez Harbour. Defects of the posix sporadic server and how to correct them, (revised september 16, 2011). Technical report, September 2011.

Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605585772. doi: 10.1145/1755913.1755935. URL `https://doi.org/10.1145/1755913.1755935`.

Udo Steinberg, Alexander Böttcher, and Bernhard Kauer. Timeslice donation in component-based systems. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, pages 16–22, July 2010.

Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *IEEE Real-Time Systems Symposium, pages 239–243*, 2007.

Marcus Völp, Adam Lackorzynski, and Hermann Härtig. On the expressiveness of fixed-priority scheduling contexts for mixed-criticality scheduling. 2013.